e3nn

Release 0.5.0

e3nn Developers

Aug 08, 2022

CONTENTS

CHAPTER

ONE

WHAT IS `E3NN`?

`e3nn` is a python library based on pytorch to create equivariant neural networks for the group $O(3)$.

CHAPTER

TWO

WHERE TO START?

- Guide to the `e3nn.o3.Irreps`: *Irreducible representations*
- Guide to implement a *Convolution*
- The simplest example to start with is *Tetris Polynomial Example*.
- Guide to implement a *Transformer*

2.1 e3nn API

2.1.1 o3

All functions in this module are accessible via the `o3` submodule:

```
from e3nn import o3

R = o3.rand_matrix(10)
D = o3.Irreps.spherical_harmonics(4).D_from_matrix(R)
```

Overview

Parametrization of Rotations

Matrix Parametrization

e3nn.o3.**rand_matrix**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

random rotation matrix

Parameters

***shape** (*int*) –

Returns

tensor of shape $(\text{shape}, 3, 3)$

Return type

`torch.Tensor`

e3nn.o3.**matrix_x**(*angle: Tensor*) → Tensor

matrix of rotation around X axis

Parameters
angle (torch.Tensor) – tensor of any shape $(...)$

Returns
matrices of shape $(..., 3, 3)$

Return type
torch.Tensor

e3nn.o3.**matrix_y**(*angle: Tensor*) → Tensor

matrix of rotation around Y axis

Parameters
angle (torch.Tensor) – tensor of any shape $(...)$

Returns
matrices of shape $(..., 3, 3)$

Return type
torch.Tensor

e3nn.o3.**matrix_z**(*angle: Tensor*) → Tensor

matrix of rotation around Z axis

Parameters
angle (torch.Tensor) – tensor of any shape $(...)$

Returns
matrices of shape $(..., 3, 3)$

Return type
torch.Tensor

Euler Angles Parametrization

e3nn.o3.**identity_angles**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

angles of the identity rotation

Parameters
***shape** (*int*) –

Returns

- **alpha** (torch.Tensor) – tensor of shape (shape)
- **beta** (torch.Tensor) – tensor of shape (shape)
- **gamma** (torch.Tensor) – tensor of shape (shape)

e3nn.o3.**rand_angles**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

random rotation angles

Parameters
***shape** (*int*) –

Returns

- **alpha** (torch.Tensor) – tensor of shape (shape)
- **beta** (torch.Tensor) – tensor of shape (shape)
- **gamma** (torch.Tensor) – tensor of shape (shape)

e3nn.o3.**compose_angles**(*a1*, *b1*, *c1*, *a2*, *b2*, *c2*)

compose angles

Computes (a, b, c) such that $R(a, b, c) = R(a_1, b_1, c_1) \circ R(a_2, b_2, c_2)$

Parameters

- **a1** (torch.Tensor) – tensor of shape $(...)$, (applied second)
- **b1** (torch.Tensor) – tensor of shape $(...)$, (applied second)
- **c1** (torch.Tensor) – tensor of shape $(...)$, (applied second)
- **a2** (torch.Tensor) – tensor of shape $(...)$, (applied first)
- **b2** (torch.Tensor) – tensor of shape $(...)$, (applied first)
- **c2** (torch.Tensor) – tensor of shape $(...)$, (applied first)

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**inverse_angles**(*a*, *b*, *c*)

angles of the inverse rotation

Parameters

- **a** (torch.Tensor) – tensor of shape $(...)$
- **b** (torch.Tensor) – tensor of shape $(...)$
- **c** (torch.Tensor) – tensor of shape $(...)$

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

Quaternion Parametrization

e3nn.o3.**identity_quaternion**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

quaternion of identity rotation

Parameters

***shape** (*int*) –

Returns

tensor of shape $(\text{shape}, 4)$

Return type

torch.Tensor

e3nn.o3.**rand_quaternion**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

generate random quaternion

Parameters

***shape** (*int*) –

Returns
tensor of shape $(\text{shape}, 4)$

Return type
torch.Tensor

e3nn.o3.**compose_quaternion**(*q1*, *q2*)

compose two quaternions: $q_1 \circ q_2$

Parameters

- **q1** (torch.Tensor) – tensor of shape $(..., 4)$, (applied second)
- **q2** (torch.Tensor) – tensor of shape $(..., 4)$, (applied first)

Returns
tensor of shape $(..., 4)$

Return type
torch.Tensor

e3nn.o3.**inverse_quaternion**(*q*)

inverse of a quaternion

Works only for unit quaternions.

Parameters
q (torch.Tensor) – tensor of shape $(..., 4)$

Returns
tensor of shape $(..., 4)$

Return type
torch.Tensor

Axis-Angle Parametrization

e3nn.o3.**rand_axis_angle**(**shape*, *requires_grad=False*, *dtype=None*, *device=None*)

generate random rotation as axis-angle

Parameters
***shape** (*int*) –

Returns

- **axis** (torch.Tensor) – tensor of shape $(\text{shape}, 3)$
- **angle** (torch.Tensor) – tensor of shape (shape)

e3nn.o3.**compose_axis_angle**(*axis1*, *angle1*, *axis2*, *angle2*)

compose $(\vec{x}_1, \alpha_1)$ with $(\vec{x}_2, \alpha_2)$

Parameters

- **axis1** (torch.Tensor) – tensor of shape $(..., 3)$, (applied second)
- **angle1** (torch.Tensor) – tensor of shape $(...)$, (applied second)
- **axis2** (torch.Tensor) – tensor of shape $(..., 3)$, (applied first)
- **angle2** (torch.Tensor) – tensor of shape $(...)$, (applied first)

Returns

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Convertions

e3nn.o3.**angles_to_matrix**(*alpha*, *beta*, *gamma*)

conversion from angles to matrix

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

Returns

matrices of shape $(..., 3, 3)$

Return type

torch.Tensor

e3nn.o3.**matrix_to_angles**(*R*)

conversion from matrix to angles

Parameters

R (torch.Tensor) – matrices of shape $(..., 3, 3)$

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**angles_to_quaternion**(*alpha*, *beta*, *gamma*)

conversion from angles to quaternion

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

Returns

matrices of shape $(..., 4)$

Return type

torch.Tensor

e3nn.o3.**matrix_to_quaternion**(*R*)

conversion from matrix R to quaternion q

Parameters

R (torch.Tensor) – tensor of shape $(..., 3, 3)$

Returns

tensor of shape $(..., 4)$

Return type
torch.Tensor

e3nn.o3.**axis_angle_to_quaternion**(*xyz*, *angle*)

convertion from axis-angle to quaternion

Parameters

- **xyz** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Returns
tensor of shape $(..., 4)$

Return type
torch.Tensor

e3nn.o3.**quaternion_to_axis_angle**(*q*)

convertion from quaternion to axis-angle

Parameters
q (torch.Tensor) – tensor of shape $(..., 4)$

Returns

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**matrix_to_axis_angle**(*R*)

conversion from matrix to axis-angle

Parameters
R (torch.Tensor) – tensor of shape $(..., 3, 3)$

Returns

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**angles_to_axis_angle**(*alpha*, *beta*, *gamma*)

conversion from angles to axis-angle

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

Returns

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**axis_angle_to_matrix**(*axis*, *angle*)

conversion from axis-angle to matrix

Parameters

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Returns
tensor of shape $(..., 3, 3)$

Return type
torch.Tensor

e3nn.o3.**quaternion_to_matrix**(*q*)

convertion from quaternion to matrix

Parameters
q (torch.Tensor) – tensor of shape $(..., 4)$

Returns
tensor of shape $(..., 3, 3)$

Return type
torch.Tensor

e3nn.o3.**quaternion_to_angles**(*q*)

convertion from quaternion to angles

Parameters
q (torch.Tensor) – tensor of shape $(..., 4)$

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

e3nn.o3.**axis_angle_to_angles**(*axis*, *angle*)

convertion from axis-angle to angles

Parameters

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$

Convertions to point on the sphere

e3nn.o3.**angles_to_xyz**(*alpha*, *beta*)

convert (α, β) into a point (x, y, z) on the sphere

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$

Returns
tensor of shape $(..., 3)$

Return type
torch.Tensor

Examples

```
>>> angles_to_xyz(torch.tensor(1.7), torch.tensor(0.0)).abs()
tensor([0., 1., 0.])
```

e3nn.o3.**xyz_to_angles**(*xyz*)

convert a point $\vec{r} = (x, y, z)$ on the sphere into angles (α, β)

$$\vec{r} = R(\alpha, \beta, 0)\vec{e}_z$$

Parameters
xyz (torch.Tensor) – tensor of shape $(..., 3)$

Returns

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$

Irreps

A group representation (D, V) describe the action of a group G on a vector space V

$$D : G \longrightarrow \text{linear map on } V.$$

The irreducible representations, in short *irreps* (definition of irreps) are the "smallest" representations.

- Any representation can be decomposed via a change of basis into a direct sum of irreps
- Any physical quantity, under the action of $O(3)$, transforms with a representation of $O(3)$

The irreps of $SO(3)$ are called the wigner matrices D^L. The irreps of the group of inversion $(\{e, I\})$ are the trivial representation σ_+ and the sign representation σ_-

$$\sigma_p(g) = \begin{cases} 1 \text{ if } g = e \\ p \text{ if } g = I \end{cases}.$$

The group $O(3)$ is the direct product of $SO(3)$ and inversion

$$g = ri, \quad r \in SO(3), i \in \text{inversion}.$$

The irreps of $O(3)$ are the product of the irreps of $SO(3)$ and inversion. An instance of the class *e3nn.o3.Irreps* represent a direct sum of irreps of $O(3)$:

$$g = ri \mapsto \bigoplus_{j=1}^{n} m_j \times \sigma_{p_j}(i) D^{L_j}(r)$$

where $(m_j \in \mathbb{N}, p_j = \pm 1, L_j = 0, 1, 2, 3, \dots)_{j=1}^{n}$ defines the *e3nn.o3.Irreps*.

Irreps of $O(3)$ are often confused with the spherical harmonics, the relation between the irreps and the spherical harmonics is explained at *Spherical Harmonics*.

class e3nn.o3.**Irrep**(*l: Union[int, Irrep, str, tuple]*, *p=None*)

Bases: tuple

Irreducible representation of $O(3)$

This class does not contain any data, it is a structure that describe the representation. It is typically used as argument of other classes of the library to define the input and output representations of functions.

Parameters

- **l** (*int*) – non-negative integer, the degree of the representation, $l = 0, 1, \dots$
- **p** (*{1, -1}*) – the parity of the representation

Examples

Create a scalar representation ($l = 0$) of even parity.

```
>>> Irrep(0, 1)
0e
```

Create a pseudotensor representation ($l = 2$) of odd parity.

```
>>> Irrep(2, -1)
2o
```

Create a vector representation ($l = 1$) of the parity of the spherical harmonics (-1^l gives odd parity).

```
>>> Irrep("1y")
1o
```

```
>>> Irrep("2o").dim
5
```

```
>>> Irrep("2e") in Irrep("1o") * Irrep("1o")
True
```

```
>>> Irrep("1o") + Irrep("2o")
1x1o+1x2o
```

Methods:

D_from_angles(alpha, beta, gamma[, k])	Matrix $p^k D^l(\alpha, \beta, \gamma)$
D_from_axis_angle(axis, angle)	Matrix of the representation, see *Irrep.D_from_angles*
D_from_matrix(R)	Matrix of the representation, see *Irrep.D_from_angles*
D_from_quaternion(q[, k])	Matrix of the representation, see *Irrep.D_from_angles*
count(_value)	Return number of occurrences of value.
index(_value)	Return first index of value.
is_scalar()	Equivalent to `l == 0 and p == 1`
iterator([lmax])	Iterator through all the irreps of $O(3)$

Attributes:

dim	The dimension of the representation, $2l + 1$.
l	The degree of the representation, $l = 0, 1, \ldots$.
p	The parity of the representation, $p = \pm 1$.

D_from_angles(*alpha*, *beta*, *gamma*, *k=None*)

Matrix $p^k D^l(\alpha, \beta, \gamma)$

(matrix) Representation of $O(3)$. D is the representation of $SO(3)$, see *wigner_D*.

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$ Rotation α around Y axis, applied third.
- **beta** (torch.Tensor) – tensor of shape $(...)$ Rotation β around X axis, applied second.
- **gamma** (torch.Tensor) – tensor of shape $(...)$ Rotation γ around Y axis, applied first.
- **k** (torch.Tensor, optional) – tensor of shape $(...)$ How many times the parity is applied.

Returns
tensor of shape $(..., 2l + 1, 2l + 1)$

Return type
torch.Tensor

See also:

o3.wigner_D, *Irreps.D_from_angles*

D_from_axis_angle(*axis*, *angle*)

Matrix of the representation, see *Irrep.D_from_angles*

Parameters

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Returns
tensor of shape $(..., 2l + 1, 2l + 1)$

Return type
torch.Tensor

D_from_matrix(*R*)

Matrix of the representation, see *Irrep.D_from_angles*

Parameters

- **R** (torch.Tensor) – tensor of shape $(..., 3, 3)$
- **k** (torch.Tensor, optional) – tensor of shape $(...)$

Returns
tensor of shape $(..., 2l + 1, 2l + 1)$

Return type
torch.Tensor

Examples

```
>>> m = Irrep(1, -1).D_from_matrix(-torch.eye(3))
>>> m.long()
tensor([[-1,  0,  0],
        [ 0, -1,  0],
        [ 0,  0, -1]])
```

D_from_quaternion(*q*, *k=None*)

Matrix of the representation, see *Irrep.D_from_angles*

Parameters

- **q** (torch.Tensor) – tensor of shape $(..., 4)$
- **k** (torch.Tensor, optional) – tensor of shape $(...)$

Returns

tensor of shape $(..., 2l+1, 2l+1)$

Return type

torch.Tensor

count(*_value*)

Return number of occurrences of value.

property dim: int

The dimension of the representation, $2l+1$.

index(*_value*)

Return first index of value.

Raises ValueError if the value is not present.

is_scalar() → bool

Equivalent to `l == 0 and p == 1`

classmethod iterator(*lmax=None*)

Iterator through all the irreps of $O(3)$

Examples

```
>>> it = Irrep.iterator()
>>> next(it), next(it), next(it), next(it)
(0e, 0o, 1o, 1e)
```

property l: int

The degree of the representation, $l = 0, 1, \dots$.

property p: int

The parity of the representation, $p = \pm 1$.

class e3nn.o3.**Irreps**(*irreps=None*)

Bases: tuple

Direct sum of irreducible representations of $O(3)$

This class does not contain any data, it is a structure that describe the representation. It is typically used as argument of other classes of the library to define the input and output representations of functions.

dim

the total dimension of the representation

Type
int

num_irreps

number of irreps. the sum of the multiplicities

Type
int

ls

list of l values

Type
list of int

lmax

maximum l value

Type
int

Examples

Create a representation of 100 $l = 0$ of even parity and 50 pseudo-vectors.

```
>>> x = Irreps([(100, (0, 1)), (50, (1, 1))])
>>> x
100x0e+50x1e
```

```
>>> x.dim
250
```

Create a representation of 100 $l = 0$ of even parity and 50 pseudo-vectors.

```
>>> Irreps("100x0e + 50x1e")
100x0e+50x1e
```

```
>>> Irreps("100x0e + 50x1e + 0x2e")
100x0e+50x1e+0x2e
```

```
>>> Irreps("100x0e + 50x1e + 0x2e").lmax
1
```

```
>>> Irrep("2e") in Irreps("0e + 2e")
True
```

Empty Irreps

```
>>> Irreps(), Irreps("")
(, )
```

Methods:

D_from_angles(alpha, beta, gamma[, k])	Matrix of the representation
D_from_axis_angle(axis, angle)	Matrix of the representation
D_from_matrix(R)	Matrix of the representation
D_from_quaternion(q[, k])	Matrix of the representation
count(ir)	Multiplicity of `ir`.
index(_object)	Return first index of value.
randn(*size[, normalization, requires_grad, ...])	Random tensor.
remove_zero_multiplicities()	Remove any irreps with multiplicities of zero.
simplify()	Simplify the representations.
slices()	List of slices corresponding to indices for each irrep.
sort()	Sort the representations.
spherical_harmonics(lmax[, p])	representation of the spherical harmonics

D_from_angles(*alpha*, *beta*, *gamma*, *k=None*)

Matrix of the representation

Parameters

- **alpha** (torch.Tensor) – tensor of shape $(...)$
- **beta** (torch.Tensor) – tensor of shape $(...)$
- **gamma** (torch.Tensor) – tensor of shape $(...)$
- **k** (torch.Tensor, optional) – tensor of shape $(...)$

Returns

tensor of shape $(..., \dim, \dim)$

Return type

torch.Tensor

D_from_axis_angle(*axis*, *angle*)

Matrix of the representation

Parameters

- **axis** (torch.Tensor) – tensor of shape $(..., 3)$
- **angle** (torch.Tensor) – tensor of shape $(...)$

Returns

tensor of shape $(..., \dim, \dim)$

Return type

torch.Tensor

D_from_matrix(*R*)

Matrix of the representation

Parameters

R (torch.Tensor) – tensor of shape $(..., 3, 3)$

Returns

tensor of shape $(..., \dim, \dim)$

Return type

torch.Tensor

D_from_quaternion(*q*, *k=None*)

Matrix of the representation

Parameters

- **q** (torch.Tensor) – tensor of shape $(..., 4)$
- **k** (torch.Tensor, optional) – tensor of shape $(...)$

Returns

tensor of shape $(..., \dim, \dim)$

Return type

torch.Tensor

count(*ir*) → int

Multiplicity of `ir`.

Parameters

ir (*e3nn.o3.Irrep*) –

Returns

total multiplicity of `ir`

Return type

int

index(*_object*)

Return first index of value.

Raises ValueError if the value is not present.

randn(**size*, *normalization='component'*, *requires_grad=False*, *dtype=None*, *device=None*)

Random tensor.

Parameters

- ***size** (*list of int*) – size of the output tensor, needs to contains a `-1`
- **normalization** (*{'component', 'norm'}*) –

Returns

tensor of shape `size` where `-1` is replaced by `self.dim`

Return type

torch.Tensor

Examples

```
>>> Irreps("5x0e + 10x1o").randn(5, -1, 5, normalization='norm').shape
torch.Size([5, 35, 5])
```

```
>>> random_tensor = Irreps("2o").randn(2, -1, 3, normalization='norm')
>>> random_tensor.norm(dim=1).sub(1).abs().max().item() < 1e-5
True
```

remove_zero_multiplicities()

Remove any irreps with multiplicities of zero.

Return type

e3nn.o3.Irreps

Examples

```
>>> Irreps("4x0e + 0x1o + 2x3e").remove_zero_multiplicities()
4x0e+2x3e
```

simplify()

Simplify the representations.

Return type
e3nn.o3.Irreps

Examples

Note that simplify does not sort the representations.

```
>>> Irreps("1e + 1e + 0e").simplify()
2x1e+1x0e
```

Equivalent representations which are separated from each other are not combined.

```
>>> Irreps("1e + 1e + 0e + 1e").simplify()
2x1e+1x0e+1x1e
```

slices()

List of slices corresponding to indices for each irrep.

Examples

```
>>> Irreps('2x0e + 1e').slices()
[slice(0, 2, None), slice(2, 5, None)]
```

sort()

Sort the representations.

Returns

- **irreps** (*e3nn.o3.Irreps*)
- **p** (*tuple of int*)
- **inv** (*tuple of int*)

Examples

```
>>> Irreps("1e + 0e + 1e").sort().irreps
1x0e+1x1e+1x1e
```

```
>>> Irreps("2o + 1e + 0e + 1e").sort().p
(3, 1, 0, 2)
```

```
>>> Irreps("2o + 1e + 0e + 1e").sort().inv
(2, 1, 3, 0)
```

static spherical_harmonics(*lmax*, *p=-1*)

representation of the spherical harmonics

Parameters

- **lmax** (*int*) – maximum l
- **p** (*{1, -1}*) – the parity of the representation

Returns

representation of $(Y^0, Y^1, \dots, Y^{\text{lmax}})$

Return type

e3nn.o3.Irreps

Examples

```
>>> Irreps.spherical_harmonics(3)
1x0e+1x1o+1x2e+1x3o
```

```
>>> Irreps.spherical_harmonics(4, p=1)
1x0e+1x1e+1x2e+1x3e+1x4e
```

Tensor Product

All tensor products — denoted $\otimes$ — share two key characteristics:

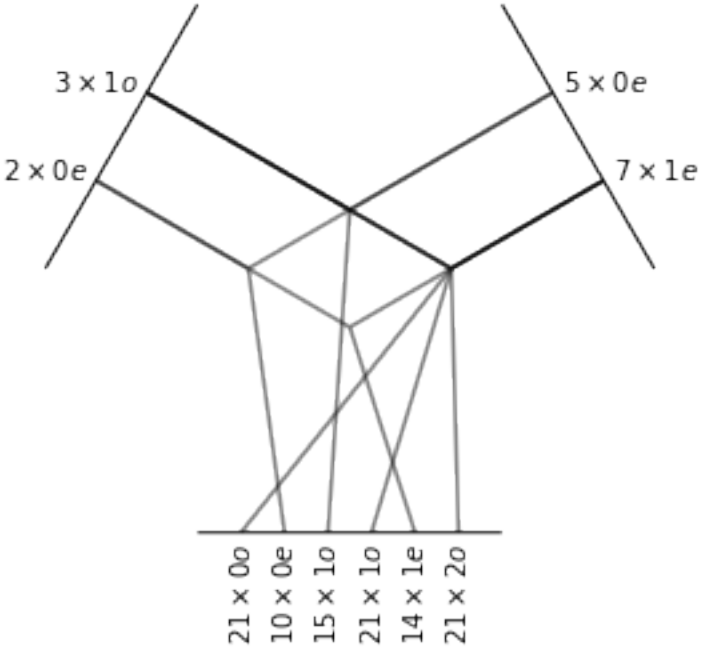
1. The tensor product is *bilinear*: $(\alpha x_1 + x_2) \otimes y = \alpha x_1 \otimes y + x_2 \otimes y$ and $x \otimes (\alpha y_1 + y_2) = \alpha x \otimes y_1 + x \otimes y_2$
2. The tensor product is *equivariant*: $(Dx) \otimes (Dy) = D(x \otimes y)$ where D is the representation of some symmetry operation from $E(3)$ (sorry for the very loose notation)

The class *e3nn.o3.TensorProduct* implements all possible tensor products between finite direct sums of irreducible representations (*e3nn.o3.Irreps*). While *e3nn.o3.TensorProduct* provides maximum flexibility, a number of sublcasses provide various typical special cases of the tensor product:

- *e3nn.o3.FullTensorProduct*:

```
tp = o3.FullTensorProduct(
    irreps_in1='2x0e + 3x1o',
    irreps_in2='5x0e + 7x1e'
)
print(tp)
tp.visualize();
```

```
FullTensorProduct(2x0e+3x1o x 5x0e+7x1e -> 21x0o+10x0e+36x1o+14x1e+21x2o | 102 paths | 0␣
↪weights)
```

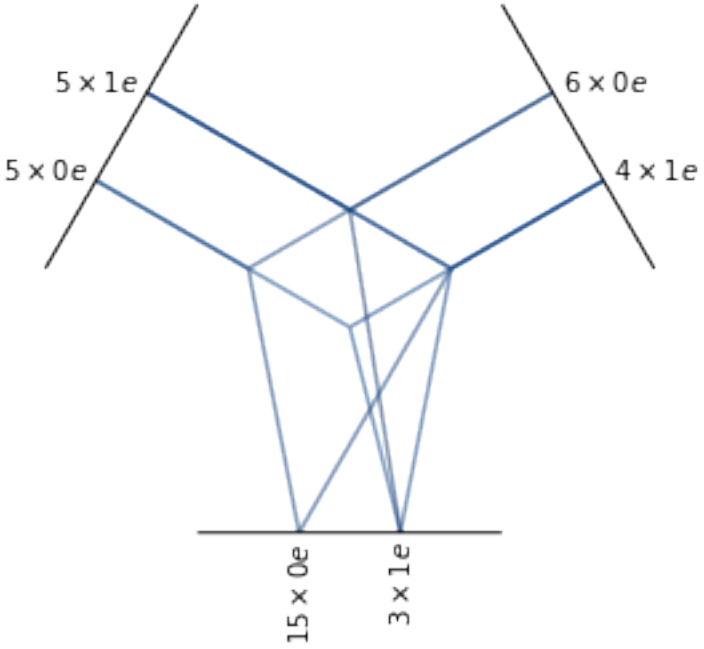


The full tensor product is the "natural" one. Every possible output — each output irrep for every pair of input irreps — is created and returned independently. The outputs are not mixed with each other. Note how the multiplicities of the outputs are the product of the multiplicities of the respective inputs.

