EquiTorch: A Modularized Package for Flexibly Constructing Equivariant GNNs Building upon Pytorch-Geometric

Tong Wang [1]
Chuan Chen [1]

Abstract

Equivariant graph neural networks have recently gained significant attention due to their demonstrated effectiveness in geometric deep learning applications for scientific problems. However, despite the significant progress in developing equivariant graph neural networks, the implementations show slight diversity in their conventions, setting up a little barrier for pure AI researchers to get engaged in this fascinating field. Starting from this point, we would like to introduce our package, *EquiTorch*, which aims to collect the operations related to equivariant neural networks in a standardized style, following the framework and idea of "Message Passing Neural Networks" (MPNN) used in Pytorch-Geometric, which is familiar to classical AI researchers on GNNs. Besides, by aligning to the framework of MPNN, we presents the basic operations in a modularized way, which enables researchers to compose these operations flexibly and explore the larger space of design of equivariant GNNs. In near future, more comprehensive documentation and tutorials will be made available. The package can now be found at https://github.com/Xenadon/equitorch.

1. Introduction

The rapid advancements in scientific domains, such as molecular modeling, particle physics material science and fluid dynamics (Zhang et al., 2023; Wang et al., 2023) have led to an increased demand for powerful deep learning techniques that can effectively capture the underlying symmetries and geometric properties, where the relevant information is equivariant or invariant to certain transformations like rotation, translation or permutation. Graph neural networks (GNNs), which are built upon the message passing paradigm (Gilmer et al., 2017), have shown promise to handle the permutation symmetries through utilizing permutation equivariant aggregation and pooling functions. However, when it comes to rotation symmetries, conventional GNNs may fail to capture exact equivariance features beyond invariances on limited data (Satorras et al., 2022).

To address this challenge, equivariant[†] GNNs has emerged as a promising solution that can effectively exploit the inherent symmetries and geometric properties of the input data, leading to improved performance and enhanced generalization comparing to conventional GNNs. Specifically, equivariant GNNs achieve their advantages by incorporating specialized operations like tensor product of vectors in irreducible spaces(Thomas et al., 2018; Fuchs et al., 2020; Brandstetter et al., 2022; Liao & Smidt, 2023; Liao et al., 2024) and equivariant frames scalarization-tensorization (Duval et al.; Zitnick et al., 2022; Du et al., 2022; Joshi et al., 2023) into the neural network architecture. These operations are designed to respect the symmetries of the input data, aligning the model parameters and internal embeddings with the physical priors.

However, while numerous equivariant layers and architectures have been proposed for equivariant GNNs, the convention of model organization and equivaiant data storage may not be always consistent across different implementations. Even researchers with sufficient background knowledge may spend a while in combining modules from different works, a newcomer with pure AI background can further more easily get lost in tensor analysis or group representation when trying to aligning different conventions, which may discourage them form stepping into this fascinating field.

Therefore, we presents *EquiTorch*, a modularized package for flexibly constructing equivariant GNNs. We aims to collect most of the currently proposed equivariant operations

[1]Department of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. Correspondence to: Tong Wang <wangt328@mail2.sysu.edu.cn>, Chuan Chen <chenchuan@mail.sysu.edu.cn>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

[†]Here, by "equivariant" or "equivariances", we refer in particular to rotational and translational equivariances for simplicity, since the permutation equivariances of GNNs have long been studied.

in a modularized way and aligning them to a single convention for easy combination of modules from different works. Besides, the package is built upon Pytorch-Geometric (Fey & Lenssen, 2019), which is a widely used package for building GNNs for classical AI researchers. By modularizing the operations, aligning them to a single convention and fit the model into the message passing fashion of Pytorch-Geometric, we hope we can make it easier for experienced researchers in combining different techniques and explore the larger space of model designation as well as for new comers to smoothly step into the world of equivariant GNNs.

2. Background

2.1. Group Representations and Equivariant Functions

Suppose we are given a group G, such as the SO(3) group of rotations in $\mathbb{R}^3$. Beyond the original transformation on $\mathbb{R}^3$ by definition, we may further **represent** the group elements as transformations on other linear spaces, while preserving the relational structure of combination of original transform. If we are able to do that, we are said to have found a **representation** of that group.

