

RESEARCH ARTICLE | APRIL 12 2023

## SchNetPack 2.0: A neural network toolbox for atomistic machine learning

Special Collection: [Software for Atomistic Machine Learning](#)

Kristof T. Schütt ; Stefaan S. P. Hessmann; Niklas W. A. Gebauer; ... et. al



*J. Chem. Phys.* 158, 144801 (2023)

<https://doi.org/10.1063/5.0138367>



View  
Online



Export  
Citation

CrossMark

### Articles You May Be Interested In

REANN: A PyTorch-based end-to-end multi-functional deep neural network package for molecular, reactive, and periodic systems

*J. Chem. Phys.* (March 2022)

Underwater acoustic localization using a modular differentiable model for acoustic wave propagation

*J Acoust Soc Am* (April 2022)



Time to get excited.  
Lock-in Amplifiers – from DC to 8.5 GHz

 Find out more

Zurich  
Instruments

# SchNetPack 2.0: A neural network toolbox for atomistic machine learning

Cite as: J. Chem. Phys. 158, 144801 (2023); doi: 10.1063/5.0138367

Submitted: 11 December 2022 • Accepted: 21 March 2023 •

Published Online: 12 April 2023



View Online



Export Citation



CrossMark

Kristof T. Schütt,<sup>1,2,a)</sup> Stefaan S. P. Hessmann,<sup>1,2</sup> Niklas W. A. Gebauer,<sup>1,2,3</sup> Jonas Lederer,<sup>1,2</sup> and Michael Gastegger<sup>1,3,b)</sup>

## AFFILIATIONS

<sup>1</sup> Machine Learning Group, Technische Universität Berlin, 10587 Berlin, Germany

<sup>2</sup> Berlin Institute for the Foundations of Learning and Data, 10587 Berlin, Germany

<sup>3</sup> BASLEARN–TU Berlin/BASF Joint Lab for Machine Learning, Technische Universität Berlin, 10587 Berlin, Germany

**Note:** This paper is part of the JCP Special Topic on Software for Atomistic Machine Learning.

**a)**Author to whom correspondence should be addressed: [kristof.schuetz@tu-berlin.de](mailto:kristof.schuetz@tu-berlin.de)

**b)**Electronic mail: [michael.gastegger@tu-berlin.de](mailto:michael.gastegger@tu-berlin.de)

## ABSTRACT

SchNetPack is a versatile neural network toolbox that addresses both the requirements of method development and the application of atomistic machine learning. Version 2.0 comes with an improved data pipeline, modules for equivariant neural networks, and a PyTorch implementation of molecular dynamics. An optional integration with PyTorch Lightning and the Hydra configuration framework powers a flexible command-line interface. This makes SchNetPack 2.0 easily extendable with a custom code and ready for complex training tasks, such as the generation of 3D molecular structures.

Published under an exclusive license by AIP Publishing. <https://doi.org/10.1063/5.0138367>

## I. INTRODUCTION

In recent years, machine-learning (ML) techniques have become valuable tools in atomistic modeling of molecules and materials.<sup>1–6</sup> They have been shown to accurately predict chemical properties<sup>7–15</sup> and accelerate molecular dynamics (MD) simulations.<sup>16–20</sup> Machine-learning force fields have been applied to systems ranging from small molecules<sup>21–23</sup> over biomolecular systems<sup>24</sup> to materials with millions of atoms.<sup>25,26</sup> Although modeling of potential energy surfaces is arguably the most prominent application, machine learning is integrated into more and more steps of molecular and materials modeling workflows.<sup>27</sup> It has ventured into the prediction of electronic densities,<sup>28–30</sup> molecular orbitals,<sup>31,32</sup> and excited states.<sup>33–35</sup> Techniques such as reinforcement learning and generative neural networks have enabled complex tasks such as molecular manipulation<sup>36</sup> or the inverse design of 3D molecular structures.<sup>37–41</sup> Beyond that, unsupervised learning has been applied to learn molecular kinetics,<sup>42</sup> identify chemical moieties,<sup>43</sup> and learn representations of wavefunctions.<sup>44,45</sup> These developments come with increasingly diverse demands on an ML toolbox for atomistic modeling.

When the first version of SchNetPack was released,<sup>46</sup> the aim was to provide a software package that makes neural network potentials easily accessible for researchers in atomistic modeling and machine learning. This has been achieved by making atomistic benchmark sets readily available, establishing a unified structure for neural network potentials and providing a scalable training framework based on PyTorch<sup>47</sup> that takes large parts of the boilerplate code off the researcher. However, the rapid development of the field described above demands a more flexible approach that is able to adapt to new tasks such as generative models. Furthermore, the growing PyTorch ecosystem with training frameworks such as PyTorch Lightning<sup>48</sup> or Ignite<sup>49</sup> makes maintaining its own training framework in SchNetPack an unwarranted technical debt.

With the release of SchNetPack 2.0, major parts of the code base have been rewritten to address the changing demands on a neural network toolbox for atomistic modeling. We have designed SchNetPack for (1) the developers of atomistic ML models and (2) the researchers that aim to apply those methods. Although SchNetPack includes the implementations of ML force fields developed by some of us, it is not supposed to be a collection of methods. It

is rather a framework that provides commonly required functionality and building blocks to develop, evaluate, and run atomistic ML models. This enables developers to publish SchNetPack extensions as third-party packages, as demonstrated in the example of the generative model cG-SchNet in Sec. VI C. The main focus of this paper lies in the design principles of SchNetPack. For readers that are new to the field of atomistic ML, we recommend to read some of the plethora of excellent reviews that appeared over the previous years.<sup>2,4,6,18,91</sup>

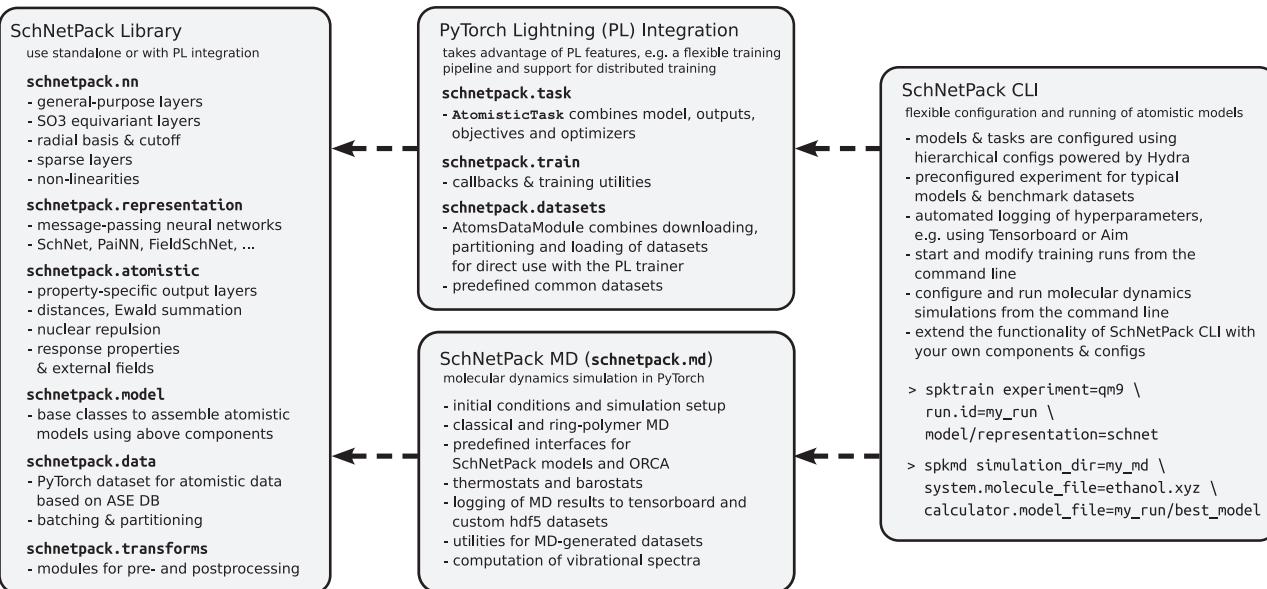
SchNetPack 2.0 is structured into components that can be used individually or as a unified framework. This makes it straightforward to combine some or all SchNetPack components with other PyTorch-based libraries, such as e3nn<sup>50</sup> or TorchMD.<sup>51</sup> Figure 1 gives an overview of the functionalities contained in each component. The core of the package is the SchNetPack library that consists of the tools required to load atomistic data and define neural network models. The library not only includes the implementations of representations such as SchNet and PaiNN but also provides commonly required layers to build custom atomistic representations. Furthermore, the SchNetPack library includes specific modules for the prediction of common targets, such as energies and response properties. The library can be used with pure PyTorch or a training framework of choice, as described in detail in Sec. II.

Alternatively, SchNetPack provides an integration of the library with the PyTorch Lightning (PL) framework.<sup>48</sup> This enables the use of a plethora of PL features, such as customizable training loops with callbacks, distributed training supporting various accelerators, and extensive support for experiment loggers such as Tensorboard<sup>52</sup> or Aim.<sup>53</sup> SchNetPack integrates with PL over the AtomisticTask, which is composed of models defined using the

SchNetPack library, training objectives, and optimizers. We further provide interfaces to popular datasets on the basis of Lightning-DataModule that enables automatic download and parsing of common benchmark datasets. The PL interface is described in detail in Sec. III.

Besides implementing neural networks with the Python API, SchNetPack 2.0 features a command-line interface (CLI) for composing models from the supplied or custom modules, which will be described in Sec. IV. It is powered by the Hydra<sup>54</sup> framework, which allows us to build hierarchical YAML configurations. The structure of this hierarchy is closely oriented to the SchNetPack PL integration so that the complex models and training tasks can be described. Configured training runs can be started and modified from the command line, which makes it easy to quickly scan a large number of hyperparameters and models. User extensions to SchNetPack can be directly incorporated into the existing CLI.

Finally, SchNetPack 2.0 contains a molecular dynamics code, which makes it possible to directly apply the SchNetPack models in simulations with little communication overhead. Like the rest of the code package, the MD suite is implemented in PyTorch offering full CUDA support. It retains the batch structure of the neural network toolbox, making it possible to simulate multiple systems in parallel. Building on this feature, the MD code implements an efficient way to perform ring-polymer MD simulations. A collection of thermostats and barostats is available to sample different thermodynamic ensembles. The MD code also features full CLI integration and a series of utilities to simplify logging and the analysis of simulation results, e.g., computation of different vibrational spectra. In addition, SchNetPack provides an interface to the molecular dynamics simulation software Large-scale Atomic/Molecular



**FIG. 1.** Overview of the five major components of the SchNetPack toolbox: the atomistic neural network library, PyTorch Lightning integration, command-line interface, and molecular dynamics code. The arrows indicate dependencies between the components, i.e., components that can be used independently of components on their right.

Massively Parallel Simulator (LAMMPS).<sup>52</sup> Thus, any force field model that has been trained in the SchNetPack framework can be used as a pair style in LAMMPS.

Code, documentation, and tutorials for SchNetPack are available on GitHub<sup>55</sup> and ReadTheDocs.<sup>56</sup>

## II. NEURAL NETWORK LIBRARY

The SchNetPack 2.0 library provides tools and functionality to build atomistic neural networks and process datasets of molecules and materials. We have designed the library so that it can be used with vanilla PyTorch, i.e., without the need to integrate with PyTorch Lightning or the Hydra configurations. Instead, we define common interfaces for datasets and models that make them ready to use with the other SchNetPack components.

A major change compared with SchNetPack 1.0 is that the data format is now fully sparse. This is achieved by concatenating all atoms of the entire batch instead of having a separate batch dimension in the input and output tensors. Thus, we no longer have to pad atomic environments with varying numbers of neighbors. This required a rewrite of all atomwise operations including the data pipeline and the message-passing and output layers. In the following, we will introduce the improved data pipeline, the pre- and post-processing modules, and the neural network models.

### A. Data pipeline

The SchNetPack data pipeline mainly consists of ASEAtomsData, adhering to the PyTorch dataset interface, and AtomsLoader, being a PyTorch data loader with a customized collate function for batching of atomistic data. An important feature of ASEAtomsData is that one can provide a list of preprocessing transforms. They are applied in sequence to single data instances before the latter are batched in the AtomsLoader. This is particularly useful for calculating neighbor lists, removing offsets, or casting properties. The transforms will be described in detail in Sec. II B. Figure 2 gives an overview of the data pipeline and the format of the data before and after batching.

As in SchNetPack 1.0, the default PyTorch dataset ASEAtomsData is based on the database of the Atomic Simulation Environment (ASE).<sup>57</sup> However, the internal format has slightly changed in that it expects a dictionary of units for all properties and the atom positions. This enables automatic unit conversion without requiring internal default units. We have included the script spkconvert to add this information to older datasets. The following code snippet demonstrates how to create a new dataset:

The atomistic systems are added by providing the lists of ASE Atoms and dictionaries of NumPy<sup>58</sup> arrays containing the corresponding properties. In addition, we may provide single-atom reference energies (atomrefs), which can be used for preprocessing the target properties. Both the single-atom energies and property units are stored as metadata in the ASE DB. Beyond that, additional custom metadata can be stored, such as predefined train/test splits in the code example above.

