Getting Started

1 Contents 3
Python Module Index 111
Index 113
SchNetPack is a toolbox for the development and application of deep neural networks to the prediction of potential energy surfaces and other quantum-chemical properties of molecules and materials. It contains basic building blocks of atomistic neural networks, manages their training and provides simple access to common benchmark datasets. This allows for an easy implementation and evaluation of new models.
SchNetPack aims to provide accessible atomistic neural networks that can be trained and applied out-of-the-box, while still being extensible to custom atomistic architectures.

1.1 Installation

1.1.1 Requirements

- Python (>=3.6)
- NumPy
- Pytorch (>=1.1)
- ASE (>=3.16)
- TensorboardX (For improved training visualization)
- h5py
- tqdm
- PyYaml

1.1.2 Building from source

You can directly build a path from source through git clone. To do so, just type:

```
$ git clone https://github.com/atomistic-machine-learning/schnetpack.git <dest_dir>
```

then move in the new directory <dest_dir>:

```
$ cd <dest_dir>
```
install both requirements and schnetpack:

```bash
$ pip install -r requirements.txt
$ pip install .
```

and run tests to be sure everything runs as expected:

```bash
$ pytest
```

Once that’s done, you are ready to go!

**Note:** If your OS doesn’t have numpy, pytorch, and ase packages installed, and the previous command didn’t work for you, you can install those requirements through:

```bash
$ pip install --upgrade --user numpy torch ase
```

### 1.1.3 Visualization with Tensorboard

While SchNetPack is based on PyTorch, it is possible to use Tensorboard, which comes with TensorFlow, to visualize the learning progress. While this is more convenient for visualization, you need to install TensorBoard in order to view the event files SchNetPack produces. Even though there is a standalone version, the easiest way to get Tensorboard is by installing TensorFlow, e.g. using pip:

```bash
$ pip install tensorflow
```

### 1.2 Scripts for benchmark data sets

The best place to start is training a SchNetPack model on a common benchmark dataset. Scripts for common datasets are provided by SchNetPack and inserted into your PATH during installation.

The example script allows to train and evaluate both SchNet and wACSF neural networks. In the following, we focus on using the script for the QM9 dataset, but the same procedure applies for the other benchmark datasets as well. The training can be started using:

```bash
$ spk_run.py train <schnet/wacsf> <qm9/ani1/...> <dbpath> <modeldir> --split num_˓
→train num_val [--cuda]
```

where num_train and num_val need to be replaced by the number of training and validation datapoints respectively. You can choose between SchNet and wACSF networks and have to provide a directory to store the model and the location of the dataset, which has to be a ASE DB file (.db or .json). It will be downloaded automatically if it does not exist.

**Note:** Please be warned that the ANI-1 dataset is huge (more than 20gb).

With the --cuda flag, you can activate GPU training. The default hyper-parameters should work fine, however, you can change them through command-line arguments. Please refer to the help at the command line:

```bash
$ spk_run.py train <schnet/wacsf> --help
```
The training progress will be logged in <modeldir>/log. The default is a basic logging with CSV files. Advanced logging with TensorBoard event files can be activated using --logger tensorboard (see above).

To evaluate the trained model that showed the best validation error during training (i.e., early stopping), call:

```
$ spk_run.py eval <datapath> <modeldir> [--split train val test] [--cuda]
```

which will write a result file evaluation.txt into the model directory.

**Tip:** <modeldir> should point to a directory in which a pre-trained model is stored. As an argument for the --split flag for evaluation you should choose among one of training, validation or test subsets.

### 1.3 Using Scripts with custom Datasets

The script for benchmark data can also train a model on custom data sets, by using:

```
$ spk_run.py train <schnet/wacsf> custom <dbpath> <modeldir> --split num_train num_val --property your_property [--cuda]
```

Depending on your data you will need to define some settings that have already been pre-selected for the benchmark data. In order to show how to use the script on arbitrary data sets, we will use the MD17 data set and treat it as a custom data set. First of all we need to define the property that we want to use for training. In this example we will train the model on the energy labels. If we want to use the forces during training, we need to add the --derivative argument and also set --negative_dr, because the gradient of the energy predictions corresponds to the negative forces. Since energy is a property that depends on the total number of atoms we select --aggregation_mode sum. Other properties (e.g. homo, lumo, ...) do not depend on the total number of atoms and will therefore use the mean aggregation mode. While most properties should be trained with the spk.nn.Atomwise output module which is selected by default, some properties require special output modules. Models using the spk.SchNet representation support dipole_moment and electronic.spatial.extent. Note that if your model is based on the spk.BehlerSFBlock representation you need to select between elemental_atomwise and elemental_dipole_moment. The output module selection is defined with --output_module <atomwise/elemental/atomwise/dipole_moment/...>. The final command for the MD17 example would be:

```
$ spk_run.py train <schnet/wacsf> custom <dbpath> <modeldir> --split num_train num_val --property energy --derivative forces --negative_dr --aggregation_mode sum [--cuda]
```

The command for training a QM9-like data set on dipole moments would be:

```
$ spk_run.py train <schnet/wacsf> custom <dbpath> <modeldir> --split num_train num_val --property dipole_moment --output_module dipole_moment --aggregation_mode sum [--cuda]
```

The evaluation of the trained model uses the same commands as any pre-implemented data set.

### 1.4 Using Argument Files for Training

An argument file with all training arguments is created at the beginning of every training session and can be found at <modeldir>/args.json. These argument files can be modified and used for new training sessions. In order to build a file with default settings run:

```

1.3. Using Scripts with custom Datasets

5
```
$ spk_run.py train <schnet/wacsf> custom <dbpath> <modeldir>

This will create the <modeldir> which contains the argument file, while the training session will fail because --split is not selected. You can now modify the arguments and use them for training:

$ spk_run.py from_json <modeldir>/args.json

1.5 Supported Models

SchNetPack currently supports SchNet and (w)ACSF.

1.5.1 SchNet

SchNet is an end-to-end deep neural network architecture based on continuous-filter convolutions. It follows the deep tensor neural network framework, i.e. atom-wise representations are constructed by starting from embedding vectors that characterize the atom type before introducing the configuration of the system by a series of interaction blocks.

1.5.2 ACSF & (w)ACSF

ACSFs describe the local chemical environment around a central atom via a combination of radial and angular distribution functions. Those model come from Behler–Parrinello networks, based on atom centered symmetry functions (ACSFs). Moreover, wACSF comes as an extensions of this latest. It uses weighted atom-centered symmetry functions (wACSF). Whereas for SchNet, features are learned by the network, for ACSFs (and wACSFs) we need to introduce some handcrafted features before training.