Mathematically, we can define a representation as follows:

Definition 2.1 (Group Representation)**.** Given a group G and a linear space X, a representation of G on V is a mapping $\rho_X : G \to \mathrm{GL}(X)$, such that for any two members a, b of G, it holds that

$$\rho_X(a) \cdot \rho_X(b) = \rho_X(a \cdot b),$$

where the $\mathrm{GL}(X)$ is the general linear group on X, that is, all invertible linear transformations on it.

Sometimes, we may also call the space X in the definition above a representation of G.

If we have two representations ρ_X, ρ_Y for a group G on linear spaces X and Y, respectively, a function $f : X \to Y$ is said to be **equivariant** if

$$f(\rho_X(g)x) = \rho_Y(g)f(x)$$

holds for any g in G and x in X. In particular, if $\rho_Y \equiv I_Y$, that is the identity on Y, the equivariance will become

$$f(\rho_X(g)x) = f(x)$$

for all g in G and x in X. In this special case, we call the function f to be **invariant**.

When it comes to the case of SO(3), modeling features in representation spaces with equivariant feature transformations has been shown to be a promising approach to achieve exact equivariances.

According to the group representation theory, any finite-dimensional representation space of SO(3) can be decomposed into the direct sum of a series of irreducible representations (irreps), which reminds us that we can model the equivariant features in the irreps. The irreps of SO(3) can be indexed with a degree $l = 0, 1, 2, \ldots$, and the irrep of degree-l is $(2l+1)$-dimensional with each dimension called an order.

Commonly, the spherical harmonics $\mathbf{Y}^{(l)}(\hat{\mathbf{r}}) = \{Y_m^{(l)}(\hat{\mathbf{r}})\}_{m=-l,\ldots,l}$ with $\hat{\mathbf{r}}$ lying on the unit sphere are chosen as the basis of degree-l irreps and the corresponding representation for a rotation $\mathbf{R} \in \mathrm{SO}(3)$ is the Wigner-D matrices $\mathbf{D}^{(l)}(\mathbf{R}) \in \mathbb{R}^{(2l+1)\times(2l+1)}$ that additionally satisfies $\mathbf{D}^{(l)}(\mathbf{R})\mathbf{Y}^{(l)}(\hat{\mathbf{r}}) = \mathbf{Y}^{(l)}(\mathbf{R}\hat{\mathbf{r}})$.

When we represent an equivariant feature in a series of irreps, its components in each irrep of degree-l_i ought to transform via the corresponding representation $\mathbf{D}^{(l_i)}(\mathbf{R})$ under a rotation $\mathbf{R}$ in the physical space. To perform the transformation on such features while perserve the equivariance, one key tequnique is to use a "tensor product"(Thomas et al., 2018), which is a bilinear equivariant operation. Especially, when we fixed one input of this bilinear operator, it will be a linear operator with respect to the remaining input, which makes it a useful building block for constructing deep equivariant neural networks(Brandstetter et al., 2022).

After linear operations, we may turn to nonlinear operations. However, it is important to note that the $(2l+1)$ components of a vector in a degree-l representation should transform as a cohesive unit under orthogonal and scaling transformations, in order to maintain the equivariance. This means we cannot simply apply element-wise nonlineariteis or bias on the equivariant features. Instead, we should restrict ourselves to degree-wise nonlinearities and biases that operate on the norm of the equivariant features. This can ensure the transformation properties of the vector are preserved.

2.2. Graph Neural Networks and Message Passing Paradigm

Throughout the past several years, GNNs have been a powerful approach for dealing with unstructured data such as graphs, point clouds and meshes. In GNNs, an attributed graph of N nodes and E edges is usually represented as a tuple $\mathcal{G} = (\mathbf{X}, (\mathbf{I}, \mathbf{E}))$ of F-dimensional node features $\mathbf{X} \in \mathbb{R}^{N\times F}$ and edge info containing the end-point indices $\mathbf{I} \in \mathbb{R}^{2\times E}$ and D-dimensional edge features $\mathbf{E} \in \mathbb{R}^{E\times D}$.