ASEAtomsData is an implementation of the abstract base class BaseAtomsData, which defines a general interface to SchNetPack

```
# create new dataset
new_dataset = ASEAtomsData.create(
    './new_dataset.db', distance_unit='Ang',
    property_unit_dict={
        'energy': 'kcal/mol', 'forces': 'kcal/mol/Ang'
    },
    atomrefs = {
        'energy': [0.0, -313.5150902000774, ...]
    }
)

atoms_list = [
    ase.Atoms(numbers=[1, 1, 8], positions=[...]), ...
]
property_list = [{{
    'energy': np.array([-450.2]),
    'forces': np.array([[0.8, -0.4, 1.3], ...])}
}]
new_dataset.add_systems(property_list, atoms_list)
# add metadata for custum train/test split
new_dataset.update_metadata(
    train_idx=[0, 12, 53, ...])

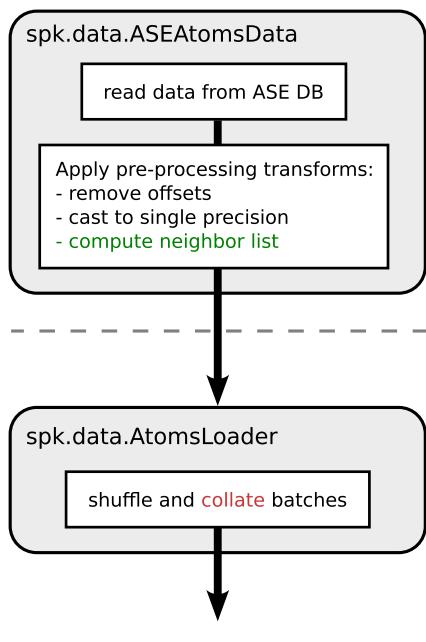
# create training dataset and retrieve an entry
train_data:ASEAtomsData = new_dataset.subset(
    new_dataset.metadata["train_idx"])
some_molecule = train_data[2]
```

datasets. This makes it possible to extend SchNetPack with custom data formats, for example, for distributed datasets or special data types such as wave function files. Independent of the concrete implementation of BaseAtomsData, the format of retrieved data is a dictionary mapping from strings to PyTorch tensors, as shown in the example in Fig. 2 (right).

The AtomsLoader batches the preprocessed inputs after optional shuffling. Since systems can have a varying number of atoms, the batch dimension for atomwise properties, such as positions and forces, runs over the atoms instead of the systems. The index of the corresponding system in the batch is encoded in the PyTorch tensor idx\_m. Index tensors, e.g., generated by neighbor lists (idx\_i, idx\_j), have to be treated differently since they refer to the atom indices within a single data example. Therefore, the collate function shifts the indices to refer to the correct position within the batch.

### B. Pre- and post-processing transforms

SchNetPack transforms are PyTorch modules that have no trainable parameters and are used for preprocessing of inputs or postprocessing of model results. Preprocessing transforms are applied before batching, i.e., they operate on single inputs. For example, virtually every SchNetPack model requires a preprocessing transform that constructs a neighbor list. However, different types of postprocessing may be demanded in the training and prediction phases. For example, a preprocessor might



```

aseAtoms = aseAtoms(number=[1, 1, 8], positions=[...])
{
  properties.Z: [1, 1, 8],
  properties.positions: [r1, r2, r3],
  properties.cell: zeros(3,3),
  properties.pbc: False,
  properties.energy: -1365,
  properties.idx_i: [0, 0, 1, 1, 2, 2],
  properties.idx_j: [1, 2, 0, 2, 0, 1],
  properties.offsets: zeros(6, 3),
}
  
```

```

{
  properties.Z: [1, 1, 8, 1, 1, 8],
  properties.positions: [r1, r2, r3, r4, r5, r6],
  properties.cell: zeros(2, 3, 3),
  properties.pbc: [False, False],
  properties.energy: [-1365, -1264],
  properties.idx_i: [0, 0, 1, 1, 2, 2,
                    3, 3, 4, 4, 5, 5],
  properties.idx_j: [1, 2, 0, 2, 0, 1,
                    4, 5, 3, 5, 3, 4],
  properties.offsets: zeros(12, 3),
  properties.idx_m: [0, 0, 0, 0, 0, 0,
                    1, 1, 1, 1, 1, 1]
}
  
```

**FIG. 2.** The SchNetPack data pipeline. Left: ASEAtomsData provides an interface to the data stored in an ASE DB and applies a sequence of preprocessing transforms. The AtomsLoader loads these data with multiprocessing and builds batches to be passed to the model. Right: example of the input dictionaries after preprocessing (top) and batching (bottom). All values of the dictionary are PyTorch tensors.

need to perform data augmentation during training, but not during predicting. Another example is the SkinNeighborList that takes advantage of the structural similarity of sequential examples, which can be encountered in prediction tasks such as molecular dynamics simulations or structure relaxation, but not during training.

Postprocessing transforms act on batches in the result dictionary and are part of the AtomisticModel described in Sec. II C. However, the loss function is supposed to be independent of post-processing. Thus, these transforms are only enabled for prediction but not during training and evaluation. The currently supported pre- and post-processing transforms are listed in Table I.

In the following, we illustrate the usage of transforms in the use case of casting between single and double precisions: On one hand, double precision is required to accurately represent the comparatively small energy differences compared with the much larger scale of the total energy. On the other hand, single or even half-precision is needed for the fast processing of neural networks

on graphics processing units (GPUs). We solve this by applying a chain of pre- and post-processing transforms. First, offsets are removed from the energy targets using the RemoveOffsets transform at double precision. Only then do we use the CastTo32 transform to cast all floating point inputs to a single precision. For instance, one might supply precomputed single-atom reference energies  $E_Z$  for all atomic numbers  $Z$  in the data to be subtracted from the total energy, effectively predicting the atomization energy instead:

$$E_{\text{atomization}} = E - \sum_{i=0}^{n_{\text{atoms}}} E_{Z_i}$$

With these offsets removed, single precision can represent the lower decimals with sufficient accuracy in many cases. The preprocessing modules can be set as follows:

Since neural network is trained to predict the target without offsets, postprocessing modules are employed to recast the neural network prediction to double precision (CastTo64) and only afterward adding the single-atom energies (AddOffsets). Should single-precision still not be sufficient, one can also switch to double-precision training, if it is supported by the hardware. Obtaining single-atom references or other data-dependent initialization can be taken care of automatically when using PyTorch Lighting, as described in Sec. III.

### C. Atomistic models

All neural networks implemented with SchNetPack are supposed to subclass AtomisticModel. It is a PyTorch module with additional functionality that is commonly required for atomistic

---

```

atomrefs = train_data.atomrefs
train_data.transforms = [
    RemoveOffsets(
        remove_atomrefs=True,
        atomrefs=train_data.atomrefs
    ),
    MatScipyNeighborList(cutoff=5.),
    CastTo32(),
]
  
```

---

**TABLE I.** List of pre- and post-processing transforms.

| Category         | Transform            | Usage    | Description  |
|------------------|----------------------|----------|--|
| Neighbor lists   | MatScipyNeighborList | Pre      | Neighbor list implementation based on Matscipy. <sup>59</sup> This should be preferred   |
|                  | ASENeighborList      | Pre      | Neighbor list based on atomic simulation environment   |
|                  | TorchNeighborList    | Pre      | Neighbor list implemented in PyTorch   |
|                  | CachedNeighborList   | Pre      | Wrapper for other neighbor list transforms that caches the results   |
|                  | SkinNeighborList     | Pre      | Wrapper around neighbor list transform that only recalculates neighbor indices after atom positions change more than a given threshold.<br>This can be useful for structure relaxation |
|                  | FilterNeighbors      | Pre      | Filter previously calculated neighbor indices  |
|                  | CountNeighbors       | Pre      | Count and store number of neighbors for each atom  |
|                  | WrapPositions        | Pre      | Wrap atom position into periodic cell  |
| Casting          | CastMap              | Pre/Post | Cast all properties according to supplied type map   |
|                  | CastTo32             | Pre/Post | Cast all double precision inputs to single precision   |
|                  | CastTo64             | Pre/Post | Cast all single precision inputs to double precision   |
| Scale and offset | ScaleProperty        | Pre/Post | Scale an input or result by data mean, standard deviation or given factor  |
|                  | RemoveOffsets        | Pre/Post | Remove single-atom reference and/or mean from an input or result   |
|                  | AddOffsets           | Pre/Post | Add single-atom reference and/or mean to an input or result  |

machine learning. In particular, it offers support for the previously described postprocessors, filtering of result dictionaries, and a convenient mechanism to initialize and collect automatic derivatives.

Although `AtomisticModel` can be used to implement neural networks for a broad class of tasks, SchNetPack also includes its more structured `NeuralNetworkPotential`. This is tailored toward ML potentials and makes use of all the features of `AtomisticModel`. We recommend using `NeuralNetworkPotential` whenever possible as it allows for easier integration with the SchNetPack CLI. On the other hand, `AtomisticModel` may be employed for more general tasks. This will be demonstrated in the example of the generative model cG-SchNet in Sec. VI C.

Figure 3 depicts the processing flow of `NeuralNetworkPotential`. The main concept is to pass along an input dictionary that is modified by submodules. This enables the definition of complex neural networks with multiple inputs and outputs as just a sequence of modules. An immediate advantage is that such a model can be easily composed using configuration files and the SchNetPack CLI.

Many neural network potentials are employed for predicting response variables such as atomic forces or electric multipoles. Therefore, it is often required to calculate the derivatives of model outputs w.r.t. input variables. The derivative tracking of these input variables needs to be activated before any of the modules are applied. The method `AtomisticModel.collect_derivatives()` can be called during initialization to scan all submodules of `NeuralNetworkPotential` for required derivatives. Based on this list, we enable the generation of corresponding backward graphs using the `AtomisticModel.initialize_derivatives()` method during the forward pass.

In the next step, the dictionary is passed to a sequence of so-called `input_modules`. These may carry out preparatory steps such as

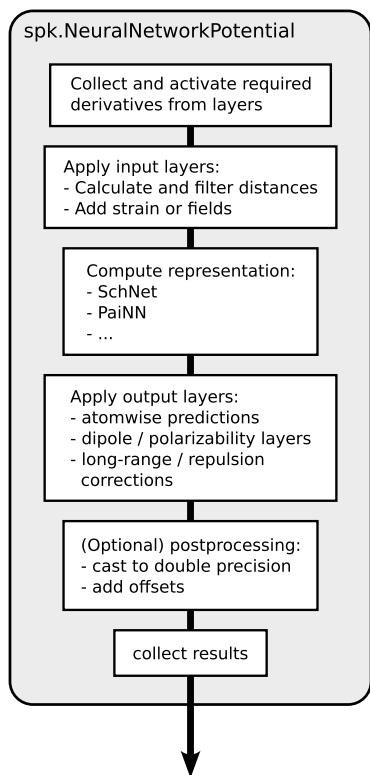
calculating distances, applying strain to a lattice or adding auxiliary inputs such as external fields. The inputs then are passed on to the `representation` module, which constructs atomwise features based on the input dictionary. This is often achieved by a message-passing neural network that is designed to be equivariant to the symmetries of an atomistic structure (see Sec. II D). The resulting features can be scalars, vectors, or general multipoles and are stored under a corresponding key in the dictionary.

Finally, a sequence of `output_modules` computes the final results. For example, energies are often predicted as a sum over atomwise energy contributions, which is achieved by the `Atomwise` module. We refer to both input and output modules as *atomistic modules*, since they are usually tailored to the specifics of atomistic data—in contrast to the general purpose layers such as convolutions. We describe the supported operations in detail in Sec. II E.

Before the results are returned, `NeuralNetworkPotential` makes use of two more features of `AtomisticModel`: First, a sequence of postprocessors is applied, if they have been enabled. This is usually only done in prediction mode, but not during training. Second, the input dictionary was updated without removing information as it passed through the model. Therefore, it still contains all the raw inputs and intermediate features. The `extract_outputs` method filters only the results that are supposed to be returned by the model. This is achieved in a semi-automatic fashion by scanning all submodules for potential model outputs during initialization.

#### D. Message-passing and equivariant neural networks

SchNetPack provides the tools to build various atomistic machine-learning models, even beyond neural networks. However,



```

# initialize derivatives for response properties
inputs = self.initialize_derivatives(inputs)

for mod in self.input_modules:
    inputs = mod(inputs)

inputs = self.representation(inputs)

for mod in self.output_modules:
    inputs = mod(inputs)

inputs = self.postprocess(inputs)

results = self.extract_outputs(inputs)

results = {
    properties.energy: [-1365.2, -1264.1],
    properties.dipole_moment: [...]
}
  
```

**FIG. 3.** Data processing in NeuralNetworkPotential with corresponding codes.

our focus remains on end-to-end neural networks that build atomwise representations. In recent years, the two concepts that have dominated this field are neural message-passing<sup>9,63</sup> and equivariant neural networks.<sup>61,62</sup> SchNetPack comes with four implemented atomistic representations:

- **SchNet:**<sup>10,21</sup> The name-giving continuous-filter convolutional network motivated the creation of SchNetPack. It uses rotationally invariant parameter tensors, in neural networks often referred to as filters. Although not being the most accurate model, five years after it was first proposed, it is pretty lightweight.
- **FieldSchNet:**<sup>64</sup> An extension of SchNet that makes use of atomic dipole features to handle external fields to model response properties and solvent effects.
- **PaiNN:**<sup>65</sup> The successor to SchNet that uses equivariant message passing in Cartesian space.
- **SO3net:** A minimalist SO(3)-equivariant neural network in the spirit of Tensor Field Networks<sup>61</sup> or NequIP<sup>66</sup> that showcases the spherical harmonics and Clebsch–Gordan tensor product modules of SchNetPack.

The architecture of SO3net is explained in detail in the [supplementary material](#). For a detailed description of the other models, please refer to the respective publications. Table II gives an overview of related modules that we provide for building atomistic representations. This includes radial basis and cutoff functions,

nonlinearities and SO(3) equivariant layers. Naturally, these are only additions to the large variety of modules already included in PyTorch, e.g., the SiLU nonlinearity,<sup>67</sup> which has been employed in many recent atomistic neural networks.<sup>15,65,66</sup>

By convention, the atomistic representations in SchNetPack have the shape  $(n_{\text{atoms}}, [n_{\text{spatial}}] n_{\text{features}})$ , where the first dimension runs over all atoms in the batch, the second is reserved for optional directional channels of equivariant representations and the last is reserved for feature channels. In this format, it is straightforward to define the message-passing operations using the atom and neighbor indices obtained from neighbor list transforms during preprocessing. For example, the continuous-filter convolution of SchNet<sup>10</sup> can be implemented as follows:

The second (optional) dimension of atomwise representations is reserved for the directional features of equivariant neural networks. For the vectorial features, e.g., in the case of PaiNN, this corresponds to the Cartesian  $(x, y, z)$ -directions ( $n_{\text{spatial}} = 3$ ). In the case of SO(3)-equivariant features, the dimensions correspond to the real spherical harmonics  $Y_{l,m}$  as follows:  $[Y_{0,0}, Y_{1,-1}, Y_{1,0}, Y_{1,1}, Y_{2,-2}, Y_{2,-1}, \dots]$ .