1.6 Benchmark Datasets

SchNetPack provides convenient interfaces to popular benchmark datasets in order to train and test its model.

1.6.1 QM9

The qm9 dataset contains 133,885 organic molecules with up to nine heavy atoms from C, O, N and F.

---

1.6.2 MD17

The md17 dataset allows to do molecular dynamics of small molecules containing molecular forces\(^9\).

1.6.3 ANI1

The ani1 dataset consists of more than 20 million conformations for 57454 small organic molecules from C, O and N\(^8\).

1.6.4 Materials Project

A repository of bulk crystals containing atom types ranging across the whole periodic table up to \(Z = 94\)\(^11\).

1.6.5 OMDB

The omdb dataset contains data from Organic Materials Database (OMDB) of bulk organic crystals. This database contains DFT (PBE) band gap (OMDB-GAP1 database) for 12500 non-magnetic materials. The registration to the OMDB is free for academic users.\(^10\).

1.7 References

1.8 Benchmarks and Trained Models

1.8.1 QM9

The QM9 benchmarks use a SchNet model with 128 features, 6 interaction blocks, 50 gaussians and a cosine cutoff-function with a cutoff at 10. The model is trained with a batch size of 100 and an initial learning rate of 1e-4. The learning rate is scheduled using spk.train.ReduceOnPlateauHook with a learning rate patience of 25, a learning rate decay of 0.8 and a minimum learning rate of 1e-6. 100000 data-points have been used for training and 10000 data-points are used for validation. The remaining dataset is used for evaluating the test-set.

---

\(^9\) Quantum-Machine.org


\(^10\) Organic Materials Database (OMDB)
<table>
<thead>
<tr>
<th>Property</th>
<th>MAE</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>heat_capacity</td>
<td>0.034</td>
<td>Kcal/mol</td>
</tr>
<tr>
<td>zpve</td>
<td>1.616</td>
<td>meV</td>
</tr>
<tr>
<td>gap</td>
<td>0.074</td>
<td>eV</td>
</tr>
<tr>
<td>energy_U0</td>
<td>0.012</td>
<td>eV</td>
</tr>
<tr>
<td>enthalpy_H</td>
<td>0.012</td>
<td>eV</td>
</tr>
<tr>
<td>homo</td>
<td>0.047</td>
<td>eV</td>
</tr>
<tr>
<td>electronic.spatial.extent</td>
<td>0.158</td>
<td>Bohr**2</td>
</tr>
<tr>
<td>energy_U</td>
<td>0.012</td>
<td>eV</td>
</tr>
<tr>
<td>free_energy</td>
<td>0.013</td>
<td>eV</td>
</tr>
<tr>
<td>isotropic.polarizability</td>
<td>0.124</td>
<td>Bohr**3</td>
</tr>
<tr>
<td>lumo</td>
<td>0.039</td>
<td>eV</td>
</tr>
<tr>
<td>dipole_moment</td>
<td>0.021</td>
<td>Debye</td>
</tr>
</tbody>
</table>

### 1.8.2 MD17

The MD17 benchmarks use a SchNet model with 64 features, 6 interaction blocks, 25 gaussians and a cosine cutoff-function with a cutoff at 5. The model is trained with a batch size of 100 and an initial learning rate of 1e-3. The learning rate is scheduled using spk.train.ReduceOnPlateauHook with a learning rate patience of 150, a learning rate decay of 0.8 and a minimum learning rate of 1e-6. The loss tradeoff is set to multiply the energy loss with 0.01 and the forces loss value with 0.99. 950 data-points have been used for training and 50 data-points are used for validation. The remaining dataset is used for evaluating the test-set.

<table>
<thead>
<tr>
<th>Property</th>
<th>MAE</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy</td>
<td>0.069</td>
<td>eV</td>
</tr>
</tbody>
</table>

### 1.8.3 Trained Models

The trained benchmark-models can be downloaded [here](#).

### 1.9 Preparing and loading your data

This tutorial introduces how SchNetPack stores and loads data. Before we can start training neural networks with SchNetPack, we need to prepare our data. This is because SchNetPack has to streams the reference data from disk during training in order to be able to handle large datasets. Therefore, it is crucial to use data format that allows for fast random read access. We found that the ASE database format fulfills perfectly. To further improve the performance, we internally encode properties in binary. However, as long as you only access the ASE database via the provided SchNetPack AtomsData class, you don’t have to worry about that.

[1]: `from schnetpack import AtomsData`

### 1.9.1 Predefined datasets

SchNetPack supports several benchmark datasets that can be used without preparation. Each one can be accessed using a corresponding class that inherits from DownloadableAtomsData, which supports automatic download and conversion. Here, we show how to use these data sets at the example of the QM9 benchmark.
First, we have to import the dataset class and instantiate it. This will automatically download the data to the specified location.

```python
from schnetpack.datasets import QM9
qm9data = QM9('./qm9.db', download=True)
```

Let's have a closer look at this dataset. We can find out how large it is and which properties it supports:

```python
print('Number of reference calculations:', len(qm9data))
print('Available properties:]
for p in qm9data.available_properties:
    print('-', p)
```

Number of reference calculations: 133885
Available properties:
- rotational_constant_A
- rotational_constant_B
- rotational_constant_C
- dipole_moment
- isotropic_polarizability
- homo
- lumo
- gap
- electronic_spatial_extent
- zpve
- energy_U0
- energy_U
- enthalpy_H
- free_energy
- heat_capacity

We can load data points using zero-base indexing. The result is a dictionary containing the geometry and properties:

```python
example = qm9data[0]
print('Properties:]
for k, v in example.items():
    print('-', k, ':', v.shape)
```

Properties:
- rotational_constant_A : torch.Size([1])
- rotational_constant_B : torch.Size([1])
- rotational_constant_C : torch.Size([1])
- dipole_moment : torch.Size([1])
- isotropic_polarizability : torch.Size([1])
- homo : torch.Size([1])
- lumo : torch.Size([1])
- gap : torch.Size([1])
- electronic_spatial_extent : torch.Size([1])
- zpve : torch.Size([1])
- energy_U0 : torch.Size([1])
- energy_U : torch.Size([1])
- enthalpy_H : torch.Size([1])
- free_energy : torch.Size([1])
- heat_capacity : torch.Size([1])
- _atomic_numbers : torch.Size([5])
- _positions : torch.Size([5, 3])

(continues on next page)
We see that all available properties have been loaded as torch tensors with the given shapes. Keys with an underscore indicate that these names are reserved for internal use. This includes the geometry (_atomic_numbers, _positions, _cell), the index within the dataset (_idx) as well as information about neighboring atoms and periodic boundary conditions (_neighbors, _cell_offset).