- *e3nn.o3.FullyConnectedTensorProduct*

```
tp = o3.FullyConnectedTensorProduct(
    irreps_in1='5x0e + 5x1e',
    irreps_in2='6x0e + 4x1e',
    irreps_out='15x0e + 3x1e'
)
print(tp)
tp.visualize();
```

```
FullyConnectedTensorProduct(5x0e+5x1e x 6x0e+4x1e -> 15x0e+3x1e | 960 paths | 960␣
↪weights)
```



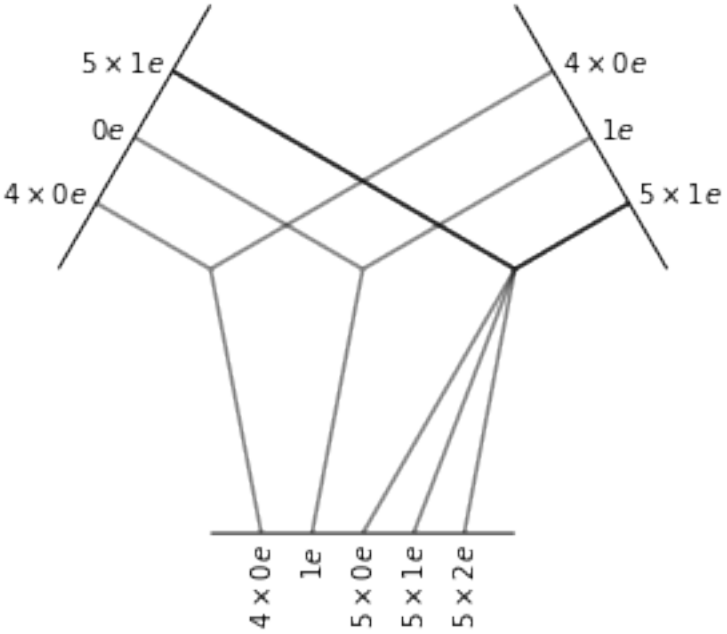
In a fully connected tensor product, all paths that lead to any of the irreps specified in `irreps_out` are created. Unlike *e3nn.o3.FullTensorProduct*, each output is a learned weighted sum of compatible paths. This allows *e3nn.o3.FullyConnectedTensorProduct* to produce outputs with any multiplicity; note that the example above has $5 \times 6 + 5 \times 4 = 50$ ways of creating scalars (`0e`), but the specified `irreps_out` has only 15 scalars, each of which is a learned weighted combination of those 50 possible scalars. The blue color in the visualization indicates that the path has these learnable weights.

All possible output irreps do **not** need to be included in `irreps_out` of a *e3nn.o3.FullyConnectedTensorProduct*: `o3.FullyConnectedTensorProduct(irreps_in1='5x1o', irreps_in2='3x1o', irreps_out='20x0e')` will only compute inner products between its inputs, since `1e`, the output irrep of a vector cross product, is not present in `irreps_out`. Note also in this example that there are 20 output scalars, even though the given inputs can produce only 15 unique scalars — this is again allowed because each output is a learned linear combination of those 15 scalars, placing no restrictions on how many or how few outputs can be requested.

- *e3nn.o3.ElementwiseTensorProduct*

```
tp = o3.ElementwiseTensorProduct(
    irreps_in1='5x0e + 5x1e',
    irreps_in2='4x0e + 6x1e'
)
print(tp)
tp.visualize();
```

```
ElementwiseTensorProduct(5x0e+5x1e x 4x0e+6x1e -> 4x0e+1x1e+5x0e+5x1e+5x2e | 20 paths |␣
↪0 weights)
```

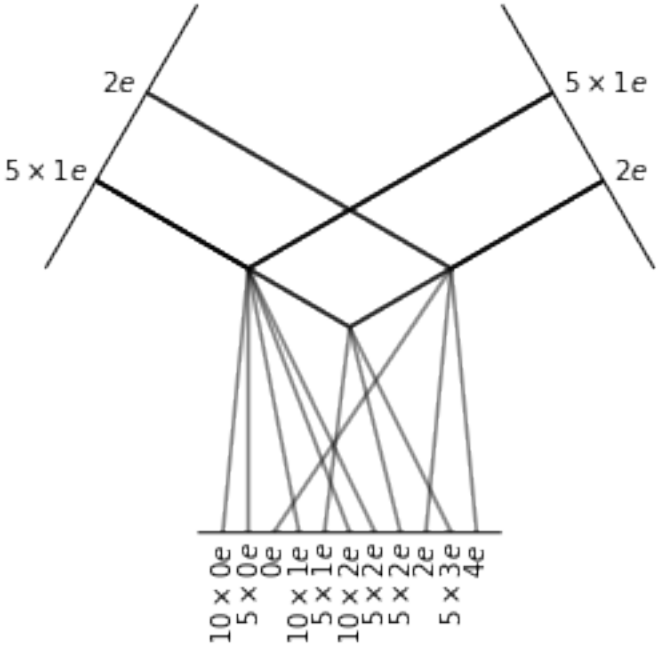


In the elementwise tensor product, the irreps are multiplied one-by-one. Note in the visualization how the inputs have been split and that the multiplicities of the outputs match with the multiplicities of the input.

- *e3nn.o3.TensorSquare*

```
tp = o3.TensorSquare("5x1e + 2e")
print(tp)
tp.visualize();
```

```
TensorSquare(5x1e+1x2e -> 16x0e+15x1e+21x2e+5x3e+1x4e | 58 paths | 0 weights)
```



The tensor square operation only computes the non-zero entries of a tensor times itself. It also applies different normalization rules taking into account that a tensor time itself is statistically different from the product of two independent tensors.

class e3nn.o3.**TensorProduct**(*irreps_in1:* Irreps, *irreps_in2:* Irreps, *irreps_out:* Irreps, *instructions: List[tuple]*, *in1_var: Optional[Union[List[float], Tensor]] = None*, *in2_var: Optional[Union[List[float], Tensor]] = None*, *out_var: Optional[Union[List[float], Tensor]] = None*, *irrep_normalization: Optional[str] = None*, *path_normalization: Optional[str] = None*, *internal_weights: Optional[bool] = None*, *shared_weights: Optional[bool] = None*, *compile_left_right: bool = True*, *compile_right: bool = False*, *normalization=None*, *_specialized_code: Optional[bool] = None*, *_optimize_einsums: Optional[bool] = None*)

Bases: `CodeGenMixin`, `Module`

Tensor product with parametrized paths.

Parameters

- **irreps_in1** (*e3nn.o3.Irreps*) – Irreps for the first input.
- **irreps_in2** (*e3nn.o3.Irreps*) – Irreps for the second input.
- **irreps_out** (*e3nn.o3.Irreps*) – Irreps for the output.
- **instructions** (*list of tuple*) – List of instructions `(i_1, i_2, i_out, mode, train[, path_weight])`.

  Each instruction puts `in1[i_1]` $\otimes$ `in2[i_2]` into `out[i_out]`.

  – `mode`: `str`. Determines the way the multiplicities are treated, `"uvw"` is fully connected. Other valid

  options are: `'uvw'`, `'uvu'`, `'uvv'`, `'uuw'`, `'uuu'`, and `'uvuv'`. * `train`: `bool`. `True` if this path should have learnable weights, otherwise `False`. * `path_weight`: `float`. A fixed

multiplicative weight to apply to the output of this path. Defaults to 1. Note that setting `path_weight` breaks the normalization derived from `in1_var/in2_var/out_var`.

- **in1_var** (*list of float, Tensor, or None*) – Variance for each irrep in `irreps_in1`. If `None`, all default to `1.0`.
- **in2_var** (*list of float, Tensor, or None*) – Variance for each irrep in `irreps_in2`. If `None`, all default to `1.0`.
- **out_var** (*list of float, Tensor, or None*) – Variance for each irrep in `irreps_out`. If `None`, all default to `1.0`.
- **irrep_normalization** (*{'component', 'norm'}*) – The assumed normalization of the input and output representations. If it is set to "norm":

$$\|x\| = \|y\| = 1 \implies \|x \otimes y\| = 1$$

- **path_normalization** (*{'element', 'path'}*) – If set to `element`, each output is normalized by the total number of elements (independently of their paths). If it is set to `path`, each path is normalized by the total number of elements in the path, then each output is normalized by the number of paths.
- **internal_weights** (*bool*) – whether the *e3nn.o3.TensorProduct* contains its learnable weights as a parameter
- **shared_weights** (*bool*) – whether the learnable weights are shared among the input's extra dimensions
  - `True` $z_i = w x_i \otimes y_i$
  - `False` $z_i = w_i x_i \otimes y_i$

  where here i denotes a *batch-like* index. `shared_weights` cannot be `False` if `internal_weights` is `True`.
- **compile_left_right** (*bool*) – whether to compile the forward function, true by default
- **compile_right** (*bool*) – whether to compile the `.right` function, false by default

Examples

Create a module that computes elementwise the cross-product of 16 vectors with 16 vectors $z_u = x_u \wedge y_u$

```
>>> module = TensorProduct(
...     "16x1o", "16x1o", "16x1e",
...     [
...         (0, 0, 0, "uuu", False)
...     ]
... )
```

Now mix all 16 vectors with all 16 vectors to makes 16 pseudo-vectors $z_w = \sum_{u,v} w_{uvw} x_u \wedge y_v$

```
>>> module = TensorProduct(
...     [(16, (1, -1))],
...     [(16, (1, -1))],
...     [(16, (1,  1))],
...     [
...         (0, 0, 0, "uvw", True)
```

(continues on next page)

(continued from previous page)

```
...     ]
... )
```

With custom input variance and custom path weights:

```
>>> module = TensorProduct(
...     "8x0o + 8x1o",
...     "16x1o",
...     "16x1e",
...     [
...         (0, 0, 0, "uvw", True, 3),
...         (1, 0, 0, "uvw", True, 1),
...     ],
...     in2_var=[1/16]
... )
```

Example of a dot product:

```
>>> irreps = o3.Irreps("3x0e + 4x0o + 1e + 2o + 3o")
>>> module = TensorProduct(irreps, irreps, "0e", [
...     (i, i, 0, 'uuw', False)
...     for i, (mul, ir) in enumerate(irreps)
... ])
```

Implement $z_u = x_u \otimes (\sum_v w_{uv} y_v)$

```
>>> module = TensorProduct(
...     "8x0o + 7x1o + 3x2e",
...     "10x0e + 10x1e + 10x2e",
...     "8x0o + 7x1o + 3x2e",
...     [
...         # paths for the l=0:
...         (0, 0, 0, "uvu", True),  # 0x0->0
...         # paths for the l=1:
...         (1, 0, 1, "uvu", True),  # 1x0->1
...         (1, 1, 1, "uvu", True),  # 1x1->1
...         (1, 2, 1, "uvu", True),  # 1x2->1
...         # paths for the l=2:
...         (2, 0, 2, "uvu", True),  # 2x0->2
...         (2, 1, 2, "uvu", True),  # 2x1->2
...         (2, 2, 2, "uvu", True),  # 2x2->2
...     ]
... )
```

Tensor Product using the xavier uniform initialization:

```
>>> irreps_1 = o3.Irreps("5x0e + 10x1o + 1x2e")
>>> irreps_2 = o3.Irreps("5x0e + 10x1o + 1x2e")
>>> irreps_out = o3.Irreps("5x0e + 10x1o + 1x2e")
>>> # create a Fully Connected Tensor Product
>>> module = o3.TensorProduct(
...     irreps_1,
...     irreps_2,
```

(continues on next page)

(continued from previous page)

```
...     irreps_out,
...     [
...         (i_1, i_2, i_out, "uvw", True, mul_1 * mul_2)
...         for i_1, (mul_1, ir_1) in enumerate(irreps_1)
...         for i_2, (mul_2, ir_2) in enumerate(irreps_2)
...         for i_out, (mul_out, ir_out) in enumerate(irreps_out)
...         if ir_out in ir_1 * ir_2
...     ]
... )
>>> with torch.no_grad():
...     for weight in module.weight_views():
...         mul_1, mul_2, mul_out = weight.shape
...         # formula from torch.nn.init.xavier_uniform_
...         a = (6 / (mul_1 * mul_2 + mul_out))**0.5
...         new_weight = torch.empty_like(weight)
...         new_weight.uniform_(-a, a)
...         weight[:] = new_weight
tensor(...)
>>> n = 1_000
>>> vars = module(irreps_1.randn(n, -1), irreps_2.randn(n, -1)).var(0)
>>> assert vars.min() > 1 / 3
>>> assert vars.max() < 3
```

Methods:

forward(x, y[, weight])	Evaluate $wx \otimes y$.
right(y[, weight])	Partially evaluate $wx \otimes y$.
visualize([weight, plot_weight, ...])	Visualize the connectivity of this *e3nn.o3.TensorProduct*
weight_view_for_instruction(instruction[, ...])	View of weights corresponding to `instruction`.
weight_views([weight, yield_instruction])	Iterator over weight views for each weighted instruction.

forward(*x*, *y*, *weight: Optional[Tensor] = None*)

Evaluate $wx \otimes y$.

Parameters

- **x** (torch.Tensor) – tensor of shape `(..., irreps_in1.dim)`
- **y** (torch.Tensor) – tensor of shape `(..., irreps_in2.dim)`
- **weight** (torch.Tensor or list of torch.Tensor, optional) – required if `internal_weights` is `False` tensor of shape `(self.weight_numel,)` if `shared_weights` is `True` tensor of shape `(..., self.weight_numel)` if `shared_weights` is `False` or list of tensors of shapes `weight_shape` / `(...)` + `weight_shape`. Use `self.instructions` to know what are the weights used for.

Returns

tensor of shape `(..., irreps_out.dim)`

Return type

torch.Tensor

right(*y*, *weight: Optional[Tensor] = None*)

Partially evaluate $wx \otimes y$.

It returns an operator in the form of a tensor that can act on an arbitrary x.

For example, if the tensor product above is expressed as

$$w_{ijk} x_i y_j \to z_k$$

then the right method returns a tensor b_{ik} such that

$$w_{ijk} y_j \to b_{ik} x_i b_{ik} \to z_k$$

The result of this method can be applied with a tensor contraction:

```
torch.einsum("...ik,...i->...k", right, input)
```

Parameters

- **y** (torch.Tensor) – tensor of shape `(..., irreps_in2.dim)`
- **weight** (torch.Tensor or list of torch.Tensor, optional) – required if `internal_weights` is `False` tensor of shape `(self.weight_numel,)` if `shared_weights` is `True` tensor of shape `(..., self.weight_numel)` if `shared_weights` is `False` or list of tensors of shapes `weight_shape` / `(...)` + `weight_shape`. Use `self.instructions` to know what are the weights used for.

Returns
tensor of shape `(..., irreps_in1.dim, irreps_out.dim)`

Return type
torch.Tensor

visualize(*weight: Optional[Tensor] = None*, *plot_weight: bool = True*, *aspect_ratio=1*, *ax=None*)

Visualize the connectivity of this *e3nn.o3.TensorProduct*

Parameters

- **weight** (torch.Tensor, optional) – like `weight` argument to `forward()`
- **plot_weight** (bool, default True) – Whether to color paths by the sum of their weights.
- **ax** (`matplotlib.Axes`, default None) – The axes to plot on. If `None`, a new figure will be created.

Returns
The figure and axes on which the plot was drawn.

Return type
(fig, ax)

weight_view_for_instruction(*instruction: int*, *weight: Optional[Tensor] = None*) → Tensor

View of weights corresponding to `instruction`.

Parameters

- **instruction** (*int*) – The index of the instruction to get a view on the weights for. `self.instructions[instruction].has_weight` must be `True`.
- **weight** (torch.Tensor, optional) – like `weight` argument to `forward()`

Returns
A view on `weight` or this object's internal weights for the weights corresponding to the `instruction` th instruction.

Return type
`torch.Tensor`

weight_views(*weight: Optional[Tensor] = None*, *yield_instruction: bool = False*)

Iterator over weight views for each weighted instruction.

Parameters

- **weight** (`torch.Tensor`, optional) – like `weight` argument to `forward()`
- **yield_instruction** (`bool`, default False) – Whether to also yield the corresponding instruction.

Yields

- If `yield_instruction` is `True`, yields `(instruction_index, instruction, weight_view)`.
- Otherwise, yields `weight_view`.

class e3nn.o3.**FullyConnectedTensorProduct**(*irreps_in1*, *irreps_in2*, *irreps_out*, *irrep_normalization: Optional[str] = None*, *path_normalization: Optional[str] = None*, ***kwargs*)

Bases: *TensorProduct*

Fully-connected weighted tensor product

All the possible path allowed by $|l_1 - l_2| \leq l_{out} \leq l_1 + l_2$ are made. The output is a sum on different paths:

$$z_w = \sum_{u,v} w_{uvw} x_u \otimes y_v + \cdots \text{other paths}$$

where u, v, w are the indices of the multiplicities.

Parameters

- **irreps_in1** (*e3nn.o3.Irreps*) – representation of the first input
- **irreps_in2** (*e3nn.o3.Irreps*) – representation of the second input
- **irreps_out** (*e3nn.o3.Irreps*) – representation of the output
- **irrep_normalization** (*{'component', 'norm'}*) – see *e3nn.o3.TensorProduct*
- **path_normalization** (*{'element', 'path'}*) – see *e3nn.o3.TensorProduct*
- **internal_weights** (*bool*) – see *e3nn.o3.TensorProduct*
- **shared_weights** (*bool*) – see *e3nn.o3.TensorProduct*

class e3nn.o3.**FullTensorProduct**(*irreps_in1: Irreps*, *irreps_in2: Irreps*, *filter_ir_out: Optional[Iterator[Irrep]] = None*, *irrep_normalization: Optional[str] = None*, ***kwargs*)

Bases: *TensorProduct*

Full tensor product between two irreps.

$$z_{uv} = x_u \otimes y_v$$

where u and v run over the irreps. Note that there are no weights. The output representation is determined by the two input representations.

Parameters

- **irreps_in1** (*e3nn.o3.Irreps*) – representation of the first input
- **irreps_in2** (*e3nn.o3.Irreps*) – representation of the second input
- **filter_ir_out** (iterator of *e3nn.o3.Irrep*, optional) – filter to select only specific *e3nn.o3.Irrep* of the output
- **irrep_normalization** (*{'component', 'norm'}*) – see *e3nn.o3.TensorProduct*

class e3nn.o3.**ElementwiseTensorProduct**(*irreps_in1*, *irreps_in2*, *filter_ir_out=None*, *irrep_normalization: Optional[str] = None*, ***kwargs*)

Bases: *TensorProduct*

Elementwise connected tensor product.

$$z_u = x_u \otimes y_u$$

where u runs over the irreps. Note that there are no weights. The output representation is determined by the two input representations.

Parameters

- **irreps_in1** (*e3nn.o3.Irreps*) – representation of the first input
- **irreps_in2** (*e3nn.o3.Irreps*) – representation of the second input
- **filter_ir_out** (iterator of *e3nn.o3.Irrep*, optional) – filter to select only specific *e3nn.o3.Irrep* of the output
- **irrep_normalization** (*{'component', 'norm'}*) – see *e3nn.o3.TensorProduct*

Examples

Elementwise scalar product

```
>>> ElementwiseTensorProduct("5x1o + 5x1e", "10x1e", ["0e", "0o"])
ElementwiseTensorProduct(5x1o+5x1e x 10x1e -> 5x0o+5x0e | 10 paths | 0 weights)
```

class e3nn.o3.**TensorSquare**(*irreps_in: Irreps*, *irreps_out: Optional[Irreps] = None*, *filter_ir_out: Optional[Iterator[Irrep]] = None*, *irrep_normalization: Optional[str] = None*, ***kwargs*)

Bases: *TensorProduct*

Compute the square tensor product of a tensor and reduce it in irreps

If *irreps_out* is given, this operation is fully connected. If *irreps_out* is not given, the operation has no parameter and is like full tensor product.

Parameters

- **irreps_in** (*e3nn.o3.Irreps*) – representation of the input
- **irreps_out** (*e3nn.o3.Irreps*, optional) – representation of the output
- **filter_ir_out** (iterator of *e3nn.o3.Irrep*, optional) – filter to select only specific *e3nn.o3.Irrep* of the output
- **irrep_normalization** (*{'component', 'norm'}*) – see *e3nn.o3.TensorProduct*

Methods:

forward(x[, weight])	Evaluate $wx \otimes y$.

forward(*x*, *weight: Optional[Tensor] = None*)

Evaluate $wx \otimes y$.

Parameters

- **x** (torch.Tensor) – tensor of shape `(..., irreps_in1.dim)`
- **y** (torch.Tensor) – tensor of shape `(..., irreps_in2.dim)`
- **weight** (torch.Tensor or list of torch.Tensor, optional) – required if `internal_weights` is `False` tensor of shape `(self.weight_numel,)` if `shared_weights` is `True` tensor of shape `(..., self.weight_numel)` if `shared_weights` is `False` or list of tensors of shapes `weight_shape` / `(...)` + `weight_shape`. Use `self.instructions` to know what are the weights used for.

Returns
tensor of shape `(..., irreps_out.dim)`

Return type
torch.Tensor

Spherical Harmonics

The spherical harmonics $Y^l(x)$ are functions defined on the sphere S^2. They form a basis of the space on function on the sphere:

$$\mathcal{F} = \{S^2 \longrightarrow \mathbb{R}\}$$

On this space it is nautal how the group $O(3)$ acts, Given p_a, p_v two scalar representations:

$$[L(g)f](x) = p_v(g)f(p_a(g)R(g)^{-1}x), \quad \forall f \in \mathcal{F}, x \in S^2$$

L is representation of $O(3)$. But L is not irreducible. It can be decomposed via a change of basis into a sum of irreps,

$$Y^T L(g) Y = 0 \oplus 1 \oplus 2 \oplus 3 \oplus \dots$$

where the change of basis are the spherical harmonics!

As a consequence, the spherical harmonics are equivariant,

$$Y^l(R(g)x) = D^l(g)Y^l(x)$$

```
r = s2_grid()
```

`r` is a grid on the sphere.

Each point on the sphere has 3 components. If we plot the value of each of the 3 component separately we obtain the following figure:

```
plot(r, radial_abs=False)
```

x, y and z are represented as 3 scalar fields on 3 different spheres. To obtain a nicer figure (that looks like the spherical harmonics shown on Wikipedia) we can deform the spheres into a shape that has its radius equal to the absolute value of the plotted quantity:

```
plot(r)
```

Y^1 is the identity function. Now let's compute Y^2, for this we take the tensor product $r \otimes r$ and extract the $L = 2$ part of it.

```
tp = o3.ElementwiseTensorProduct("1o", "1o", ['2e'], irrep_normalization='norm')
y2 = tp(r, r)
plot(y2)
```

Similarly, the next spherical harmonic function Y^3 is the $L = 3$ part of $r \otimes r \otimes r$:

```
tp = o3.ElementwiseTensorProduct("2e", "1o", ['3o'], irrep_normalization='norm')
y3 = tp(y2, r)
plot(y3)
```

The functions below are more efficient versions not using *e3nn.o3.ElementwiseTensorProduct*:

Details

e3nn.o3.**spherical_harmonics**(*l: Union[int, List[int], str,* Irreps*], x: Tensor, normalize: bool, normalization: str = 'integral'*)

Spherical harmonics

Polynomials defined on the 3d space $Y^l : \mathbb{R}^3 \longrightarrow \mathbb{R}^{2l+1}$
Usually restricted on the sphere (with `normalize=True`) $Y^l : S^2 \longrightarrow \mathbb{R}^{2l+1}$
who satisfies the following properties:

- are polynomials of the cartesian coordinates `x, y, z`
- is equivariant $Y^l(Rx) = D^l(R)Y^l(x)$
- are orthogonal $\int_{S^2} Y^l_m(x)Y^j_n(x)dx = \text{cste}\ \delta_{lj}\delta_{mn}$

The value of the constant depends on the choice of normalization.

It obeys the following property:

$$\begin{aligned} Y^{l+1}_i(x) &= \text{cste}(l) & C_{ijk}Y^l_j(x)x_k \\ \partial_k Y^{l+1}_i(x) &= \text{cste}(l)\ (l+1) & C_{ijk}Y^l_j(x) \end{aligned}$$

Where C are the *wigner_3j*.

Note: This function match with this table of standard real spherical harmonics from Wikipedia when `normalize=True`, `normalization='integral'` and is called with the argument in the order `y,z,x` (instead of `x,y,z`).