For the implementation of tensor product modules required for SO(3)-equivariant models, the Clebsch–Gordan coefficients  $C_{l_1 m_1 l_2 m_2}^{lm}$  are precomputed during initialization and stored in a sparse format with the non-zero coefficients clebsch\_gordan and three combined index tensors idx\_in\_1, idx\_in\_2 and idx\_out corresponding to the tuples  $(l_1, m_1)$ ,  $(l_2, m_2)$ , and  $(l, m)$ ,

**TABLE II.** List of modules for message-passing and equivariant neural networks.

| Category           | Module  | Description  |
|--------------------|---|--|
| Radial basis       | GaussianRBF <sup>9,60</sup>                                 | Gaussian radial basis functions centered on an equidistant grid in the interval [start, cutoff]  |
|                    | GaussianRBCentered <sup>60</sup><br>BesselRBF <sup>15</sup> | Gaussian radial basis functions centered at zero with varying widths<br>Zeroth-order Bessel radial basis functions: $f(r) = \sin(\omega r)/r$                |
| Cutoff             | CosineCutoff <sup>60</sup>                                  | $f(r) = \frac{1}{2} \left[ 1 + \cos \left( \frac{\pi r}{r_{\text{cutoff}}} \right) \right], \quad r < r_{\text{cutoff}}$                                     |
|                    | MollifierCutoff   | $f(r) = \exp \left[ 1 - \frac{1}{1 - \left( \frac{r}{r_{\text{cutoff}}} \right)^2} \right], \quad r < r_{\text{cutoff}}$                                     |
| Nonlinearity       | shifted_softplus <sup>10</sup>                              | $f(x) = \ln(0.5 + 0.5e^{-x})$  |
| Sparse             | scatter_add   | Sum over a sparse dimension, e.g., for sums of atoms and neighbors as well as Clebsch-Gordon tensor products   |
| SO(3) equivariance | RealSphericalHarmonics                                      | Generates the real spherical harmonics for a batch of unit vectors   |
|                    | SO3TensorProduct <sup>61</sup>                              | $f(x, y)_{(lm)} = \sum_{l_1 m_1 l_2 m_2} x_{l_1 m_1} y_{l_2 m_2} C_{l_1 m_1 l_2 m_2}^{lm}$   |
|                    | SO3Convolution <sup>61</sup>                                | $f(x)_{i(lm)} = \sum_{j \in \text{nbh}[i]} \sum_{l_1 m_1 l_2 m_2} x_j_{(l_1 m_1)} W_{l_2}(r_{ij}) Y_{l_2 m_2}(\vec{r}_{ij}/r_{ij}) C_{l_1 m_1 l_2 m_2}^{lm}$ |
|                    | SO3GatedNonlinearity  | $f(x)_{i(lm)} = x_{i(l,m)} \sigma(x_{i(0,0)})$   |
|                    | SO3ParametricGatedNonlinearity <sup>62</sup>                | $f(x)_{i(lm)} = x_{i(l,m)} \sigma(g_\theta(x_{i(0,0)}))$   |

---

```
# inputs:
# x: atom-wise representation
# Wij: radial convolution kernels
# idx_i, idx_j: neighbor list indices
x_j = x[idx_j]
x_ij = x_j * Wij
y = scatter_add(x_ij, idx_i, dim_size=x.shape[0])
```

---

respectively. This enables the fast calculation of tensor products. For example, the SO3TensorProduct module calculating

$$f(x, y)_{(lm)} = \sum_{l_1 m_1 l_2 m_2} x_{l_1 m_1} y_{l_2 m_2} C_{l_1 m_1 l_2 m_2}^{lm},$$

can be implemented as follows:

---

```
# x, y: atom-wise features [n_atoms, (lmax + 1)^2, n_features]
h = x[:, idx_in_1, :] \
    * y[:, idx_in_2, :] \
    * clebsch_gordan[None, :, None]
z = scatter_add(h, idx_out,
                dim_size=(lmax + 1) ** 2, dim=1)
```

---

The combined indices `idx_in_1` and `idx_in_2` select the spherical basis functions to be multiplied by the non-zero Clebsch-Gordan coefficients. The third index tensor is used for the summation of terms mapping to the same  $(l, m)$  using `scatter_add` analogous to

the atom accumulation in the message passing shown above. Similarly, SO3Convolutions or other custom SO(3)-equivariant modules can be implemented in a straightforward fashion.

## E. Atomistic modules

*Atomistic modules* are specific to the properties to be predicted and informed by the regularities of the underlying physics. In contrast to the general purpose layers, such as non-linearities or convolutions, they are supposed to directly operate on the input dictionary that is passed through the model, as described in Sec. II C. This makes it easy to compose complex atomistic neural networks. Thus, the input and output modules of a `NeuralNetworkPotential` are usually atomistic layers.

Table III describes the atomistic modules currently supported by SchNetPack. A range of output modules is concerned with predicting atomistic properties from the previously generated representations. The most common is `Atomwise`, which predicts a property as a sum of atomwise contributions, e.g., when predicting potential energy surfaces.<sup>8,9,16</sup> We have also included specialized layers for tensorial properties such as dipole moments and polarizability tensors. Beyond that, it is often helpful for generalization to include physically inspired terms in the energy prediction. This includes electrostatic modules and ZBL potentials<sup>70</sup> for nuclear–nuclear repulsion.

Virtually, every neural network potential requires the `PairwiseDistances` module as an input layer, which makes use of the indices calculated by a neighbor list during preprocessing. The distance calculation being part of the model enables straightforward automatic differentiation w.r.t. to atom positions. The `Forces` module provides atomic forces and the stress tensor as derivatives w.r.t.

**TABLE III.** List of atomistic modules for preparing the raw inputs and predicting various properties.

| Category   | Module  | Description   |
|------------|---|---|
| Distances  | <code>PairwiseDistances</code><br><code>FilterShortRange</code>   | Compute pair-wise distances from indices provided by a neighbor list transform<br>Separate distances below a short-range cutoff from all supplied distances   |
| Properties | <code>Atomwise</code> <sup>16</sup><br><br><code>DipoleMoment</code> <sup>65,68,69</sup><br><br><code>Polarizability</code> <sup>65</sup><br><br><code>Aggregation</code> | Predicts a property from atom-wise contributions and accumulates global prediction using sum or average, e.g., for the energy $E = \sum_{i=1}^{n_{\text{atoms}}} E(x_i)$<br>Predicts dipole moments from latent partial charges and (optionally) local, atomic dipoles: $\vec{\mu} = \sum_{i=1}^{n_{\text{atoms}}} q(x_i) \vec{r}_i + [\vec{\mu}_{\text{atomic}}(\vec{x}_i)]$<br>Predicts polarizability using the tensor rank factorization: $\alpha = \sum_{i=1}^N \alpha_0(x_i) I_3 + \vec{v}(\vec{x}_i) \otimes \vec{r}_i + \vec{r}_i \otimes \vec{v}(\vec{x}_i)$<br>Aggregate predictions from multiple atomistic modules to a single output variable, e.g., calculating the energy prediction as the sum of short-range and long-range energies |
| Potentials | <code>EnergyCoulomb</code><br><code>EnergyEwald</code><br><br><code>ZBLRepulsionEnergy</code> <sup>70,71</sup>  | Compute Coulomb energy from a set of (latent) point charges<br>Compute the Coulomb energy of a set of (latent) point charges inside a periodic box using Ewald summation<br>Computes a Ziegler-Biersack-Littmark-style repulsion energy   |
| Response   | <code>StaticExternalFields</code><br><br><code>Strain</code><br><br><code>Forces</code><br><br><code>Output</code>  | Input module for setting up dummy external fields.<br>These do not receive inputs, but are only used to calculate response properties with autograd<br>Input module for setting up dummy strain. It does not receive inputs, but is only used to calculate the stress tensor using autograd<br>Output module that predicts forces and stress as response of the energy prediction w.r.t. the atom positions and strain<br>Output module that computes different response properties by taking derivatives of an energy model. Supports forces, stress, Hessian, dipole moment (and its derivatives) polarizability (and its derivatives), shielding tensor, and nuclear spin–spin coupling  |

atomic positions and strain. Beyond that, SchNetPack includes the Response module, which additionally supports response properties w.r.t. external (electric or magnetic) fields and higher-order derivatives, e.g., for polarizability or shielding tensors. The FieldSchNet example in Sec. VI B demonstrates how this can be employed in practice.

### III. PYTORCH LIGHTNING INTEGRATION

Although it is possible to use the previously described SchNetPack library on its own, a third-party framework that takes care of the boilerplate code required for training and validation can significantly speed up the development process. We chose PyTorch Lightning<sup>48</sup> as the default training framework for SchNetPack, as it supports a wide variety of hardware devices and distribution strategies. In the following, we describe how LightningModule and LightningDataModule are employed to predefine common datasets, tasks, and workflows.

The PyTorch Lightning trainer expects a `LightningModule` that defines the learning task, i.e., a combination of model definition, objectives, and optimizers. SchNetPack provides the `AtomisticTask`,

which integrates the `AtomisticModel`, as described in Sec. II C, with PyTorch Lightning. The task configures the optimizer; defines the training, validation, and test steps; and calculates the loss and logs training metrics. For this purpose, `AtomisticTask` expects a list of `ModelOutputs`, which map a target property to a key in the results dictionary of the `AtomisticModel`. In addition, `ModelOutput` requires a loss function and (optionally) a list of Metrics that are tracked during training and validation. Given a model `my_neural_network_potential` that stores the predictions energy and forces in the result dictionary, the task of training a neural network potential can be defined as follows:

To define a dataset, we subclass PyTorch Lightning data modules with `AtomsDataModule`. This combines the data classes introduced in Sec. II A with codes for preparation, setup, and partitioning into train, validation, and test splits. The default splitting strategy is the random sampling; however, other strategies, such as sub-sampling predefined partitions or keeping certain groups of structures in the same partition, are supported as well. One may provide separate preprocessing transforms for train, validation, and test splits if needed, e.g., when data augmentation is only required for the training data.

```
output_energy = ModelOutput(  
    name="energy",  
    loss_fn=torch.nn.MSELoss(),  
    loss_weight=0.01,  
    metrics={"MAE": torchmetrics.MeanAbsoluteError()})  
  
output_forces = ModelOutput(  
    name="forces",  
    loss_fn=torch.nn.MSELoss(),  
    loss_weight=0.99,  
    metrics={"MAE": torchmetrics.MeanAbsoluteError()})  
  
task = AtomisticTask(  
    model=my_neural_network_potential,  
    outputs=[output_energy, output_forces],  
    optimizer_cls=torch.optim.AdamW,  
    optimizer_args={"lr": 1e-4})  
)
```

Beyond that, data modules may take care of setting up the data for distributed training, e.g., copying the data to local storage. We provide specialized AtomsDataModules for common benchmark sets, which automatically download and parse the data. Currently, supported datasets include QM9,<sup>72</sup> (r)MD17,<sup>22,73</sup> MD22,<sup>74</sup> OMDB,<sup>75</sup> and the MaterialsProject.<sup>76</sup> Additional datasets can be added by sub-classing AtomsDataModule and overriding the prepare\_data method with custom code for downloading and parsing.

Here is an example of how the MD17 datamodule can be used:

```
ethanol_data = MD17(  
    "/path/to/data.db",  
    molecule='ethanol',  
    batch_size=10,  
    num_train=1000,  
    num_val=1000,  
    transforms=[  
        MatScipyNeighborList(cutoff=5.),  
        RemoveOffsets(MD17.energy, remove_mean=True),  
        CastTo32()  
    ],  
    num_workers=1  
)  
ethanol_data.prepare_data()  
ethanol_data.setup()  
  
# iterate over training batches  
for batch in ethanol_data.train_dataloader():  
    print(batch)
```

The passed transforms are applied to all partitions in this case, and RemoveOffsets is automatically initialized with the training data statistics. Manual calling of prepare\_data, which downloads

and parses the data and setup, which creates and loads the partitions, is necessary here because we retrieve the data loader and iterate over the training data. Instead, one may pass the data module directly to the PyTorch Lightning trainer class, which ensures that prepare\_data is called exactly once. However, the setup method is called in every process of distributed training.

Finally, we put everything together by passing the task and data module to the PyTorch Lightning trainer, which executes the training loop:

```
trainer = pl.Trainer()  
trainer.fit(task, datamodule=ethanol_data)
```

The training process can be adapted by callbacks and loggers and specifying options to train on several (distributed) devices. Please refer to the PyTorch Lightning documentation<sup>77</sup> for more information.

#### IV. CONFIGURATION AND COMMAND-LINE INTERFACE

SchNetPack training runs can be defined using the hierarchical configurations framework Hydra.<sup>54</sup> This enables the configuration of complex neural network potentials using YAML files, provides powerful command-line tools, and makes it easy for developers to extend SchNetPack with external codes.

The main command of the SchNetPack CLI is spktrain, which creates a new run directory and starts the training of a configured model. We refer to a predefined configuration including model, task, data, etc. as an *experiment*, which can be started as follows:

```
spktrain experiment=qm9_atomwise
```

This starts the training with the default settings for energies of the QM9 dataset. The script first prints the flattened config of the run, i.e., the config when specified in a single YAML file. Figure 4 instead shows the hierarchical experiment configuration that this has been derived from.