**Note:** Neighbors are collected using an `EnvironmentProvider`, that can be passed to the `AtomsData` constructor. The default is the `SimpleEnvironmentProvider`, which constructs the neighbor list using a full distance matrix. This is suitable for small molecules. We supply environment providers using a cutoff (`AseEnvironmentProvider`, `TorchEnvironmentProvider`) that are able to handle larger molecules and periodic boundary conditions.

We can directly obtain an ASE atoms object as follows:

```python
[5]: at = qm9data.get_atoms(idx=0)
print('Atoms object:', at)

at2, props = qm9data.get_properties(idx=0)
print('Atoms object (not the same):', at2)
print('Equivalent:', at2 == at, '; not the same object:', at2 is at)
```

Atoms object: Atoms(symbols='CH4', pbc=False)
Atoms object (not the same): Atoms(symbols='CH4', pbc=False)
Equivalent: True ; not the same object: False

Alternatively, all property names are pre-defined as class-variable for convenient access:

```python
[6]: print('Total energy at 0K:', props[QM9.U0])
print('HOMO:', props[QM9.homo])
```

Total energy at 0K: tensor([-1101.4878])
HOMO: tensor([-10.5499])

### 1.9.2 Preparing your own data

In the following we will create an ASE database from our own data. For this tutorial, we will use a dataset containing a molecular dynamics (MD) trajectory of ethanol, which can be downloaded here.

```python
[7]: import os
if not os.path.exists('./ethanol_dft.zip'):
    !wget http://quantum-machine.org/gdml/data/xyz/ethanol_dft.zip
if not os.path.exists('./ethanol.xyz'):
    !unzip ./ethanol_dft.zip
```

The data set is in xyz format with the total energy given in the comment row. For this kind of data, we supply a script that converts it into the SchNetPack ASE DB format.

`schnetpack_parse.py ./ethanol.xyz ./ethanol.db`
In the following, we show how this can be done in general, so that you apply this to any other data format.

First, we need to parse our data. For this we use the IO functionality supplied by ASE. In order to create a SchNetPack DB, we require a list of ASE “Atoms” objects as well as a corresponding list of dictionaries \([\{\text{property\_name1: property1\_molecule1}, \{\text{property\_name1: property1\_molecule2}, \ldots\}\}\) containing the mapping from property names to values.

```python
from ase.io import read
import numpy as np

# load atoms from xyz file. Here, we only parse the first 10 molecules
atoms = read('./ethanol.xyz', index=':10')

# comment line is weirdly stored in the info dictionary as key by ASE. here it
→ corresponds to the energy
print('Energy:', atoms[0].info)
print()

# parse properties as list of dictionaries
property_list = []
for at in atoms:
    # All properties need to be stored as numpy arrays.
    # Note: The shape for scalars should be (1,), not ()
    # Note: GPUs work best with float32 data
    energy = np.array([float(list(at.info.keys())[0])], dtype=np.float32)
    property_list.append(
        {'energy': energy}
    )

print('Properties:', property_list)

Energy: {'-97208.40600498248': True}
Properties: [{'energy': array([-97208.41], dtype=float32)},
             {'energy': array([-97208.375], dtype=float32)},
             {'energy': array([-97208.04], dtype=float32)},
             {'energy': array([-97207.5], dtype=float32)},
             {'energy': array([-97206.84], dtype=float32)},
             {'energy': array([-97206.1], dtype=float32)},
             {'energy': array([-97205.266], dtype=float32)},
             {'energy': array([-97204.29], dtype=float32)},
             {'energy': array([-97203.16], dtype=float32)},
             {'energy': array([-97201.875], dtype=float32)}]

Once we have our data in this format, it is straightforward to create a new SchNetPack DB and store it.

```python
% rm './new_dataset.db'
new_dataset = AtomsData('./new_dataset.db', available_properties=['energy'])
new_dataset.add_systems(atoms, property_list)
```

Now we can have a look at the data in the same way we did before for QM9:

```python
print('Number of reference calculations:', len(new_dataset))
print('Available properties:')
for p in new_dataset.available_properties:
    print('-', p)
print()
example = new_dataset[0]
print('Properties of molecule with id 0:')
```

(continues on next page)
for k, v in example.items():
    print('-', k, ':', v.shape)

Number of reference calculations: 10
Available properties:
- energy

Properties of molecule with id 0:
- energy : torch.Size([1])
- _atomic_numbers : torch.Size([9])
- _positions : torch.Size([9, 3])
- _cell : torch.Size([3, 3])
- _neighbors : torch.Size([9, 8])
- _cell_offset : torch.Size([9, 8, 3])
- _idx : torch.Size([1])

The same way, we can store multiple properties, including atomic properties such as forces, or tensorial properties such as polarizability tensors.

In the following tutorials, we will describe how these datasets can be used to train neural networks.

1.10 Training a neural network on QM9

This tutorial will explain how to use SchNetPack for training a model on the QM9 dataset and how the trained model can be used for further.

First, we import the necessary modules and create a new directory for the data and our model.

```
[1]: import os
    import schnetpack as spk

qm9tut = './qm9tut'
if not os.path.exists('qm9tut'):
    os.makedirs(qm9tut)
```

1.10.1 Loading the data

As explained in the previous tutorial, datasets in SchNetPack are loaded with the AtomsLoader class or one of the sub-classes that are specialized for common benchmark datasets. The QM9 dataset class will download and convert the data. We will only use the inner energy at 0K $U_0$, so all other properties do not need to be loaded:

```
[2]: from schnetpack.datasets import QM9

qm9data = QM9('./qm9.db', download=True, load_only=[QM9.U0], remove__uncharacterized=True)
```

Splitting the data

Next, we split the data into training, validation and test set. Here, we choose to use 1000 training examples, 1000 examples for validation and the remaining data as test set. The corresponding indices are stored in the split.npz file.
Next, we will create an `AtomsLoader` for each split. This will take care of shuffling, batching and asynchronously loading the data during training and evaluation.