Since (Gilmer et al., 2017) proposed the paradigm of message passing neural networks, it has become the standard framework of GNNs. The key process of updating latent

node embedding at the k-th layer can be expressed as

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \bigoplus_{j\in\mathcal{N}(i)} \phi^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{ij}^{(k-1)}\right)\right),$$

where ϕ is a message function that generate messages from both source node, target node and the edge; $\bigoplus$ is a differentiable, permutation-invariant function that aggregates messages from the neighbor $\mathcal{N}(i)$ and γ is the update function that computes the new node embeddings considering the current embedding and the aggregated messages.

By substituting all the message, aggregation and updating functions of equivariant features with equivariant functions, we will obtain an equivariant message passing neural network.

In the context of AI for science tasks, the message passing processes are also interpreted as the "interactions" between particles. Therefore, some work will refer to a message passing layer as an "interaction block".

3. Related Works

The increasing interest in building equivariant neural networks has led to the development of several software packages that provide equivariant operations.

One such framework is **e3nn** (Geiger & Smidt, 2022), which is a generalized framework for creating E(3) equivariant trainable functions. It provides a wide range of fundamental operations, with support for both PyTorch and JAX backends. In e3nn, the shape of equivariant features is identified by the `Irreps` class. Data are arranged in a (l, c, m) manner, that is the data are first grouped by degree l, then by channel c in each degree, with each channel containing a dim-$(2l+1)$ vector of which the components are indexed by order m. These degree-l vectors for all degrees are concentrated one by one to give the final equivariant feature. This data organization allows for flexible, dense data storage with varying number of channels across different degrees, while might sacrifice a little performance when doing transformation.

Another notable package is **SchNetPack 2.0** (Schütt et al., 2023), a neural network toolbox that addresses both the development and application of models on atomistic machine learning. In addition to implementing equivariant operations, it also includes a variety of utility functions, such as cutoffs, that are particularly useful for building atomistic models on the top of PyTorch. SchNetPack organizes equivariant features in an (l, m, c) or (s, c) layout, where s is a flattened index of (l, m). It considers the data to have c equivariant channels, with each channel's feature composed of one equivariant vector of all degrees from the given degree range (0 to the maximum degree). This organization enables efficient channel-mixing and representation rotation through direct matrix multiplications. However, when differnet degrees require different numbers of channels, padding may be necessory, potentially leading to non-dense data storage.

E3x (Unke & Maennel, 2024) is a JAX library that focuses on constructing efficient E(3)-equivariant neural networks. It offers comprehensive set of equivariant linear and nonlinear functions as well as utility functions, to facilitate the development of E(3)-equivariant neural networks. E3x employs a data layout of (p, s, c) that is similar to SchNetPack in (s, c) part, but with an additional dimension p to indicate feature parity under reflection that will always have a length of 1 or 2 (odd and even).

4. Design of EquiTorch

In this section we will disscuss the design of EquiTorch. First we will clarify how the equivariant data are stored for the convention on dimension-order. Then we will list some basic modules of equivariant GNNs that have been implemented, which enables flexible combination to more novel equivariant architectures.

4.1. Storage of Equivariant Data

In EquiTorch, we do not rely on additional data structures to represent equivariant data. Instead, we establish a convention where the first dimension of any equivariant data should always represent the data dimension, which could be either a node index or an edge index, and the last dimension should always represent the channel dimension, if it exists. The internal dimensions are used to represent the coordinates. For equivariant features, this exactly coincides with the (s, c) layout used in SchNetPack, while for equivariant transformations, there will be two internal dimensions that the latter is the input dimension and the former is the output dimension. We adopt this layout mainly because, after examing most of the morden equivariant networks, we find that the case of varying channel numbers in different degree only occurs in the first and final layers, while in the hidden layers, they typically set a constant number of channels for different degrees. Regarding for issues of parity under reflection, we also find that most networks do not explicitly take it into consideration, assuming such one-element discrete symmetry will be relatively easy to learn from data in contrast to continuous 3D rotation. Fig. 1 gives two examples of the rotation matrices and the features consisting components of degree 0 to 2 of four data points with four channels.