##### A. Structure of the configuration

The structure of configurations is heavily oriented on the building blocks introduced in Secs. II and III. Due to its hierarchy, we only have to override the defaults and make use of reusable, predefined config groups, e.g., for the model or the dataset. The major config groups of SchNetPack are:

- **run**: Definition of run-specific variables, such as the run id and directories, where the metrics are logged and the trained model and the data will be stored. If no run id is given, SchNetPack will create a unique hash.
- **globals**: Custom variables to be used across the whole config can be added here. This is possible through the use of the interpolation syntax \${globals.variable}.

```

1 defaults:
2   - override /model: nnp
3   - override /data: qm9
4
5 run:
6   experiment: qm9_${globals.property}
7
8 globals:
9   cutoff: 5.
10  lr: 5e-4
11  property: energy_U0
12  aggregation: sum
13
14 data:
15   transforms:
16     - _target_: schnetpack.transform.SubtractCenterOfMass
17     - _target_: schnetpack.transform.RemoveOffsets
18     property: ${globals.property}
19     remove_atomrefs: True
20     remove_mean: True
21     - _target_: schnetpack.transform.MatScipyNeighborList
22     cutoff: ${globals.cutoff}
23     - _target_: schnetpack.transform.CastTo32
24
25 model:
26   output_modules:
27     - _target_: schnetpack.atomistic.Atomwise
28     output_key: ${globals.property}
29     n_in: ${model.representation.n_atom_basis}
30     aggregation_mode: ${globals.aggregation}
31   postprocessors:
32     - _target_: schnetpack.transform.CastTo64
33     - _target_: schnetpack.transform.AddOffsets
34     property: ${globals.property}
35     add_mean: True
36     add_atomrefs: True
37
38 task:
39   outputs:
40     - _target_: schnetpack.task.ModelOutput
41     name: ${globals.property}
42     loss_fn:
43       - _target_: torch.nn.MSELoss
44     metrics:
45       mae:
46         - _target_: torchmetrics.regression.MeanAbsoluteError
47       rmse:
48         - _target_: torchmetrics.regression.MeanSquaredError
49         squared: False
50     loss_weight: 1.

```

**FIG. 4.** SchNetPack experiment config for prediction tasks of QM9 properties with the Atomwise output layer.

- **data:** Definition of the dataset as described by AtomsDataModule (see Sec. III).
- **model:** Definition of the AtomisticModel (see Sec. II C).
- **task:** Configuration of the AtomisticTask, including model outputs, losses, and optimizers.
- **trainer:** Arguments to be passed to the PyTorch Lightning Trainer.
- **callbacks:** A list of callbacks to be passed to the Trainer.
- **logger:** Configuration of training metric loggers, such as Tensorboard<sup>52</sup> or Aim.<sup>53</sup>
- **seed:** Sets the random seed for PyTorch and Pytorch Lightning.

The first lines of the configuration in Fig. 4 show how the model and data are overridden with the default config templates (nnp and qm9) for the neural network potential and the QM9 dataset, respectively. The data and model templates are further modified

in lines 14–36 by specifying pre- and post-processing transforms, output modules, and model outputs with losses and metrics to be tracked. The special key `_target_` enables the automatic instantiating of objects. For example, in the list of model outputs in Fig. 4 (lines 26–30), the Atomwise module is initialized with the given parameters. Here, the target object is not restricted to be a SchNetPack module but can also be a class provided by a third-party Python package. This makes it straightforward to extend SchNetPack with custom layers, losses, and models.

## B. Modifying configurations from the command line

We have shown above how a training run specified by the experiment config can be started from the command line. Beyond that, the configuration of a run can also be directly modified using the CLI. For example, training neural network potentials with a different representation and using a larger learning rate than the default can be achieved as follows:

---

```
spktrain experiment=qm9_atomwise \
  model/representation=schnet globals.lr=1e-3
```

---

Note that when setting the config groups to a preconfigured template, a slash “/” is used, whereas when setting a value in the YAML, the dot “.” is used to navigate the hierarchy. Finally, custom experiment configs can be added by setting a user config directory:

---

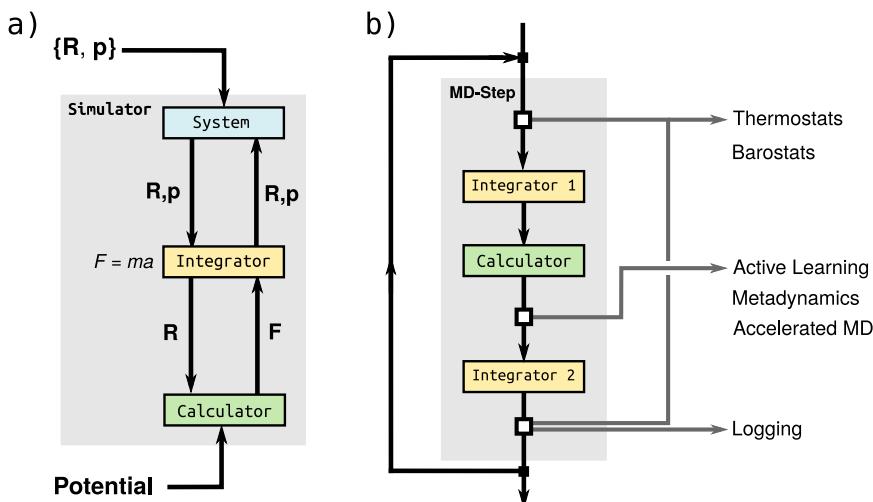
```
spktrain -config-dir=/configdir \
  experiment=my_experiment
```

---

The config directory needs to follow the structure of the config, e.g., the experiment should be located at `/configdir/experiment/my_experiment.yaml`.

## C. Extending SchNetPack

The modular design of SchNetPack configurations allows for a seamless extension of the framework with additional models and learning tasks, including their integration into the CLI. For example, the data pipeline and training flow of the framework could be kept while changing the representation block of a neural network potential. This enables a quick and standardized comparison of different approaches. To this end, one may implement a small python package containing the network building blocks and a custom configuration file for instantiating the new representation. The training can be started with spktrain using the familiar CLI. By providing Hydra with the location of the additional configuration files in the extension package, the representation can simply be switched to the new network as if it was part of SchNetPack. This means that developers are able to build atomistic ML models on top of SchNetPack by exchanging only selected building blocks and avoiding as much boilerplate code as possible. The example in Sec. VI C shows that this is not only possible with neural network potentials but also with



**FIG. 5.** (a) Basic MD workflow. (b) Internal structure of a single simulation step, indicating points at which different hooks can be applied to modify a simulation.

more complex atomistic learning tasks such as a generative model for molecules.

## V. MOLECULAR DYNAMICS SIMULATIONS

In addition to the neural network library, SchNetPack 2.0 contains the schnetpack.md code for carrying out molecular dynamics simulations. This MD environment is structured in a modular way to facilitate the development and interfacing with different ML potentials. Since all routines are implemented in PyTorch, models based on `AtomisticModel` can be used with minimal communication overhead and full GPU capabilities.

The MD code performs several core tasks during each simulation step. It keeps track of the positions  $R$  and momenta  $p$  of all nuclei, computes the forces  $F$  acting on them, and uses the latter to integrate the equations of motion. In SchNetPack, these tasks are distributed between different modules, which are sketched in Fig. 5(a). The `md.System` class contains all information on the current system state (e.g., nuclear positions and momenta). The `md.Integrator` propagates the positions and momenta of the system. To carry out this update, nuclear forces are required. These are computed by an `md.Calculator`, which takes the positions of atoms and returns the forces due to the potential. Typically, the Calculator connects to a previously trained machine-learning model. All these modules are linked together in the `md.Simulator` class, which contains the main MD loop and calls the three previous modules. The simulator can further be modified with so-called simulation hooks to control aspects like sampling (e.g., thermostats or barostats) or logging [Fig. 5(b)]. In the following, we will describe the MD modules in detail.

### A. System

The `md.System` class keeps track of the state of the simulated system and collects all associated quantities (e.g., positions, momenta, atom types, and simulation cells). This information is stored in multi-dimensional PyTorch tensors, which makes

it possible to vectorize many operations and e.g., simulate several different molecules and different replicas of a molecule in a single step. The shape convention of the system tensors is similar to the one used in the atomistic representations, with an added replica dimension ( $n_{\text{replicas}}, n_{\text{atoms}}, \dots$ ). This replica dimension collects different replicas of the same molecules and can be used, for example, in ring-polymer MD simulations. In addition, the `md.System` class provides utilities for computing different quantities relevant to MD simulations, such as temperature and pressure.

Structural information can be added to a `md.System` instance via the `load_molecules` function using ASE `Atoms` objects. The function operates on either a single object or a list of `Atoms` and automatically takes care of simulation cells and periodic boundary conditions. If a list is provided, multiple molecules can be loaded at once. During loading, it is necessary to specify the number of replicas and the units of length used for the input structures. The latter is required, since MD codes use their own internal unit system and automatically convert the inputs. The creation of a `md.System` instance is demonstrated in the following example:

```
# Read structures with ASE
molecule = ase.io.read("molecule.xyz")

# Create system instance and load molecule
md_system = schnetpack.md.System()
md_system.load_molecules(
    molecule,
    n_replicas=1,
    position_unit_input="Angstrom"
)
```

By default, momenta are set to zero on creating a `System`. If desired, a `md.Initializer` can be used to draw the momenta from

different distributions corresponding to a certain temperature. The following example uses the `UniformInit` initializer, which draws momenta from a random uniform distribution and rescales them to a certain temperature. Other routines, e.g., drawing from a Maxwell–Boltzmann distribution, are also available.

---

```
md_initializer = md.UniformInit(
    300, # system temperature in K
    remove_center_of_mass=True,
    remove_translation=True,
    remove_rotation=True,
)

# Initialize the system momenta
md_initializer.initialize_system(md_system)
```

---

A `md.Initializer` may also be used to center the molecular structure on the center of mass and remove all translational and rotational components of the momenta, as demonstrated in the code snippet above.

## B. Integrator

As the name suggests, the `md.Integrator` is used to integrate the equations of motion based on the nuclear forces. Two integration schemes are implemented in SchNetPack 2.0: The first `md.VelocityVerlet` implements the Velocity Verlet algorithm that evolves the system in a purely classical manner. The second integrator `md.RingPolymer` performs ring-polymer MD simulations, which recover a certain degree of nuclear quantum effects. The basic Velocity Verlet integrator is implemented in a three-step scheme,

$$\mathbf{p}(t + \frac{\Delta t}{2}) = \mathbf{p}(t) + \frac{\Delta t}{2} \mathbf{F}[\mathbf{R}(t)], \quad (1)$$

$$\mathbf{R}(t + \Delta t) = \mathbf{R}(t) + \Delta t \frac{\mathbf{p}(t + \frac{\Delta t}{2})}{\mathbf{m}}, \quad (2)$$

$$\mathbf{p}(t + \Delta t) = \mathbf{p}(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2} \mathbf{F}[\mathbf{R}(t + \Delta t)], \quad (3)$$

$\mathbf{m}$  contains the masses of all atoms and  $\Delta t$  is the integration time step. The `Integrator 1` block in Fig. 5(b) collects the first two steps, whereas `Integrator 2` corresponds to the third step. The ring-polymer MD integrator uses a modified version of the second step.<sup>84</sup> It transforms the positions and momenta of the ring-polymer beads into their normal mode representations  $\tilde{\mathbf{R}}$  and  $\tilde{\mathbf{p}}$  and propagates both according to

$$\begin{bmatrix} \tilde{\mathbf{p}}_k \\ \tilde{\mathbf{R}}_k \end{bmatrix} = \begin{bmatrix} \cos \omega_k \Delta t & -\mathbf{m} \omega_k \sin \omega_k \Delta t \\ \sin \omega_k \Delta t & \mathbf{m} \omega_k \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{p}}_k \\ \tilde{\mathbf{R}}_k \end{bmatrix}. \quad (4)$$

The subscript  $k$  indicates the respective normal mode and  $\omega_k$  is the associated normal mode frequency. Before momenta and positions

are updated in Eq. (3), both are transformed back into the original bead representation. Special NPT variants of the Velocity Verlet and ring-polymer integrator are provided for performing constant pressure simulations. To initialize an `md.Integrator`, one has to specify the integration time step in units of femtoseconds. In the following example, we use the Velocity Verlet algorithm with a time step of  $\Delta t = 0.5$  fs:

---

```
time_step = 0.5 # fs

# Set up the integrator
md_integrator = md.VelocityVerlet(time_step)
```

---

## C. Calculator

The nuclear forces required for the integrator are provided by a `md.Calculator`. It serves as an interface between a computation method (e.g., the neural network potential) and the MD code. The calculator takes the current positions of the `md.System` and other structural properties (simulation cells, etc.) and converts them to a format suitable for the computation method. Once all requested properties (e.g., forces) have been computed, the `md.Calculator` collects the results and reshapes the tensors back into the `md.System` format. Different custom calculators can be derived from the `MDCalculator` base class. SchNetPack comes with several predefined calculators, such as the `SchNetPackCalculator` that can be directly used with trained SchNetPack models, the `OrcaCalculator` for the ORCA quantum chemistry package,<sup>78</sup> and ensemble variants of these calculators for adaptive sampling. In the following, we describe the basic usage of the `SchNetPackCalculator`.