```python
[4]: train_loader = spk.AtomsLoader(train, batch_size=100, shuffle=True)
val_loader = spk.AtomsLoader(val, batch_size=100)
```

### Dataset statistics

Before building the model, we need some statistic about our target property for good initial conditions. We will get this from the training examples. For QM9, we also have single-atom reference values stored in the metadata:

```python
[6]: atomrefs = qm9data.get_atomref(QM9.U0)
print('U0 of hydrogen:', '{:.2f}'.format(atomrefs[QM9.U0][1][0]), 'eV')
print('U0 of carbon:', '{:.2f}'.format(atomrefs[QM9.U0][6][0]), 'eV')
print('U0 of oxygen:', '{:.2f}'.format(atomrefs[QM9.U0][8][0]), 'eV')
```

These can be used together with the mean and standard deviation of the energy per atom to initialize the model with a good guess of the energy of a molecule. When calculating these statistics, we pass the atomref to take into account, that the model will add these atomrefs to the predicted energy later, so that this part of the energy does not have to be considered in the statistics, i.e.

\[
\mu U_0 = \frac{1}{n_{\text{train}}} \sum_{n=1}^{n_{\text{train}}} \left( U_{0,n} - \sum_{i=1}^{n_{\text{atoms},n}} U_{0,Z_{n,i}} \right)
\]  

for the mean and analog for the standard deviation. In this case, this corresponds to the mean and std. dev of the atomization energy per atom.

```python
[7]: means, stddevs = train_loader.get_statistics(
    QM9.U0, get_atomwise_statistics=True, single_atom_ref=atomrefs
)
print('Mean atomization energy / atom:', means[QM9.U0])
print('Std. dev. atomization energy / atom:', stddevs[QM9.U0])
```

### 1.10.2 Building the model

The next step is to build the neural network model. This consists of 2 parts:

1. The representation part which either constructs atom-wise features, e.g. with SchNet, or build a fixed descriptor such as atom-centered symmetry functions.
2. One or more output modules for property prediction.
We will use a SchNet module with 3 interaction layers, a 5 Angstrom cosine cutoff with pairwise distances expanded on 20 Gaussians and 50 atomwise features and convolution filters here, since we only have a few training examples (click here for details on SchNet). Then, we will use one Atomwise modules to predict the energy, which takes mean and standard deviation per atom of the property for initialization. Both modules are then combined to an AtomisticModel.

```python
[8]: schnet = spk.representation.SchNet(
    n_atom_basis=30, n_filters=30, n_gaussians=20, n_interactions=5,
    cutoff=4., cutoff_network=spk.nn.cutoff.CosineCutoff
)
output_U0 = spk.atomistic.Atomwise(n_in=30, atomref=atomrefs['QM9.U0'], property='QM9.U0',
    mean=means['QM9.U0'], stddev=stddevs['QM9.U0'])
model = spk.AtomisticModel(representation=schnet, output_modules=output_U0)
```

### 1.10.3 Training the model

To train the model, we will use the Trainer class that comes with SchNetPack. For this, we need to first define a loss function and select an optimizer. As the loss function, we will use the mean squared error of the energy

$$\ell(E_{\text{ref}}, E_{\text{pred}}) = \frac{1}{n_{\text{train}}} \sum_{n=1}^{n_{\text{train}}} (E_{\text{ref}} - E_{\text{pred}})^2$$

This will be minimized using the Adam optimizer from PyTorch.

```python
[9]: from torch.optim import Adam

# loss function
def mse_loss(batch, result):
    diff = batch['QM9.U0'] - result['QM9.U0']
    err_sq = torch.mean(diff ** 2)
    return err_sq

# build optimizer
optimizer = Adam(model.parameters(), lr=1e-2)
```

We can give the Trainer hooks, that are called at certain points during the training loop. This is useful to customize the training process, e.g. with logging, learning rate schedules or stopping criteria. Here, we set up a basis logging as well as a learning rate schedule that reduces the learning rate by factor 0.8 after 5 epochs without improvement of the validation loss.

The logger takes a list of validation metrics that specify what is going to be stored. In this example, we want to log the mean absolute and root mean squared error of the $U_0$ energy prediction.

```python
[10]: from schnetpack.train import CSVHook, ReduceLROnPlateauHook

# before setting up the trainer, remove previous training checkpoints and logs
%rm -r ./qm9tut/checkpoints
%rm -r ./qm9tut/log.csv

import schnetpack.train as trn

loss = trn.build_mse_loss(['QM9.U0'])
metrics = [spk.metrics.MeanAbsoluteError('QM9.U0')]
hooks = [
    trn.CSVHook(log_path=qm9tut, metrics=metrics),
    trn.ReduceLROnPlateauHook(optimizer,
        patience=5, factor=0.8, min_lr=1e-6,
    )
]
```
We can run the training for a given number of epochs. If we don’t give a number, the trainer runs until a stopping criterion is met. For the purpose of this tutorial, we let it train for 200 epochs (on GPU this should take about x minutes).

```python
import numpy as np
import matplotlib.pyplot as plt
from ase.units import kcal, mol
```

```python
results = np.loadtxt(os.path.join(qm9tut, 'log.csv'), skiprows=1, delimiter=',')
time = results[:,0]-results[0,0]
learning_rate = results[:,1]
train_loss = results[:,2]
val_loss = results[:,3]
val_mae = results[:,4]
```

```python
print('Final validation MAE:', np.round(val_mae[-1], 2), 'eV =', np.round(val_mae[-1] / (kcal/mol), 2), 'kcal/mol')
```

Final validation MAE: 0.14 eV = 3.2 kcal/mol

You can also call this method several times to continue training. For the training to run until convergence, remove the `n_epochs` argument or set it to a very large number.

Let us finally have a look at the CSV log:
Of course, the prediction can be improved by letting the training run longer, increasing the patience, the number of neurons and interactions or using regularization techniques.

1.10.4 Using the model

Having trained a model for QM9, we are going to use it to obtain some predictions. First, we need to load the model. The Trainer stores the best model in the model directory which can be loaded using PyTorch:

```python
import torch
best_model = torch.load(os.path.join(qm9tut, 'best_model'))
```

To evaluate it on the test data, we create a data loader for the test split:

```python
test_loader = spk.AtomsLoader(test, batch_size=100)

err = 0
for count, batch in enumerate(test_loader):
    batch = {k: v.to(device) for k, v in batch.items()}
    pred = best_model(batch)
    tmp = torch.sum(torch.abs(pred[QM9.U0]-batch[QM9.U0]))
    tmp = tmp.detach().cpu().numpy()
    err += tmp

percent = '{:3.2f}'.format(count/len(test_loader)*100)
print('Progress: ', percent+'%', end='')

err /= len(test)
print('Test MAE', np.round(err, 2), 'eV =', np.round(err / (kcal/mol), 2), 'kcal/mol')
```

If your data is not already in SchNetPack format, a more convenient way is to use ASE atoms objects with the provided AtomsConverter.