Parameters

- **l** (*int or list of int*) – degree of the spherical harmonics.
- **x** (torch.Tensor) – tensor x of shape `(..., 3)`.
- **normalize** (*bool*) – whether to normalize the `x` to unit vectors that lie on the sphere before projecting onto the spherical harmonics
- **normalization** (*{'integral', 'component', 'norm'}*) – normalization of the output tensors — note that this option is independent of `normalize`, which controls the processing of the *input*, rather than the output. Valid options: * *component*: $\|Y^l(x)\|^2 = 2l+1, x \in S^2$ * *norm*: $\|Y^l(x)\| = 1, x \in S^2$, `component / sqrt(2l+1)` * *integral*: $\int_{S^2} Y^l_m(x)^2 dx = 1$, `component / sqrt(4pi)`

Returns

a tensor of shape `(..., 2l+1)`

$$Y^l(x)$$

Return type
torch.Tensor

Examples

```
>>> spherical_harmonics(0, torch.randn(2, 3), False, normalization='component')
tensor([[1.],
        [1.]])
```

See also:

wigner_D, *wigner_3j*

e3nn.o3.**spherical_harmonics_alpha_beta**(*l*, *alpha*, *beta*, ***, *normalization='integral'*)

Spherical harmonics of $\vec{r} = R_y(\alpha)R_x(\beta)e_y$

$$Y^l(\alpha, \beta) = S^l(\alpha)P^l(\cos(\beta))$$

where P^l are the *Legendre* polynomials

Parameters

- **l** (*int or list of int*) – degree of the spherical harmonics.
- **alpha** (torch.Tensor) – tensor of shape `(...)`.
- **beta** (torch.Tensor) – tensor of shape `(...)`.

Returns
a tensor of shape `(..., 2l+1)`

Return type
torch.Tensor

e3nn.o3.**Legendre**(**args*, ***kwargs*)

GraphModule is an nn.Module generated from an fx.Graph. Graphmodule has a `graph` attribute, as well as `code` and `forward` attributes generated from that `graph`.

Warning: When `graph` is reassigned, `code` and `forward` will be automatically regenerated. However, if you edit the contents of the `graph` without reassigning the `graph` attribute itself, you must call `recompile()` to update the generated code.

Note: Backwards-compatibility for this API is guaranteed.

Reduction of Tensors in Irreps

class e3nn.o3.**ReducedTensorProducts**(*formula*, *filter_ir_out=None*, *filter_ir_mid=None*, *eps=1e-09*, ***irreps*)

Bases: `CodeGenMixin`, `Module`

reduce a tensor with symmetries into irreducible representations

Parameters

- **formula** (*str*) – String made of letters - and = that represent the indices symmetries of the tensor. For instance `ij=ji` means that the tensor has to indices and if they are exchanged, its value is the same. `ij=-ji` means that the tensor change its sign if the two indices are exchanged.
- **filter_ir_out** (list of *e3nn.o3.Irrep*, optional) – Optional, list of allowed irrep in the output
- **filter_ir_mid** (list of *e3nn.o3.Irrep*, optional) – Optional, list of allowed irrep in the intermediary operations
- ****kwargs** (dict of *e3nn.o3.Irreps*) – each letter present in the formula has to be present in the `irreps` dictionary, unless it can be inferred by the formula. For instance if the formula is `ij=ji` you can provide the representation of `i` only: `ReducedTensorProducts('ij=ji', i='1o')`.

irreps_in

input representations

Type
list of *e3nn.o3.Irreps*

irreps_out

output representation

Type
e3nn.o3.Irreps

change_of_basis

tensor of shape `(irreps_out.dim, irreps_in[0].dim, ..., irreps_in[-1].dim)`

Type
torch.Tensor

Examples

```
>>> tp = ReducedTensorProducts('ij=-ji', i='1o')
>>> x = torch.tensor([1.0, 0.0, 0.0])
>>> y = torch.tensor([0.0, 1.0, 0.0])
>>> tp(x, y) + tp(y, x)
tensor([0., 0., 0.])
```

```
>>> tp = ReducedTensorProducts('ijkl=jikl=ikjl=ijlk', i="1e")
>>> tp.irreps_out
1x0e+1x2e+1x4e
```

```
>>> tp = ReducedTensorProducts('ij=ji', i='1o')
>>> x, y = torch.randn(2, 3)
>>> a = torch.einsum('zij,i,j->z', tp.change_of_basis, x, y)
>>> b = tp(x, y)
>>> assert torch.allclose(a, b, atol=1e-3, rtol=1e-3)
```

Methods:

forward(*xs)	Defines the computation performed at every call.

forward(**xs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Grid Signal on the Sphere

e3nn.o3.**s2_grid**(*res_beta*, *res_alpha*, *dtype=None*, *device=None*)

grid on the sphere

Parameters

- **res_beta** (*int*) – N
- **res_alpha** (*int*) – M
- **dtype** (*torch.dtype or None*) – `dtype` of the returned tensors. If `None` then set to `torch.get_default_dtype()`.
- **device** (*torch.device or None*) – `device` of the returned tensors. If `None` then set to the default device of the current context.

Returns

- **betas** (`torch.Tensor`) – tensor of shape `(res_beta)`
- **alphas** (`torch.Tensor`) – tensor of shape `(res_alpha)`

e3nn.o3.**spherical_harmonics_s2_grid**(*lmax*, *res_beta*, *res_alpha*, *dtype=None*, *device=None*)

spherical harmonics evaluated on the grid on the sphere

$$f(x) = \sum_{l=0}^{l_{max}} F^l \cdot Y^l(x)$$

$$f(\beta, \alpha) = \sum_{l=0}^{l_{max}} F^l \cdot S^l(\alpha) P^l(\cos(\beta))$$

Parameters

- **lmax** (*int*) – l_{max}
- **res_beta** (*int*) – N
- **res_alpha** (*int*) – M

Returns

- **betas** (torch.Tensor) – tensor of shape `(res_beta)`
- **alphas** (torch.Tensor) – tensor of shape `(res_alpha)`
- **shb** (torch.Tensor) – tensor of shape `(res_beta, (lmax + 1)**2)`
- **sha** (torch.Tensor) – tensor of shape `(res_alpha, 2 lmax + 1)`

e3nn.o3.**rfft**(*x*, *l*)

Real fourier transform

Parameters

- **x** (torch.Tensor) – tensor of shape `(..., 2 l + 1)`
- **res** (*int*) – output resolution, has to be an odd number

Returns

tensor of shape `(..., res)`

Return type

torch.Tensor

Examples

```
>>> lmax = 8
>>> res = 101
>>> _betas, _alphas, _shb, sha = spherical_harmonics_s2_grid(lmax, res, res)
>>> x = torch.randn(res)
>>> (rfft(x, lmax) - x @ sha).abs().max().item() < 1e-4
True
```

e3nn.o3.**irfft**(*x*, *res*)

Inverse of the real fourier transform

Parameters

- **x** (torch.Tensor) – tensor of shape `(..., 2 l + 1)`
- **res** (*int*) – output resolution, has to be an odd number

Returns
positions on the sphere, tensor of shape `(..., res, 3)`

Return type
`torch.Tensor`

Examples

```
>>> lmax = 8
>>> res = 101
>>> _betas, _alphas, _shb, sha = spherical_harmonics_s2_grid(lmax, res, res)
>>> x = torch.randn(2 * lmax + 1)
>>> (irfft(x, res) - sha @ x).abs().max().item() < 1e-4
True
```

class e3nn.o3.**ToS2Grid**(*lmax=None*, *res=None*, *normalization='component'*, *dtype=None*, *device=None*)

Bases: `Module`

Transform spherical tensor into signal on the sphere

The inverse transformation of *FromS2Grid*

Parameters

- **lmax** (*int*) –
- **res** (*int, tuple of int*) – resolution in `beta` and in `alpha`
- **normalization** (*{'norm', 'component', 'integral'}*) –
- **dtype** (*torch.dtype or None, optional*) –
- **device** (*torch.device or None, optional*) –

Examples

```
>>> m = ToS2Grid(6, (100, 101))
>>> x = torch.randn(3, 49)
>>> m(x).shape
torch.Size([3, 100, 101])
```

ToS2Grid and *FromS2Grid* are inverse of each other

```
>>> m = ToS2Grid(6, (100, 101))
>>> k = FromS2Grid((100, 101), 6)
>>> x = torch.randn(3, 49)
>>> y = k(m(x))
>>> (x - y).abs().max().item() < 1e-4
True
```

grid
positions on the sphere, tensor of shape `(res_beta, res_alpha, 3)`

Type
`torch.Tensor`

Methods:

forward(x)	Evaluate

forward(*x*)

Evaluate

Parameters
x (torch.Tensor) – tensor of shape `(..., (l+1)^2)`

Returns
tensor of shape `[..., beta, alpha]`

Return type
torch.Tensor

class e3nn.o3.**FromS2Grid**(*res=None*, *lmax=None*, *normalization='component'*, *lmax_in=None*, *dtype=None*, *device=None*)

Bases: `Module`

Transform signal on the sphere into spherical tensor

The inverse transformation of *ToS2Grid*

Parameters

- **res** (*int, tuple of int*) – resolution in `beta` and in `alpha`
- **lmax** (*int*) –
- **normalization** (*{'norm', 'component', 'integral'}*) –
- **lmax_in** (*int, optional*) –
- **dtype** (*torch.dtype or None, optional*) –
- **device** (*torch.device or None, optional*) –

Examples

```
>>> m = FromS2Grid((100, 101), 6)
>>> x = torch.randn(3, 100, 101)
>>> m(x).shape
torch.Size([3, 49])
```

ToS2Grid and *FromS2Grid* are inverse of each other

```
>>> m = FromS2Grid((100, 101), 6)
>>> k = ToS2Grid(6, (100, 101))
>>> x = torch.randn(3, 100, 101)
>>> x = k(m(x))  # remove high frequencies
>>> y = k(m(x))
>>> (x - y).abs().max().item() < 1e-4
True
```

grid

positions on the sphere, tensor of shape `(res_beta, res_alpha, 3)`

Type
torch.Tensor

Methods:

forward(x)	Evaluate

forward(x)

Evaluate

Parameters
x (torch.Tensor) – tensor of shape `[..., beta, alpha]`

Returns
tensor of shape `(..., (l+1)^2)`

Return type
torch.Tensor

Wigner Functions

e3nn.o3.**wigner_D**(*l*, *alpha*, *beta*, *gamma*)

Wigner D matrix representation of $SO(3)$.

It satisfies the following properties:

- $D(\text{identity rotation}) = \text{identity matrix}$
- $D(R_1 \circ R_2) = D(R_1) \circ D(R_2)$
- $D(R^{-1}) = D(R)^{-1} = D(R)^T$
- $D(\text{rotation around Y axis})$ has some property that allows us to use FFT in *ToS2Grid*

Parameters

- **l** (*int*) – l
- **alpha** (torch.Tensor) – tensor of shape $(...)$ Rotation α around Y axis, applied third.
- **beta** (torch.Tensor) – tensor of shape $(...)$ Rotation β around X axis, applied second.
- **gamma** (torch.Tensor) – tensor of shape $(...)$ Rotation γ around Y axis, applied first.

Returns
tensor $D^l(\alpha, \beta, \gamma)$ of shape $(2l+1, 2l+1)$

Return type
torch.Tensor

e3nn.o3.**wigner_3j**(*l1*, *l2*, *l3*, *dtype=None*, *device=None*)

Wigner 3j symbols C_{lmn}.

It satisfies the following two properties:

$$C_{lmn} = C_{ijk} D_{il}(g) D_{jm}(g) D_{kn}(g) \qquad \forall g \in SO(3)$$

where D are given by *wigner_D*.

$$C_{ijk} C_{ijk} = 1$$

Parameters

- **l1** (*int*) – l_1
- **l2** (*int*) – l_2
- **l3** (*int*) – l_3
- **dtype** (*torch.dtype or None*) – `dtype` of the returned tensor. If `None` then set to `torch.get_default_dtype()`.
- **device** (*torch.device or None*) – `device` of the returned tensor. If `None` then set to the default device of the current context.

Returns

tensor C of shape $(2l_1 + 1, 2l_2 + 1, 2l_3 + 1)$

Return type

`torch.Tensor`

2.1.2 nn

Overview

Gate

class e3nn.nn.**Activation**(*irreps_in*, *acts*)

Bases: `Module`

Scalar activation function.

Odd scalar inputs require activation functions with a defined parity (odd or even).

Parameters

- **irreps_in** (*e3nn.o3.Irreps*) – representation of the input
- **acts** (*list of function or None*) – list of activation functions, `None` if non-scalar or identity

Examples

```
>>> a = Activation("256x0o", [torch.abs])
>>> a.irreps_out
256x0e
```

```
>>> a = Activation("256x0o+16x1e", [None, None])
>>> a.irreps_out
256x0o+16x1e
```

Methods:

forward(features[, dim])	evaluate

forward(*features*, *dim=-1*)

evaluate

Parameters
features (torch.Tensor) – tensor of shape (...)

Returns
tensor of shape the same shape as the input

Return type
torch.Tensor

class e3nn.nn.**Gate**(*irreps_scalars*, *act_scalars*, *irreps_gates*, *act_gates*, *irreps_gated*)

Bases: Module

Gate activation function.

The gate activation is a direct sum of two sets of irreps. The first set of irreps is `irreps_scalars` passed through activation functions `act_scalars`. The second set of irreps is `irreps_gated` multiplied by the scalars `irreps_gates` passed through activation functions `act_gates`. Mathematically, this can be written as:

$$\left(\bigoplus_i \phi_i(x_i)\right) \oplus \left(\bigoplus_j \phi_j(g_j) y_j\right)$$

where x_i and ϕ_i are from `irreps_scalars` and `act_scalars`, and g_j, ϕ_j, and y_j are from `irreps_gates`, `act_gates`, and `irreps_gated`.

The parameters passed in should adhere to the following conditions:

1. `len(irreps_scalars) == len(act_scalars)`.
2. `len(irreps_gates) == len(act_gates)`.
3. `irreps_gates.num_irreps == irreps_gated.num_irreps`.

Parameters

- **irreps_scalars** (*e3nn.o3.Irreps*) – Representation of the scalars that will be passed through the activation functions `act_scalars`.
- **act_scalars** (*list of function or None*) – Activation functions acting on the scalars.
- **irreps_gates** (*e3nn.o3.Irreps*) – Representation of the scalars that will be passed through the activation functions `act_gates` and multiplied by the `irreps_gated`.
- **act_gates** (*list of function or None*) – Activation functions acting on the gates. The number of functions in the list should match the number of irrep groups in `irreps_gates`.
- **irreps_gated** (*e3nn.o3.Irreps*) – Representation of the gated tensors. `irreps_gates.num_irreps == irreps_gated.num_irreps`

Examples

```
>>> g = Gate("16x0o", [torch.tanh], "32x0o", [torch.tanh], "16x1e+16x1o")
>>> g.irreps_out
16x0o+16x1o+16x1e
```

Methods:

forward(features)	Evaluate the gated activation function.

Attributes:

irreps_in	Input representations.
irreps_out	Output representations.

forward(*features*)

Evaluate the gated activation function.

Parameters
features (torch.Tensor) – tensor of shape (..., irreps_in.dim)

Returns
tensor of shape (..., irreps_out.dim)

Return type
torch.Tensor

property irreps_in

Input representations.

property irreps_out

Output representations.

Fully Connected Neural Network

class e3nn.nn.**FullyConnectedNet**(*hs*, *act=None*, *variance_in=1*, *variance_out=1*, *out_act=False*)

Bases: Sequential

Fully-connected Neural Network

Parameters

- **hs** (*list of int*) – input, internal and output dimensions
- **act** (*function*) – activation function ϕ, it will be automatically normalized by a scaling factor such that

$$\int_{-\infty}^{\infty} \phi(z)^2 \frac{e^{-z^2/2}}{\sqrt{2\pi}} dz = 1$$

Batch Normalization

class e3nn.nn.**BatchNorm**(*irreps*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *reduce='mean'*, *instance=False*, *normalization='component'*)

Bases: `Module`

Batch normalization for orthonormal representations

It normalizes by the norm of the representations. Note that the norm is invariant only for orthonormal representations. Irreducible representations *wigner_D* are orthonormal.

Parameters

- **irreps** (*o3.Irreps*) – representation
- **eps** (*float*) – avoid division by zero when we normalize by the variance
- **momentum** (*float*) – momentum of the running average
- **affine** (*bool*) – do we have weight and bias parameters
- **reduce** (*{'mean', 'max'}*) – method used to reduce
- **instance** (*bool*) – apply instance norm instead of batch norm

Methods:

forward(input)	evaluate

forward(*input*)

evaluate

Parameters
input (torch.Tensor) – tensor of shape `(batch, ..., irreps.dim)`

Returns
tensor of shape `(batch, ..., irreps.dim)`

Return type
torch.Tensor

Spherical Activation

class e3nn.nn.**S2Activation**(*irreps:* Irreps, *act*, *res*, *normalization='component'*, *lmax_out=None*, *random_rot=False*)

Bases: `Module`

Apply non linearity on the signal on the sphere

Maps to the sphere, apply the non linearity point wise and project back.
The signal on the sphere is a quasiregular representation of $O(3)$ and we can apply a pointwise operation on these representations.

$$\{A^l\}_l \mapsto \{\int \phi(\sum_l A^l \cdot Y^l(x)) Y^j(x) dx\}_j$$

Parameters

- **irreps** (*o3.Irreps*) – input representation of the form `[(1, (l, p_val * (p_arg)^l)) for l in [0, ..., lmax]]`
- **act** (*function*) – activation function ϕ
- **res** (*int*) – resolution of the grid on the sphere (the higher the more accurate)
- **normalization** (*{'norm', 'component'}*) –
- **lmax_out** (*int, optional*) – maximum `l` of the output
- **random_rot** (*bool*) – rotate randomly the grid

Examples

```
>>> from e3nn import io
>>> m = S2Activation(io.SphericalTensor(5, p_val=+1, p_arg=-1), torch.tanh, 100)
```

Methods:

forward(features)	evaluate

forward(*features*)

evaluate

Parameters
features (`torch.Tensor`) – tensor $\{A^l\}_l$ of shape `(..., self.irreps_in.dim)`

Returns
tensor of shape `(..., self.irreps_out.dim)`

Return type
`torch.Tensor`

Norm-Based Activation

class e3nn.nn.**NormActivation**(*irreps_in*, *scalar_nonlinearity: Callable*, *normalize: bool = True*, *epsilon: Optional[float] = None*, *bias: bool = False*)

Bases: `Module`

Norm-based activation function Applies a scalar nonlinearity to the norm of each irrep and ouputs a (normalized) version of that irrep multiplied by the scalar output of the scalar nonlinearity. :param irreps_in: representation of the input :type irreps_in: *e3nn.o3.Irreps* :param scalar_nonlinearity: scalar nonlinearity such as `torch.sigmoid` :type scalar_nonlinearity: callable :param normalize: whether to normalize the input features before multiplying them by the scalars from the nonlinearity :type normalize: bool :param epsilon: when `normalize``ing, norms smaller than ``epsilon` will be clamped up to `epsilon` to avoid division by zero and

NaN gradients. Not allowed when `normalize` is False.

Parameters
bias (*bool*) – whether to apply a learnable additive bias to the inputs of the `scalar_nonlinearity`

Examples

```
>>> n = NormActivation("2x1e", torch.sigmoid)
>>> feats = torch.ones(1, 2*3)
>>> print(feats.reshape(1, 2, 3).norm(dim=-1))
tensor([[1.7321, 1.7321]])
>>> print(torch.sigmoid(feats.reshape(1, 2, 3).norm(dim=-1)))
tensor([[0.8497, 0.8497]])
>>> print(n(feats).reshape(1, 2, 3).norm(dim=-1))
tensor([[0.8497, 0.8497]])
```

Methods:

forward(features)	evaluate :param features: tensor of shape `(..., irreps_in.dim)` :type features: `torch.Tensor`

forward(*features*)

evaluate :param features: tensor of shape `(..., irreps_in.dim)` :type features: `torch.Tensor`

Returns

tensor of shape `(..., irreps_in.dim)`

Return type

`torch.Tensor`

nn - Models

Overview

Models of March 2021

Simple Network

Let's create a simple network and evaluate it on random data.

```
import torch
from e3nn.nn.models.v2103.gate_points_networks import SimpleNetwork

net = SimpleNetwork(
    irreps_in="3x0e + 2x1o",
    irreps_out="1x1o",
    max_radius=2.0,
    num_neighbors=3.0,
    num_nodes=5.0
)

pos = torch.randn(5, 3)
x = net.irreps_in.randn(5, -1)

net({
    'pos': pos,
```

(continues on next page)

(continued from previous page)

```
    'x': x
})
```

```
tensor([[-0.8386, -1.1427,  0.7933]], grad_fn=<DivBackward0>)
```

If we rotate the inputs,

```
from e3nn import o3

rot = o3.matrix_x(torch.tensor(3.14 / 3.0))
rot
```

```
tensor([[ 1.0000,  0.0000,  0.0000],
        [ 0.0000,  0.5005, -0.8658],
        [ 0.0000,  0.8658,  0.5005]])
```

```
net({
    'pos': pos @ rot.T,
    'x': x @ net.irreps_in.D_from_matrix(rot).T
})
```

```
tensor([[-0.8386, -1.2587, -0.5923]], grad_fn=<DivBackward0>)
```

it gives the same result as rotating the outputs.

```
net({
    'pos': pos,
    'x': x
}) @ net.irreps_out.D_from_matrix(rot).T
```

```
tensor([[-0.8386, -1.2587, -0.5923]], grad_fn=<MmBackward0>)
```

Network for a graph with node/edge attributes

A graph is made of nodes and edges. The nodes and edges can have attributes. Usually their only attributes are the positions of the nodes $\vec{r}_i$ and the relative positions of the edges $\vec{r}_i - \vec{r}_j$. We typically don't use the node positions because they change with the global translation of the graph. The nodes and edges can have other attributes like for instance atom type or bond type and so on.

The attributes defines the graph properties. They don't change layer after layer (in this example). The data (`node_input`) flow through this graph layer after layer.

In the following network, the edges attributes are the spherical harmonics $Y^l(\vec{r}_i - \vec{r}_j)$ plus the extra attributes provided by the user.

```
from e3nn.nn.models.v2103.gate_points_networks import NetworkForAGraphWithAttributes
from torch_cluster import radius_graph

max_radius = 3.0
```

(continues on next page)

(continued from previous page)

```
net = NetworkForAGraphWithAttributes(
    irreps_node_input="0e+1e",
    irreps_node_attr="0e+1e",
    irreps_edge_attr="0e+1e",  # attributes in extra of the spherical harmonics
    irreps_node_output="0e+1e",
    max_radius=max_radius,
    num_neighbors=4.0,
    num_nodes=5.0,
)

num_nodes = 5
pos = torch.randn(num_nodes, 4)
edge_index = radius_graph(pos, max_radius)
num_edges = edge_index.shape[1]

net({
    'pos': pos,
    'edge_index': edge_index,
    'node_input': torch.randn(num_nodes, 4),
    'node_attr': torch.randn(num_nodes, 4),
    'edge_attr': torch.randn(num_edges, 4),
})
```

```
tensor([[-13.9198,   0.7475,  -0.2959,   5.0120]], grad_fn=<DivBackward0>)
```

Model Gate of January 2021

Multipurpose equivariant neural network for point-clouds. Made with *e3nn.o3.TensorProduct* for the linear part and *e3nn.nn.Gate* for the nonlinearities.