Since core of the calculator is a SchNetPack `AtomisticModel`, interatomic distances need to be computed during the MD. This is done with the `NeighborListMD` wrapper, which can be used with any neighbor list transform (Table I). It takes the basic neighbor list module, a cutoff radius, and a buffer region as input (both use the same length units as the model). The introduction of a buffer region improves performance, since a MD neighbor list only needs to be recomputed once the structural changes exceed the buffer region. In the following, we use the `MatScipyNeighborlist`, with a 5 Å cutoff and 2 Å buffer region:

---

```
from schnetpack.md.neighborlist_md \
    import NeighborListMD
from schnetpack.transform \
    import MatScipyNeighborList

md_neighborlist = NeighborListMD(
    cutoff=5.0, # cutoff
    cutoff_shell=2.0, # buffer region
    MatScipyNeighborList,
)
```

---

To initialize `md.SchNetPackCalculator`, we need to provide it with the path to a trained model and the key to the forces in the output dictionary. Moreover, the units that the calculator expects for positions and energy to properly convert between MD and calculator unit systems need to be specified. Force units are inferred automatically based on these two inputs. Optionally, one may provide an `energy_key` to store the potential energies in the `md.System` class. Additional properties (e.g., dipole moments for infrared spectra, etc.) can be requested with the `required_properties` argument. A typical SchNetPack `NeuralNetworkPotential` trained on the MD17 data would be initialized as follows:

```
from schnetpack.md.calculators \
    import SchNetPackCalculator

md_calculator = SchNetPackCalculator(
    "<PATH/TO/MODEL>", # path to stored model
    "forces", # force key
    "kcal/mol", # energy units
    "Angstrom", # length units
    md_neighborlist, # neighbor list defined above
    energy_key="energy", # potential energies
    required_properties=[], # additional properties
)
```

## D. Simulator

The `md.Simulator` loops over a series of time steps, calls the individual MD modules in the appropriate order, and updates the system state. Its internal structure is depicted in Fig. 5(b). Assuming that `md.System`, `md.Integrator`, and `md.Calculator` have been created as described above, the setup is straightforward:

```
md_simulator = schnetpack.md.Simulator(
    md_system,
    md_integrator,
    md_calculator,
    simulation_hooks=[],
)
```

Simulation hooks can be used to modify a simulation analogous to transforms in the neural network library and are described in Sec. V E. The required floating point precision and the computational device to be used can be set by the usual PyTorch directives:

```
md_device = "cpu"
md_precision = torch.float32

md_simulator = md_simulator.to(md_precision)
md_simulator = md_simulator.to(md_device)
```

The `simulate` routine runs the simulation with the number of simulation steps as an argument:

```
# simulate for 100 steps
md_simulator.simulate(100)
```

Since `md.Simulator` keeps track of the simulation and system state, repeated calls of `calculate` will resume the simulation.

## E. Simulation hooks

Simulation hooks can be used to tailor a simulation to specific needs, improving the customizability of the SchNetPack 2.0 MD code. A `SimulationHook` can be thought of as a set of instructions for the `Simulator`, which are performed at certain points of a MD step. Depending on which time they are called and which actions they encode, simulation hooks can achieve a wide range of tasks.

Figure 5(b) shows an overview at which points simulation hooks can be applied during a `Simulator` step. If a `SimulationHook` is applied before and after the integration half-step updating the momenta, it could control the temperature and pressure of the system in the form of thermostats and barostats. Acting directly after the `md.Calculator`, a hook may implement custom sampling schemes such as metadynamics or adaptive sampling.<sup>68,79</sup> When called after the second integration step, simulation hooks can be used to collect and store information on the system for analysis. A list of multiple hooks can be passed to a simulator, which enables control of a simulation in various manners. An overview of all simulation hooks currently implemented in SchNetPack 2.0 is shown in Table IV. In the following, we will give a brief overview of how to apply thermostat hooks and collect simulation data.

### 1. Temperature and pressure control

Constant temperature (NVT) and constant pressure (NPT) simulations can be performed using thermostat and barostat hooks. A simple example is the stochastic Langevin thermostat implemented in `LangevinThermostat`. This thermostat modifies the momenta before and after the integrator steps by applying random noise,

$$\mathbf{p} = c_1 \mathbf{p} + c_2 \sqrt{k_B T_{\text{bath}}} \mathbf{m} \xi, \quad (5)$$

where  $k_B$  is the Boltzmann constant,  $T_{\text{bath}}$  is the target Temperature of the ensemble, and  $\xi$  a vector of random noise drawn from a standard normal distribution. The coefficients  $c_1$  and  $c_2$  are defined as  $c_1 = \exp(-\frac{1}{2}\gamma\Delta t)$  and  $c_2 = \sqrt{1 - c_1^2}$ , with  $\gamma$  being friction coefficient of the thermostat.

The Langevin thermostat requires a time constant (in fs, used to determine  $\gamma$ ) and bath temperature (in K) to initialize:

Similar schemes can be used for ring-polymer MD.<sup>84</sup> In this case, thermostats act on the normal mode representation of the ring-polymer momenta.

**TABLE IV.** Overview of the simulation hooks (and data streams).

| Category    | Hook                              | Description  |
|-------------|-----------------------------------|--|
| Basic       | SimulationHook<br>RemoveCOMMotion | Abstract base class for deriving hooks<br>Periodically remove center of mass translation and rotation    |
| Thermostats | BerendsenThermostat               | Simple velocity rescaling thermostat <sup>80</sup>   |
|             | LangevinThermostat                | Basic stochastic Langevin thermostat <sup>81</sup>   |
|             | NHCThermostat                     | Nosé–Hoover chain thermostat <sup>82</sup>   |
|             | GLEThermostat                     | Stochastic generalized Langevin equation (GLE) colored noise thermostat <sup>83</sup>                    |
|             | PILELocalThermostat               | Path integral Langevin equation (PILE) thermostat for ring-polymer MD <sup>84</sup>                      |
|             | PILEGlobalThermostat              | Global variant of PILE, applies stochastic velocity rescaling to the ring-polymer centroid <sup>84</sup> |
|             | TRPMDTHERMOSTAT                   | Thermostated ring-polymer variant of the PILE thermostat <sup>85</sup>                                   |
|             | RPMGLETHERMOSTAT                  | GLE colored noise thermostat for ring-polymer MD <sup>83</sup>   |
|             | PIGLETThermostat                  | Version of GLE where every normal mode is thermostated separately <sup>86</sup>                          |
|             | NHCRingPolymerThermostat          | Nosé–Hoover chain thermostat for ring-polymer MD <sup>84</sup>   |
| Barostats   | NHCBarostatIsotropic              | Combined Nosé–Hoover chain thermostat and barostat for isotropic cell fluctuations <sup>87</sup>         |
|             | NHCBarostatAnisotropic            | Combined Nosé–Hoover chain thermostat and barostat for anisotropic cell fluctuations <sup>87</sup>       |
|             | PILEbarostat                      | Stochastic PILE barostat for ring-polymer MD <sup>88</sup>   |
| Logging     | CheckPoint                        | Periodically store the system state for restarting   |
|             | TensorBoardLogger                 | Log system information (e.g., temperature, energy) in TensorBoard format                                 |
|             | FileLogger                        | Log system information to a custom HDF5 dataset. Data streams are used to store different data groups    |
|             | MoleculeStream                    | Data stream for storing structural information with the FileLogger                                       |
|             | PropertyStream                    | Data stream for storing system properties with the FileLogger  |

---

```
from schnetpack.md.simulation_hooks \
    import LangevinThermostat

thermostat_hook = LangevinThermostat(
    300.0, # bath temperature in K
    100.0, # time constant in fs
)
```

---

## 2. Callbacks and logging

The FileLogger hook is the primary way to collect and store MD simulation data. This kind of information stored is controlled via two types of data streams: The MoleculeStream stores structural information and velocities during the simulation, whereas PropertyStream collects properties computed by the md.Calculator. To reduce file I/O overhead, the FileLogger collects a certain number of steps into a buffer before it is written to an HDF5 file at once.

The FileLogger requires the destination path, the buffer size, and the data streams. In addition, the logging frequency can be specified:

## F. Using the HDF5 dataset

SchNetPack 2.0 simulation data, stored in a HDF5 dataset as described previously, can be accessed with the HDF5Loader class to retrieve structures or perform analysis. Properties collected during simulation can be extracted with the `get_property` function. It takes the name of a property and an indicator, whether the property relates to the whole system or particular atoms:

The HDF5Loader comes with multiple predefined functions to extract specific properties, such as temperature or pressure. All of those receive a molecule index `mol_idx` and replica index `replica_idx` as additional inputs. The former is used to extract information about a particular system when multiple are simulated at the same time, whereas the latter is used to target specific replicas, e.g., in ring-polymer MD. The default behavior is to target the first system and compute the average (centroid) over all replicas. Finally, `convert_to_atoms` allows structures to be extracted as a list of ASE `Atoms` objects.

SchNetPack 2.0 provides the module `md.data.spectra` to compute different vibrational spectra directly from an HDF5Loader instance. Routines for power spectra (`PowerSpectrum`, requires momenta), infrared (IR) spectra (`IRSpectrum`, requires dipole moments), and Raman spectra (`RamanSpectrum`, requires polarizabilities) are available. All spectra are computed by Fourier transforming the time autocorrelation functions of the respective properties.<sup>93</sup>

```
from schnetpack.md.simulation_hooks \
    import callback_hooks

# Set up data streams
data_streams = [
    # store positions and velocities
    callback_hooks.MoleculeStream(
        store_velocities=True
    ),
    # store energies
    callback_hooks.PropertyStream(
        target_properties=["energy"]
    ),
]

# Create the file logger
file_logger_hook = callback_hooks.FileLogger(
    "simulation.hdf5", # path to the log file
    100, # size of the buffer
    data_streams=data_streams,
    every_n_steps=1, # logging frequency
)

from schnetpack.md.data \
    import HDF5Loader

md_data = HDF5Loader("simulation.hdf5")

energy = md_data.get_property(
    "energy", atomistic=False
)
```

For example, the power spectrum can be computed as follows:

```
from schnetpack.md.data \
    import PowerSpectrum

# Initialize the spectrum
spectrum = PowerSpectrum(md_data, resolution=2048)

# Compute the spectrum
spectrum.compute_spectrum()

# Get frequencies and intensities
frequencies, intensities = spectrum.get_spectrum()
```

## G. Molecular dynamics configuration and command line interface

Similar to the neural network package, MD simulations can be configured using the Hydra framework. The central command of the

SchNetPack MD CLI is spkmd, which sets up everything for a basic MD simulation and creates a simulation directory. Runs can be configured via predefined config groups and command-line overrides. Moreover, since Hydra CLI is able to instantiate classes from YAML configs, it is straightforward to integrate external modules, such as custom calculators for simulations.

A standard spkmd run requires the following inputs:

```
spkmd simulation_dir=<DIR>
    ↳ system.molecule_file=<XYZ>
    ↳ calculator.model_file=<MODEL>
    ↳ calculator.neighbor_list.cutoff=<CUTOFF>
```

where `simulation_dir` indicates the simulation directory, `system.molecule_file` the file containing the structures to be simulated and `calculator.model_file` and `calculator.neighbor_list.cutoff` specify the path to the previously trained neural network potential and the cutoff radius used in the model. This starts a MD run with a predefined default configuration, which corresponds to a NVE simulation where features such as logging and check pointing have already been set up.

Like in the neural network package, an MD config is structured into different config groups:

- uncategorized, general settings, such as device, precision, random seed, and simulation directory.
- `calculator`: Definition of the MD calculator (see Sec. V C).
- `system`: Definition of the MD system, including loading of structures and initial conditions (see Sec. V A).
- `dynamics`: Definition of the overall MD loop (Sec. V D). Contains the subgroups:

- `integrator`: Integrator settings (Sec. V B)
- `thermostat`: Temperature control.
- `barostat`: Pressure control.
- `simulation_hook`: General hooks for sampling (Sec. V E).

- `callbacks`: Definition of hooks for callback and logging.

These groups can be used to further configure the simulation, e.g., by adding a thermostat or changing integrator settings (see e.g., Sec. VI B).

MD simulations can be started using full or partial config files in YAML format as input. For example, it is possible to create a basic config file by calling spkmd with the `-cfg` job flag and edit it for a particular application. An existing config file can then be loaded with the `load_config` option:

```
spkmd simulation_dir=<DIR> load_config=<CONFIG-FILE>
```

It is still possible to further modify the simulation via other command line overwrites.

## H. LAMMPS interface

In addition to the integrated `schnetpack.md` code, SchNetPack 2.0 comes with an interface to LAMMPS.<sup>92</sup> First, a trained model needs to be deployed using the torch just-in-time (JIT) compiler as follows:

```
spkdeploy <model_path> <deployed_path>
```

SchNetPack supplies a pair style that allows us to utilize such a deployed model in LAMMPS. Detailed instructions are available in the SchNetPack documentation.<sup>90</sup> This enables researchers to use the full functionality of LAMMPS when simulating systems with SchNetPack potentials.

## VI. EXAMPLE APPLICATIONS

This section features some basic and advanced applications of SchNetPack. First, we demonstrate how to train atomistic neural networks on the supported benchmark datasets. In a second example, we showcase the prediction of response properties in the example of a custom dataset. Finally, we demonstrate how SchNetPack can be extended with custom code to train a generative neural network for the inverse design of 3D structures.

### A. Potential energy surfaces and property prediction

The datasets QM9<sup>72</sup> and MD17<sup>22</sup> have become established benchmarks in the development of atomistic representations. Here, we use the revised MD17 (rMD17) dataset for which the energies and forces have been recomputed with higher accuracy.<sup>73</sup> We have trained models using three representations SchNet, PaiNN, and SO3net. For the latter, we have explored setting the maximum angular moment to  $l_{\max} \in \{1, 2\}$ .

We have predicted the inner energy  $U_0$  and the total dipole moment  $\mu$  from the properties in QM9 with two separate models. The configuration `qm9_atomwise` for  $U_0$  is given in Fig. 4. The only difference of the `qm9_dipole` configuration is the use of the `DipoleMoment` output module:

```
model:
  output_modules:
    - _target_: schnetpack.atomistic.DipoleMoment
      dipole_key: ${globals.property}
      n_in: ${model.representation.n_atom_basis}
      predict_magnitude: True
      use_vector_representation: False
  postprocessors:
    - _target_: schnetpack.transform.CastTo64
```

If the representation supports equivariant vector features, which is the case for PaiNN and SO3net, we use atomic dipoles in the output layer<sup>65</sup> by setting `use_vector_representation = True` (see Table III).

The training runs have been started using the SchNetPack CLI. The initial learning rate was set such that training was still stable,

```
spktrain experiment=qm9_dipole
  task.scheduler_args.patience=25
  trainer.max_epochs=5000
  model/representation/radial_basis=bessel
  task.optimizer_args.weight_decay=0.01
  globals.lr={1e-3,5e-4}
  model/representation=[schnet,painn,so3net]
  [model.representation.lmax={1,2}]
  model.output_modules.0.
  use_vector_representation={True,False}
```

```
spktrain experiment=md17 data=rmd17
  data.molecule=[aspirin,paracetamol]
  globals.lr={1e-3,5e-4}
  task.optimizer_args.weight_decay=0.01
  model/representation=[schnet,painn,so3net]
  [model.representation.lmax={1,2}]
```

which we found to be  $5 \times 10^{-4}$  for SchNet and PaiNN and  $1 \times 10^{-3}$  for SO3net. We use a learning rate scheduler that decays when the validation error does not decrease within the given number of patience epochs. The training of dipole moments can be reproduced by calling:

Different hyperparameter selections are indicated by curly brackets and `lmax` can only be set for SO3net (indicated by `[]`).