4.2. Basic Modules Implementation

A key goal of the EquiTorch package is to decompose existing equivariant GNN models into more fundamental, mod-

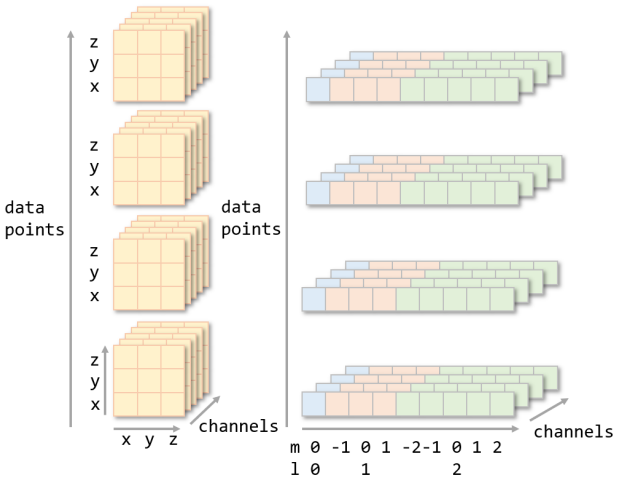

Figure 1. Data organization examples of rotation matrices (left) and equivariant features (right). The tensor containing rotation matrices is of shape $4 \times 3 \times 3 \times 4$, and the tensor containing equivariant features is of shape $4 \times 9 \times 4$.

ular building blocks. This allows for greater flexibility in constructing and exploring novel equivariant architectures. The core modules currently implemented in EquiTorch include:

Tensor Products: Tensor products are fundamental operations for combining equivariant feature representations. We provide several implementations, including `TensorProduct` for direct contraction of the Clebsch-Gordan coefficients and two input equivariant features with no weights and `WeightedTensorProduct` for the commonly used tensor products (Thomas et al., 2018; Brandstetter et al., 2022; Fuchs et al., 2020; Yu et al.).

Notably, in the implementation, we reduced the computational complexity to $O(L^5)$ by exploiting the sparsity of Clebsch-Gordan coefficients. This optimization also greatly enhanced the actual performances. Moreover, our sparse implementation of tensor products can be readily adapted to many-body approaches, such as MACE(Batatia et al., 2023), with minor modifications.

For the operations `WeightedTensorProdcut` that involve weights, we provide an `external_weights` option, allowing users to choose whether the weights are passed from external or let the layer keep a set of weights that is independent on data.

Linear Operations: Once we fix one factor of the tensor product and focus on the other, it will turn to a linear operation - an essential building block in deep learning architectures. We have implemented `SO3Linear`(Thomas et al., 2018; Fuchs et al., 2020; Brandstetter et al., 2022) using similar sparse techniques as tensor products to achieve a complexity of O(L^5) for the input of degrees up to L; `SO2Linear` as derived in (Passaro & Zitnick) and used in (Liao et al., 2024) without explicitly looping over the order m; and `DWLinear` (Degree-Wise Linear) used for self-interaction transforms in (Thomas et al., 2018; Fuchs et al., 2020).

For these linear operations, we provide two additional options: `external_weights` and `channel_wise`. Similar to the tensor products, the `external_weights` option allows users to choose whether to pass external weights; and the `channel_wise` option determines whether weights are applied as per-channel scaling factors or as general linear transformations mixing all input channels for all output channels.

To the best of our knowledge, we are the first to release our implementation that reduces the time complexity of fully connected SO(3) linear transformations to $O(L^5C + L^4CC')$ without any precomputation for the maximal degree L, input channel C and output channel C'. Though (Milesi, 2021) also provides an solution with same time complexity, it requires the edge features to be fixed during the entire training process to reuse a precomputed tensor. This may be suitable for simple classification processes, but not for the generation processes, or the cases where noises are injected to the node positions, causing edge features to change across epochs. The implementation in SchNetPack does use similar sparse techniques but only supports channel wise scaling with contraction to spherical harmonics rather than a fully connected linear transformation with contraction to arbitrary equivariant features. A performance comparison with other implementations is presented in Table 1.