Convolution

The linear part, module `Convolution`, is inspired from the `Depth wise Separable Convolution` idea. The main operation of the Convolution module is `tp`. It makes the atoms interact with their neighbors but does not mix the channels. To mix the channels, it is sandwiched between `lin1` and `lin2`.

```
@compile_mode("script")
class Convolution(torch.nn.Module):
    r"""equivariant convolution

    Parameters
    ----------
    irreps_in : `e3nn.o3.Irreps`
        representation of the input node features

    irreps_node_attr : `e3nn.o3.Irreps`
        representation of the node attributes

    irreps_edge_attr : `e3nn.o3.Irreps`
```

(continues on next page)

(continued from previous page)

```
        representation of the edge attributes

    irreps_out : `e3nn.o3.Irreps` or None
        representation of the output node features

    number_of_basis : int
        number of basis on which the edge length are projected

    radial_layers : int
        number of hidden layers in the radial fully connected network

    radial_neurons : int
        number of neurons in the hidden layers of the radial fully connected network

    num_neighbors : float
        typical number of nodes convolved over
    """

    def __init__(
        self,
        irreps_in,
        irreps_node_attr,
        irreps_edge_attr,
        irreps_out,
        number_of_basis,
        radial_layers,
        radial_neurons,
        num_neighbors,
    ) -> None:
        super().__init__()
        self.irreps_in = o3.Irreps(irreps_in)
        self.irreps_node_attr = o3.Irreps(irreps_node_attr)
        self.irreps_edge_attr = o3.Irreps(irreps_edge_attr)
        self.irreps_out = o3.Irreps(irreps_out)
        self.num_neighbors = num_neighbors

        self.sc = FullyConnectedTensorProduct(self.irreps_in, self.irreps_node_attr,␣
↪self.irreps_out)

        self.lin1 = FullyConnectedTensorProduct(self.irreps_in, self.irreps_node_attr,␣
↪self.irreps_in)

        irreps_mid = []
        instructions = []
        for i, (mul, ir_in) in enumerate(self.irreps_in):
            for j, (_, ir_edge) in enumerate(self.irreps_edge_attr):
                for ir_out in ir_in * ir_edge:
                    if ir_out in self.irreps_out:
                        k = len(irreps_mid)
                        irreps_mid.append((mul, ir_out))
                        instructions.append((i, j, k, "uvu", True))
        irreps_mid = o3.Irreps(irreps_mid)
```

(continues on next page)

(continued from previous page)

```
        irreps_mid, p, _ = irreps_mid.sort()

        instructions = [(i_1, i_2, p[i_out], mode, train) for i_1, i_2, i_out, mode,␣
↪train in instructions]

        tp = TensorProduct(
            self.irreps_in,
            self.irreps_edge_attr,
            irreps_mid,
            instructions,
            internal_weights=False,
            shared_weights=False,
        )
        self.fc = FullyConnectedNet(
            [number_of_basis] + radial_layers * [radial_neurons] + [tp.weight_numel],␣
↪torch.nn.functional.silu
        )
        self.tp = tp

        self.lin2 = FullyConnectedTensorProduct(irreps_mid, self.irreps_node_attr, self.
↪irreps_out)

    def forward(self, node_input, node_attr, edge_src, edge_dst, edge_attr, edge_length_
↪embedded) -> torch.Tensor:
        weight = self.fc(edge_length_embedded)

        x = node_input

        s = self.sc(x, node_attr)
        x = self.lin1(x, node_attr)

        edge_features = self.tp(x[edge_src], edge_attr, weight)
        x = scatter(edge_features, edge_dst, dim=0, dim_size=x.shape[0]).div(self.num_
↪neighbors**0.5)

        x = self.lin2(x, node_attr)

        c_s, c_x = math.sin(math.pi / 8), math.cos(math.pi / 8)
        m = self.sc.output_mask
        c_x = (1 - m) + c_x * m
        return c_s * s + c_x * x
```

Network

The network is a simple succession of `Convolution` and *e3nn.nn.Gate*. The activation function is ReLU when dealing with even scalars and tanh of abs when dealing with even scalars. When the parities (`p` in *e3nn.o3.Irrep*) are provided, network is equivariant to `O(3)`. To relax this constraint and make it equivariant to `SO(3)` only, one can simply pass all the `irreps` parameters to be even (`p=1` in *e3nn.o3.Irrep*). This is why `irreps_sh` is a parameter of the class `Network`, one can use specific `l` of the spherical harmonics with the correct parity `p=(-1)^l` (one can use *e3nn.o3.Irreps.spherical_harmonics* for that) or consider that `p=1` in order to **not** be equivariant to parity.

```
class Network(torch.nn.Module):
    r"""equivariant neural network

    Parameters
    ----------
    irreps_in : `e3nn.o3.Irreps` or None
        representation of the input features
        can be set to ``None`` if nodes don't have input features

    irreps_hidden : `e3nn.o3.Irreps`
        representation of the hidden features

    irreps_out : `e3nn.o3.Irreps`
        representation of the output features

    irreps_node_attr : `e3nn.o3.Irreps` or None
        representation of the nodes attributes
        can be set to ``None`` if nodes don't have attributes

    irreps_edge_attr : `e3nn.o3.Irreps`
        representation of the edge attributes
        the edge attributes are :math:`h(r) Y(\vec r / r)`
        where :math:`h` is a smooth function that goes to zero at ``max_radius``
        and :math:`Y` are the spherical harmonics polynomials

    layers : int
        number of gates (non linearities)

    max_radius : float
        maximum radius for the convolution

    number_of_basis : int
        number of basis on which the edge length are projected

    radial_layers : int
        number of hidden layers in the radial fully connected network

    radial_neurons : int
        number of neurons in the hidden layers of the radial fully connected network

    num_neighbors : float
        typical number of nodes at a distance ``max_radius``

    num_nodes : float
```

(continues on next page)

(continued from previous page)

```
        typical number of nodes in a graph
    """

    def __init__(
        self,
        irreps_in,
        irreps_hidden,
        irreps_out,
        irreps_node_attr,
        irreps_edge_attr,
        layers,
        max_radius,
        number_of_basis,
        radial_layers,
        radial_neurons,
        num_neighbors,
        num_nodes,
        reduce_output=True,
    ) -> None:
        super().__init__()
        self.max_radius = max_radius
        self.number_of_basis = number_of_basis
        self.num_neighbors = num_neighbors
        self.num_nodes = num_nodes
        self.reduce_output = reduce_output

        self.irreps_in = o3.Irreps(irreps_in) if irreps_in is not None else None
        self.irreps_hidden = o3.Irreps(irreps_hidden)
        self.irreps_out = o3.Irreps(irreps_out)
        self.irreps_node_attr = o3.Irreps(irreps_node_attr) if irreps_node_attr is not␣
↪None else o3.Irreps("0e")
        self.irreps_edge_attr = o3.Irreps(irreps_edge_attr)

        self.input_has_node_in = irreps_in is not None
        self.input_has_node_attr = irreps_node_attr is not None

        irreps = self.irreps_in if self.irreps_in is not None else o3.Irreps("0e")

        act = {
            1: torch.nn.functional.silu,
            -1: torch.tanh,
        }
        act_gates = {
            1: torch.sigmoid,
            -1: torch.tanh,
        }

        self.layers = torch.nn.ModuleList()

        for _ in range(layers):
            irreps_scalars = o3.Irreps(
                [
```

(continues on next page)

(continued from previous page)

```
                    (mul, ir)
                    for mul, ir in self.irreps_hidden
                    if ir.l == 0 and tp_path_exists(irreps, self.irreps_edge_attr, ir)
                ]
            )
            irreps_gated = o3.Irreps(
                [(mul, ir) for mul, ir in self.irreps_hidden if ir.l > 0 and tp_path_
↪exists(irreps, self.irreps_edge_attr, ir)]
            )
            ir = "0e" if tp_path_exists(irreps, self.irreps_edge_attr, "0e") else "0o"
            irreps_gates = o3.Irreps([(mul, ir) for mul, _ in irreps_gated])

            gate = Gate(
                irreps_scalars,
                [act[ir.p] for _, ir in irreps_scalars],  # scalar
                irreps_gates,
                [act_gates[ir.p] for _, ir in irreps_gates],  # gates (scalars)
                irreps_gated,  # gated tensors
            )
            conv = Convolution(
                irreps,
                self.irreps_node_attr,
                self.irreps_edge_attr,
                gate.irreps_in,
                number_of_basis,
                radial_layers,
                radial_neurons,
                num_neighbors,
            )
            irreps = gate.irreps_out
            self.layers.append(Compose(conv, gate))

        self.layers.append(
            Convolution(
                irreps,
                self.irreps_node_attr,
                self.irreps_edge_attr,
                self.irreps_out,
                number_of_basis,
                radial_layers,
                radial_neurons,
                num_neighbors,
            )
        )

    def forward(self, data: Union[Data, Dict[str, torch.Tensor]]) -> torch.Tensor:
        """evaluate the network

        Parameters
        ----------
        data : `torch_geometric.data.Data` or dict
            data object containing
```

(continues on next page)

(continued from previous page)

```
            - ``pos`` the position of the nodes (atoms)
            - ``x`` the input features of the nodes, optional
            - ``z`` the attributes of the nodes, for instance the atom type, optional
            - ``batch`` the graph to which the node belong, optional
        """
        if "batch" in data:
            batch = data["batch"]
        else:
            batch = data["pos"].new_zeros(data["pos"].shape[0], dtype=torch.long)

        edge_index = radius_graph(data["pos"], self.max_radius, batch)
        edge_src = edge_index[0]
        edge_dst = edge_index[1]
        edge_vec = data["pos"][edge_src] - data["pos"][edge_dst]
        edge_sh = o3.spherical_harmonics(self.irreps_edge_attr, edge_vec, True,␣
↪normalization="component")
        edge_length = edge_vec.norm(dim=1)
        edge_length_embedded = soft_one_hot_linspace(
            x=edge_length, start=0.0, end=self.max_radius, number=self.number_of_basis,␣
↪basis="gaussian", cutoff=False
        ).mul(self.number_of_basis**0.5)
        edge_attr = smooth_cutoff(edge_length / self.max_radius)[:, None] * edge_sh

        if self.input_has_node_in and "x" in data:
            assert self.irreps_in is not None
            x = data["x"]
        else:
            assert self.irreps_in is None
            x = data["pos"].new_ones((data["pos"].shape[0], 1))

        if self.input_has_node_attr and "z" in data:
            z = data["z"]
        else:
            assert self.irreps_node_attr == o3.Irreps("0e")
            z = data["pos"].new_ones((data["pos"].shape[0], 1))

        for lay in self.layers:
```

model with self-interactions and gates

Exact equivariance to $E(3)$

version of january 2021

Classes:

Compose(first, second)	

Convolution(irreps_in, irreps_node_attr, ...)	equivariant convolution
Network(irreps_in, irreps_hidden, ...[, ...])	equivariant neural network

class e3nn.nn.models.gate_points_2101.**Compose**(*first*, *second*)

Bases: Module

Methods:

forward(*input)	Defines the computation performed at every call.

forward(**input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class e3nn.nn.models.gate_points_2101.**Convolution**(*irreps_in*, *irreps_node_attr*, *irreps_edge_attr*, *irreps_out*, *number_of_basis*, *radial_layers*, *radial_neurons*, *num_neighbors*)

Bases: `Module`

equivariant convolution

Parameters

- **irreps_in** (*e3nn.o3.Irreps*) – representation of the input node features
- **irreps_node_attr** (*e3nn.o3.Irreps*) – representation of the node attributes
- **irreps_edge_attr** (*e3nn.o3.Irreps*) – representation of the edge attributes
- **irreps_out** (*e3nn.o3.Irreps* or None) – representation of the output node features
- **number_of_basis** (*int*) – number of basis on which the edge length are projected
- **radial_layers** (*int*) – number of hidden layers in the radial fully connected network
- **radial_neurons** (*int*) – number of neurons in the hidden layers of the radial fully connected network
- **num_neighbors** (*float*) – typical number of nodes convolved over

Methods:

forward(node_input, node_attr, edge_src, ...)	Defines the computation performed at every call.

forward(*node_input*, *node_attr*, *edge_src*, *edge_dst*, *edge_attr*, *edge_length_embedded*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class e3nn.nn.models.gate_points_2101.**Network**(*irreps_in*, *irreps_hidden*, *irreps_out*, *irreps_node_attr*, *irreps_edge_attr*, *layers*, *max_radius*, *number_of_basis*, *radial_layers*, *radial_neurons*, *num_neighbors*, *num_nodes*, *reduce_output=True*)

Bases: `Module`

equivariant neural network

Parameters

- **irreps_in** (*e3nn.o3.Irreps* or None) – representation of the input features can be set to `None` if nodes don't have input features
- **irreps_hidden** (*e3nn.o3.Irreps*) – representation of the hidden features
- **irreps_out** (*e3nn.o3.Irreps*) – representation of the output features
- **irreps_node_attr** (*e3nn.o3.Irreps* or None) – representation of the nodes attributes can be set to `None` if nodes don't have attributes
- **irreps_edge_attr** (*e3nn.o3.Irreps*) – representation of the edge attributes the edge attributes are $h(r)Y(\vec{r}/r)$ where h is a smooth function that goes to zero at `max_radius` and Y are the spherical harmonics polynomials
- **layers** (*int*) – number of gates (non linearities)
- **max_radius** (*float*) – maximum radius for the convolution
- **number_of_basis** (*int*) – number of basis on which the edge length are projected
- **radial_layers** (*int*) – number of hidden layers in the radial fully connected network
- **radial_neurons** (*int*) – number of neurons in the hidden layers of the radial fully connected network
- **num_neighbors** (*float*) – typical number of nodes at a distance `max_radius`
- **num_nodes** (*float*) – typical number of nodes in a graph

Methods:

forward(data)	evaluate the network

forward(*data: Union[Data, Dict[str, Tensor]]*) → Tensor

evaluate the network

Parameters

data (`torch_geometric.data.Data` or dict) – data object containing - `pos` the position of the nodes (atoms) - `x` the input features of the nodes, optional - `z` the attributes of the nodes, for instance the atom type, optional - `batch` the graph to which the node belong, optional

2.1.3 io

This submodule contains subclasses of *e3nn.o3.Irreps* for specialized representations.

Overview

Spherical Tensor

There exists 4 types of function on the sphere depending on how the parity affects it. The representation of the coefficients are affected by this choice:

```
import torch
from e3nn.io import SphericalTensor

print(SphericalTensor(lmax=2, p_val=1, p_arg=1))
print(SphericalTensor(lmax=2, p_val=1, p_arg=-1))
print(SphericalTensor(lmax=2, p_val=-1, p_arg=1))
print(SphericalTensor(lmax=2, p_val=-1, p_arg=-1))
```

```
1x0e+1x1e+1x2e
1x0e+1x1o+1x2e
1x0o+1x1o+1x2o
1x0o+1x1e+1x2o
```

```
import plotly.graph_objects as go

def plot(traces):
    traces = [go.Surface(**d) for d in traces]
    fig = go.Figure(data=traces)
    fig.show()
```

In the following graph we show the four possible behavior under parity for a function on the sphere.

1. This first ball shows $f(x)$ unaffected by the parity
2. Then `p_val=1` but `p_arg=-1` so we see the signal flipped over the sphere but the colors are unchanged
3. For `p_val=-1` and `p_arg=1` only the value of the signal flips its sign
4. For `p_val=-1` and `p_arg=-1` both in the same time, the signal flips over the sphere and the value flip its sign

```
lmax = 1
x = torch.tensor([0.8] + [0.0, 0.0, 1.0])

parity = -torch.eye(3)

x = torch.stack([
    SphericalTensor(lmax, p_val, p_arg).D_from_matrix(parity) @ x
    for p_val in [+1, -1]
    for p_arg in [+1, -1]
])
centers = torch.tensor([
    [-3.0, 0.0, 0.0],
    [-1.0, 0.0, 0.0],
    [1.0, 0.0, 0.0],
    [3.0, 0.0, 0.0],
])
```

(continues on next page)

(continued from previous page)

```
st = SphericalTensor(lmax, 1, 1)  # p_val and p_arg set arbitrarily here
plot(st.plotly_surface(x, centers=centers, radius=False))
```

class e3nn.io.**SphericalTensor**(*lmax*, *p_val*, *p_arg*)

Bases: *Irreps*

representation of a signal on the sphere

A *SphericalTensor* contains the coefficients A^l of a function f defined on the sphere

$$f(x) = \sum_{l=0}^{l_{\max}} A^l \cdot Y^l(x)$$

The way this function is transformed by parity $f \longrightarrow Pf$ is described by the two parameters p_v and p_a

$$\begin{aligned}(Pf)(x) &= p_v f(p_a x) \\ &= \sum_{l=0}^{l_{\max}} p_v p_a^l A^l \cdot Y^l(x)\end{aligned}$$

Parameters

- **lmax** (*int*) – $l_{\max}$
- **p_val** (*{+1, -1}*) – p_v
- **p_arg** (*{+1, -1}*) – p_a

Examples

```
>>> SphericalTensor(3, 1, 1)
1x0e+1x1e+1x2e+1x3e
```

```
>>> SphericalTensor(3, 1, -1)
1x0e+1x1o+1x2e+1x3o
```

Methods:

find_peaks(signal[, res])	Locate peaks on the sphere
from_samples_on_s2(positions, values[, res])	Convert a set of position on the sphere and values into a spherical tensor
norms(signal)	The norms of each l component
plot(signal[, center, res, radius, relu, ...])	Create surface in order to make a plot
plotly_surface(signals[, centers, res, ...])	Create traces for plotly
signal_on_grid(signal[, res, normalization])	Evaluate the signal on a grid on the sphere
signal_xyz(signal, r)	Evaluate the signal on given points on the sphere
sum_of_diracs(positions, values)	Sum (almost-) dirac deltas
with_peaks_at(vectors[, values])	Create a spherical tensor with peaks

find_peaks(*signal*, *res=100*)

Locate peaks on the sphere

Examples

```
>>> s = SphericalTensor(4, 1, -1)
>>> pos = torch.tensor([
...     [4.0, 0.0, 4.0],
...     [0.0, 5.0, 0.0],
... ])
>>> x = s.with_peaks_at(pos)
>>> pos, val = s.find_peaks(x)
>>> pos[val > 4.0].mul(10).round().abs()
tensor([[ 7.,  0.,  7.],
        [ 0., 10.,  0.]])
>>> val[val > 4.0].mul(10).round().abs()
tensor([57., 50.])
```

from_samples_on_s2(*positions: Tensor*, *values: Tensor*, *res=100*) → Tensor

Convert a set of position on the sphere and values into a spherical tensor

Parameters

- **positions** (torch.Tensor) – tensor of shape `(..., N, 3)`
- **values** (torch.Tensor) – tensor of shape `(..., N)`

Returns

tensor of shape `(..., self.dim)`

Return type

torch.Tensor

Examples

```
>>> s = SphericalTensor(2, 1, 1)
>>> pos = torch.tensor([
...     [
...         [0.0, 0.0, 1.0],
...         [0.0, 0.0, -1.0],
...     ],
...     [
...         [0.0, 1.0, 0.0],
...         [0.0, -1.0, 0.0],
...     ],
... ], dtype=torch.float64)
>>> val = torch.tensor([
...     [
...         1.0,
...         -1.0,
...     ],
...     [
...         1.0,
...         -1.0,
...     ],
... ], dtype=torch.float64)
>>> s.from_samples_on_s2(pos, val, res=200).long()
```

(continues on next page)

(continued from previous page)

```
tensor([[0, 0, 0, 3, 0, 0, 0, 0, 0],
        [0, 0, 3, 0, 0, 0, 0, 0, 0]])
```

```
>>> pos = torch.empty(2, 0, 10, 3)
>>> val = torch.empty(2, 0, 10)
>>> s.from_samples_on_s2(pos, val)
tensor([], size=(2, 0, 9))
```

norms(*signal*)

The norms of each l component

Parameters

signal (torch.Tensor) – tensor of shape `(..., dim)`

Returns

tensor of shape `(..., lmax+1)`

Return type

torch.Tensor

Examples

Examples

```
>>> s = SphericalTensor(1, 1, -1)
>>> s.norms(torch.tensor([1.5, 0.0, 3.0, 4.0]))
tensor([1.5000, 5.0000])
```

plot(*signal*, *center=None*, *res=100*, *radius=True*, *relu=False*, *normalization='integral'*)

Create surface in order to make a plot

plotly_surface(*signals*, *centers=None*, *res=100*, *radius=True*, *relu=False*, *normalization='integral'*)

Create traces for plotly

Examples

```
>>> import plotly.graph_objects as go
>>> x = SphericalTensor(4, +1, +1)
>>> traces = x.plotly_surface(x.randn(-1))
>>> traces = [go.Surface(**d) for d in traces]
>>> fig = go.Figure(data=traces)
```

signal_on_grid(*signal*, *res=100*, *normalization='integral'*)

Evaluate the signal on a grid on the sphere

signal_xyz(*signal*, *r*)

Evaluate the signal on given points on the sphere

$$f(\vec{x}/\|\vec{x}\|)$$

Parameters

- **signal** (torch.Tensor) – tensor of shape `(*A, self.dim)`

- **r** (torch.Tensor) – tensor of shape `(*B, 3)`

Returns
tensor of shape `(*A, *B)`

Return type
torch.Tensor

Examples

```
>>> s = SphericalTensor(3, 1, -1)
>>> s.signal_xyz(s.randn(2, 1, 3, -1), torch.randn(2, 4, 3)).shape
torch.Size([2, 1, 3, 2, 4])
```

sum_of_diracs(*positions: Tensor*, *values: Tensor*) → Tensor

Sum (almost-) dirac deltas

$$f(x) = \sum_i v_i \delta^L(\vec{r}_i)$$

where δ^L is the apporximation of a dirac delta.

Parameters

- **positions** (torch.Tensor) – $\vec{r}_i$ tensor of shape `(..., N, 3)`
- **values** (torch.Tensor) – v_i tensor of shape `(..., N)`

Returns
tensor of shape `(..., self.dim)`

Return type
torch.Tensor

Examples

```
>>> s = SphericalTensor(7, 1, -1)
>>> pos = torch.tensor([
...     [1.0, 0.0, 0.0],
...     [0.0, 1.0, 0.0],
... ])
>>> val = torch.tensor([
...     -1.0,
...     1.0,
... ])
>>> x = s.sum_of_diracs(pos, val)
>>> s.signal_xyz(x, torch.eye(3)).mul(10.0).round()
tensor([-10.,  10.,  -0.])
```

```
>>> s.sum_of_diracs(torch.empty(1, 0, 2, 3), torch.empty(2, 0, 1)).shape
torch.Size([2, 0, 64])
```

```
>>> s.sum_of_diracs(torch.randn(1, 3, 2, 3), torch.randn(2, 1, 1)).shape
torch.Size([2, 3, 64])
```

with_peaks_at(*vectors*, *values=None*)

Create a spherical tensor with peaks

The peaks are located in $\vec{r}_i$ and have amplitude $\|\vec{r}_i\|$

Parameters

- **vectors** (torch.Tensor) – $\vec{r}_i$ tensor of shape `(N, 3)`
- **values** (torch.Tensor, optional) – value on the peak, tensor of shape `(N)`

Returns

tensor of shape `(self.dim,)`

Return type

torch.Tensor

Examples

```
>>> s = SphericalTensor(4, 1, -1)
>>> pos = torch.tensor([
...     [1.0, 0.0, 0.0],
...     [3.0, 4.0, 0.0],
... ])
>>> x = s.with_peaks_at(pos)
>>> s.signal_xyz(x, pos).long()
tensor([1, 5])
```

```
>>> val = torch.tensor([
...     -1.5,
...     2.0,
... ])
>>> x = s.with_peaks_at(pos, val)
>>> s.signal_xyz(x, pos)
tensor([-1.5000,  2.0000])
```

Cartesian Tensor

class e3nn.io.**CartesianTensor**(*formula*)

Bases: *Irreps*

representation of a cartesian tensor into irreps

Parameters

formula (*str*) –

Examples

```
>>> import torch
>>> CartesianTensor("ij=-ji")
1x1e
```

```
>>> x = CartesianTensor("ijk=-jik=-ikj")
>>> x.from_cartesian(torch.ones(3, 3, 3))
tensor([0.])
```

```
>>> x.from_vectors(torch.ones(3), torch.ones(3), torch.ones(3))
tensor([0.])
```

```
>>> x = CartesianTensor("ij=ji")
>>> t = torch.arange(9).to(torch.float).view(3,3)
>>> y = x.from_cartesian(t)
>>> z = x.to_cartesian(y)
>>> torch.allclose(z, (t + t.T)/2, atol=1e-5)
True
```

Methods:

from_cartesian(data[, rtp])	convert cartesian tensor into irreps
from_vectors(*xs[, rtp])	convert $x_1 \otimes x_2 \otimes x_3 \otimes \dots$
reduced_tensor_products([data])	reduced tensor products
to_cartesian(data[, rtp])	convert irreps tensor to cartesian tensor

from_cartesian(*data*, *rtp=None*)

convert cartesian tensor into irreps

Parameters
data (torch.Tensor) – cartesian tensor of shape `(..., 3, 3, 3, ...)`

Returns
irreps tensor of shape `(..., self.dim)`

Return type
torch.Tensor

from_vectors(**xs*, *rtp=None*)

convert $x_1 \otimes x_2 \otimes x_3 \otimes \dots$

Parameters
xs (list of torch.Tensor) – list of vectors of shape `(..., 3)`

Returns
irreps tensor of shape `(..., self.dim)`

Return type
torch.Tensor

reduced_tensor_products(*data: Optional[Tensor] = None*) → *ReducedTensorProducts*

reduced tensor products

Returns
reduced tensor products

Return type
e3nn.ReducedTensorProducts

to_cartesian(*data*, *rtp=None*)

convert irreps tensor to cartesian tensor

This is the symmetry-aware inverse operation of `from_cartesian()`.

Parameters
data (torch.Tensor) – irreps tensor of shape `(..., D)`, where D is the dimension of the irreps, i.e. `D=self.dim`.

Returns
cartesian tensor of shape `(..., 3, 3, 3, ...)`

Return type
torch.Tensor

2.1.4 math

e3nn.math.**direct_sum**(**matrices*)

Direct sum of matrices, put them in the diagonal

e3nn.math.**orthonormalize**()

orthonomalize vectors

Parameters

- **original** (torch.Tensor) – list of the original vectors x
- **eps** (*float*) – a small number

Returns

- **final** (torch.Tensor) – list of orthonomalized vectors y
- **matrix** (torch.Tensor) – the matrix A such that $y = Ax$

e3nn.math.**complete_basis**()

e3nn.math.**soft_one_hot_linspace**(*x: Tensor*, *start*, *end*, *number*, *basis=None*, *cutoff=None*)

Projection on a basis of functions

Returns a set of $\{y_i(x)\}_{i=1}^N$,

$$y_i(x) = \frac{1}{Z} f_i(x)$$

where x is the input and f_i is the ith basis function. Z is a constant defined (if possible) such that,

$$\langle \sum_{i=1}^{N} y_i(x)^2 \rangle_x \approx 1$$

See the last plot below. Note that `bessel` basis cannot be normalized.