We have selected the aspirin and paracetamol molecules for our rMD17 experiments and predicted energies and forces. We call the `md17` experimental configuration of SchNetPack with the following settings:

Table V shows the errors for the trained SchNetPack models. The results for SchNet and PaiNN are similar to what has been observed in earlier work,<sup>21,65</sup> although hyperparameters such as the learning rate schedule, the radial basis, and the weight decay may differ. Note, that these experiments are only meant to demonstrate the capabilities of SchNetPack. For an accurate comparison between atomistic ML models, an extensive hyperparameter search should be performed. Table VI shows the average time per epoch of the performed experiments. Although SchNet and PaiNN have more parameters than SO3net, this does not correspond directly to the number of floating point operations. In particular, when setting the maximum angular momentum  $l_{\max} > 1$ , the required compute rises faster than the model size due to increased parameter sharing in our tensor product layer implementation.

### B. Modeling response properties

The FieldSchNet representation,<sup>64</sup> included in SchNetPack 2.0, is able to model response properties and solvent effects. Here, we demonstrate how to train a FieldSchNet model for potential energies  $E$  and corresponding response properties, namely atomic forces  $F$ , dipole moments  $\mu$ , polarizabilities  $\alpha$ , and nuclear-shielding tensors  $\sigma$ . We use the reference data of ethanol in vacuum published in Ref. 68.

**TABLE V.** Mean absolute and mean squared error for various neural networks trained with SchNetPack on prediction task on the QM9 and rMD17 benchmark datasets. The errors are averaged over three repetitions.

| Dataset           | Property | Unit             | Metric | SchNet | PaiNN | SO3net ( $l_{\max} = 1$ ) | SO3net ( $l_{\max} = 2$ ) |
|-------------------|----------|------------------|--------|--------|-------|---------------------------|---------------------------|
| QM9               | $U_0$    | meV              | MAE    | 9.6    | 5.7   | 6.8                       | 6.4                       |
|                   |          |                  | RMSE   | 21.9   | 15.3  | 16.2                      | 17.1                      |
| QM9               | $\mu$    | Debye            | MAE    | 0.022  | 0.011 | 0.018                     | 0.014                     |
|                   |          |                  | RMSE   | 0.044  | 0.026 | 0.038                     | 0.033                     |
| rMD17/Aspirin     | $E$      | meV              | MAE    | 13.5   | 3.8   | 3.8                       | 2.6                       |
|                   |          |                  | RMSE   | 18.3   | 5.9   | 5.7                       | 3.8                       |
|                   | $F$      | meV/Å            | MAE    | 33.2   | 12.8  | 12.7                      | 9.0                       |
|                   |          |                  | RMSE   | 49.5   | 21.7  | 19.6                      | 14.5                      |
| rMD17/Paracetamol | $E$      | meV              | MAE    | 8.4    | 2.1   | 2.2                       | 1.4                       |
|                   |          |                  | RMSE   | 11.2   | 2.9   | 3.0                       | 1.9                       |
|                   | $F$      | meV/Å<br>plateau | MAE    | 26.1   | 9.0   | 8.9                       | 6.0                       |
|                   |          |                  | RMSE   | 40.0   | 14.7  | 13.8                      | 10.0                      |

**TABLE VI.** Training and validation time per epoch for models trained on QM9 and rMD17 tasks using an Nvidia A100.

| Task                               | Model                     | No. Params (k) | Time/epoch |
|------------------------------------|---------------------------|----------------|------------|
| QM9, $U_0$ 110/10 k bs 100         | SchNet                    | 432            | 1 min 14s  |
|                                    | PaiNN                     | 589            | 1 min 13s  |
|                                    | SO3net ( $l_{\max} = 1$ ) | 283            | 1 min 16s  |
|                                    | SO3net ( $l_{\max} = 2$ ) | 341            | 2 min 29s  |
| rMD17, E + F, aspirin 950/50 bs 10 | SchNet                    | 432            | 7s         |
|                                    | PaiNN                     | 589            | 6s         |
|                                    | SO3net ( $l_{\max} = 1$ ) | 283            | 7s         |
|                                    | SO3net ( $l_{\max} = 2$ ) | 341            | 10s        |

To model the target response properties, we first need to construct a FieldSchNet model of the energy  $E(\mathbf{R}, \boldsymbol{\varepsilon}, \mathbf{B}, \mathbf{I})$  that depends on positions, electric field ( $\boldsymbol{\varepsilon}$ ), magnetic field ( $\mathbf{B}$ ), and nuclear magnetic moments ( $\mathbf{I}$ ). The response properties can then be obtained by taking the appropriate derivatives of this model:<sup>94</sup>

$$\begin{aligned}\mathbf{F}_i &= \frac{\partial E}{\partial \mathbf{R}_i}, \\ \boldsymbol{\mu} &= -\frac{\partial E}{\partial \boldsymbol{\varepsilon}}, \\ \boldsymbol{\alpha} &= -\frac{\partial^2 E}{\partial \boldsymbol{\varepsilon}^2}, \\ \sigma_i &= \frac{\partial^2 E}{\partial \mathbf{B} \partial \mathbf{I}_i}.\end{aligned}$$

To implement this, two modifications to the standard neural network potential configuration are required: First, a `StaticExternalFields` module needs to be added to the input modules to set up the required auxiliary fields (electric and magnetic). Second, a `Response` output module is appended after the `Atomwise` layer to compute the different derivatives

corresponding to the response properties. Both modules automatically determine the required fields and derivatives based on the requested response properties (in this case specified in `globals.response_properties`):

These changes and settings such as loss, metrics, and trade-offs are predefined in the experiment config `response`, which only requires the dataset, batch size, and splits to be specified:

This command trains a FieldSchNet model with a cutoff of 9.449 bohrs (5 Å), 128 features, and five interactions, using 9000 points of the ethanol dataset for training and validation and the remaining 1000 points for testing. The average test errors over three runs can be found in Table VII.

Once the model is trained, we can use the following `spkmd` command to perform a 25 ps ring-polymer simulation with 16 beads:

Since reference data uses atomic units, we set the energy and position units in the calculator to Hartree and Bohr. The simulation temperature is kept at 300 K with a ring-polymer Nosé-Hoover chain thermostat (see Sec. IV). The overrides `calculator.required_properties` and `callbacks.hdf5`.  
`data_streams.1.target_properties` provide instructions on which properties the `md.Calculator` needs to compute and on

```

globals:
  energy_key: energy
  response_properties:
    - forces
    - dipole_moment
    - polarizability
    - shielding

model:
  input_modules:
    - _target_: schnetpack.atomistic.PairwiseDistances
    - _target_: schnetpack.atomistic.StaticExternalFields
      response_properties: ${globals.response_properties}
  output_modules:
    - _target_: schnetpack.atomistic.Atomwise
      output_key: ${globals.energy_key}
      n_in: ${model.representation.n_atom_basis}
      aggregation_mode: sum
    - _target_: schnetpack.transform.ScaleProperty
      input_key: ${globals.energy_key}
      output_key: ${globals.energy_key}
    - _target_: schnetpack.atomistic.Response
      energy_key: ${globals.energy_key}
      response_properties: ${globals.response_properties}

```

```

spktrain experiment=response
→ data.datapath=<PATH/TO/DB> data.num_train=8000
→ data.num_val=1000 data.batch_size=20

```

**TABLE VII.** Prediction errors of the FieldSchNet model for the ethanol molecule.

| Property   | Unit                                   | MAE   |
|------------|--|-------|
| E          | Kcal mol <sup>-1</sup>                 | 0.023 |
| F          | Kcal mol <sup>-1</sup> Å <sup>-1</sup> | 0.158 |
| $\mu$      | D                                      | 0.004 |
| $\alpha$   | bohr <sup>3</sup>                      | 0.009 |
| $\sigma_H$ | ppm                                    | 0.045 |
| $\sigma_C$ | ppm                                    | 0.215 |
| $\sigma_O$ | ppm                                    | 0.469 |

which ones should be logged by the FileLogger. In this case, we select the potential energy, dipole moment, and polarizability to compute vibrational infrared and Raman spectra.

The HDF5Loader is used to load the simulation.hdf5 file generated by the MD, where we skip an initial equilibration period of 5 ps (25 000 steps). Finally, we compute the infrared and Raman spectra with the IRSpectrum and RamanSpectrum routines, using a temperature of 300 K and laser frequency of 514 nm for the latter. The predicted and experimental spectra can be found in Fig. 6.

```

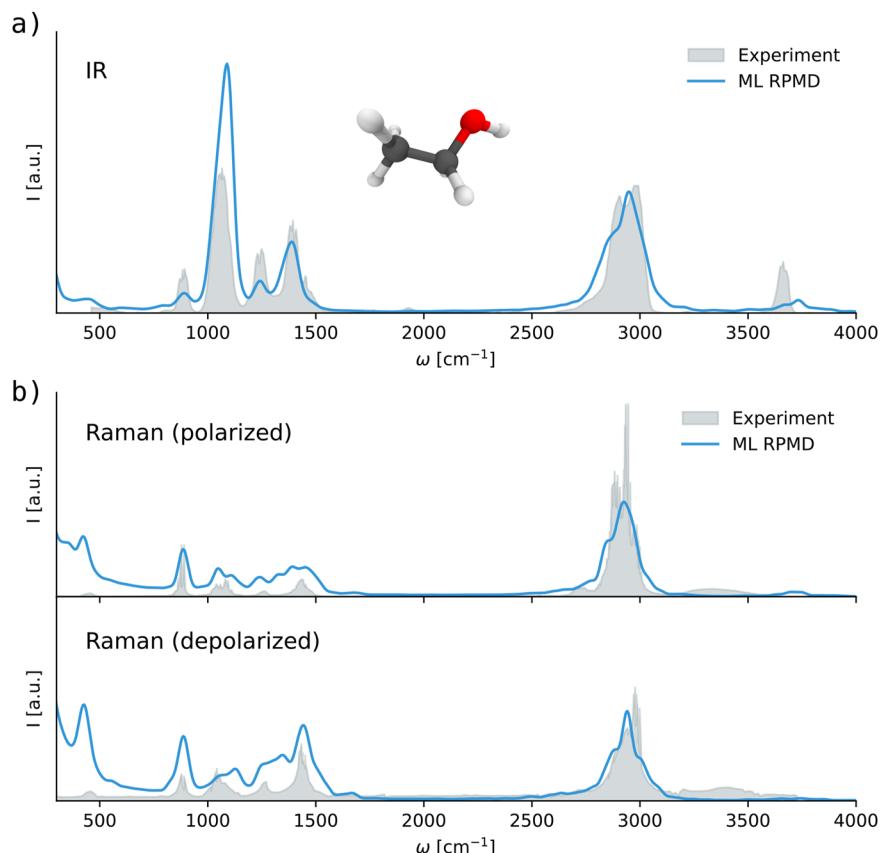
spkmd simulation_dir=<DIR>
→ system.molecule_file=<ethanol.xyz>
→ calculator.model_file=<MODEL>
→ calculator.neighbor_list.cutoff=9.449
→ calculator.energy_unit=Hartree
→ calculator.position_unit=Bohr
→ system.n_replicas=16 dynamics/integrator=rpmd
+ dynamics/thermostat=pi_nhc_global
→ dynamics.n_steps=125000
→ calculator.required_properties=[ energy,
  dipole_moment, polarizability ]
→ callbacks.hdf5.data_streams.1.target_properties=[ energy, dipole_moment, polarizability ]

```

## C. Building generative models

Generative SchNet (G-SchNet)<sup>89</sup> is a deep autoregressive neural network model for the inverse design of 3D molecular structures. It learns to sample molecules by sequentially placing atoms in 3D space. At each step, the type of the next atom and its distances to preceding atoms are predicted. Recently, the model has been extended to learn conditional distributions by taking target property values as additional inputs.<sup>37</sup> We have implemented an updated version of this conditional G-SchNet (cG-SchNet) as an extension of SchNetPack 2.0. The package is called schnetpack-gschnet and available on Github.<sup>95</sup> Compared with previous implementations, it aims at simple integration of custom datasets and improves the scalability of cG-SchNet both in terms of memory and computational complexity to make it applicable to larger molecules.

Since cG-SchNet is not a neural network potential but a generative model, several new modules are required for the architecture, the atomistic task, and the data processing. The core network implementation consists of the classes `ConditionalGenerativeSchNet`, a subclass of `AtomisticModel`, and `ConditioningModule`, which takes any amount of `ConditionEmbedding` networks to extract a combined feature vector representing all conditioning targets. Three subclasses of `ConditionEmbedding` are provided for the embedding of scalar properties, vectorial properties, and the atomic composition of molecules. Furthermore, `ConditionalGenerativeSchNetTask`, a subclass of `AtomisticTask`, customizes the learning task including the loss functions applied to predicted distributions and some task-specific setup, e.g., making sure that the molecular properties required as target conditions are loaded by the data module. To learn the sequential placements of atoms with cG-SchNet, training molecules need to be sliced into trajectories where the structure grows atom by atom. This is implemented in a preprocessing Transform called `BuildAtomsTrajectory`, which allows us to sample a random trajectory for each data point in each epoch. The process depends on a few prerequisites, e.g., a certain ordering of atoms and different neighbor lists, which are also computed in custom transforms. In addition, we require a filter to exclude disconnected structures that is evaluated once before determining the training, validation, and test splits. Thus, it is not implemented as a transform but in the setup stage of `GenerativeAtomsDataModule`, which is a subclass of `AtomsDataModule` and serves as the base



**FIG. 6.** Vibrational spectra of ethanol obtained by a ring-polymer MD using a FieldSchNet model. Shown are (a) the infrared and (b) polarized and depolarized Raman spectra along with respective experimental references (gray).

```

configs/
└── data/
    └── gschnet_qm9.yaml
experiment/
└── gschnet_qm9_gap_relenergy.yaml
model/
└── conditioning/
    └── gap_relenergy.yaml
    └── gschnet.yaml
task/
└── gschnet_task.yaml
...