Activations. In equivairant neural networks, activations on equivariant features need to be specially designed to perserve the equivariances. We have implemented `NormActivation` that act on the norm of each degree of equivariant features as introduced in (Thomas et al., 2018); `GatedActivation` in (Liao & Smidt, 2023; Brandstetter et al., 2022); `S2Activation` and `SeparableS2Activation` in (Zitnick et al., 2022; Passaro & Zitnick; Liao et al., 2024).

Radial Basis. Radial basis modules will enable us to expand a single distance feature to a vector. We have implemented `GaussianBasisExpansion` (Schütt et al., 2023; 2017; Thomas et al., 2018; Liao & Smidt, 2023; Thölke & De Fabritiis, 2022) with option to train or fix basis parameters.

Cutoff Modules. When constructing the molecule graphs, a cutoff distance is set to sparsify the interactions. However, a hard cutoff by threshold may lead to unsmooth predictions, therefore cutoff modules will offer a soft weight to the edges that smoothly decays to 0 at the cutoff threshold. We have implemented three widely used cutoffs including `CosineCutoff` (Schütt et al.,

2017; Thölke & De Fabritiis, 2022; Simeon & de Fabritiis, 2023), `MollifierCutoff` (Schütt et al., 2023) and `PolynomialCutoff` (Gasteiger et al., 2022) with option to choose starting and ending points of the cutoff functions.

Besides these modules, we have also implemented many utility functions. These include computing the shapes of equivariant features for a given degree range, extracting information of specified degrees, generating local frames on given edge vectors, and performing order-wise operations such as dot product and norm computation. The semantics and conventions used in all these functions are stated as clearly as possible in our documentation.

5. Experiments

To demonstrate the efficiency of the operations we implemented, we provide a comparison of the fully connected SO(3) equivariant linear transformation between our implementation (`SO3Linear`) and several other implementations.

The compared implementations include:

- Directly performing dense contraction between Clebsch-Gordan coefficients and two input features (dense).
- Pre-computing a basis via contraction of Clebsch-Gordan coefficients and then using it to contract with the other input as suggested in (Milesi, 2021)(cached).
- Calling the `FullyConnectedTensorProduct` provided by e3nn(Geiger & Smidt, 2022).

To evaluate the performance of all the implementations, we benchmark the execution time for both the forward and backward passes. For forward benchmarking, we call the function on randomly generated batched inputs consisting of 200 samples and 64 channels, with maximal degrees ranging from 1 to 8. For backward benchmarking, we first perform the forward pass and then compute the gradients by backpropagating the sum of the output tensor. Each function is called 100 times, and the minimum execution time among these iterations is recorded. The benchmarking is conducted on a system with an Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz and an NVIDIA GeForce RTX 3090 GPU. The results of the benchmarking experiments are presented in Table 1.

Our implementation of SO(3) equivariant linear transformations consistently outperforms existing methods across all tested degrees (L=1 to 8) for both forward and backward passes, as shown in Table 1. It demonstrates superior speed, with execution times up to 50 times faster than e3nn for forward passes and up to near 2000 times faster for backward passes at L=8. Moreover, our implementation exhibits remarkable memory efficiency, maintaining stable performance where other methods encounter out-of-memory issues.