Parameters

- **x** (torch.Tensor) – tensor of shape $(...)$
- **start** (*float*) – minimum value span by the basis
- **end** (*float*) – maximum value span by the basis

- **number** (*int*) – number of basis functions N
- **basis** (*{'gaussian', 'cosine', 'smooth_finite', 'fourier', 'bessel'}*) – choice of basis family; note that due to the $1/x$ term, `bessel` basis does not satisfy the normalization of other basis choices
- **cutoff** (*bool*) – if `cutoff=True` then for all x outside of the interval defined by `(start, end)`, $\forall i,\ f_i(x) \approx 0$

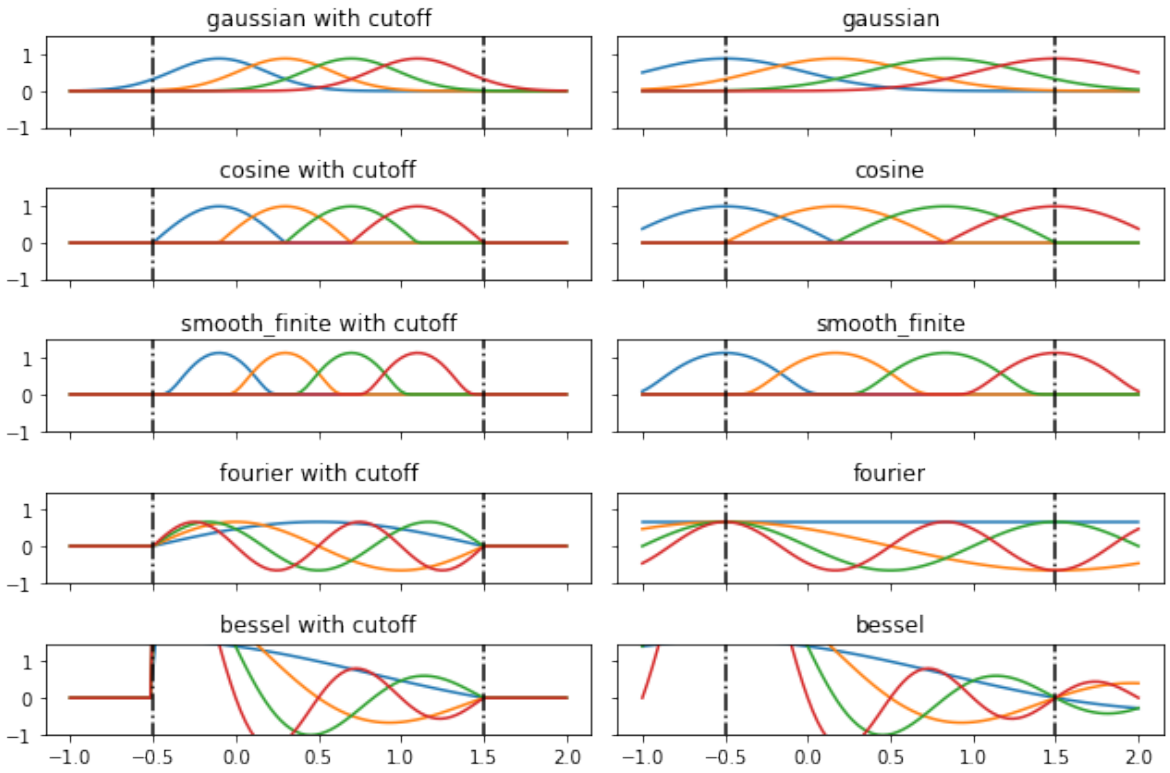
Returns
tensor of shape $(..., N)$

Return type
`torch.Tensor`

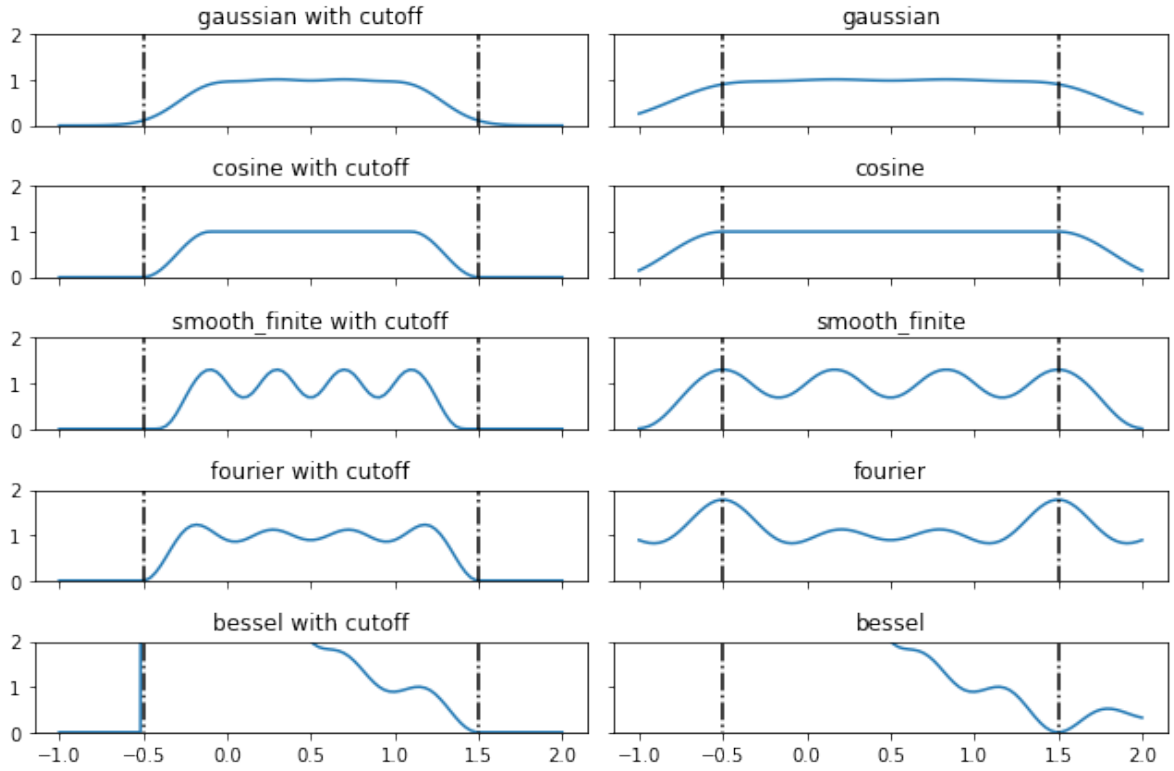
Examples

```
bases = ['gaussian', 'cosine', 'smooth_finite', 'fourier', 'bessel']
x = torch.linspace(-1.0, 2.0, 100)
```

```
fig, axss = plt.subplots(len(bases), 2, figsize=(9, 6), sharex=True, sharey=True)

for axs, b in zip(axss, bases):
    for ax, c in zip(axs, [True, False]):
        plt.sca(ax)
        plt.plot(x, soft_one_hot_linspace(x, -0.5, 1.5, number=4, basis=b,␣
↪cutoff=c))
        plt.plot([-0.5]*2, [-2, 2], 'k-.')
        plt.plot([1.5]*2, [-2, 2], 'k-.')
        plt.title(f"{b}" + (" with cutoff" if c else ""))

plt.ylim(-1, 1.5)
plt.tight_layout()
```




```
fig, axss = plt.subplots(len(bases), 2, figsize=(9, 6), sharex=True, sharey=True)

for axs, b in zip(axss, bases):
    for ax, c in zip(axs, [True, False]):
        plt.sca(ax)
        plt.plot(x, soft_one_hot_linspace(x, -0.5, 1.5, number=4, basis=b,␣
↪cutoff=c).pow(2).sum(1))
        plt.plot([-0.5]*2, [-2, 2], 'k-.')
        plt.plot([1.5]*2, [-2, 2], 'k-.')
        plt.title(f"{b}" + (" with cutoff" if c else ""))

plt.ylim(0, 2)
plt.tight_layout()
```



e3nn.math.**soft_unit_step**(x)

smooth C^∞ version of the unit step function

$$x \mapsto \theta(x)e^{-1/x}$$

Parameters

x (torch.Tensor) – tensor of shape $(...)$

Returns

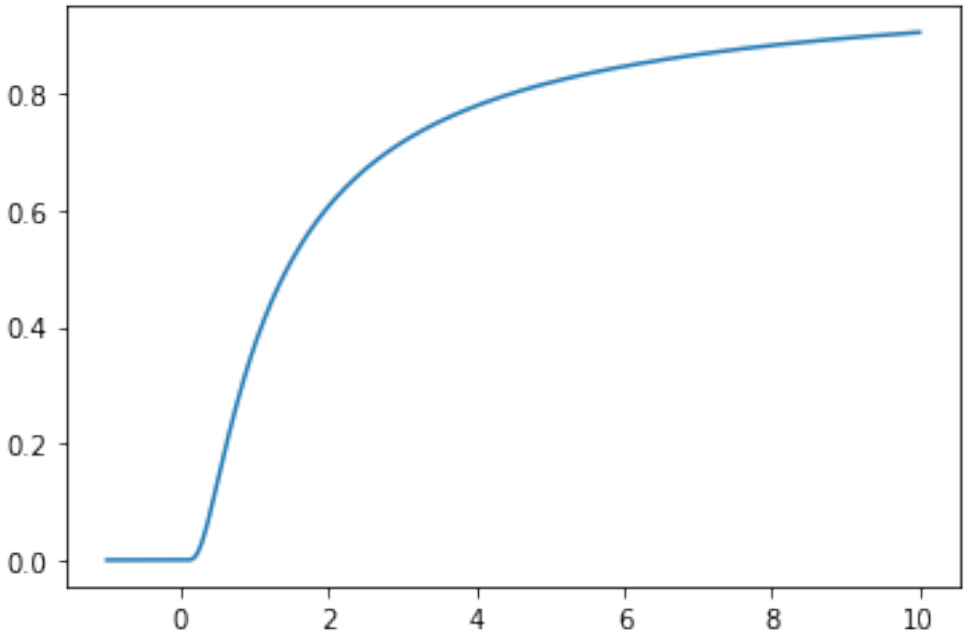
tensor of shape $(...)$

Return type

torch.Tensor

Examples

```
x = torch.linspace(-1.0, 10.0, 1000)
plt.plot(x, soft_unit_step(x));
```



2.1.5 util

Helper functions.

Overview

JIT - wrappers for TorchScript

Functions:

compile(mod[, n_trace_checks, ...])	Recursively compile a module and all submodules according to their decorators.
compile_mode(mode)	Decorator to set the compile mode of a module.
get_compile_mode(mod)	Get the compilation mode of a module.
get_tracing_inputs(mod[, n, device, dtype])	Get random tracing inputs for `mod`.
script(mod[, in_place])	Script a module.
trace(mod[, example_inputs, check_inputs, ...])	Trace a module.
trace_module(mod[, inputs, check_inputs, ...])	Trace a module.

e3nn.util.jit.**compile**(*mod: Module*, *n_trace_checks: int = 1*, *script_options: Optional[dict] = None*, *trace_options: Optional[dict] = None*, *in_place: bool = True*)

Recursively compile a module and all submodules according to their decorators.

(Sub)modules without decorators will be unaffected.

Parameters

- **mod** (*torch.nn.Module*) – The module to compile. The module will have its submodules compiled replaced in-place.
- **n_trace_checks** (*int, default = 1*) – How many random example inputs to generate when tracing a module. Must be at least one in order to have a tracing input. Extra example inputs will be pased to `torch.jit.trace` to confirm that the traced copmute graph doesn't change.
- **script_options** (*dict, default = {}*) – Extra kwargs for `torch.jit.script`.
- **trace_options** (*dict, default = {}*) – Extra kwargs for `torch.jit.trace`.

Return type
Returns the compiled module.

e3nn.util.jit.**compile_mode**(*mode: str*)

Decorator to set the compile mode of a module.

Parameters
mode (*str*) – 'script', 'trace', or None

e3nn.util.jit.**get_compile_mode**(*mod: Module*) → str

Get the compilation mode of a module.

Parameters
mod (*torch.nn.Module*) –

Return type
'script', 'trace', or None if the module was not decorated with @compile_mode

e3nn.util.jit.**get_tracing_inputs**(*mod: Module*, *n: int = 1*, *device: Optional[device] = None*, *dtype: Optional[dtype] = None*)

Get random tracing inputs for `mod`.

First checks if `mod` has a `_make_tracing_inputs` method. If so, calls it with `n` as the single argument and returns its results.

Otherwise, attempts to infer the input signature of the module using `e3nn.util._argtools._get_io_irreps`.

Parameters

- **mod** (*torch.nn.Module*) –
- **n** (*int, default = 1*) – A hint for how many inputs are wanted. Usually n will be returned, but modules don't necessarily have to.
- **device** (*torch.device*) – The device to do tracing on. If `None` (default), will be guessed.
- **dtype** (*torch.dtype*) – The dtype to trace with. If `None` (default), will be guessed.

Returns
Tracing inputs in the format of `torch.jit.trace_module`: dicts mapping method names like `'forward'` to tuples of arguments.

Return type
list of dict

e3nn.util.jit.**script**(*mod: Module*, *in_place: bool = True*)

Script a module.

Like `torch.jit.script`, but first recursively compiles `mod` using :func:`compile`.

Parameters
mod (*torch.nn.Module*) –

Return type
Scripted module.

e3nn.util.jit.**trace**(*mod: Module*, *example_inputs: Optional[tuple] = None*, *check_inputs: Optional[list] = None*, *in_place: bool = True*)

Trace a module.

Identical signature to `torch.jit.trace`, but first recursively compiles `mod` using :func:`compile`.

Parameters

- **mod** (*torch.nn.Module*) –
- **example_inputs** (*tuple*) –
- **check_inputs** (*list of tuple*) –

Return type
Traced module.

e3nn.util.jit.**trace_module**(*mod: Module*, *inputs: Optional[dict] = None*, *check_inputs: Optional[list] = None*, *in_place: bool = True*)

Trace a module.

Identical signature to `torch.jit.trace_module`, but first recursively compiles `mod` using `compile`.

Parameters

- **mod** (*torch.nn.Module*) –
- **inputs** (*dict*) –
- **check_inputs** (*list of dict*) –

Return type
Traced module.

test - helpers for unit testing

Functions:

assert_auto_jitable(func[, ...])	Assert that submodule `func` is automatically JITable.
assert_equivariant(func[, args_in, ...])	Assert that `func` is equivariant.
assert_normalized(func[, irreps_in, ...])	Assert that `func` is normalized.
equivariance_error(func, args_in[, ...])	Get the maximum equivariance error for `func` over `ntrials`
format_equivariance_error(errors)	Format the dictionary returned by `equivariance_error` into a readable string.
random_irreps([n, lmax, mul_min, mul_max, ...])	Generate random irreps parameters for testing.
set_random_seeds()	Set the random seeds to try to get some reproducibility

e3nn.util.test.**assert_auto_jitable**(*func*, *error_on_warnings=True*, *n_trace_checks=2*, *strict_shapes=True*)

Assert that submodule `func` is automatically JITable.

Parameters

- **func** (*Callable*) – The function to trace.
- **error_on_warnings** (*bool*) – If True (default), TracerWarnings emitted by `torch.jit.trace` will be treated as errors.
- **n_random_tests** (*int*) – If `args_in` is `None` and arguments are being automatically generated, this many random arguments will be generated as test inputs for `torch.jit.trace`.
- **strict_shapes** (*bool*) – Test that the traced function errors on inputs with feature dimensions that don't match the input irreps.

Return type
The traced TorchScript function.

`e3nn.util.test.`**assert_equivariant**(*func*, *args_in=None*, *irreps_in=None*, *irreps_out=None*, *tolerance=None*, ***kwargs*) → dict

Assert that `func` is equivariant.

Parameters

- **args_in** (*list or None*) – the original input arguments for the function. If `None` and the function has `irreps_in` consisting only of `o3.Irreps` and `'cartesian'`, random test inputs will be generated.
- **irreps_in** (*object*) – see `equivariance_error`
- **irreps_out** (*object*) – see `equivariance_error`
- **tolerance** (*float or None*) – the threshold below which the equivariance error must fall. If `None`, (the default), `FLOAT_TOLERANCE[torch.get_default_dtype()]` is used.
- ****kwargs** (*kwargs*) – passed through to `equivariance_error`.

Returns
The same as ``equivariance_error``

Return type
a dictionary mapping tuples `(parity_k, did_translate)` to errors

`e3nn.util.test.`**assert_normalized**(*func: Module*, *irreps_in=None*, *irreps_out=None*, *normalization: str = 'component'*, *n_input: int = 10000*, *n_weight: Optional[int] = None*, *weights: Optional[Iterable[Parameter]] = None*, *atol: float = 0.1*) → None

Assert that `func` is normalized.

See https://docs.e3nn.org/en/stable/guide/normalization.html for more information on the normalization scheme.

`atol`, `n_input`, and `n_weight` may need to be significantly higher in order to converge the statistics to pass the test.

Parameters

- **func** (*torch.nn.Module*) – the module to test
- **irreps_in** (*object*) – see `equivariance_error`
- **irreps_out** (*object*) – see `equivariance_error`
- **normalization** (*str, default "component"*) – one of "component" or "norm". Note that this is defined for both the inputs and the outputs; if you need seperate normalizations for input and output please file a feature request.

- **n_input** (*int, default 10_000*) – the number of input samples to use for each weight init
- **n_weight** (*int, default 20*) – the number of weight initializations to sample
- **weights** (*optional iterable of parameters*) – the weights to reinitialize `n_weight` times. If `None` (default), `func.parameters()` will be used.
- **atol** (*float, default 0.1*) – tolerance for checking moments. Higher values for this prevent explosive computational costs for this test.

e3nn.util.test.**equivariance_error**(*func*, *args_in*, *irreps_in=None*, *irreps_out=None*, *ntrials=1*, *do_parity=True*, *do_translation=True*, *transform_dtype=torch.float64*)

Get the maximum equivariance error for `func` over `ntrials`

Each trial randomizes the equivariant transformation tested.

Parameters

- **func** (*callable*) – the function to test
- **args_in** (*list*) – the original inputs to pass to `func`.
- **irreps_in** (list of *e3nn.o3.Irreps* or *e3nn.o3.Irreps*) – the input irreps for each of the arguments in `args_in`. If left as the default of `None`, `get_io_irreps` will be used to try to infer them. If a sequence is provided, valid elements are also the string `'cartesian'`, which denotes that the corresponding input should be dealt with as cartesian points in 3D, and `None`, which indicates that the argument should not be transformed.
- **irreps_out** (list of *e3nn.o3.Irreps* or *e3nn.o3.Irreps*) – the out irreps for each of the return values of `func`. Accepts similar values to `irreps_in`.
- **ntrials** (*int*) – run this many trials with random transforms
- **do_parity** (*bool*) – whether to test parity
- **do_translation** (*bool*) – whether to test translation for `'cartesian'` inputs

Returns

- dictionary mapping tuples `(parity_k, did_translate)` to an array of errors,
- *each entry the biggest over all trials for that output, in order.*

e3nn.util.test.**format_equivariance_error**(*errors: dict*) → str

Format the dictionary returned by `equivariance_error` into a readable string.

Parameters
errors (*dict*) – A dictionary of errors returned by `equivariance_error`.

Return type
A string.

e3nn.util.test.**random_irreps**(*n: int = 1*, *lmax: int = 4*, *mul_min: int = 0*, *mul_max: int = 5*, *len_min: int = 0*, *len_max: int = 4*, *clean: bool = False*, *allow_empty: bool = True*)

Generate random irreps parameters for testing.

Parameters

- **n** (*int, optional*) – How many to generate; defaults to 1.
- **lmax** (*int, optional*) – The maximum L to generate (inclusive); defaults to 4.
- **mul_min** (*int, optional*) – The smallest multiplicity to generate, defaults to 0.
- **mul_max** (*int, optional*) – The largest multiplicity to generate, defaults to 5.

- **len_min** (*int, optional*) – The smallest number of irreps to generate, defaults to 0.
- **len_max** (*int, optional*) – The largest number of irreps to generate, defaults to 4.
- **clean** (*bool, optional*) – If `True`, only `o3.Irreps` objects will be returned. If `False` (the default), `e3nn.o3.Irreps`-like objects like strings and lists of tuples can be returned.
- **allow_empty** (*bool, optional*) – Whether to allow generating empty `e3nn.o3.Irreps`.

Return type
An irreps-like object if `n == 1` or a list of them if `n > 1`

`e3nn.util.test.set_random_seeds()`
Set the random seeds to try to get some reproducibility

2.2 User Guide

Beginner

2.2.1 Install

Dependencies

PyTorch

e3nn requires PyTorch >=1.8.0 For installation instructions, please see the PyTorch homepage.

optional: torch_geometric

First you have to install pytorch_geometric. For `torch` 1.11 and no CUDA support:

```
CUDA=cpu

pip install --upgrade --force-reinstall torch-scatter -f https://data.pyg.org/whl/torch-
↪1.11.0+${CUDA}.html
pip install --upgrade --force-reinstall torch-sparse -f https://data.pyg.org/whl/torch-1.
↪11.0+${CUDA}.html
pip install torch-geometric
```

See here to get cuda support or newer versions.

e3nn

Stable (PyPI)

```
$ pip install e3nn
```

Unstable (Git)

```
$ git clone https://github.com/e3nn/e3nn.git
$ cd e3nn/
$ pip install .
```

2.2.2 Irreducible representations

This page is a beginner introduction to the main object of `e3nn` library: *e3nn.o3.Irreps*. All the core component of `e3nn` can be found in `e3nn.o3`. `o3` stands for the group of 3d orthogonal matrices, which is equivalently the group of rotation and inversion.

```
from e3nn.o3 import Irreps
```

An instance of *e3nn.o3.Irreps* describe how some data behave under rotation. The mathematical describtion of irreps can be found in the API *Irreps*.

```
irreps = Irreps("1o")
irreps
```

```
1x1o
```

`irreps` does not contain any data. Under the hood it is simply a tuple of made of other tuples and ints.

```
# Tuple[Tuple[int, Tuple[int, int]]]
# ((multiplicity, (l, p)), ...)

print(len(irreps))
mul_ir = irreps[0]  # a tuple

print(mul_ir)
print(len(mul_ir))
mul = mul_ir[0]  # an int
ir = mul_ir[1]  # another tuple

print(mul)

print(ir)
# print(len(ir))  ir is a tuple of 2 ints but __len__ has been disabled since it is␣
↪always 2
l = ir[0]
p = ir[1]

print(l, p)
```

```
1
1x1o
2
1
1o
1 -1
```

Our `irreps` means "transforms like a vector". `irreps` is able to provide the matrix to transform the data under a rotation

```
import torch
t = torch.tensor

# show the transformation matrix corresponding to the inversion
irreps.D_from_angles(alpha=t(0.0), beta=t(0.0), gamma=t(0.0), k=t(1))
```

```
tensor([[-1., -0., -0.],
        [-0., -1., -0.],
        [-0., -0., -1.]])
```

```
# a small rotation around the y axis
irreps.D_from_angles(alpha=t(0.1), beta=t(0.0), gamma=t(0.0), k=t(0))
```

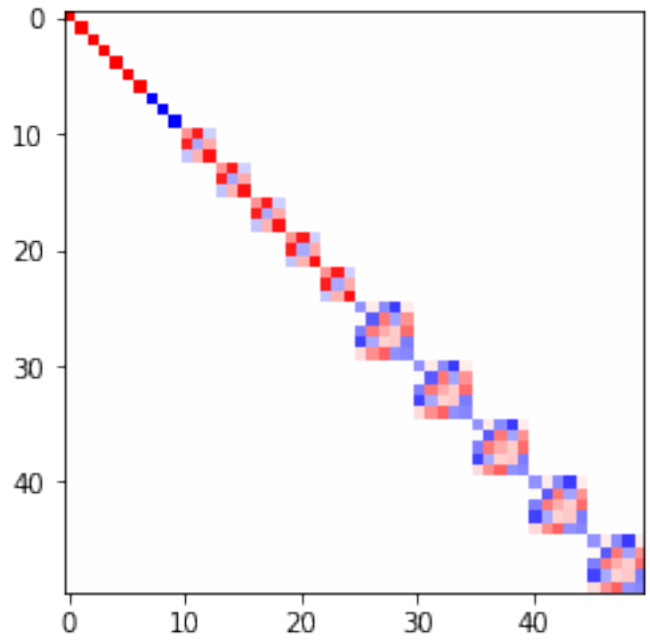
```
tensor([[ 0.9950,  0.0000,  0.0998],
        [ 0.0000,  1.0000,  0.0000],
        [-0.0998,  0.0000,  0.9950]])
```