```

```

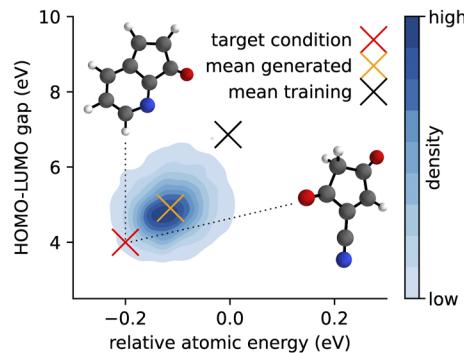
1  defaults:
2  - override /data: gschnet_qm9
3  - override /task: gschnet_task
4  - override /model: gschnet
5  - override /model/conditioning: gap_relenergy
...
45 data:
46 transforms:
47 - _target_: schnetpack.transform.SubtractCenterOfMass
48 - _target_: schnetpack_gschnet.transform.OrderByDistanceToOrigin
...
65 - _target_: schnetpack_gschnet.transform.BuildAtomsTrajectory
66 centered: True
67 origin_type: ${globals.origin_type}
68 focus_type: ${globals.focus_type}
69 stop_type: ${globals.stop_type}
70 draw_random_samples: ${globals.draw_random_samples}
71 sort_idx_i: False
72 - _target_: schnetpack.transform.CastTo32

```

(a)

(b)

(c)



**FIG. 7.** (a) Additional config files from schnetpack-gschnet. The directory tree follows the structure induced by SchNetPack 2.0 to enable the composition of schnetpack and schnetpack-gschnet config files. (b) Excerpt from the experiment config `gschnet_qm9_gap_relenergy.yaml` where we override defaults from schnetpack with the new config files and use transforms from both packages to train a cG-SchNet model conditioned on HOMO-LUMO gap and energy of molecules on the QM9 dataset. **c** Density plot showing the HOMO-LUMO gap and energy of 20 k molecules generated after training with schnetpack-gschnet. We use particularly low values of gap and energy as targets (red cross) and show the two generated molecules closest to it. The mean gap and energy of training structures (black cross) and generated structures (orange cross) are also provided. Energy and gap of generated structures are predicted with PaiNN models trained with standard settings from SchNetPack 2.0.

class for datasets used with cG-SchNet. The package contains QM9Gen, an example dataset class for the QM9 benchmark dataset.

The hierarchical configuration framework Hydra allows us to easily integrate the new modules with SchNetPack. It requires corresponding YAML files for the model, task, data, and experiment, where the directory tree should follow the config groups of SchNetPack as described in Sec. IV [see Fig. 7(a)]. We start the training of cG-SchNet just like for other models via the SchNetPack CLI by supplying the new configs as additional sources:

```
spktrain --config-dir=<PATH/TO/CONFIGS>
  ↳ experiment=gschnet_qm9_gap_relenergy
```

Here, we use a custom experiment that overrides the model, task, and data configs to train a cG-SchNet model on QM9 that is conditioned on the energy and the HOMO–LUMO gap. All remaining configs, e.g., for the trainer and the run, are loaded from SchNetPack. In Fig. 7(b), we show the integration of new configs and how Transform modules from both schnetpack and schnetpack-gschnet can be accessed in the experiment configuration.

Although the training reuses code and configs from the SchNetPack framework, the inference with cG-SchNet consists of sampling molecules from scratch, which is quite different from predicting the properties of given molecules. Therefore, the generation of molecules is implemented in the package as a separate CLI with its own, hierarchical Hydra configuration. Exemplary results of molecules generated with cG-SchNet after training with our SchNetPack extension are shown in Fig. 7(c).

## VII. CONCLUSIONS

We have presented SchNetPack 2.0 that constitutes a major upgrade in functionality over the first version. The new data pipeline comes with preprocessing transforms and a sparse data format. Due to precalculated indices, sparse operations within the model, such as aggregation of neighbors in message passing or Clebsch–Gordan tensor products can be written in a couple of lines. The switch to versatile training and configuration frameworks makes it easy for developers to extend SchNetPack with custom modules, datasets, and configs. SchNetPack 2.0 moves beyond neural network potentials by enabling a flexible definition of complex training tasks, as we have shown in the example of a generative neural network for 3D molecules. Finally, SchNetPack comes with its own molecular dynamics simulation code so that trained models can directly be applied. We are confident that these changes and additions in SchNetPack 2.0 will prove useful for both users and developers of atomistic neural networks.

## SUPPLEMENTARY MATERIAL

See the [supplementary material](#) for description of the SO3net neural network architecture.

## ACKNOWLEDGMENTS

We thank all contributors to SchNetPack on Github. K.T.S., S.S.P.H., N.W.A.G., and J.L. acknowledge support by the Federal Ministry of Education and Research (BMBF) for the Berlin Institute for the Foundations of Learning and Data (BIFOLD) (Grant No. 01IS18037A). M.G. and N.W.A.G. work at the BASLEARN – TU Berlin/BASF Joint Lab for Machine Learning, co-financed by TU Berlin and BASF SE.

## AUTHOR DECLARATIONS

### Conflict of Interest

The authors have no conflicts to disclose.

### Author Contributions

**Kristof T. Schütt:** Conceptualization (lead); Methodology (equal); Software (equal); Visualization (equal); Writing – original draft (lead); Writing – review & editing (equal). **Stefaan S. P. Hessmann:** Methodology (supporting); Software (supporting); Writing – review & editing (equal). **Niklas W. A. Gebauer:** Conceptualization (supporting); Methodology (supporting); Software (supporting); Visualization (equal); Writing – original draft (supporting); Writing – review & editing (supporting). **Jonas Lederer:** Conceptualization (supporting); Methodology (supporting); Software (supporting); Writing – review & editing (supporting). **Michael Gastegger:** Methodology (equal); Software (equal); Visualization (supporting); Writing – original draft (supporting); Writing – review & editing (equal).

## DATA AVAILABILITY

The datasets used in this work are openly available Figshare under <https://doi.org/10.6084/m9.figshare.c.978904.v5> (Ref. 72, QM9), at <https://doi.org/10.6084/m9.figshare.12672038.v3> (Ref. 73, rMD17), and at DepositOnce under <https://doi.org/10.14279/depositonce-12244> (Ref. 64, response properties of ethanol).

## REFERENCES

- H. J. Kulik, T. Hammerschmidt, J. Schmidt, S. Botti, M. A. L. Marques, M. Boley, M. Scheffler, M. Todorović, P. Rinke, C. Oses *et al.*, “Roadmap on machine learning in electronic structure,” *Electron. Struct.* **4**, 023004 (2022).
- O. A. von Lilienfeld, K.-R. Müller, and A. Tkatchenko, “Exploring chemical compound space with quantum-based machine learning,” *Nat. Rev. Chem.* **4**, 347–358 (2020).
- J. A. Keith, V. Vassilev-Galindo, B. Cheng, S. Chmiela, M. Gastegger, K.-R. Müller, and A. Tkatchenko, “Combining machine learning and computational chemistry for predictive insights into chemical systems,” *Chem. Rev.* **121**, 9816–9872 (2021).
- F. Noé, A. Tkatchenko, K.-R. Müller, and C. Clementi, “Machine learning for molecular simulation,” *Annu. Rev. Phys. Chem.* **71**, 361–390 (2020).
- J. Behler, “Four generations of high-dimensional neural network potentials,” *Chem. Rev.* **121**, 10037–10072 (2021).
- P. O. Dral, “Quantum chemistry in the age of machine learning,” *J. Phys. Chem. Lett.* **11**, 2336–2347 (2020).

- <sup>7</sup>M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld, "Fast and accurate modeling of molecular atomization energies with machine learning," *Phys. Rev. Lett.* **108**, 058301 (2012).
- <sup>8</sup>A. P. Bartók, M. C. Payne, R. Kondor, and G. Csányi, "Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons," *Phys. Rev. Lett.* **104**, 136403 (2010).
- <sup>9</sup>K. T. Schütt, F. Arbabzadah, S. Chmiela, K. R. Müller, and A. Tkatchenko, "Quantum-chemical insights from deep tensor neural networks," *Nat. Commun.* **8**, 13890 (2017).
- <sup>10</sup>K. T. Schütt, P.-J. Kindermans, H. E. Saucedo, S. Chmiela, A. Tkatchenko, and K.-R. Müller, "SchNet: A continuous-filter convolutional neural network for modeling quantum interactions," in *Advances in Neural Information Processing Systems* (Curran Associates, Inc., 2017), pp. 991–1001.
- <sup>11</sup>F. A. Faber, L. Hutchison, B. Huang, J. Gilmer, S. S. Schoenholz, G. E. Dahl, O. Vinyals, S. Kearnes, P. F. Riley, and O. A. von Lilienfeld, "Prediction errors of molecular machine learning models lower than hybrid DFT error," *J. Chem. Theory Comput.* **13**, 5255–5264 (2017).
- <sup>12</sup>F. A. Faber, A. S. Christensen, B. Huang, and O. A. von Lilienfeld, "Alchemical and structural distribution based representation for universal quantum machine learning," *J. Chem. Phys.* **148**, 241717 (2018).
- <sup>13</sup>O. T. Unke and M. Meuwly, "PhysNet: A neural network for predicting energies, forces, dipole moments, and partial charges," *J. Chem. Theory Comput.* **15**, 3678–3693 (2019).
- <sup>14</sup>B. Huang and O. A. von Lilienfeld, "Quantum machine learning using atom-in-molecule-based fragments selected on the fly," *Nat. Chem.* **12**, 945–951 (2020).
- <sup>15</sup>J. Gasteiger, J. Groß, and S. Günnemann, "Directional message passing for molecular graphs," in International Conference on Learning Representations (ICLR), 2020.
- <sup>16</sup>J. Behler and M. Parrinello, "Generalized neural-network representation of high-dimensional potential-energy surfaces," *Phys. Rev. Lett.* **98**, 146401 (2007).
- <sup>17</sup>J. Behler, "Constructing high-dimensional neural network potentials: A tutorial review," *Int. J. Quantum Chem.* **115**, 1032–1050 (2015).
- <sup>18</sup>O. T. Unke, S. Chmiela, H. E. Saucedo, M. Gastegger, I. Poltavsky, K. T. Schütt, A. Tkatchenko, and K.-R. Müller, "Machine learning force fields," *Chem. Rev.* **121**, 10142–10186 (2021).
- <sup>19</sup>I. Batatia , D. P. Kovacs , G. Simm , C. Ortner , and G. Csányi , "MACE: Higher order equivariant message passing neural networks for fast and accurate force fields," in Proceedings of Advances in Neural Information Processing Systems 35 (Curran Associates, Inc., 2022), pp. 11423–11436; available at [https://proceedings.neurips.cc/paper\\_files/paper/2022/hash/4a36c3c51af11ed9f34615b81edb5bbc-Abstract-Conference.html](https://proceedings.neurips.cc/paper_files/paper/2022/hash/4a36c3c51af11ed9f34615b81edb5bbc-Abstract-Conference.html).
- <sup>20</sup>A. P. Bartók, J. Kermode, N. Bernstein, and G. Csányi, "Machine learning a general-purpose interatomic potential for silicon," *Phys. Rev. X* **8**, 041048 (2018).
- <sup>21</sup>K. T. Schütt, H. E. Saucedo, P.-J. Kindermans, A. Tkatchenko, and K.-R. Müller, "SchNet—A deep learning architecture for molecules and materials," *J. Chem. Phys.* **148**, 241722 (2018).
- <sup>22</sup>S. Chmiela, A. Tkatchenko, H. E. Saucedo, I. Poltavsky, K. T. Schütt, and K.-R. Müller, "Machine learning of accurate energy-conserving molecular force fields," *Sci. Adv.* **3**, e1603015 (2017).
- <sup>23</sup>S. Chmiela, H. E. Saucedo, K.-R. Müller, and A. Tkatchenko, "Towards exact molecular dynamics simulations with machine-learned force fields," *Nat. Commun.* **9**, 3887 (2018).
- <sup>24</sup>O. T. Unke, M. Stöhr, S. Ganscha, T. Unterthiner, H. Maennel, S. Kashubin, D. Ahlin, M. Gastegger, L. M. Sandonas, A. Tkatchenko *et al.*, "Accurate machine learned quantum-mechanical force fields for biomolecular simulations," [arXiv:2205.08306](https://arxiv.org/abs/2205.08306) (2022).
- <sup>25</sup>D. Lu, H. Wang, M. Chen, L. Lin, R. Car, W. E, W. Jia, and L. Zhang, "86 PFLOPS deep potential molecular dynamics simulation of 100 million atoms with *ab initio* accuracy," *Comput. Phys. Commun.* **259**, 107624 (2021).
- <sup>26</sup>A. Musaelian, S. Batzner, A. Johansson, L. Sun, C. J. Owen, M. Kornbluth, and B. Kozinsky, "Learning local equivariant representations for large-scale atomistic dynamics," *Nat. Commun.* **14**, 579 (2023).
- <sup>27</sup>J. Westermayr, M. Gastegger, K. T. Schütt, and R. J. Maurer, "Perspective on integrating machine learning into computational chemistry and materials science," *J. Chem. Phys.* **154**, 230903 (2021).
- <sup>28</sup>L. Li, S. Hoyer, R. Pederson, R. Sun, E. D. Cubuk, P. Riley, and K. Burke, "Kohn-Sham equations as regularizer: Building prior knowledge into machine-learned physics," *Phys. Rev. Lett.* **126**, 036401 (2021).
- <sup>29</sup>F. Brockherde, L. Vogt, L. Li, M. E. Tuckerman, K. Burke, and K.-R. Müller, "Bypassing the Kohn-Sham equations with machine learning," *Nat. Commun.* **8**, 872 (2017).
- <sup>30</sup>A. Fabrizio, A. Grisafi, B. Meyer, M. Ceriotti, and C. Corminboeuf, "Electron density learning of non-covalent systems," *Chem. Sci.* **10**, 9424–9432 (2019).
- <sup>31</sup>K. T. Schütt, M. Gastegger, A. Tkatchenko, K.-R. Müller, and R. J. Maurer, "Unifying machine learning and quantum chemistry with a deep neural network for molecular wavefunctions," *Nat. Commun.* **10**, 5024 (2019).
- <sup>32</sup>O. Unke, M. Bogojeski, M. Gastegger, M. Geiger, T. Smidt, and K.-R. Müller, "SE(3)-equivariant prediction of molecular wavefunctions and electronic densities," *Adv. Neural Inf. Process. Syst.* **34**, 14434–14447 (2021).
- <sup>33</sup>K. Ghosh, A. Stuke, M. Todorović, P. B. Jørgensen, M. N. Schmidt, A. Vehtari, and P. Rinke, "Deep learning spectroscopy: Neural networks for molecular excitation spectra," *Adv. Sci.* **6**, 1801367 (2019).
- <sup>34</sup>J. Westermayr and P. Marquetand, "Machine learning and excited-state molecular dynamics," *Mach. Learn.: Sci. Technol.* **1**, 043001 (2020).
- <sup>35</sup>J. Westermayr and R. J. Maurer, "Physically inspired deep learning of molecular excitations and photoemission spectra," *Chem. Sci.* **12**, 10755–10764 (2021).
- <sup>36</sup>P. Leinen, M. Esders, K. T. Schütt, C. Wagner, K.-R. Müller, and F. S. Tautz, "Autonomous robotic nanofabrication with reinforcement learning," *Sci. Adv.* **6**, eabb6987 (2020).
- <sup>37</sup>N. W. A. Gebauer, M. Gastegger, S. S. P. Hessmann, K.-R. Müller, and K. T. Schütt, "Inverse design of 3d molecular structures with conditional generative neural networks," *Nat. Commun.* **13**, 973 (2022).
- <sup>38</sup>J. Köhler, L. Klein, and F. Noé, "Equivariant flows: Sampling configurations for multi-body systems with symmetric energies," in Proceedings of the 37th International Conference on Machine Learning, 2019.
- <sup>39</sup>Q. Liu, M. Allamanis, M. Brockschmidt, and A. Gaunt, "Constrained graph variational autoencoders for molecule design," in *Advances in Neural Information Processing Systems* (Curran Associates, Inc., 2018), pp. 7795–7804.
- <sup>40</sup>G. N. C. Simm, R. Pinsler, G. Csányi, and J. M. Hernández-Lobato, "Symmetry-aware actor-critic for 3D molecular design," in International Conference on Learning Representations, 2021.
- <sup>41</sup>R. P. Joshi, N. W. A. Gebauer, M. Bontha, M. Khazaieli, R. M. James, J. B. Brown, and N. Kumar, "3D-Scaffold: A deep learning framework to generate 3D coordinates of drug-like molecules with desired scaffolds," *J. Phys. Chem. B* **125**, 12166–12176 (2021).
- <sup>42</sup>A. Mardt, L. Pasquali, H. Wu, and F. Noé, "VAMPnets for deep learning of molecular kinetics," *Nat. Commun.* **9**, 5 (2018).
- <sup>43</sup>J. Lederer, M. Gastegger, K. T. Schütt, M. Kampffmeyer, K.-R. Müller, and O. T. Unke, "Automatic identification of chemical moieties," [arXiv:2203.16205](https://arxiv.org/abs/2203.16205) (2022).
- <sup>44</sup>J. Hermann, Z. Schätzle, and F. Noé, "Deep-neural-network solution of the electronic Schrödinger equation," *Nat. Chem.* **12**, 891–897 (2020).
- <sup>45</sup>D. Pfau, J. Spencer, A. d. G. Matthews, and W. Foulkes, "Ab-initio solution of the many-electron Schrödinger equation with deep neural networks," *Phys. Rev. Res.* **2**, 033429 (2020).
- <sup>46</sup>K. T. Schütt, P. Kessel, M. Gastegger, K. A. Nicoli, A. Tkatchenko, and K.-R. Müller, "SchNetPack: A deep learning toolbox for atomistic systems," *J. Chem. Theory Comput.* **15**, 448–455 (2018).
- <sup>47</sup>A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in NeurIPS 2017 Workshop Autodiff, 2017.
- <sup>48</sup>W. Falcon, J. Borovac, A. Wälchli, N. Eggert, J. Schock, J. Jordan, N. Skafte, V. Bereznnyk, E. Harris, T. Murrell *et al.*, Pytorchlightning/pytorch-lightning: 0.7. 6 Release, 2020.