```python
converter = spk.data.AtomsConverter(device=device)
inputs = converter(at)
print('Keys:', list(inputs.keys()))
print('Truth:', props[QM9.U0].cpu().numpy()[0])
pred = model(inputs)
print('Prediction:', pred[QM9.U0].detach().cpu().numpy()[0,0])
```

Alternatively, one can use the SpkCalculator as an interface to ASE:
1.10.5 Summary

In this tutorial, we have trained and evaluated a SchNet model on a small subset of QM9. A full training script with is available here.

In the next tutorial, we will show how to learn potential energy surfaces and forces field as well as performing molecular dynamics simulations with SchNetPack.

1.11 Training a model on forces and energies

In addition to the energy, machine learning models can also be used to model molecular forces. These are \( N_{\text{atoms}} \times 3 \) arrays describing the Cartesian force acting on each atom due to the overall (potential) energy. They are formally defined as the negative gradient of the energy \( E_{\text{pot}} \) with respect to the nuclear positions \( \mathbf{R} \):

\[
\mathbf{F}^{(\alpha)} = -\frac{\partial E_{\text{pot}}}{\partial \mathbf{R}^{(\alpha)}},
\]

where \( \alpha \) is the index of the nucleus.

The above expression offers a straightforward way to include forces in machine learning models by simply defining a model for the energy and taking the appropriate derivatives. The resulting model can directly be trained on energies and forces. Moreover, in this manner energy conservation and the correct behaviour under rotations of the molecule is guaranteed.

Using forces in addition to energies to construct a machine learning model offers several advantages. Accurate force predictions are important for molecular dynamics simulations, which will be covered in the subsequent tutorial. Forces also encode a greater wealth of information than the energies. For every molecule, only one energy is present, while there are \( 3N_{\text{atoms}} \) force entries. This property, combined with the fact that reference forces can be computed at the same cost as energies, makes models trained on forces and energies very data efficient.

In the following, we will show how to train such force models and how to use them in practical applications.

1.11.1 Preparing the data

The process of preparing the data is similar to the tutorial on QM9. We begin by importing all relevant packages and generating a directory for the tutorial experiments.

```python
import schnetpack as spk
import os
forcetut = './forcetut'
if not os.path.exists(forcetut):
    os.makedirs(forcetut)
```

Next, the data needs to be loaded from a suitable dataset. For convenience, we use the MD17 dataset class provided in SchNetPack, which automatically downloads and builds suitable databases containing energies and forces for a range of small organic molecules. In this case, we use the ethanol molecule as an example.
For custom datasets, the data would have to be loaded via the SchNetPack AtomsData class. In this case, one needs to make sure, that the naming of properties is kept consistent. The schnetpack.Properties module provides standard names for a wide range of properties. Here, we use the definitions provided with the MD17 class.

In order to train force models, forces need to be included in the reference data. Once the dataset was loaded, this can e.g. be checked by calling the get_properties function for sample entries. Here we look at the configuration with the index 0. The function returns an ASE Atoms object and a dictionary containing the loaded properties:

```
[3]: atoms, properties = ethanol_data.get_properties(0)

print('Loaded properties:
', *
['{:s}:
'.format(i) for i in properties.keys()])
```

```
Loaded properties:
  energy
  forces
  _atomic_numbers
  _positions
  _cell
  _neighbors
  _cell_offset
```

As can be seen, energy and forces are included in the properties dictionary. To have a look at the forces array and check whether it has the expected dimensions, we can call:

```
[4]: print('Forces:
', properties[MD17.forces])
print('Shape:
', properties[MD17.forces].shape)
```

```
Forces:
tensor([[ 1.4517e+00,  6.0192e+00,  5.2068e-07],
  [ 1.7953e+01, -5.1624e+00,  3.4900e-07],
  [-4.0884e+00,  2.2590e+01,  3.3088e+00],
  [-1.1416e+00, -9.7469e+00,  7.6473e+00],
  [-1.1416e+00, -9.7469e+00, -7.6473e+00],
  [-2.4821e+00,  4.9335e+00,  4.3700e+00],
  [-2.4821e+00,  4.9335e+00, -4.3700e+00],
  [-5.5148e+00, -3.0207e+01, -8.9093e-09],
  [-2.4393e+00, -1.0838e+01, -6.0721e-08]])
Shape: torch.Size([9, 3])
```

The atoms object can e.g. be used to visualize the ethanol molecule:

```
[5]: from ase.visualize import view
view(atoms, viewer='x3d')
```

```
<IPython.core.display.HTML object>
```

Next, the data is split into training (1000 points), test (500 points) and validation set (remainder) and data loaders are created. This is done in the same way as described in the QM9 tutorial.

```
[6]: train, val, test = spk.train_test_split(
    data=ethanol_data,
    num_train=1000,
    (continues on next page)
```
Once again, we want to use the mean and standard deviation of the energies in the training data to precondition our model. This only needs to be done for the energies, since the forces are obtained as derivatives and automatically capture the scale of the data. Unlike in the case of QM9, the subtraction of atomic reference energies is not necessary, since only configurations of the same molecule are loaded. All this can be done via the `get_statistics` function of the `AtomsLoader` class:

```python
[7]: means, stddevs = train_loader.get_statistics(
    spk.datasets.MD17.energy, get_atomwise_statistics=True
)
print('Mean atomization energy / atom: {:.12f} [kcal/mol]'.format(means[MD17.˓→energy][0]))
print('Std. dev. atomization energy / atom: {:.12f} [kcal/mol]'.format(stddevs[MD17.˓→energy][0]))
```

Mean atomization energy / atom: -10799.5244 [kcal/mol]
Std. dev. atomization energy / atom: 0.4679 [kcal/mol]

### 1.1.2 Building the model

After having prepared the data in the above way, we can now build and train the force model. This is done in the same two steps as described in QM9 tutorial:

1. Building the representation
2. Defining an output module

For the representation we can use the same SchNet layer as in the previous tutorial:

```python
[8]: n_features = 128

schnet = spk.representation.SchNet(
    n_atom_basis=n_features,
    n_filters=n_features,
    n_gaussians=25,
    n_interactions=3,
    cutoff=5.0,
    cutoff_network=spk.nn.cutoff.CosineCutoff
)
```

Since we want to model forces, the `Atomwise` output module needs to be adapted slightly. We will still use one module to predict the energy, preconditioning with the mean and standard deviation per atom of the energy.

However, since the forces should be described as the derivative of the energy, we have to indicate that the corresponding derivative of the model should be computed. This is done by specifying `derivative=MD17.forces`, which also assigns the computed derivative to the property `MD17.forces`. Since the forces are the negative gradient, we also need to enable `negative_dr=True`, which simply multiplies the derivative with -1.

### 1.11 Training a model on forces and energies
energy_model = spk.atomistic.Atomwise(
    n_in=n_features,
    property=MD17.energy,
    mean=means[MD17.energy],
    stddev=stddevs[MD17.energy],
    derivative=MD17.forces,
    negative_dr=True
)

Both modules are then combined to an AtomisticModel.

def model = spk.AtomisticModel(representation=schnet, output_modules=energy_model)

## 1.11.3 Training the model

To train the model on energies and forces, we need to update the loss function to include the latter. This combined loss function is:

\[
\mathcal{L}(E_{\text{ref}}, F_{\text{ref}}, E_{\text{pred}}, F_{\text{pred}}) = \frac{1}{n_{\text{train}}} \sum_{n=1}^{n_{\text{min}}} \left[ \rho (E_{\text{ref}} - E_{\text{pred}})^2 + \frac{(1 - \rho)}{3N_{\text{atoms}}} \sum_{\alpha} \left\| F_{\text{ref}}^{(\alpha)} - F_{\text{pred}}^{(\alpha)} \right\|^2 \right], \tag{1.3}
\]

where we take the predicted forces to be:

\[
F_{\text{pred}}^{(\alpha)} = -\frac{\partial E_{\text{pred}}}{\partial R^{(\alpha)}}. \tag{1.4}
\]

We have introduced a parameter \( \rho \) in order to control the tradeoff between energy and force loss. By varying this parameter, the accuracy on energies and forces can be tuned. Setting \( \rho = 0 \) only forces are trained, while in the case of \( \rho = 1 \) only energies are learned. Using PyTorch, we can implement this loss function in the following way:

```python
import torch
rho_tradeoff = 0.1

def loss(batch, result):
    # compute the mean squared error on the energies
    diff_energy = batch[MD17.energy]-result[MD17.energy]
    err_sq_energy = torch.mean(diff_energy ** 2)

    # compute the mean squared error on the forces
    diff_forces = batch[MD17.forces]-result[MD17.forces]
    err_sq_forces = torch.mean(diff_forces ** 2)

    # build the combined loss function
    err_sq = rho_tradeoff*err_sq_energy + (1-rho_tradeoff)*err_sq_forces
    return err_sq
```

Next, we proceed in the same manner as in the QM9 tutorial. First, we specify that the Adam optimizer from PyTorch should be used to train the model:

```python
from torch.optim import Adam
optimizer = Adam(model.parameters(), lr=5e-4)
```
Then, we construct the trainer hooks to monitor the training process and anneal the learning rate. Since we also learn forces in addition to the energies, we include a corresponding metric into the logger.

```python
# before setting up the trainer, remove previous training checkpoints and logs
%rm -rf ./forcetut/checkpoints
%rm -rf ./forcetut/log.csv

import schnetpack.train as trn

# set up metrics
metrics = [
    spk.metrics.MeanAbsoluteError(MD17.energy),
    spk.metrics.MeanAbsoluteError(MD17.forces)
]

# construct hooks
hooks = [
    trn.CSVHook(log_path=forcetut, metrics=metrics),
    trn.ReduceLROnPlateauHook(
        optimizer,
        patience=5, factor=0.8, min_lr=1e-6,
        stop_after_min=True
    )
]
```

Finally, we build the SchNetPack Trainer and pass the optimizer, loss function, hooks and data loaders.

```python
trainer = trn.Trainer(
    model_path=forcetut,
    model=model,
    hooks=hooks,
    loss_fn=loss,
    optimizer=optimizer,
    train_loader=train_loader,
    validation_loader=val_loader,
)
```

We then train our model for 300 epochs, which should take approximately 10 minutes on a notebook GPU.

```python
# check if a GPU is available and use a CPU otherwise
if torch.cuda.is_available():
    device = "cuda"
else:
    device = "cpu"

# determine number of epochs and train
n_epochs = 300
trainer.train(device=device, n_epochs=n_epochs)
```

Training will produce several files in the model_path directory, which is forcetut in our case. The split is stored in split.npz. Checkpoints are written to checkpoints periodically, which can be used to restart training. A copy of the best model is stored in best_model, which can directly be accessed using the torch.load function. Since we specified the CSV logger, the training progress is saved to log.csv.

Using the CSV file, the training progress can be visualized. An example showing the evolution of the mean absolute errors (MAEs) during training is shown below. Besides the schnetpack.train.CSVHook, it is also possible to use the schnetpack.train.tensorboardHook. This makes it possible to monitor the training in real time with TensorBoard.
```python
import numpy as np
import matplotlib.pyplot as plt
from ase.units import kcal, mol

# Load logged results
results = np.loadtxt(os.path.join(forcetut, 'log.csv'), skiprows=1, delimiter=',')

# Determine time axis
time = results[:,0] - results[0,0]

# Load the validation MAEs
energy_mae = results[:,4]
forces_mae = results[:,5]

# Get final validation errors
print('Validation MAE:
energy: {:10.3f} kcal/mol'.format(energy_mae[-1]))
print('forces: {:10.3f} kcal/mol/\u212B'.format(forces_mae[-1]))

# Construct figure
plt.figure(figsize=(14,5))

# Plot energies
plt.subplot(1,2,1)
plt.plot(time, energy_mae)
plt.title('Energy')
plt.ylabel('MAE [kcal/mol]')
plt.xlabel('Time [s]')

# Plot forces
plt.subplot(1,2,2)
plt.plot(time, forces_mae)
plt.title('Forces')
plt.ylabel('MAE [kcal/mol/\u212B]')
plt.xlabel('Time [s]')

plt.show()
```

Validation MAE:
energy: 0.414 kcal/mol
forces: 1.099 kcal/mol/Å
It should be noted, that the model trained here is used exclusively for demonstrative purposes. Accordingly, its size and the training time have been reduced significantly. This puts strong constraints on the accuracy that can be obtained. For practical applications, one would e.g. increase the number of features, the interaction layers, the learning rate schedule and train until convergence (removing the n_epochs keyword from the trainer).

SchNetPack also provides the script schnetpack_md17.py, which fully automates the whole training process for MD17 and comes with a series of reasonable default settings.

### 1.11.4 Using the model

Since all models in SchNetPack are stored in the same way, we can use the trained force model in exactly the same manner as described in the QM9 tutorial.

To load the model stored in the best_model file, we use the torch.load function. It will automatically be moved to the device it was trained on.

```python
[17]: best_model = torch.load(os.path.join(forcetut, 'best_model'))
```

To evaluate its performance on the test data, we use an adapted version of the loop introduced in the previous tutorial. Since the MD17 dataset of ethanol contains approximately 550000 configurations, the full evaluation takes approximately 10 minutes on GPU.