In this example

```
irreps = Irreps("7x0e + 3x0o + 5x1o + 5x2o")
```

the `irreps` tell us how 7 scalars, 3 pseudoscalars, 5 vectors and 5 odd representation of `l=2` transforms. They all transforms independently, this can be seen by visualizing the matrix

```
from e3nn import o3
rot = -o3.rand_matrix()

D = irreps.D_from_matrix(rot)

import matplotlib.pyplot as plt
plt.imshow(D, cmap='bwr', vmin=-1, vmax=1);
```



2.2.3 Convolution

In this document we will implement an equivariant convolution with `e3nn`. We will implement this formula:

$$f_i' = \frac{1}{\sqrt{z}} \sum_{j \in \partial(i)} f_j \ \otimes(h(\|x_{ij}\|)) \ Y(x_{ij}/\|x_{ij}\|)$$

where

- f_j, f_i' are the nodes input and output
- z is the average degree of the nodes
- $\partial(i)$ is the set of neighbors of the node i
- x_{ij} is the relative vector
- h is a multi layer perceptron
- Y is the spherical harmonics
- $x \ \otimes(w) \ y$ is a tensor product of x with y parametrized by some weights w

Boilerplate imports

```
import torch
from torch_cluster import radius_graph
from torch_scatter import scatter
from e3nn import o3, nn
from e3nn.math import soft_one_hot_linspace
import matplotlib.pyplot as plt
```

Let's first define the irreps of the input and output features.

```
irreps_input = o3.Irreps("10x0e + 10x1e")
irreps_output = o3.Irreps("20x0e + 10x1e")
```

And create a random graph using random positions and edges when the relative distance is smaller than `max_radius`.

```
# create node positions
num_nodes = 100
pos = torch.randn(num_nodes, 3)  # random node positions

# create edges
max_radius = 1.8
edge_src, edge_dst = radius_graph(pos, max_radius, max_num_neighbors=num_nodes - 1)

print(edge_src.shape)

edge_vec = pos[edge_dst] - pos[edge_src]

# compute z
num_neighbors = len(edge_src) / num_nodes
num_neighbors
```

```
torch.Size([3964])
```

```
39.64
```

`edge_src` and `edge_dst` contain the indices of the nodes for each edge. And we can also create some random input features.

```
f_in = irreps_input.randn(num_nodes, -1)
```

Note that out data is generated with a normal distribution. We will take care of having all the data following the `component` normalization (see *Normalization*).

```
f_in.pow(2).mean()  # should be close to 1
```

```
tensor(0.9921)
```

Let's start with

$$Y(x_{ij}/\|x_{ij}\|)$$

```
irreps_sh = o3.Irreps.spherical_harmonics(lmax=2)
print(irreps_sh)

sh = o3.spherical_harmonics(irreps_sh, edge_vec, normalize=True, normalization='component
↪')
# normalize=True ensure that x is divided by |x| before computing the sh

sh.pow(2).mean()  # should be close to 1
```

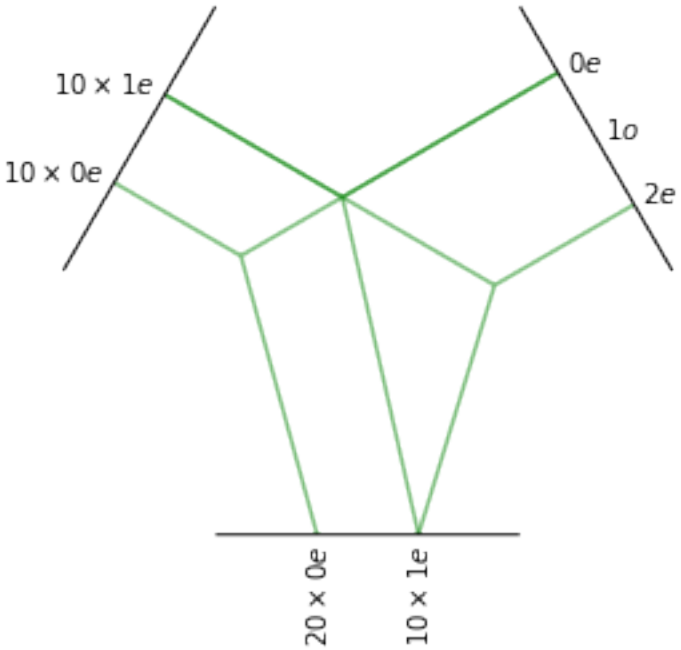
```
1x0e+1x1o+1x2e
```

```
tensor(1.)
```

Now we need to compute $\otimes(w)$ and h. Let's create the tensor product first, it will tell us how many weights it needs.

```
tp = o3.FullyConnectedTensorProduct(irreps_input, irreps_sh, irreps_output, shared_
↪weights=False)

print(f"{tp} needs {tp.weight_numel} weights")

tp.visualize();
```

```
FullyConnectedTensorProduct(10x0e+10x1e x 1x0e+1x1o+1x2e -> 20x0e+10x1e | 400 paths |␣
↪400 weights) needs 400 weights
```



in this particual choice of irreps we can see that the l=1 component of the spherical harmonics cannot be used in the tensor product. In this example it's the equivariance to inversion that prohibit the use of l=1. If we don't want the equivariance to inversion we can declare all irreps to be even (`irreps_sh = Irreps("0e + 1e + 2e")`).

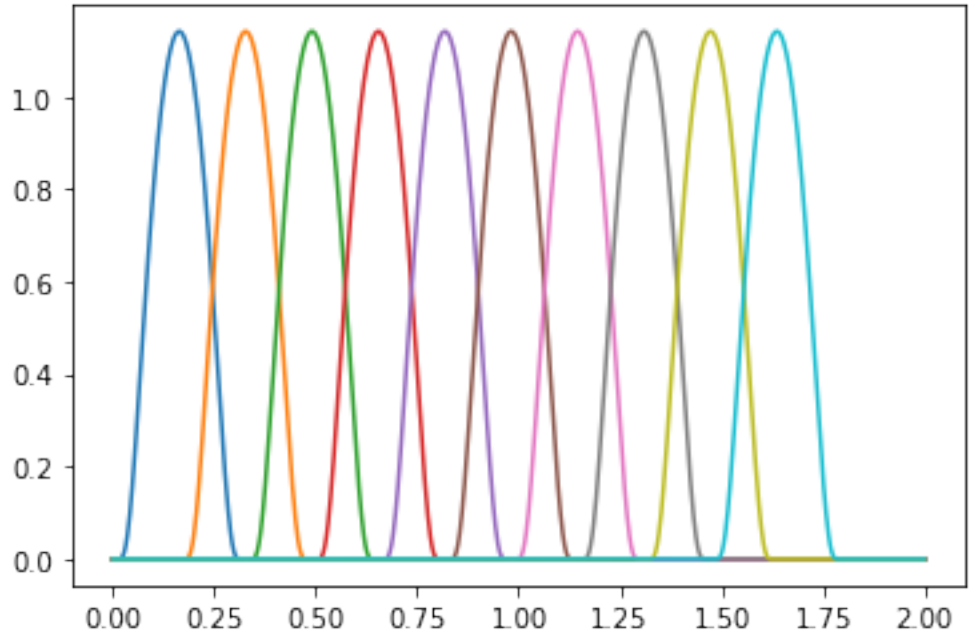
To implement h that has to map the relative distances to the weights of the tensor product we will embed the distances using a basis function and then feed this embedding to a neural network. Let's create that embedding. Here is the base functions we will use:

```
num_basis = 10

x = torch.linspace(0.0, 2.0, 1000)
y = soft_one_hot_linspace(
    x,
    start=0.0,
```

(continues on next page)

(continued from previous page)

```
    end=max_radius,
    number=num_basis,
    basis='smooth_finite',
    cutoff=True,
)

plt.plot(x, y);
```



Note that this set of functions are all smooth and are strictly zero beyond `max_radius`. This is useful to get a convolution that is smooth although the sharp cutoff at `max_radius`.

Let's use this embedding for the edge distances and normalize it properly (`component` i.e. second moment close to 1).

```
edge_length_embedding = soft_one_hot_linspace(
    edge_vec.norm(dim=1),
    start=0.0,
    end=max_radius,
    number=num_basis,
    basis='smooth_finite',
    cutoff=True,
)
edge_length_embedding = edge_length_embedding.mul(num_basis**0.5)

print(edge_length_embedding.shape)
edge_length_embedding.pow(2).mean()  # the second moment
```

```
torch.Size([3964, 10])
```

```
tensor(0.9072)
```

Now we can create a MLP and feed it

```
fc = nn.FullyConnectedNet([num_basis, 16, tp.weight_numel], torch.relu)
weight = fc(edge_length_embedding)

print(weight.shape)
print(len(edge_src), tp.weight_numel)

# For a proper notmalization, the weights also need to be mean 0
print(weight.mean(), weight.std())  # should close to 0 and 1
```

```
torch.Size([3964, 400])
3964 400
tensor(0.0162, grad_fn=<MeanBackward0>) tensor(0.9845, grad_fn=<StdBackward0>)
```

Now we can compute the term

$$f_j \otimes (h(\|x_{ij}\|))\, Y(x_{ij}/\|x_{ij}\|)$$

The idea is to compute this quantity per edges, so we will need to "lift" the input feature to the edges. For that we use `edge_src` that contains, for each edge, the index of the source node.

```
summand = tp(f_in[edge_src], sh, weight)

print(summand.shape)
print(summand.pow(2).mean())  # should be close to 1
```

```
torch.Size([3964, 50])
tensor(0.9660, grad_fn=<MeanBackward0>)
```

Only the sum over the neighbors is remaining

$$f'_i = \frac{1}{\sqrt{z}} \sum_{j \in \partial(i)} f_j \otimes (h(\|x_{ij}\|))\, Y(x_{ij}/\|x_{ij}\|)$$

```
f_out = scatter(summand, edge_dst, dim=0, dim_size=num_nodes)

f_out = f_out.div(num_neighbors**0.5)

f_out.pow(2).mean()  # should be close to 1
```

```
tensor(1.0832, grad_fn=<MeanBackward0>)
```

Now we can put everything into a function

```
def conv(f_in, pos):
    edge_src, edge_dst = radius_graph(pos, max_radius, max_num_neighbors=len(pos) - 1)
    edge_vec = pos[edge_dst] - pos[edge_src]
    sh = o3.spherical_harmonics(irreps_sh, edge_vec, normalize=True, normalization=
↪'component')
    emb = soft_one_hot_linspace(edge_vec.norm(dim=1), 0.0, max_radius, num_basis, basis=
↪'smooth_finite', cutoff=True).mul(num_basis**0.5)
    return scatter(tp(f_in[edge_src], sh, fc(emb)), edge_dst, dim=0, dim_size=num_nodes).
↪div(num_neighbors**0.5)
```

Now we can check the equivariance

```
rot = o3.rand_matrix()
D_in = irreps_input.D_from_matrix(rot)
D_out = irreps_output.D_from_matrix(rot)

# rotate before
f_before = conv(f_in @ D_in.T, pos @ rot.T)

# rotate after
f_after = conv(f_in, pos) @ D_out.T

torch.allclose(f_before, f_after, rtol=1e-4, atol=1e-4)
```

```
True
```

The tensor product dominates the execution time:

```
import time
wall = time.perf_counter()

edge_src, edge_dst = radius_graph(pos, max_radius, max_num_neighbors=len(pos) - 1)
edge_vec = pos[edge_dst] - pos[edge_src]
print(time.perf_counter() - wall); wall = time.perf_counter()

sh = o3.spherical_harmonics(irreps_sh, edge_vec, normalize=True, normalization='component
↪')
print(time.perf_counter() - wall); wall = time.perf_counter()

emb = soft_one_hot_linspace(edge_vec.norm(dim=1), 0.0, max_radius, num_basis, basis=
↪'smooth_finite', cutoff=True).mul(num_basis**0.5)
print(time.perf_counter() - wall); wall = time.perf_counter()

weight = fc(emb)
print(time.perf_counter() - wall); wall = time.perf_counter()

summand = tp(f_in[edge_src], sh, weight)
print(time.perf_counter() - wall); wall = time.perf_counter()

scatter(summand, edge_dst, dim=0, dim_size=num_nodes).div(num_neighbors**0.5)
print(time.perf_counter() - wall); wall = time.perf_counter()
```

```
0.0014898069998707797
0.0019875719999618013
0.0032761770000888646
0.0014120419998562284
0.006931719000021985
0.0006559060000199679
```

2.2.4 Normalization

We define two kind of normalizations: `component` and `norm`.

Definition

component

`component` normalization refers to tensors with each component of value around 1. More precisely, the second moment of each component is 1.

$$\langle x_i^2 \rangle = 1$$

Examples:

- `[1.0, -1.0, -1.0, 1.0]`
- `[1.0, 1.0, 1.0, 1.0]` the mean **don't** need to be zero
- `[0.0, 2.0, 0.0, 0.0]` this is still fine because $\|x\|^2 = n$

```
torch.randn(10)
```

```
tensor([ 0.1754, -0.1096, -0.3693,  0.6854, -1.2400,  1.1459,  0.6070, -0.2441,
        -0.3809,  1.2477])
```

norm

`norm` normalization refers to tensors of norm close to 1.

$$\|x\| \approx 1$$

Examples:

- `[0.5, -0.5, -0.5, 0.5]`
- `[0.5, 0.5, 0.5, 0.5]` the mean **don't** need to be zero
- `[0.0, 1.0, 0.0, 0.0]`

```
torch.randn(10) / 10**0.5
```

```
tensor([ 0.0138,  0.1622, -0.2619, -0.2757,  0.1137, -0.0124,  0.0621, -0.0362,
        -0.4857, -0.5307])
```

There is just a factor $\sqrt{n}$ between the two normalizations.

Motivation

Assuming that the weights distribution obey

$$\langle w_i \rangle = 0$$
$$\langle w_i w_j \rangle = \sigma^2 \delta_{ij}$$

It imply that the two first moments of $x \cdot w$ (and therefore mean and variance) are only function of the second moment of x

$$\begin{aligned}
\langle x \cdot w \rangle &= \sum_i \langle x_i w_i \rangle = \sum_i \langle x_i \rangle \langle w_i \rangle = 0 \\
\langle (x \cdot w)^2 \rangle &= \sum_i \sum_j \langle x_i w_i x_j w_j \rangle \\
&= \sum_i \sum_j \langle x_i x_j \rangle \langle w_i w_j \rangle \\
&= \sigma^2 \sum_i \langle x_i^2 \rangle
\end{aligned}$$

Testing

You can use `e3nn.util.test.assert_normalized` to check whether a function or module is normalized at initialization:

```
from e3nn.util.test import assert_normalized
from e3nn import o3
assert_normalized(o3.Linear("10x0e", "10x0e"))
```

Advanced

2.2.5 Point inputs with periodic boundary conditions

This example shows how to give point inputs with periodic boundary conditions (e.g. crystal data) to a Euclidean neural network built with `e3nn`. For a specific application, this code should be modified with a more tailored network design.

```
import torch
import e3nn
import ase
import ase.neighborlist
import torch_geometric
import torch_geometric.data

default_dtype = torch.float64
torch.set_default_dtype(default_dtype)
```

Example crystal structures

First, we create some crystal structures which have periodic boundary conditions.

```
# A lattice is a 3 x 3 matrix
# The first index is the lattice vector (a, b, c)
# The second index is a Cartesian index over (x, y, z)

# Polonium with Simple Cubic Lattice
po_lattice = torch.eye(3) * 3.340  # Cubic lattice with edges of length 3.34 AA
po_coords = torch.tensor([[0., 0., 0.,]])
po_types = ['Po']

# Silicon with Diamond Structure
si_lattice = torch.tensor([
    [0.      , 2.734364, 2.734364],
    [2.734364, 0.      , 2.734364],
    [2.734364, 2.734364, 0.      ]
])
si_coords = torch.tensor([
    [1.367182, 1.367182, 1.367182],
    [0.      , 0.      , 0.      ]
])
si_types = ['Si', 'Si']

po = ase.Atoms(symbols=po_types, positions=po_coords, cell=po_lattice, pbc=True)
si = ase.Atoms(symbols=si_types, positions=si_coords, cell=si_lattice, pbc=True)
```

Create and store periodic graph data

We use the `ase.neighborlist.neighbor_list` algorithm and a `radial_cutoff` distance to define which edges to include in the graph to represent interactions with neighboring atoms. Note that for a convolutional network, the number of layers determines the receptive field, i.e. how "far out" any given atom can see. So even if a we use a `radial_cutoff = 3.5`, a two layer network effectively sees `2 * 3.5 = 7` distance units (in this case Angstroms) away and a three layer network `3 * 3.5 = 10.5` distance units. We then store our data in `torch_geometric.data.Data` objects that we will batch with `torch_geometric.data.DataLoader` below.

```
radial_cutoff = 3.5  # Only include edges for neighboring atoms within a radius of 3.5␣
↪Angstroms.
type_encoding = {'Po': 0, 'Si': 1}
type_onehot = torch.eye(len(type_encoding))

dataset = []

dummy_energies = torch.randn(2, 1, 1)  # dummy energies for example

for crystal, energy in zip([po, si], dummy_energies):
    # edge_src and edge_dst are the indices of the central and neighboring atom,␣
↪respectively
    # edge_shift indicates whether the neighbors are in different images / copies of the␣
↪unit cell
    edge_src, edge_dst, edge_shift = ase.neighborlist.neighbor_list("ijS", a=crystal,␣
```

(continues on next page)

(continued from previous page)

```
↪cutoff=radial_cutoff, self_interaction=True)

    data = torch_geometric.data.Data(
        pos=torch.tensor(crystal.get_positions()),
        lattice=torch.tensor(crystal.cell.array).unsqueeze(0),  # We add a dimension for␣
↪batching
        x=type_onehot[[type_encoding[atom] for atom in crystal.symbols]],  # Using "dummy
↪" inputs of scalars because they are all C
        edge_index=torch.stack([torch.LongTensor(edge_src), torch.LongTensor(edge_dst)],␣
↪dim=0),
        edge_shift=torch.tensor(edge_shift, dtype=default_dtype),
        energy=energy  # dummy energy (assumed to be normalized "per atom")
    )

    dataset.append(data)

print(dataset)
```

```
[Data(x=[1, 2], edge_index=[2, 7], pos=[1, 3], lattice=[1, 3, 3], edge_shift=[7, 3],␣
↪energy=[1, 1]), Data(x=[2, 2], edge_index=[2, 10], pos=[2, 3], lattice=[1, 3, 3], edge_
↪shift=[10, 3], energy=[1, 1])]
```

The first `torch_geometric.data.Data` object is for simple cubic Polonium which has 7 edges: 6 for nearest neighbors and 1 as a "self" edge, `6 + 1 = 7`. The second `torch_geometric.data.Data` object is for diamond Silicon which has 10 edges: 4 nearest neighbors for each of the two atoms and 2 "self" edges, one for each atom, `4 * 2 + 1 * 2 = 10`. The lattice of each structure has a shape of `[1, 3, 3]` such that when we batch examples, the batched lattices will have shape `[batch_size, 3, 3]`.

Graph Batches

`torch_geometric.data.DataLoader` create batches of differently sized structures and produces `torch_geometric.data.Data` objects containing a batch when iterated over.

```
batch_size = 2
dataloader = torch_geometric.data.DataLoader(dataset, batch_size=batch_size)

for data in dataloader:
    print(data)
    print(data.batch)
    print(data.pos)
    print(data.x)
```

```
DataBatch(x=[3, 2], edge_index=[2, 17], pos=[3, 3], lattice=[2, 3, 3], edge_shift=[17,␣
↪3], energy=[2, 1], batch=[3], ptr=[3])
tensor([0, 1, 1])
tensor([[0.0000, 0.0000, 0.0000],
        [1.3672, 1.3672, 1.3672],
        [0.0000, 0.0000, 0.0000]])
tensor([[1., 0.],
        [0., 1.],
        [0., 1.]])
```

`data.batch` is the batch index which is tensor of shape `[batch_size]` that stores which points or "atoms" belong to which example. In this case, since we only have two examples in our batch, the batch tensor only contains the numbers `0` and `1`. The batch index is often passed to `scatter` operations to aggregate per examples values, e.g. the total energy for a single crystal structure.

For more details on batching with `torch_geometric`, please see this page.

Relative distance vectors of edges with periodic boundaries

To calculate the vectors associated with each edge for a given `torch_geometric.data.Data` object representing a single example, we use the following expression:

```
edge_src, edge_dst = data['edge_index'][0], data['edge_index'][1]
edge_vec = (data['pos'][edge_dst] - data['pos'][edge_src]
            + torch.einsum('ni,nij->nj', data['edge_shift'], data['lattice']))
```

The first line in the definition of `edge_vec` is simply how one normally computes relative distance vectors given two points. The second line adds the contribution to the relative distance vector due to crossing unit cell boundaries i.e. if atoms belong to different images of the unit cell. As we will see below, we can modify this expression to also include the `data['batch']` tensor when handling batched data.

One Approach: Adding a Preprocessing Method to the Network

While `edge_vec` can be stored in the `torch_geometric.data.Data` object, it can also be calculated by adding a preprocessing method to the Network. For this example, we create a modified version of the example network `SimpleNetwork` documented here with source code here. `SimpleNetwork` is a good starting point to check your data pipeline but should be replaced with a more tailored network for your specific application.

```
from e3nn.nn.models.v2103.gate_points_networks import SimpleNetwork
from typing import Dict, Union
import torch_scatter

class SimplePeriodicNetwork(SimpleNetwork):
    def __init__(self, **kwargs):
        """The keyword `pool_nodes` is used by SimpleNetwork to determine
        whether we sum over all atom contributions per example. In this example,
        we want use a mean operations instead, so we will override this behavior.
        """
        self.pool = False
        if kwargs['pool_nodes'] == True:
            kwargs['pool_nodes'] = False
            kwargs['num_nodes'] = 1.
            self.pool = True
        super().__init__(**kwargs)

    # Overwriting preprocess method of SimpleNetwork to adapt for periodic boundary data
    def preprocess(self, data: Union[torch_geometric.data.Data, Dict[str, torch.
↪Tensor]]) -> torch.Tensor:
        if 'batch' in data:
            batch = data['batch']
        else:
            batch = data['pos'].new_zeros(data['pos'].shape[0], dtype=torch.long)
```

(continues on next page)

(continued from previous page)

```
        edge_src = data['edge_index'][0]  # Edge source
        edge_dst = data['edge_index'][1]  # Edge destination

        # We need to compute this in the computation graph to backprop to positions
        # We are computing the relative distances + unit cell shifts from periodic␣
↪boundaries
        edge_batch = batch[edge_src]
        edge_vec = (data['pos'][edge_dst]
                    - data['pos'][edge_src]
                    + torch.einsum('ni,nij->nj', data['edge_shift'], data['lattice
↪'][edge_batch]))

        return batch, data['x'], edge_src, edge_dst, edge_vec

    def forward(self, data: Union[torch_geometric.data.Data, Dict[str, torch.Tensor]]) ->
↪ torch.Tensor:
        # if pool_nodes was set to True, use scatter_mean to aggregate
        output = super().forward(data)
        if self.pool == True:
            return torch_scatter.scatter_mean(output, data.batch, dim=0)  # Take mean␣
↪over atoms per example
        else:
            return output
```

We define and run the network.

```
net = SimplePeriodicNetwork(
    irreps_in="2x0e",  # One hot scalars (L=0 and even parity) on each atom to represent␣
↪atom type
    irreps_out="1x0e",  # Single scalar (L=0 and even parity) to output (for example)␣
↪energy
    max_radius=radial_cutoff, # Cutoff radius for convolution
    num_neighbors=10.0,  # scaling factor based on the typical number of neighbors
    pool_nodes=True,  # We pool nodes to predict total energy
)
```

When we apply the network to our data, we get one scalar per example.

```
for data in dataloader:
    print(net(data).shape)  # One scalar per example
```

```
torch.Size([2, 1])
```

2.2.6 Transformer

> The Transformer is a deep learning model introduced in 2017 that utilizes the mechanism of attention. It is used primarily in the field of natural language processing (NLP), but recent research has also developed its application in other tasks like video understanding. Wikipedia

In this document we will see how to implement an equivariant attention mechanism with `e3nn`. We will implement the formula (1) of SE(3)-Transformers. The output features f' are computed by

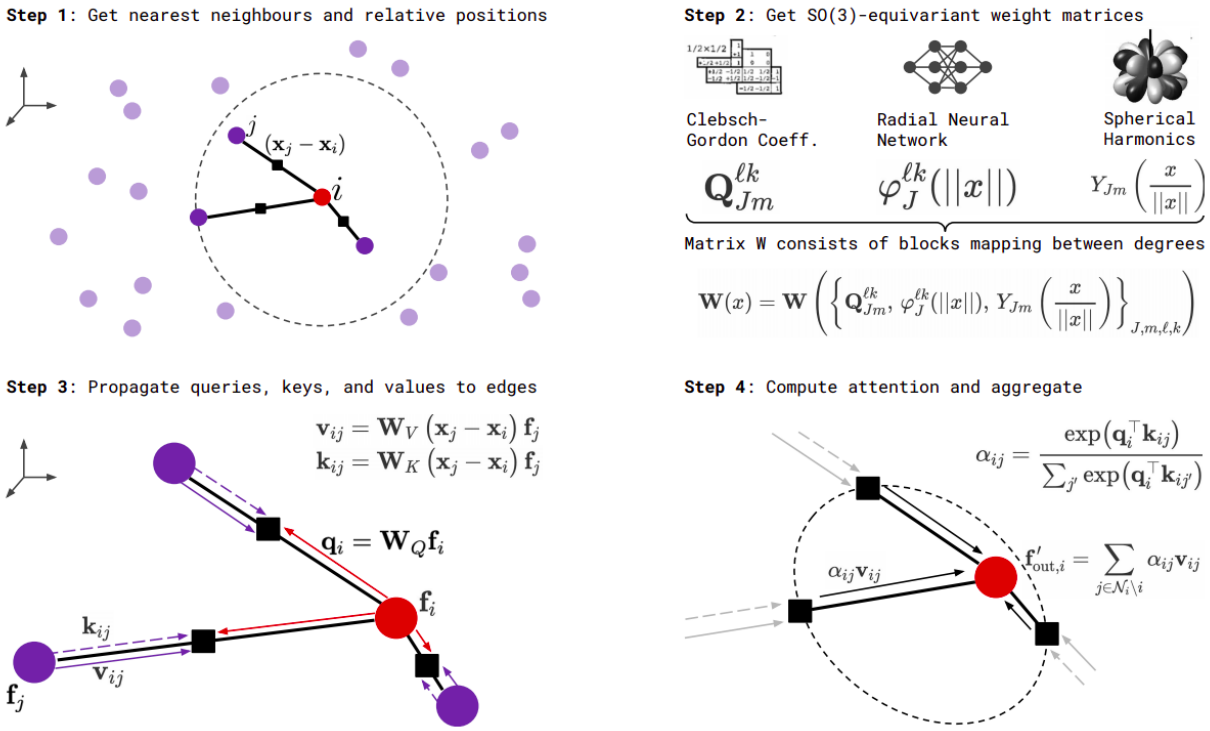
$$f'_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

$$\alpha_{ij} = \frac{\exp(q_i^T k_j)}{\sum_{j'=1}^{n} \exp(q_i^T k_{j'})}$$

where q, k, v are respectively called the queries, keys and values. They are functions of the input features f.

$$q = h_Q(f)$$
$$k = h_K(f)$$
$$v = h_V(f)$$

all these formula are well illustrated by the figure (2) of the same article.