- <sup>49</sup>V. Fomin, J. Anmol, S. Desroziers, J. Kriss, and A. Tejani, "High-level library to help with training neural networks in pytorch," <https://github.com/pytorch/ignite>, 2020.
- <sup>50</sup>M. Geiger and T. Smidt, "e3nn: Euclidean neural networks," [arXiv:2207.09453](https://arxiv.org/abs/2207.09453) (2022).
- <sup>51</sup>S. Doerr, M. Majewski, A. Pérez, A. Krämer, C. Clementi, F. Noe, T. Giorgino, and G. De Fabritiis, "TorchMD: A deep learning framework for molecular simulations," *J. Chem. Theory Comput.* **17**, 2355–2363 (2021).
- <sup>52</sup>M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," [arXiv:1603.04467](https://arxiv.org/abs/1603.04467) (2016), software available from tensorflow.org.
- <sup>53</sup>G. Arakelyan and G. Soghomonyan (2020). "Aim: An easy-to-use and performant open-source ml experiment tracking tool," Zenodo. [10.5281/zenodo.6536395](https://doi.org/10.5281/zenodo.6536395)
- <sup>54</sup>O. Yadan, Hydra—A framework for elegantly configuring complex applications, Github, 2019.
- <sup>55</sup>See <https://github.com/atomistic-machine-learning/schnetpack> for SchNetPack code.
- <sup>56</sup>See <https://schnetpack.readthedocs.io> for SchNetPack documentation.
- <sup>57</sup>A. H. Larsen, J. J. Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dułak, J. Friis, M. N. Groves, B. Hammer, C. Hargas, E. D. Hermes, P. C. Jennings, P. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rosengaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, and K. W. Jacobsen, "The atomic simulation environment—A python library for working with atoms," *J. Phys.: Condens. Matter* **29**, 273002 (2017).
- <sup>58</sup>E. Jones, T. Oliphant, P. Peterson *et al.*, SciPy: Open source scientific tools for Python, 2001.
- <sup>59</sup>J. Kermode and L. Pastewka, Matscipy, Github, 2019.
- <sup>60</sup>J. Behler, "Atom-centered symmetry functions for constructing high-dimensional neural network potentials," *J. Chem. Phys.* **134**, 074106 (2011).
- <sup>61</sup>N. Thomas, T. Smidt, S. Kearnes, L. Yang, L. Li, K. Kohlhoff, and P. Riley, "Tensor field networks: Rotation-and translation-equivariant neural networks for 3D point clouds," [arXiv:1802.08219](https://arxiv.org/abs/1802.08219) (2018).
- <sup>62</sup>M. Weiler, M. Geiger, M. Welling, B. Boomstra, and T. S. Cohen, "3D steerable CNNs: Learning rotationally equivariant features in volumetric data," in *Advances in Neural Information Processing Systems 31* (Curran Associates, Inc., 2018).
- <sup>63</sup>J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning* (PMLR, 2017), pp. 1263–1272.
- <sup>64</sup>M. Gastegger, K. T. Schütt, and K.-R. Müller, "Machine learning of solvent effects on molecular spectra and reactions," *Chem. Sci.* **12**, 11473–11483 (2021).
- <sup>65</sup>K. Schütt, O. Unke, and M. Gastegger, "Equivariant message passing for the prediction of tensorial properties and molecular spectra," in *Proceedings of the 38th International Conference on Machine Learning* (Proceedings of Machine Learning Research, PMLR, 2021), Vol. 139, pp. 9377–9388.
- <sup>66</sup>S. Batzner, A. Musaelian, L. Sun, M. Geiger, J. P. Mailoa, M. Kornbluth, N. Molinari, T. E. Smidt, and B. Kozinsky, "E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials," *Nat. Commun.* **13**, 2453 (2022).
- <sup>67</sup>D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," [arXiv:1606.08415](https://arxiv.org/abs/1606.08415) (2016).
- <sup>68</sup>M. Gastegger, J. Behler, and P. Marquetand, "Machine learning molecular dynamics for the simulation of infrared spectra," *Chem. Sci.* **8**, 6924–6935 (2017).
- <sup>69</sup>M. Veit, D. M. Wilkins, Y. Yang, R. A. DiStasio, Jr., and M. Ceriotti, "Predicting molecular dipole moments by combining atomic partial charges and atomic dipoles," *J. Chem. Phys.* **153**, 024113 (2020).
- <sup>70</sup>J. F. Ziegler and J. P. Biersack, "The stopping and range of ions in matter," in *Treatise on Heavy-Ion Science* (Springer, 1985), pp. 93–129.
- <sup>71</sup>O. T. Unke, S. Chmiela, M. Gastegger, K. T. Schütt, H. E. Sauceda, and K.-R. Müller, "SpookyNet: Learning force fields with electronic degrees of freedom and nonlocal effects," *Nat. Commun.* **12**, 7273 (2021).
- <sup>72</sup>R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld, "Quantum chemistry structures and properties of 134 kilo molecules," *Sci. Data* **1**, 140022 (2014).
- <sup>73</sup>A. S. Christensen and O. A. von Lilienfeld, "On the role of gradients for machine learning of molecular energies and forces," *Mach. Learn.: Sci. Technol.* **1**, 045018 (2020).
- <sup>74</sup>S. Chmiela, V. Vassilev-Galindo, O. T. Unke, A. Kabylda, H. E. Sauceda, A. Tkatchenko, and K.-R. Müller, "Accurate global machine learning force fields for molecules with hundreds of atoms," *Sci. Adv.* **9**(2), eadfo873 (2023).
- <sup>75</sup>B. Olsthoorn, R. M. Geilhufe, S. S. Borysov, and A. V. Balatsky, "Band gap prediction for large organic crystal structures with machine learning," *Adv. Quantum Technol.* **2**, 1900023 (2019).
- <sup>76</sup>A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, and K. A. Persson, "Commentary: The materials project: A materials genome approach to accelerating materials innovation," *APL Mater.* **1**, 011002 (2013).
- <sup>77</sup>See <https://pytorch-lightning.readthedocs.io> for PyTorch Lightning documentation.
- <sup>78</sup>F. Neese, "The ORCA program system," *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2**, 73–78 (2012).
- <sup>79</sup>A. Laio and M. Parrinello, "Escaping free-energy minima," *Proc. Natl. Acad. Sci. U. S. A.* **99**, 12562–12566 (2002).
- <sup>80</sup>H. J. C. Berendsen, J. P. M. Postma, W. F. Van Gunsteren, A. DiNola, and J. R. Haak, "Molecular dynamics with coupling to an external bath," *J. Chem. Phys.* **81**, 3684–3690 (1984).
- <sup>81</sup>G. Bussi and M. Parrinello, "Accurate sampling using Langevin dynamics," *Phys. Rev. E* **75**, 056707 (2007).
- <sup>82</sup>G. J. Martyna, M. L. Klein, and M. Tuckerman, "Nosé-Hoover chains: The canonical ensemble via continuous dynamics," *J. Chem. Phys.* **97**, 2635–2643 (1992).
- <sup>83</sup>M. Ceriotti, G. Bussi, and M. Parrinello, "Colored-noise thermostats à la carte," *J. Chem. Theory Comput.* **6**, 1170–1180 (2010).
- <sup>84</sup>M. Ceriotti, M. Parrinello, T. E. Markland, and D. E. Manolopoulos, "Efficient stochastic thermostating of path integral molecular dynamics," *J. Chem. Phys.* **133**, 124104 (2010).
- <sup>85</sup>M. Rossi, M. Ceriotti, and D. E. Manolopoulos, "How to remove the spurious resonances from ring polymer molecular dynamics," *J. Chem. Phys.* **140**, 234116 (2014).
- <sup>86</sup>F. Uhl, D. Marx, and M. Ceriotti, "Accelerated path integral methods for atomistic simulations at ultra-low temperatures," *J. Chem. Phys.* **145**, 054101 (2016).
- <sup>87</sup>G. J. Martyna, M. E. Tuckerman, D. J. Tobias, and M. L. Klein, "Explicit reversible integrators for extended systems dynamics," *Mol. Phys.* **87**, 1117–1157 (1996).
- <sup>88</sup>V. Kapil, M. Rossi, O. Marsalek, R. Petraglia, Y. Litman, T. Spura, B. Cheng, A. Cuzzocrea, R. H. Meißner, D. M. Wilkins *et al.*, "i-PI 2.0: A universal force engine for advanced molecular simulations," *Comput. Phys. Commun.* **236**, 214–223 (2019).
- <sup>89</sup>N. Gebauer, M. Gastegger, and K. Schütt, "Symmetry-adapted generation of 3d point sets for the targeted discovery of molecules," in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019), pp. 7566–7578.
- <sup>90</sup>See <https://github.com/atomistic-machine-learning/schnetpack-gschnet> for cG-SchNet code.
- <sup>91</sup>M. Ceriotti, "Unsupervised machine learning in atomistic simulations, between predictions and understanding," *J. Chem. Phys.* **150**, 150901 (2019).

<sup>92</sup>A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.* **271**, 108171 (2022).

<sup>93</sup>M. Thomas, M. Brehm, R. Fligg, P. Vöhringer, and B. Kirchner, "Computing vibrational spectra from *ab initio* molecular dynamics," *Phys. Chem. Chem. Phys.* **15**, 6608–6622 (2013).

<sup>94</sup>F. Jensen, *Introduction to Computational Chemistry* (Wiley, 2007).

<sup>95</sup>See <https://github.com/atomistic-machine-learning/schnetpack-gschnet> for cG-SchNet code.