Table 1. Forward and backward pass execution times (ms) for varying maximal degree L.

setting	L	ours	dense	cached	e3nn
forward	1	**0.1270**	0.3944	0.1490	0.3960
	2	**0.1289**	0.3999	0.1503	1.1462
	3	**0.1485**	0.3927	0.1512	2.5802
	4	**0.1374**	0.4000	0.1496	4.1335
	5	**0.1389**	1.2310	0.1683	8.5093
	6	**0.1506**	1.4214	0.5687	11.1477
	7	**0.1749**	OOM	0.3612	16.1090
	8	**0.4244**	OOM	OOM	22.9117
backward	1	**0.4859**	1.2752	0.9374	2.9601
	2	**0.5822**	12.841	0.9925	4.6638
	3	**0.5971**	1.1813	0.8188	8.6752
	4	**0.5411**	0.9646	1.0075	19.9319
	5	**0.4286**	2.2156	0.9368	164.7753
	6	**0.5939**	OOM	OOM	475.4104
	7	**0.6101**	OOM	OOM	1044.6669
	8	**1.1024**	OOM	OOM	2104.2929

6. Examples

In this section, we will give two simple examples of using EquiTorch to build the Tensor Field Network (Thomas et al., 2018) and a variant using SO(2) linear transform (Passaro & Zitnick), showing its benefits of following the interface of Pytorch-Geometric and the strengthens of the modularized design. The examples are presented in Fig. 2.

6.1. Building a TFN Layer Just Like a GCN one

In this subsection, we can see the similarity of defining a GCN(Kipf & Welling, 2017) layer and TFN layer under the framework of Pytorch-Geometric.

The left column in Fig. 2 builds a GCN layer that uses degree-normalization, sum-aggregation and a ReLU activation, which is a Hello-World example that is familiar to any researchers with some experiences on classical GNNs. While the middle column builds a TFN layer.

We may notice that, though there are more arguments of some functions, the main logic of two pieces of code does not differ a lot, given that `edge_weight` is the output of a standard invariant neural network like an MLP.

```python
class GCNLayer(MessagePassing):

    def __init__(self,
                 in_channels,
                 out_channels):
        super().__init__(aggr='add')
        self.lin = Linear(in_channels,
                          out_channels,
                          bias=False)
        self.act = ReLU()


    def forward(self,
                x, edge_index,
                norm):

        out = self.propagate(edge_index,
                             x=x,
                             norm=norm)

        out = out + x
        return self.act(out)

    def message(self, x_j,
                norm):

        x_j = self.lin(x_j)


        return norm.view(-1, 1) * x_j
```

```python
class TFNLayer(MessagePassing):

    def __init__(self,
                 in_channels, out_channels,
                 L_in, L_edge, L_out):
        super().__init__(node_dim=-3, aggr='add')
        self.lin = SO3Linear(in_channels, out_channels,
                             L_in, L_edge, L_out,
                             external_weight=True)
        self.act = NormActivation(ShiftedSoftPlus())
        self.self_int = DWLinear(in_channels, out_channels,
                                 L_out, L_in)
    def forward(self,
                x, edge_index,
                edge_feat, edge_weight, lin_weight):

        out = self.propagate(edge_index, x=x,
                             edge_feat=edge_feat,
                             lin_weight=lin_weight,
                             edge_weight=edge_weight)
        out = out + self.self_int(x)
        return self.act(out)

    def message(self, x_j, edge_feat,
                lin_weight, edge_weight):

        x_j = self.lin.forward(x_j,
                               edge_feat, lin_weight)


        return edge_weight.view(-1,1,1) * x_j
```

```python
class SO2TFNLayer(MessagePassing):

    def __init__(self,
                 in_channels, out_channels,
                 L_in, L_out):
        super().__init__(node_dim=-3, aggr='add')
        self.lin = SO2Linear(in_channels, out_channels,
                             L_in, L_out,
                             external_weight=True)
        self.act = NormActivation(ShiftedSoftPlus())
        self.self_int = DWLinear(in_channels, out_channels,
                                 L_out, L_in)
    def forward(self,
                x, edge_index,
                edge_weight, lin_weight,
                D_in, DT_out):
        out = self.propagate(edge_index, x=x,
                             lin_weight=lin_weight,
                             edge_weight=edge_weight,
                             D_in=D_in, DT_out=DT_out)
        out = out + self.self_int(x)
        return self.act(out)

    def message(self, x_j,
                lin_weight, edge_weight,
                D_in, DT_out):
        x_j = rot_on(D_in, x_j)
        x_j = self.lin.forward(x_j, lin_weight)
        x_j = rot_on(DT_out, x_j)
        return edge_weight.view(-1,1,1) * x_j
```

Figure 2. The examples of building a TFN layer that like a GCN layer and replace the SO3Linear in it with an SO2Linear. The same operations are aligned to the same lines.