First we need to define the irreps of the inputs, the queries, the keys and the outputs. Note that outputs and values share the same irreps.

```
# Just define arbitrary irreps
irreps_input = o3.Irreps("10x0e + 5x1o + 2x2e")
irreps_query = o3.Irreps("11x0e + 4x1o")
irreps_key = o3.Irreps("12x0e + 3x1o")
irreps_output = o3.Irreps("14x0e + 6x1o")  # also irreps of the values
```

Lets create a random graph on which we can apply the attention mechanism:

```
num_nodes = 20

pos = torch.randn(num_nodes, 3)
f = irreps_input.randn(num_nodes, -1)

# create graph
max_radius = 1.3
edge_src, edge_dst = radius_graph(pos, max_radius)
edge_vec = pos[edge_src] - pos[edge_dst]
edge_length = edge_vec.norm(dim=1)
```

The queries q_i are a linear combination of the input features f_i.

```
h_q = o3.Linear(irreps_input, irreps_query)
```

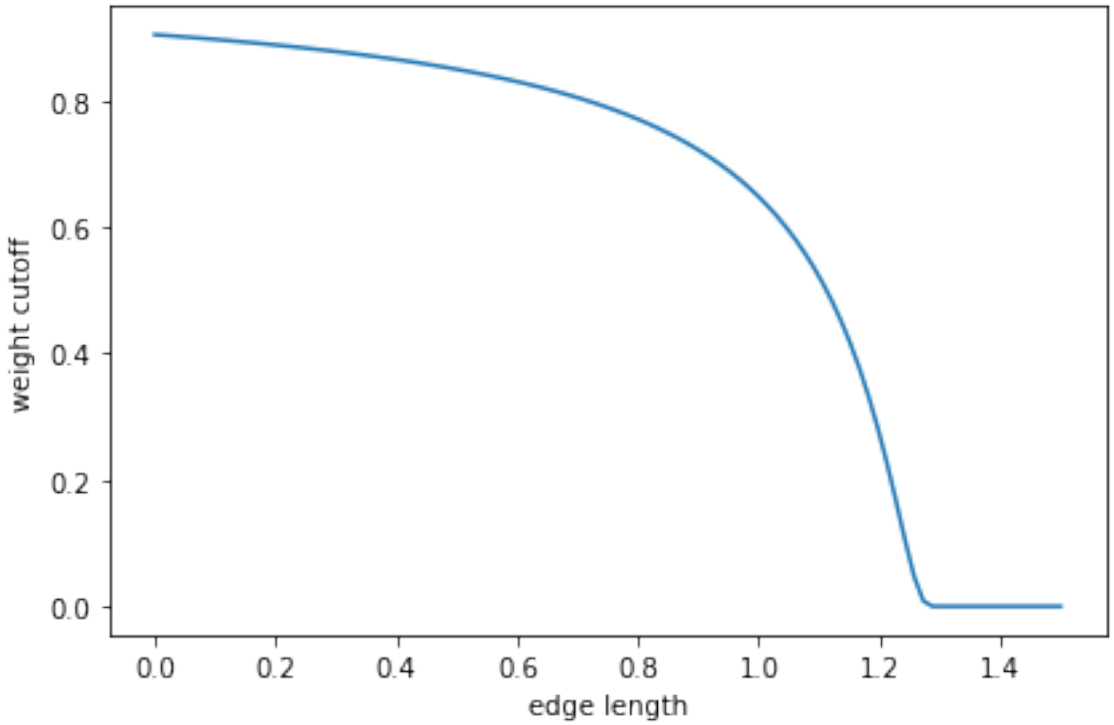
In order to generate weights that depends on the radii, we project the edges length on a basis:

```
number_of_basis = 10
edge_length_embedded = soft_one_hot_linspace(
    edge_length,
    start=0.0,
    end=max_radius,
    number=number_of_basis,
    basis='smooth_finite',
    cutoff=True  # goes (smoothly) to zero at `start` and `end`
)
edge_length_embedded = edge_length_embedded.mul(number_of_basis**0.5)
```

We will also need a number between 0 and 1 that indicates smoothly if the length of the edge is smaller than `max_radius`.

```
edge_weight_cutoff = soft_unit_step(10 * (1 - edge_length / max_radius))
```

Here is a figure of the function used:



To create the values and the keys we have to use the relative position of the edges. We will use the spherical harmonics to have a richer describtor of the relative positions:

```
irreps_sh = o3.Irreps.spherical_harmonics(3)
edge_sh = o3.spherical_harmonics(irreps_sh, edge_vec, True, normalization='component')
```

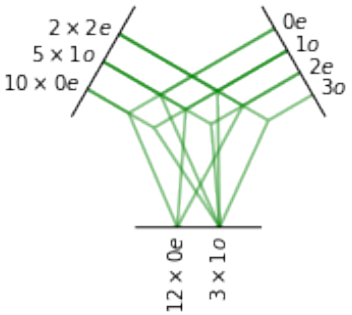
We will make a tensor prodcut between the input and the spherical harmonics to create the values and keys. Because we want the weights of these tensor products to depend on the edge length we will generate the weights using multi layer perceptrons.

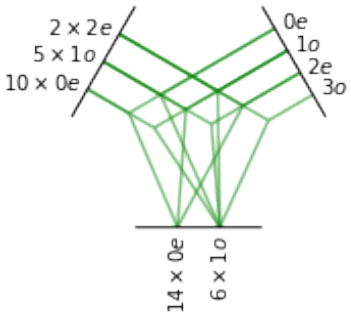
```
tp_k = o3.FullyConnectedTensorProduct(irreps_input, irreps_sh, irreps_key, shared_
↪weights=False)
fc_k = nn.FullyConnectedNet([number_of_basis, 16, tp_k.weight_numel], act=torch.nn.
↪functional.silu)

tp_v = o3.FullyConnectedTensorProduct(irreps_input, irreps_sh, irreps_output, shared_
↪weights=False)
fc_v = nn.FullyConnectedNet([number_of_basis, 16, tp_v.weight_numel], act=torch.nn.
↪functional.silu)
```

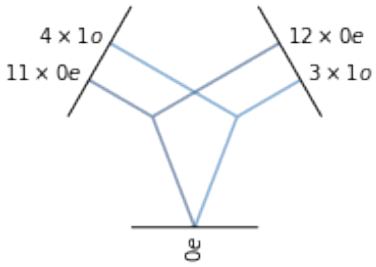
For the correpondance with the formula, `tp_v, fc_v` represent h_K and `tp_v, fc_v` represent h_V. Then we need a way to compute the dot product between the queries and the keys:

```
dot = o3.FullyConnectedTensorProduct(irreps_query, irreps_key, "0e")
```

The operations `tp_k`, `tp_v` and `dot` can be visualized as follow:







Finally we can just use all the modules we created to compute the attention mechanism:

```
# compute the queries (per node), keys (per edge) and values (per edge)
q = h_q(f)
k = tp_k(f[edge_src], edge_sh, fc_k(edge_length_embedded))
v = tp_v(f[edge_src], edge_sh, fc_v(edge_length_embedded))

# compute the softmax (per edge)
exp = edge_weight_cutoff[:, None] * dot(q[edge_dst], k).exp()  # compute the numerator
z = scatter(exp, edge_dst, dim=0, dim_size=len(f))  # compute the denominator (per nodes)
z[z == 0] = 1  # to avoid 0/0 when all the neighbors are exactly at the cutoff
alpha = exp / z[edge_dst]

# compute the outputs (per node)
f_out = scatter(alpha.relu().sqrt() * v, edge_dst, dim=0, dim_size=len(f))
```

Note that this implementation has small differences with the article.

- Special care was taken to make the whole operation smooth when we move the points (deleting/creating new edges). It was done via `edge_weight_cutoff`, `edge_length_embedded` and the property $f(0) = 0$ for the radial neural network.
- The output is weighted with $\sqrt{\alpha_{ij}}$ instead of α_{ij} to ensure a proper normalization.

Both are checked below, starting by the normalization.

```
f_out.mean().item(), f_out.std().item()
```

```
(-0.02495686337351799, 1.0274895429611206)
```

Let's put eveything into a function to check the smoothness and the equivariance.

```
def transformer(f, pos):
    edge_src, edge_dst = radius_graph(pos, max_radius)
    edge_vec = pos[edge_src] - pos[edge_dst]
    edge_length = edge_vec.norm(dim=1)

    edge_length_embedded = soft_one_hot_linspace(
        edge_length,
        start=0.0,
        end=max_radius,
        number=number_of_basis,
        basis='smooth_finite',
```

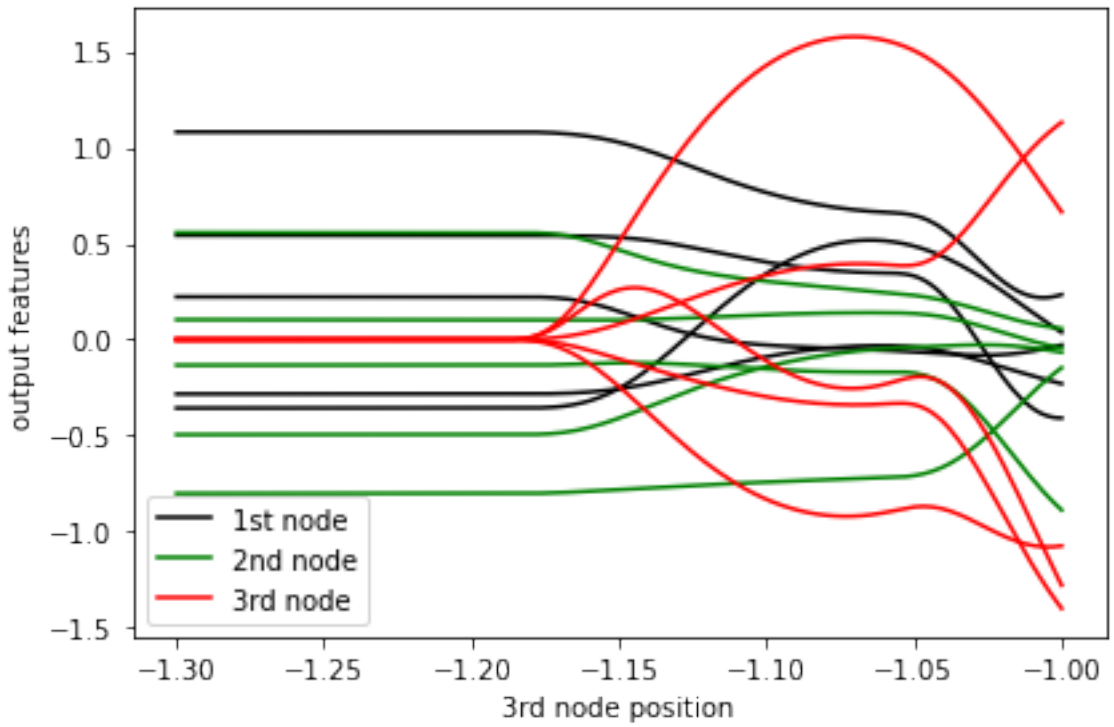
(continues on next page)

(continued from previous page)

```
        cutoff=True
    )
    edge_length_embedded = edge_length_embedded.mul(number_of_basis**0.5)
    edge_weight_cutoff = soft_unit_step(10 * (1 - edge_length / max_radius))

    edge_sh = o3.spherical_harmonics(irreps_sh, edge_vec, True, normalization='component
↪')

    q = h_q(f)
    k = tp_k(f[edge_src], edge_sh, fc_k(edge_length_embedded))
    v = tp_v(f[edge_src], edge_sh, fc_v(edge_length_embedded))

    exp = edge_weight_cutoff[:, None] * dot(q[edge_dst], k).exp()
    z = scatter(exp, edge_dst, dim=0, dim_size=len(f))
    z[z == 0] = 1
    alpha = exp / z[edge_dst]

    return scatter(alpha.relu().sqrt() * v, edge_dst, dim=0, dim_size=len(f))
```

Here is a smoothness check: tow nodes are placed at a distance 1 (`max_radius` > 1) so they see each other. A third node coming from far away moves slowly towards them.

```
f = irreps_input.randn(3, -1)

xs = torch.linspace(-1.3, -1.0, 200)
outputs = []

for x in xs:
    pos = torch.tensor([
        [0.0, 0.5, 0.0],       # this node always sees...
        [0.0, -0.5, 0.0],      # ...this node
        [x.item(), 0.0, 0.0],  # this node moves slowly
    ])

    with torch.no_grad():
        outputs.append(transformer(f, pos))

outputs = torch.stack(outputs)
plt.plot(xs, outputs[:, 0, [0, 1, 14, 15, 16]], 'k')  # plots 2 scalars and 1 vector
plt.plot(xs, outputs[:, 1, [0, 1, 14, 15, 16]], 'g')
plt.plot(xs, outputs[:, 2, [0, 1, 14, 15, 16]], 'r')
```



Finally we can check the equivariance:

```
f = irreps_input.randn(10, -1)
pos = torch.randn(10, 3)

rot = o3.rand_matrix()
D_in = irreps_input.D_from_matrix(rot)
D_out = irreps_output.D_from_matrix(rot)

f_before = transformer(f @ D_in.T, pos @ rot.T)
f_after = transformer(f, pos) @ D_out.T

torch.allclose(f_before, f_after, atol=1e-3, rtol=1e-3)
```

```
True
```

Extra sanity check of the backward pass:

```
for x in [0.0, 1e-6,  max_radius / 2, max_radius - 1e-6, max_radius, max_radius + 1e-6,␣
↪2 * max_radius]:
    f = irreps_input.randn(2, -1, requires_grad=True)
    pos = torch.tensor([
        [0.0, 0.0, 0.0],
        [x, 0.0, 0.0],
    ], requires_grad=True)
    transformer(f, pos).sum().backward()

    assert f.grad is None or torch.isfinite(f.grad).all()
    assert torch.isfinite(pos.grad).all()
```

2.2.7 Equivariance Testing

In *e3nn.util.test*, the library provides some tools for confirming that functions are equivariant. The main tool is *equivariance_error*, which computes the largest absolute change in output between the function applied to transformed arguments and the transform applied to the function:

```
import e3nn.o3
from e3nn.util.test import equivariance_error

tp = e3nn.o3.FullyConnectedTensorProduct("2x0e + 3x1o", "2x0e + 3x1o", "2x1o")

equivariance_error(
    tp,
    args_in=[tp.irreps_in1.randn(1, -1), tp.irreps_in2.randn(1, -1)],
    irreps_in=[tp.irreps_in1, tp.irreps_in2],
    irreps_out=[tp.irreps_out]
)
```

```
{(0, False): tensor([8.0559e-08]), (1, False): tensor([7.0503e-08])}
```

The keys in the output indicate the type of random transformation (`(parity, did_translation)`) and the values are the maximum componentwise error. For convenience, the wrapper function *assert_equivariant* is provided:

```
from e3nn.util.test import assert_equivariant

assert_equivariant(tp)
```

```
{(0, False): tensor([1.8066e-07]), (1, False): tensor([2.4127e-07])}
```

For typical e3nn operations *assert_equivariant* can optionally infer the input and output *e3nn.o3.Irreps*, generate random inputs when no inputs are provided, and check the error against a threshold appropriate to the current `torch.get_default_dtype()`.

In addition to *e3nn.o3.Irreps*-like objects, `irreps_in` can also contain two special values:

- `'cartesian_points'`: `(N, 3)` tensors containing XYZ points in real space that are equivariant under rotations *and* translations
- `None`: any input or output that is invariant and should be left alone

These can be used to test models that operate on full graphs that include position information:

```
import torch
from torch_geometric.data import Data
from e3nn.nn.models.v2103.gate_points_networks import SimpleNetwork
from e3nn.util.test import assert_equivariant

# kwargs = ...
f = SimpleNetwork(**kwargs)

def wrapper(pos, x):
    data = dict(pos=pos, x=x)
    return f(data)

assert_equivariant(
```

(continues on next page)

(continued from previous page)

```
    wrapper,
    irreps_in=['cartesian_points', f.irreps_in],
    irreps_out=[f.irreps_out],
)
```

```
{(0, False): tensor([3.1322e-07]),
 (0, True): tensor([2.2251e-07]),
 (1, False): tensor([2.6103e-07]),
 (1, True): tensor([1.0118e-06])}
```

To test equivariance on a specific graph, `args_in` can be used:

```
assert_equivariant(
    wrapper,
    irreps_in=['cartesian_points', f.irreps_in],
    args_in=[my_pos, my_x],
    irreps_out=[f.irreps_out],
)
```

```
{(0, False): tensor([9.7792e-07]),
 (0, True): tensor([4.9472e-06]),
 (1, False): tensor([1.1027e-06]),
 (1, True): tensor([5.6028e-06])}
```

Logging

`assert_equivariant` also logs the equivariance error to the `e3nn.util.test` logger with level `INFO` regardless of whether the test fails. When running in pytest, these logs can be seen using the "Live Logs" feature:

```
pytest tests/ --log-cli-level info
```

2.2.8 TorchScript JIT Support

PyTorch provides two ways to compile code into TorchScript: tracing and scripting. Tracing follows the tensor operations on an example input, allowing complex Python control flow if that control flow does not depend on the data itself. Scripting compiles a subset of Python directly into TorchScript, allowing data-dependent control flow but only limited Python features.

This is a problem for e3nn, where many modules — such as *e3nn.o3.TensorProduct* — use significant Python control flow based on `e3nn.o3.Irreps` as well as features like inheritance that are incompatible with scripting. Other modules like `e3nn.nn.Gate`, however, contain important but simple data-dependent control flow. Thus `e3nn.nn.Gate` needs to be scripted, even though it contains a *e3nn.o3.TensorProduct* that has to be traced.

To hide this complexity from the user and prevent difficult-to-understand errors, `e3nn` implements a wrapper for `torch.jit` — e3nn.util.jit — that recursively and automatically compiles submodules according to directions they provide. Using the `@compile_mode` decorator, modules can indicate whether they should be scripted, traced, or left alone.

Simple Example: Scripting

We define a simple module that includes data-dependent control flow:

```
import torch
from e3nn.o3 import Norm, Irreps

class MyModule(torch.nn.Module):
    def __init__(self, irreps_in):
        super().__init__()
        self.norm = Norm(irreps_in)

    def forward(self, x):
        norm = self.norm(x)
        if torch.any(norm > 7.):
            return norm
        else:
            return norm * 0.5

irreps = Irreps("2x0e + 1x1o")
mod = MyModule(irreps)
```

To compile it to TorchScript, we can try to use `torch.jit.script`:

```
try:
    mod_script = torch.jit.script(mod)
except:
    print("Compilation failed!")
```

This fails because `Norm` is a subclass of *e3nn.o3.TensorProduct* and TorchScript doesn't support inheritance. If we use `e3nn.util.jit.script`, on the other hand, it works:

```
from e3nn.util.jit import script, trace
mod_script = script(mod)
```

Internally, `e3nn.util.jit.script` recurses through the submodules of `mod`, compiling each in accordance with its `@e3nn.util.jit.compile_mode` decorator if it has one. In particular, `Norm` and other *e3nn.o3.TensorProduct* s are marked with `@compile_mode('trace')`, so `e3nn.util.jit` constructs an example input for `mod.norm`, traces it, and replaces it with the traced TorchScript module. Then when the parent module `mod` is compiled inside `e3nn.util.jit.script` with `torch.jit.script`, the submodule `mod.norm` has already been compiled and is integrated without issue.

As expected, the scripted module and the original give the same results:

```
x = irreps.randn(2, -1)
assert torch.allclose(mod(x), mod_script(x))
```

Mixing Tracing and Scripting

Say we define:

```
from e3nn.util.jit import compile_mode

@compile_mode('script')
class MyModule(torch.nn.Module):
    def __init__(self, irreps_in):
        super().__init__()
        self.norm = Norm(irreps_in)

    def forward(self, x):
        norm = self.norm(x)
        for row in norm:
            if torch.any(row > 0.1):
                return row
        return norm

class AnotherModule(torch.nn.Module):
    def __init__(self, irreps_in):
        super().__init__()
        self.mymod = MyModule(irreps_in)

    def forward(self, x):
        return self.mymod(x) + 3.
```

And trace an instance of `AnotherModule` using *e3nn.util.jit.trace*:

```
mod2 = AnotherModule(irreps)
example_inputs = (irreps.randn(3, -1),)
mod2_traced = trace(
    mod2,
    example_inputs
)
```

Note that we marked `MyModule` with `@compile_mode('script')` because it contains control flow, and that the control flow is preserved even when called from the traced `AnotherModule`:

```
print(mod2_traced(torch.zeros(2, irreps.dim)))
print(mod2_traced(irreps.randn(3, -1)))
```

```
tensor([[3., 3., 3.],
        [3., 3., 3.]])
tensor([4.3516, 4.6656, 4.5736])
```

We can confirm that the submodule `mymod` was compiled as a script, but that `mod2` was traced:

```
print(type(mod2_traced))
print(type(mod2_traced.mymod))
```

```
<class 'torch.jit._trace.TopLevelTracedModule'>
<class 'torch.jit._script.RecursiveScriptModule'>
```

Customizing Tracing Inputs

Submodules can also be compiled automatically using tracing if they are marked with `@compile_mode('trace')`. When submodules are compiled by tracing it must be possible to generate plausible input examples on the fly.

These example inputs can be generated automatically based on the `irreps_in` of the module (the specifics are the same as for `assert_equivariant`). If this is not possible or would yield incorrect results, a module can define a `_make_tracing_inputs` method that generates example inputs of correct shape and type.

```
@compile_mode('trace')
class TracingModule(torch.nn.Module):
    def forward(self, x: torch.Tensor, indexes: torch.LongTensor):
        return x[indexes].sum()

    # Because this module has no `irreps_in`, and because
    # `irreps_in` can't describe indexes, since it's a LongTensor,
    # we impliment _make_tracing_inputs
    def _make_tracing_inputs(self, n: int):
        import random
        # The compiler asks for n example inputs ---
        # this is only a suggestion, the only requirement
        # is that at least one be returned.
        return [
            {
                'forward': (
                    torch.randn(5, random.randint(1, 3)),
                    torch.arange(3)
                )
            }
            for _ in range(n)
        ]
```

To recursively compile this module and its submodules in accordance with their `@compile_mode``s, we can use ``e3nn.util.jit.compile` directly. This can be useful if the module you are compiling is annotated with `@compile_mode` and you don't want to override that annotation by using `trace` or `script`:

```
from e3nn.util.jit import compile
mod3 = TracingModule()
mod3_traced = compile(mod3)
print(type(mod3_traced))
```

```
<class 'torch.jit._trace.TopLevelTracedModule'>
```

Deciding between `'script'` and `'trace'`

The easiest way to decide on a compile mode for your module is to try both. Tracing will usually generate warnings if it encounters dynamic control flow that it cannot fully capture, and scripting will raise compiler errors for features it does not support.

In general, any module that uses inheritance or control flow based on `e3nn.o3.Irreps` in `forward()` will have to be traced.

Testing

A helper function is provided to unit test that auto-JITable modules (those annotated with `@compile_mode`) can be compiled:

```
from e3nn.util.test import assert_auto_jitable
assert_auto_jitable(mod2)
```

```
AnotherModule(
  original_name=AnotherModule
  (mymod): RecursiveScriptModule(
    original_name=MyModule
    (norm): Norm(
      original_name=Norm
      (tp): RecursiveScriptModule(
        original_name=TensorProduct
        (_compiled_main_left_right): RecursiveScriptModule(original_name=GraphModule)
        (_compiled_main_right): RecursiveScriptModule(original_name=tp_forward)
      )
    )
  )
)
```

By default, `assert_auto_jitable` will test traced modules to confirm that they reject input shapes that are likely incorrect. Specifically, it changes `x.shape[-1]` on the assumption that the final dimension is a network architecture constant. If this heuristic is wrong for your module (like it is for `TracedModule` above), it can be disabled:

```
assert_auto_jitable(mod3, strict_shapes=False)
```

```
TracingModule(original_name=TracingModule)
```

Compile mode "unsupported"

Sometimes you may write modules that use features unsupported by TorchScript regardless of whether you trace or script. To avoid cryptic errors from TorchScript if someone tries to compile a model containing such a module, the module can be marked with `@compile_mode("unsupported")`:

```
@compile_mode('unsupported')
class ChildMod(torch.nn.Module):
    pass

class Supermod(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.child = ChildMod()

mod = Supermod()
script(mod)
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Input In [13], in <cell line: 11>()
```

(continues on next page)

(continued from previous page)

```
      8         self.child = ChildMod()
     10 mod = Supermod()
---> 11 script(mod)

File ~/checkouts/readthedocs.org/user_builds/e3nn/envs/latest/lib/python3.8/site-
↪packages/e3nn/util/jit.py:266, in script(mod, in_place)
    263 setattr(mod, _E3NN_COMPILE_MODE, "script")
    265 # Compile
--> 266 out = compile(mod, in_place=in_place)
    268 # Restore old values, if we had them
    269 if old_mode is not None:

File ~/checkouts/readthedocs.org/user_builds/e3nn/envs/latest/lib/python3.8/site-
↪packages/e3nn/util/jit.py:101, in compile(mod, n_trace_checks, script_options, trace_
↪options, in_place)
     95 # == recurse to children ==
     96 # This allows us to trace compile submodules of modules we are going to script
     97 for submod_name, submod in mod.named_children():
     98     setattr(
     99         mod,
    100         submod_name,
--> 101         compile(
    102             submod,
    103             n_trace_checks=n_trace_checks,
    104             script_options=script_options,
    105             trace_options=trace_options,
    106             in_place=True,  # since we deepcopied the module above, we can do␣
↪inplace
    107         ),
    108     )
    109 # == Compile this module now ==
    110 if mode == "script":

File ~/checkouts/readthedocs.org/user_builds/e3nn/envs/latest/lib/python3.8/site-
↪packages/e3nn/util/jit.py:89, in compile(mod, n_trace_checks, script_options, trace_
↪options, in_place)
     87 mode = get_compile_mode(mod)
     88 if mode == "unsupported":
---> 89     raise NotImplementedError(f"{type(mod).__name__} does not support␣
↪TorchScript compilation")
     91 if not in_place:
     92     mod = copy.deepcopy(mod)

NotImplementedError: ChildMod does not support TorchScript compilation
```

2.2.9 Change of Basis

In the release `0.2.2`, the euler angle convention changed from the standard ZYZ to YXY. This amounts to a change of basis for e3nn.

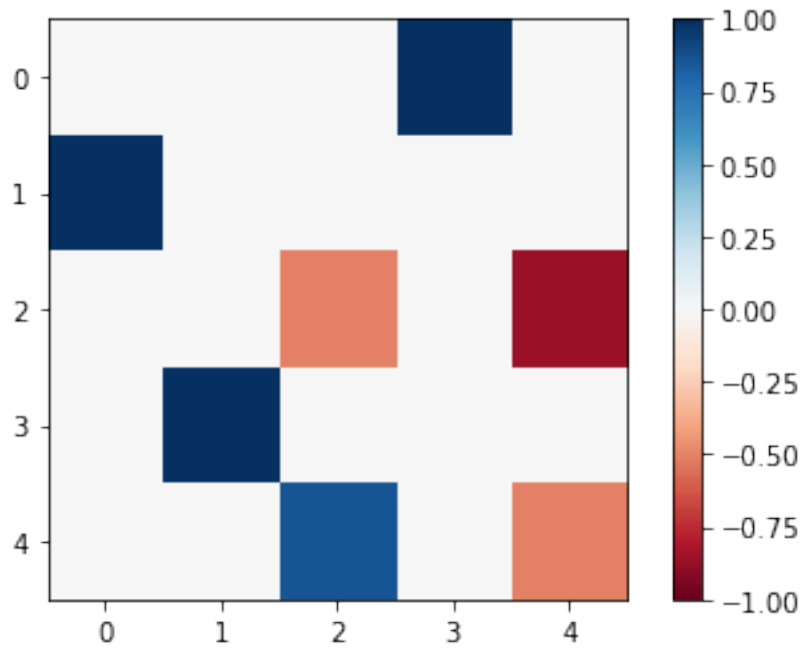
This change of basis means that the real spherical harmonics have been rotated from the "standard" real spherical harmonics (see this table of standard real spherical harmonics from Wikipedia). If your network has outputs of L=0 only, this has no effect. If your network has outputs of L=1, the components are now ordered x,y,z as opposed to the "standard" y,z,x.

If, however, your network has outputs of L=2 or greater, things are a little trickier. In this case there is no simple permutation of spherical harmonic indices that will get you back to the standard real spherical harmonics.

In this case you have two options (1) apply the change of basis to your inputs or (2) apply the change of basis to your outputs.

1. If the only inputs you have are scalars and positions, you can just permute the indices of your coordinates. You just need to permute from `y,z,x` to `x,y,z`. If you choose this method, be careful. You must keep the permuted coordinates for all subsequent analysis calculations.
2. If you want to apply the change of basis more generally, for higher L, you can grab the appropriate rotation matrices, like this example for L=2:

```
import torch
from e3nn import o3
import matplotlib.pyplot as plt

change_of_coord = torch.tensor([
    # this specifies the change of basis yzx -> xyz
    [0., 0., 1.],
    [1., 0., 0.],
    [0., 1., 0.]
])

D = o3.Irrep(2, 1).D_from_matrix(change_of_coord)

plt.imshow(D, cmap="RdBu", vmin=-1, vmax=1)
plt.colorbar();
```



Of course, you can apply the rotation method to either the inputs or the outputs – you will get the same result.

2.3 Examples

The two examples are models made to classify the toy dataset *tetris*.

2.3.1 Tetris Polynomial Example

In this example we create an *equivariant polynomial* to classify tetris.

We use the following feature of e3nn:

- *e3nn.o3.Irreps*
- *o3.spherical_harmonics*
- *e3nn.o3.FullyConnectedTensorProduct*

And the following features of pytorch_geometric

- radius_graph
- scatter

the model

```
    return data, labels


class InvariantPolynomial(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.irreps_sh = o3.Irreps.spherical_harmonics(3)
        irreps_mid = o3.Irreps("64x0e + 24x1e + 24x1o + 16x2e + 16x2o")
        irreps_out = o3.Irreps("0o + 6x0e")

        self.tp1 = FullyConnectedTensorProduct(
            irreps_in1=self.irreps_sh,
            irreps_in2=self.irreps_sh,
            irreps_out=irreps_mid,
        )
        self.tp2 = FullyConnectedTensorProduct(
            irreps_in1=irreps_mid,
            irreps_in2=self.irreps_sh,
            irreps_out=irreps_out,
        )
        self.irreps_out = self.tp2.irreps_out

    def forward(self, data) -> torch.Tensor:
        num_neighbors = 2  # typical number of neighbors
        num_nodes = 4  # typical number of nodes

        edge_src, edge_dst = radius_graph(x=data.pos, r=1.1, batch=data.batch)  #␣
↪tensors of indices representing the graph
        edge_vec = data.pos[edge_src] - data.pos[edge_dst]
        edge_sh = o3.spherical_harmonics(
            l=self.irreps_sh,
            x=edge_vec,
            normalize=False,  # here we don't normalize otherwise it would not be a␣
↪polynomial
            normalization="component",
        )

        # For each node, the initial features are the sum of the spherical harmonics of␣
↪the neighbors
        node_features = scatter(edge_sh, edge_dst, dim=0).div(num_neighbors**0.5)

        # For each edge, tensor product the features on the source node with the␣
↪spherical harmonics
        edge_features = self.tp1(node_features[edge_src], edge_sh)
        node_features = scatter(edge_features, edge_dst, dim=0).div(num_neighbors**0.5)

        edge_features = self.tp2(node_features[edge_src], edge_sh)
        node_features = scatter(edge_features, edge_dst, dim=0).div(num_neighbors**0.5)

        # For each graph, all the node's features are summed
        return scatter(node_features, data.batch, dim=0).div(num_nodes**0.5)
```

(continues on next page)

(continued from previous page)

training

```
f = InvariantPolynomial()

optim = torch.optim.Adam(f.parameters(), lr=1e-2)

# == Train ==
for step in range(200):
    pred = f(data)
    loss = (pred - labels).pow(2).sum()

    optim.zero_grad()
    loss.backward()
    optim.step()

    if step % 10 == 0:
        accuracy = pred.round().eq(labels).all(dim=1).double().mean(dim=0).item()
        print(f"epoch {step:5d} | loss {loss:<10.1f} | {100 * accuracy:5.1f}% accuracy")
```

Full code here

2.3.2 Tetris Gate Example

Build on top of *Tetris Polynomial Example*, the following is added:

- *soft_one_hot_linspace*
- *e3nn.nn.Gate*

code

```
"""Classify tetris using gate activation function

Implement a equivariant model using gates to fit the tetris dataset
Exact equivariance to :math:`E(3)`

>>> test()
"""
import logging

import torch
from torch_cluster import radius_graph
from torch_geometric.data import Data, DataLoader
from torch_scatter import scatter

from e3nn import o3
from e3nn.nn import FullyConnectedNet, Gate
```

(continues on next page)

(continued from previous page)

```
from e3nn.o3 import FullyConnectedTensorProduct
from e3nn.math import soft_one_hot_linspace
from e3nn.util.test import assert_equivariant


def tetris():
    pos = [
        [(0, 0, 0), (0, 0, 1), (1, 0, 0), (1, 1, 0)],  # chiral_shape_1
        [(0, 0, 0), (0, 0, 1), (1, 0, 0), (1, -1, 0)],  # chiral_shape_2
        [(0, 0, 0), (1, 0, 0), (0, 1, 0), (1, 1, 0)],  # square
        [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3)],  # line
        [(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0)],  # corner
        [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0)],  # L
        [(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1)],  # T
        [(0, 0, 0), (1, 0, 0), (1, 1, 0), (2, 1, 0)],  # zigzag
    ]
    pos = torch.tensor(pos, dtype=torch.get_default_dtype())

    # Since chiral shapes are the mirror of one another we need an *odd* scalar to␣
↪distinguish them
    labels = torch.tensor(
        [
            [+1, 0, 0, 0, 0, 0, 0],  # chiral_shape_1
            [-1, 0, 0, 0, 0, 0, 0],  # chiral_shape_2
            [0, 1, 0, 0, 0, 0, 0],  # square
            [0, 0, 1, 0, 0, 0, 0],  # line
            [0, 0, 0, 1, 0, 0, 0],  # corner
            [0, 0, 0, 0, 1, 0, 0],  # L
            [0, 0, 0, 0, 0, 1, 0],  # T
            [0, 0, 0, 0, 0, 0, 1],  # zigzag
        ],
        dtype=torch.get_default_dtype(),
    )

    # apply random rotation
    pos = torch.einsum("zij,zaj->zai", o3.rand_matrix(len(pos)), pos)

    # put in torch_geometric format
    dataset = [Data(pos=pos) for pos in pos]
    data = next(iter(DataLoader(dataset, batch_size=len(dataset))))

    return data, labels


def mean_std(name, x):
    print(f"{name} \t{x.mean():.1f} ± ({x.var(0).mean().sqrt():.1f}|{x.std():.1f})")


class Convolution(torch.nn.Module):
    def __init__(self, irreps_in, irreps_sh, irreps_out, num_neighbors) -> None:
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.num_neighbors = num_neighbors

        tp = FullyConnectedTensorProduct(
            irreps_in1=irreps_in,
            irreps_in2=irreps_sh,
            irreps_out=irreps_out,
            internal_weights=False,
            shared_weights=False,
        )
        self.fc = FullyConnectedNet([3, 256, tp.weight_numel], torch.relu)
        self.tp = tp
        self.irreps_out = self.tp.irreps_out

    def forward(self, node_features, edge_src, edge_dst, edge_attr, edge_scalars) ->␣
↪torch.Tensor:
        weight = self.fc(edge_scalars)
        edge_features = self.tp(node_features[edge_src], edge_attr, weight)
        node_features = scatter(edge_features, edge_dst, dim=0).div(self.num_
↪neighbors**0.5)
        return node_features


class Network(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.num_neighbors = 3.8  # typical number of neighbors
        self.irreps_sh = o3.Irreps.spherical_harmonics(3)

        irreps = self.irreps_sh

        # First layer with gate
        gate = Gate(
            "16x0e + 16x0o",
            [torch.relu, torch.abs],  # scalar
            "8x0e + 8x0o + 8x0e + 8x0o",
            [torch.relu, torch.tanh, torch.relu, torch.tanh],  # gates (scalars)
            "16x1o + 16x1e",  # gated tensors, num_irreps has to match with gates
        )
        self.conv = Convolution(irreps, self.irreps_sh, gate.irreps_in, self.num_
↪neighbors)
        self.gate = gate
        irreps = self.gate.irreps_out

        # Final layer
        self.final = Convolution(irreps, self.irreps_sh, "0o + 6x0e", self.num_neighbors)
        self.irreps_out = self.final.irreps_out

    def forward(self, data) -> torch.Tensor:
        num_nodes = 4  # typical number of nodes

        edge_src, edge_dst = radius_graph(x=data.pos, r=2.5, batch=data.batch)
        edge_vec = data.pos[edge_src] - data.pos[edge_dst]
```

(continues on next page)

(continued from previous page)

```
        edge_attr = o3.spherical_harmonics(l=self.irreps_sh, x=edge_vec, normalize=True,␣
↪normalization="component")
        edge_length_embedded = (
            soft_one_hot_linspace(x=edge_vec.norm(dim=1), start=0.5, end=2.5, number=3,␣
↪basis="smooth_finite", cutoff=True)
            * 3**0.5
        )

        x = scatter(edge_attr, edge_dst, dim=0).div(self.num_neighbors**0.5)

        x = self.conv(x, edge_src, edge_dst, edge_attr, edge_length_embedded)
        x = self.gate(x)
        x = self.final(x, edge_src, edge_dst, edge_attr, edge_length_embedded)

        return scatter(x, data.batch, dim=0).div(num_nodes**0.5)


def main():
    data, labels = tetris()
    f = Network()

    print("Built a model:")
    print(f)

    optim = torch.optim.Adam(f.parameters(), lr=1e-3)

    # == Training ==
    for step in range(200):
        pred = f(data)
        loss = (pred - labels).pow(2).sum()

        optim.zero_grad()
        loss.backward()
        optim.step()

        if step % 10 == 0:
            accuracy = pred.round().eq(labels).all(dim=1).double().mean(dim=0).item()
            print(f"epoch {step:5d} | loss {loss:<10.1f} | {100 * accuracy:5.1f}%␣
↪accuracy")

    # == Check equivariance ==
    # Because the model outputs (psuedo)scalars, we can easily directly
    # check its equivariance to the same data with new rotations:
    print("Testing equivariance directly...")
    rotated_data, _ = tetris()
    error = f(rotated_data) - f(data)
    print(f"Equivariance error = {error.abs().max().item():.1e}")

    print("Testing equivariance using `assert_equivariance`...")
    # We can also use the library's `assert_equivariant` helper
    # `assert_equivariant` also tests parity and translation, and
    # can handle non-(psuedo)scalar outputs.
```

(continues on next page)

(continued from previous page)

```
    # To "interpret" between it and torch_geometric, we use a small wrapper:

    def wrapper(pos, batch):
        return f(Data(pos=pos, batch=batch))

    # `assert_equivariant` uses logging to print a summary of the equivariance error,
    # so we enable logging
    logging.basicConfig(level=logging.INFO)
    assert_equivariant(
        wrapper,
        # We provide the original data that `assert_equivariant` will transform...
        args_in=[data.pos, data.batch],
        # ...in accordance with these irreps...
        irreps_in=[
            "cartesian_points",  # pos has vector 1o irreps, but is also translation␣
↪equivariant
            None,  # `None` indicates invariant, possibly non-floating-point data
        ],
        # ...and confirm that the outputs transform correspondingly for these irreps:
        irreps_out=[f.irreps_out],
    )


if __name__ == "__main__":
    main()


def test():
    torch.set_default_dtype(torch.float64)

    data, labels = tetris()
    f = Network()

    pred = f(data)
    loss = (pred - labels).pow(2).sum()
    loss.backward()

    rotated_data, _ = tetris()
    error = f(rotated_data) - f(data)
    assert error.abs().max() < 1e-10


def profile():
    data, labels = tetris()
    data = data.to(device="cuda")
    labels = labels.to(device="cuda")

    f = Network()
    f.to(device="cuda")

    optim = torch.optim.Adam(f.parameters(), lr=1e-2)
```

(continues on next page)

(continued from previous page)

```
    called_num = [0]

    def trace_handler(p):
        print(p.key_averages().table(sort_by="self_cuda_time_total", row_limit=-1))
        p.export_chrome_trace("test_trace_" + str(called_num[0]) + ".json")
        called_num[0] += 1

    with torch.profiler.profile(
        activities=[torch.profiler.ProfilerActivity.CPU, torch.profiler.ProfilerActivity.
↪CUDA],
        schedule=torch.profiler.schedule(wait=50, warmup=1, active=1),
        on_trace_ready=trace_handler,
    ) as p:
        for _ in range(52):
            pred = f(data)
            loss = (pred - labels).pow(2).sum()

            optim.zero_grad()
            loss.backward()
            optim.step()

            p.step()
```

Full code here

CHAPTER

THREE

DEMONSTRATION

All the functions to manipulate rotations (rotation matrices, Euler angles, quaternions, convertions, ...) can be found here *Parametrization of Rotations*. The irreducible representations of $O(3)$ (more info at *Irreps*) are represented by the class `e3nn.o3.Irrep`. The direct sum of multiple irrep is described by an object `e3nn.o3.Irreps`.

If two tensors x and y transforms as $D_x = 2 \times 1_o$ (two vectors) and $D_y = 0_e + 1_e$ (a scalar and a pseudovector) respectively, where the indices e and o stand for even and odd – the representation of parity,

```
import torch
from e3nn import o3

irreps_x = o3.Irreps('2x1o')
irreps_y = o3.Irreps('0e + 1e')

x = irreps_x.randn(-1)
y = irreps_y.randn(-1)

irreps_x.dim, irreps_y.dim
```

```
(6, 4)
```

their outer product is a 6×4 matrix of two indices $A_{ij} = x_i y_j$.

```
A = torch.einsum('i,j', x, y)
A
```

```
tensor([[-2.0980,  1.5137, -0.7972, -0.1883],
        [ 2.3531, -1.6977,  0.8941,  0.2112],
        [-0.8816,  0.6361, -0.3350, -0.0791],
        [ 2.5419, -1.8340,  0.9659,  0.2281],
        [ 0.9805, -0.7074,  0.3726,  0.0880],
        [ 0.2366, -0.1707,  0.0899,  0.0212]])
```

If a rotation is applied to the system, this matrix will transform with the representation $D_x \otimes D_y$ (the tensor product representation).

$$A = xy^t \longrightarrow A' = D_x A D_y^t$$

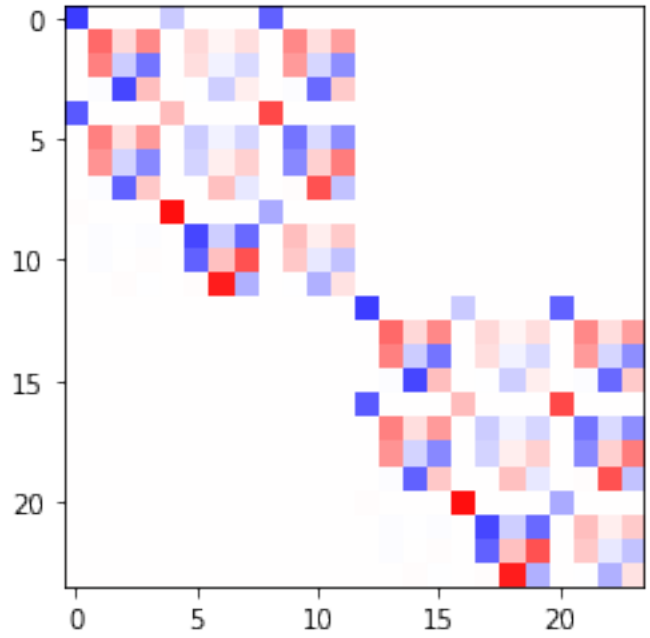
Which can be represented by

```
R = o3.rand_matrix()
D_x = irreps_x.D_from_matrix(R)
```

(continues on next page)

(continued from previous page)

```
D_y = irreps_y.D_from_matrix(R)

plt.imshow(torch.kron(D_x, D_y), cmap='bwr', vmin=-1, vmax=1);
```



This representation is not irreducible (is reducible). It can be decomposed into irreps by a change of basis. The outerproduct followed by the change of basis is done by the class *e3nn.o3.FullTensorProduct*.

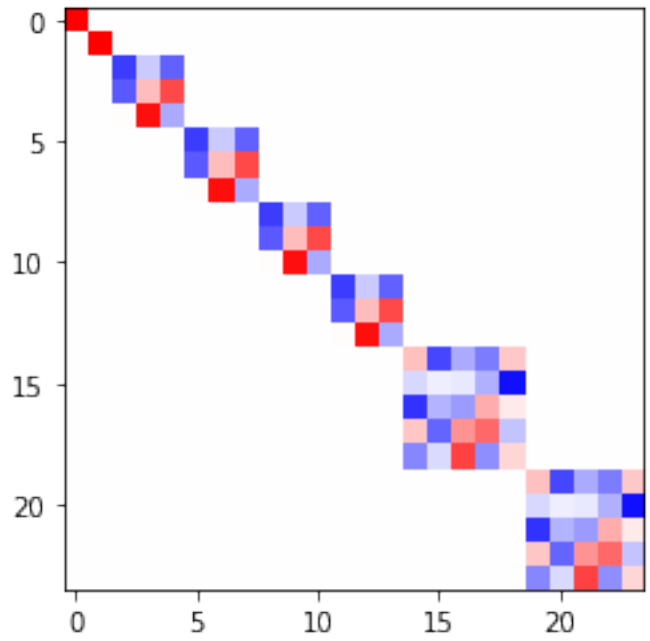
```
tp = o3.FullTensorProduct(irreps_x, irreps_y)
print(tp)

tp(x, y)
```

```
FullTensorProduct(2x1o x 1x0e+1x1e -> 2x0o+4x1o+2x2o | 8 paths | 0 weights)
```

```
tensor([ 1.3445e+00, -8.3150e-01, -2.0980e+00,  2.3531e+00, -8.8163e-01,
         2.5419e+00,  9.8047e-01,  2.3657e-01,  3.8619e-01,  5.8291e-01,
         6.3675e-01, -1.3501e-03, -2.8198e-01,  1.1832e+00,  3.1666e-01,
        -1.7642e+00,  1.4437e-01, -8.7578e-02, -1.1263e+00,  4.0601e-02,
         1.8278e-01,  1.0442e+00,  1.2578e-01,  1.3118e+00])
```

As a sanity check, we can verify that the representation of the tensor prodcut is block diagonal and of the same dimension.

```
D = tp.irreps_out.D_from_matrix(R)
plt.imshow(D, cmap='bwr', vmin=-1, vmax=1);
```



e3nn.o3.FullTensorProduct is a special case of *e3nn.o3.TensorProduct*, other ones like *e3nn.o3.FullyConnectedTensorProduct* can contained weights what can be learned, very useful to create neural networks.

PYTHON MODULE INDEX

e

INDEX

V

W

X