6.2. Building a TFN Layer with SO(2) Linear Operation

Then, suppose that we noticed the SO(2) Linear operation given by (Passaro & Zitnick) can greatly reduce the complexity. What we need to do is simply replace the SO3Linear module with the SO2Linear, as well as the related parameters, as presented in the right column of Fig. 2.

We can check that, by comparing the operations on the same lines, that there are really little modification to change from SO3Linear to SO2Linear, thanks to the modularized design.

7. Conclusion

In conclusion, EquiTorch is a modularized package that aims to provide a standardized and flexible framework for constructing equivariant graph neural networks. By aligning the package to the message passing paradigm of Pytorch-Geometric, it enables researchers to easily combine various equivariant operations and explore the design space of novel equivariant architectures. However, the current package is not yet complete, and we will continuously add more equivariant operations and functionalities to EquiTorch in the future. Moreover, we will also work on providing nice tutorials and comprehensive documentation to further lower the barrier for AI researchers to engage in this exciting field of equivariant deep learning for scientific applications. With these ongoing efforts, we hope that EquiTorch can serve as a useful tool to advance the state-of-the-art in equivariant graph neural networks.

References

Batatia, I., Kovács, D. P., Simm, G. N. C., Ortner, C., and Csányi, G. MACE: Higher order equivariant message passing neural networks for fast and accurate force fields, 2023. URL https://arxiv.org/abs/2206.07697.

Brandstetter, J., Hesselink, R., van der Pol, E., Bekkers, E. J., and Welling, M. Geometric and physical quantities improve E(3) equivariant message passing, March 2022. URL http://arxiv.org/abs/2110.02905. arXiv:2110.02905 [cs, stat].

Du, W., Zhang, H., Du, Y., Meng, Q., Chen, W., Shao, B., and Liu, T.-Y. SE(3) Equivariant Graph Neural Networks with Complete Local Frames, July 2022. URL http://arxiv.org/abs/2110.14811. arXiv:2110.14811 [physics].

Duval, A., Schmidt, V., Garcia, A. H., Miret, S., Malliaros, F. D., Bengio, Y., and Rolnick, D. FAENet: Frame Averaging Equivariant GNN for Materials Modeling.

Fey, M. and Lenssen, J. E. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019. URL http://arxiv.org/abs/1903.02428.

Fuchs, F. B., Worrall, D. E., Fischer, V., and Welling, M. SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks, November 2020. URL http://arxiv.org/abs/2006.10503. arXiv:2006.10503 [cs, stat].

Gasteiger, J., Groß, J., and Günnemann, S. Directional message passing for molecular graphs, 2022.

Geiger, M. and Smidt, T. e3nn: Euclidean neural networks, 2022.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017. URL http://arxiv.org/abs/1704.01212.

Joshi, C. K., Bodnar, C., Mathis, S. V., Cohen, T., and Liò, P. On the expressive power of geometric graph neural networks, June 2023. URL http://arxiv.org/abs/2301.09308. arXiv:2301.09308 [cs, math, stat].

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks, 2017.

Liao, Y.-L. and Smidt, T. Equiformer: Equivariant graph attention transformer for 3d atomistic graphs, February 2023. URL http://arxiv.org/abs/2206.11990. arXiv:2206.11990 [physics].

Liao, Y.-L., Wood, B., Das, A., and Smidt, T. EquiformerV2: Improved Equivariant Transformer for Scaling to Higher-Degree Representations. 2024.

Milesi, A. Accelerating se(3)-transformers training using an nvidia open-source model implementation. https://developer.nvidia.com/blog/accelerating-se3-transformers-training-using-an-nvidia-open-source-model-implementation/, 2021.

Passaro, S. and Zitnick, C. L. Reducing SO(3) convolutions to SO(2) for efficient equivariant GNNs.

Satorras, V. G., Hoogeboom, E., and Welling, M. E(n) equivariant graph neural networks, February 2022. URL http://arxiv.org/abs/2102.09844. arXiv:2102.09844 [cs, stat].

Schütt, K. T., Kindermans, P.-J., Sauceda, H. E., Chmiela, S., Tkatchenko, A., and Müller, K.-R. Schnet: A continuous-filter convolutional neural network for modeling quantum interactions, 2017.

Schütt, K. T., Hessmann, S. S. P., Gebauer, N. W. A., Lederer, J., and Gastegger, M. Schnetpack 2.0: A neural network toolbox for atomistic machine learning. *The Journal of Chemical Physics*, 158(14), April 2023. ISSN 1089-7690. doi: 10.1063/5.0138367. URL http://dx.doi.org/10.1063/5.0138367.

Simeon, G. and de Fabritiis, G. Tensornet: Cartesian tensor representations for efficient learning of molecular potentials, 2023.

Thomas, N., Smidt, T., Kearnes, S., Yang, L., Li, L., Kohlhoff, K., and Riley, P. Tensor field networks: Rotation- and translation-equivariant neural networks for 3D point clouds, May 2018. URL http://arxiv.org/abs/1802.08219. arXiv:1802.08219 [cs].

Thölke, P. and De Fabritiis, G. TorchMD-NET: Equivariant Transformers for Neural Network based Molecular Potentials, April 2022. URL http://arxiv.org/abs/2202.02541. arXiv:2202.02541 [physics].

Unke, O. T. and Maennel, H. E3x: $\mathrm{E}(3)$-equivariant deep learning made easy, 2024.

Wang, H., Fu, T., Du, Y., Gao, W., Huang, K., Liu, Z., Chandak, P., Liu, S., Katwyk, P. V., Deac, A., Anandkumar, A., Bergen, K. J., Gomes, C. P., Ho, S., Kohli, P., Lasenby, J., Leskovec, J., Liu, T.-Y., Manrai, A. K., Marks, D. S., Ramsundar, B., Song, L., Sun, J., Tang, J., Velickovic, P., Welling, M., Zhang, L., Coley, C. W., Bengio, Y., and Zitnik, M. Scientific discovery in the age of artificial intelligence. *Nature*, 620:47–60, 2023. URL https://api.semanticscholar.org/CorpusID:260384616.

Yu, H., Xu, Z., Qian, X., Qian, X., and Ji, S. Efficient and Equivariant Graph Networks for Predicting Quantum Hamiltonian.

Zhang, X., Wang, L., Helwig, J., Luo, Y., Fu, C., Xie, Y., Liu, M., Lin, Y., Xu, Z., Yan, K., Adams, K., Weiler, M., Li, X., Fu, T., Wang, Y., Yu, H., Xie, Y., Fu, X., Strasser, A., Xu, S., Liu, Y., Du, Y., Saxton, A., Ling, H., Lawrence, H., Stärk, H., Gui, S., Edwards, C., Gao, N., Ladera, A., Wu, T., Hofgard, E. F., Tehrani, A. M., Wang, R., Daigavane, A., Bohde, M., Kurtin, J., Huang, Q., Phung, T., Xu, M., Joshi, C. K., Mathis, S. V., Azizzadenesheli, K., Fang, A., Aspuru-Guzik, A., Bekkers, E., Bronstein, M., Zitnik, M., Anandkumar, A., Ermon, S., Liò, P., Yu, R., Günnemann, S., Leskovec, J., Ji, H., Sun, J., Barzilay, R., Jaakkola, T., Coley, C. W., Qian, X., Qian, X., Smidt, T., and Ji, S. Artificial intelligence for science in quantum, atomistic, and continuum systems, 2023.

Zitnick, C. L., Das, A., Kolluru, A., Lan, J., Shuaibi, M., Sriram, A., Ulissi, Z., and Wood, B. Spherical channels for modeling atomic interactions, 2022.