```python
[18]: test_loader = spk.AtomsLoader(test, batch_size=100)
energy_error = 0.0
forces_error = 0.0
for count, batch in enumerate(test_loader):
    # move batch to GPU, if necessary
    batch = {k: v.to(device) for k, v in batch.items()}

    # apply model
    pred = best_model(batch)

    # calculate absolute error of energies
    tmp_energy = torch.sum(torch.abs(pred['MD17.energy'] - batch['MD17.energy']))
    tmp_energy = tmp_energy.detach().cpu().numpy()  # detach from graph & convert to numpy
    energy_error += tmp_energy

    # calculate absolute error of forces, where we compute the mean over the n_atoms x 3 dimensions
    tmp_forces = torch.sum(torch.mean(torch.abs(pred['MD17.forces'] - batch['MD17.forces']), dim=(1,2)))
    tmp_forces = tmp_forces.detach().cpu().numpy()  # detach from graph & convert to numpy
    forces_error += tmp_forces

    # log progress
    percent = '{:3.2f}'.format(count/len(test_loader)*100)
    print('Progress: {}% '.format(percent))
energy_error /= len(test)
forces_error /= len(test)
print('Test MAE: ')
```

(continues on next page)
Progress: 99.98%
Test MAE:
  energy: 0.396 kcal/mol
  forces: 1.100 kcal/mol/Å

1.11.5 Interface to ASE

As was shown in the QM9 tutorial, one can also use the AtomsConverter to directly operate on ASE atoms objects. Having access to molecular forces also makes it possible to perform a variety of different simulations. The SpkCalculator offers a simple way to perform all computations available in the ASE package. Below, we create an ASE calculator from the trained model and the previously generated atoms object (see Preparing the data). One important point is, that the MD17 dataset uses kcal/mol and kcal/mol/Å as units for energies and forces. For the ASE interface, these need to be converted to the standard internal ASE units eV and eV/Å. This can be done by either passing the conversion factor or a string denoting the unit to the keywords energy_units and force_units.

```python
[19]:
calculator = spk.interfaces.SpkCalculator(
    model=best_model,
    device=device,
    energy=MD17.energy,
    forces=MD17.forces,
    energy_units='kcal/mol',
    forces_units='kcal/mol/Å'
)
```

```python
atoms.set_calculator(calculator)
```

```python
print('Prediction:')
print('energy:', atoms.get_total_energy())
print('forces:', atoms.get_forces())
```

Prediction:
  energy: [-4215.326]
  forces: [[ 4.8149731e-02  2.2211398e-01 -8.2710464e-08]
            [ 7.8540379e-01 -2.2908500e-01 -3.1016423e-08]
            [-9.7170897e-02  1.0150807e+00 -4.1355232e-08]
            [-2.6240392e-02 -4.0961695e-01  3.1710267e-01]
            [-2.6240392e-02 -4.0961662e-01 -3.1710231e-01]
            [-1.2042553e-01  2.2401318e-01  1.8506417e-01]
            [-1.2042536e-01  2.2401318e-01 -1.8506417e-01]
            [-3.1431168e-01 -1.4146599e-01 -2.2616142e-08]
            [-1.2873909e-01 -4.9543568e-01 -4.2001407e-09]]

Among the simulations which can be done by using ASE and a force model are geometry optimisation, normal mode analysis and simple molecular dynamics simulations.

The AseInterface of SchNetPack offers a convenient way to perform basic versions of these computations. Only a file specifying the geometry of the molecule and a pretrained model are needed.

We will first generate a XYZ file containing an ethanol configuration:

```python
[20]:
from ase import io
```
# Generate a directory for the ASE computations
ase_dir = os.path.join(forcetut, 'ase_calcs')

if not os.path.exists(ase_dir):
    os.mkdir(ase_dir)

# Write a sample molecule
molecule_path = os.path.join(ase_dir, 'ethanol.xyz')
io.write(molecule_path, atoms, format='xyz')

The AseInterface is initialized by passing the path to the molecule, the model and a computation directory. In addition, the computation device for the force model and how energies and forces are called in the output, as well as their units, need to be provided.

[21]: ethanol_ase = spk.interfaces.AseInterface(
    molecule_path,
    best_model,
    ase_dir,
    device,
    energy=MD17.energy,
    forces=MD17.forces,
    energy_units='kcal/mol',
    forces_units='kcal/mol/A'
)

## Geometry optimization

For some applications it is necessary to relax a molecule to an energy minimum. In order to perform this optimization of the molecular geometry, we can simply call

[22]: ethanol_ase.optimize(fmax=1e-4)

<table>
<thead>
<tr>
<th>Step [ FC]</th>
<th>Time</th>
<th>Energy</th>
<th>fmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFGSLineSearch: 0 [ 0] 14:04:31</td>
<td>-4215.326172</td>
<td>1.0197</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 1 [ 2] 14:04:31</td>
<td>-4215.358887</td>
<td>0.6782</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 2 [ 4] 14:04:31</td>
<td>-4215.370117</td>
<td>0.3471</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 3 [ 6] 14:04:31</td>
<td>-4215.373535</td>
<td>0.2088</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 4 [ 8] 14:04:31</td>
<td>-4215.376465</td>
<td>0.1770</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 5 [10] 14:04:31</td>
<td>-4215.377441</td>
<td>0.0837</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 6 [12] 14:04:31</td>
<td>-4215.377930</td>
<td>0.0413</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 7 [15] 14:04:31</td>
<td>-4215.377930</td>
<td>0.0323</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 8 [16] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0184</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 9 [17] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0042</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 10 [19] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0022</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 11 [20] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0016</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 12 [22] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0028</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 13 [23] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0022</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 14 [25] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0018</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 15 [28] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0006</td>
<td></td>
</tr>
<tr>
<td>BFGSLineSearch: 16 [29] 14:04:31</td>
<td>-4215.378418</td>
<td>0.0001</td>
<td></td>
</tr>
</tbody>
</table>

Since we trained only a reduced model, the accuracy of energies and forces is not optimal and several steps are needed to optimize the geometry.
Normal mode analysis

Once the geometry was optimized, normal mode frequencies can be obtained from the Hessian (matrix of second derivatives) of the molecule. The Hessian is a measure of the curvature of the potential energy surface and normal mode frequencies are useful for determining, whether an optimization has reached a minimum. Using the AseInterface, normal mode frequencies can be obtained via:

```
[23]: ethanol_ase.compute_normal_modes()
Writing ./forcetut/ase_calcs/normal_modes.eq.pckl
Writing ./forcetut/ase_calcs/normal_modes.0x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.0x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.0y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.0y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.0z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.0z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.0z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.1x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.1x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.1y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.1y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.1z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.1z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.2x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.2x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.2y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.2y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.2z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.2z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.3x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.3x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.3y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.3y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.3z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.3z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.4x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.4x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.4y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.4y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.4z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.4z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.5x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.5x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.5y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.5y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.5z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.5z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.6x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.6x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.6y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.6y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.6z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.6z+.pckl
Writing ./forcetut/ase_calcs/normal_modes.7x-.pckl
Writing ./forcetut/ase_calcs/normal_modes.7x+.pckl
Writing ./forcetut/ase_calcs/normal_modes.7y-.pckl
Writing ./forcetut/ase_calcs/normal_modes.7y+.pckl
Writing ./forcetut/ase_calcs/normal_modes.7z-.pckl
Writing ./forcetut/ase_calcs/normal_modes.7z+.pckl
```
(continues on next page)
Imaginary frequencies indicate, that the geometry optimisation has not yet reached a minimum. The AseInterface also creates an normal_modes.xyz file which can be used to visualize the vibrations with jmol.

**Molecular dynamics**

Finally, it is also possible to basic run molecular dynamics simulations using this interface. To do so, we first need to prepare the system, where we specify the simulation file. This routine automatically initializes the velocities of the atoms to a random number corresponding to a certain average kinetic energy.

```
[24]: ethanol_ase.init_md(
    'simulation'
)
```
The actual simulation is performed by calling the function `run_md` with a certain number of steps:

```
[25]: ethanol_ase.run_md(1000)
```

During simulation, energies and geometries are logged to `simulation.log` and `simulation.traj`, respectively.

We can for example visualize the evolution of the systems total and potential energies as

```
[26]: 
# Load logged results
results = np.loadtxt(os.path.join(ase_dir, 'simulation.log'), skiprows=1)

# Determine time axis
time = results[:,0]

# Load energies
energy_tot = results[:,1]
energy_pot = results[:,2]
energy_kin = results[:,3]

# Construct figure
plt.figure(figsize=(14,6))

# Plot energies
plt.subplot(2,1,1)
plt.plot(time, energy_tot, label='Total energy')
plt.plot(time, energy_pot, label='Potential energy')
plt.ylabel('E [eV]')
plt.legend()

plt.subplot(2,1,2)
plt.plot(time, energy_kin, label='Kinetic energy')
plt.ylabel('E [eV]')
plt.xlabel('Time [ps]')
plt.legend()

temperature = results[:,4]
print('Average temperature: {:.10f} K'.format(np.mean(temperature)))

plt.show()
```

Average temperature: 170.79 K
As can be seen, the potential and kinetic energies fluctuate, while the total energy (sum of potential and kinetic energy) remains approximately constant. This is a good demonstration for the energy conservation obtained by modeling forces as energy derivatives. Unfortunately, this also means that energy conservation is not a sufficient measure for the quality of the potential.

However, frequently one is interested in simulations where the system is coupled to an external heat bath. This is the same as saying that we wish to keep the average kinetic energy of the system and hence temperature close a certain value. Currently the average temperature only depends on the random velocities drawn during the initialization of the dynamics. Keeping a constant temperature average be achived by using a so-called thermostat. In the AseInterface, simulations with a thermostat (to be precise a Langevin thermostat) can be carried out by providing the temp_bath keyword. A simulation with e.g. the target temperature of 300K is performed via:

```python
[27]: ethanol_ase.init_md('simulation_300K',
 temp_bath=300,
 reset=True)
ethanol_ase.run_md(20000)
```

We can now once again plot total and potential energies. Instead of the kinetic energy, we now plot the temperature (both quantities are directly related).

```python
[28]: # Load logged results
results = np.loadtxt(os.path.join(ase_dir, 'simulation_300K.log'), skiprows=1)

# Determine time axis
time = results[:,0]

# Load energies
energy_tot = results[:,1]
energy_pot = results[:,2]

# Construct figure
plt.figure(figsize=(14,6))

# Plot energies
plt.subplot(2,1,1)
```

(continues on next page)
Since our molecule is now subjected to external influences via the thermostat the total energy is no longer conserved. However, the simulation temperature now fluctuates near to the requested 300K. This can also be seen by computing the temperature average over time, which is now close to the desired value in contrast to the previous simulation.

### 1.11.6 Summary

In this tutorial, we have trained a SchNet model on energies and forces using the MD17 ethanol dataset as an example. We have then evaluated the performance of the model and performed geometry optimisation, normal mode analysis and basic molecular dynamic simulations using the SchNetPack ASE interface.

While these simulations can already be useful for practical applications, SchNetPack also comes with its own molecular dynamics package. This package makes it possible to run efficient simulations on GPU and also offers access to advanced techniques, such as ring polymer dynamics. In the next tutorial, we will cover how to perform molecular dynamics simulations directly with SchNetPack.
1.12 Molecular dynamics in SchNetPack (experimental)

In the previous tutorial we have covered how to train machine learning models on molecular forces and use them for basic molecular dynamics (MD) simulations with the SchNetPack ASE interface.

All these simulations can also be carried out using the native MD package available in SchNetPack. The main ideas behind integrating MD functionality directly into SchNetPack are: - improve performance by reducing the communication overhead between ML models and the MD code and adding the option to use GPUs - adding extended functionality, such as sampling algorithms and ring polymer MD - providing a modular MD environment for easy development and interfacing

In the following, we first introduce the general structure of the SchNetPack-MD package. Then the simulation from the previous tutorial will be used as an example to demonstrate how to implement basic MD with SchNetPack-MD. Having done so, we will cover a few advanced simulation techniques, such as ring polymer MD.

Finally, we will show how all of these different simulations can be accessed via an input file.

1.12.1 Getting started

Before we can begin with the main tutorial, some setup is required. First, we generate a directory for holding our simulations:

```
[1]: import os

md_workdir = 'mdtut'

# Gnerate a directory of not present
if not os.path.exists(md_workdir):
    os.mkdir(md_workdir)
```

Since we want to run MD simulations, we need a SchNetPack model trained on forces and a molecular structure as a starting point. In principle, we could use the ethanol model and structure generated in the previous tutorial. However, the model trained in the force tutorial was only intended as a demonstration and is not the most accurate.

Instead, we will use a sample ethanol structure, as well as a fully converged SchNet model of ethanol provided with the data used for testing SchNetPack for this tutorial:

```
[2]: import schnetpack as spk

# Get the parent directory of SchNetPack
spk_path = os.path.abspath(os.path.join(os.path.dirname(spk.__file__), '../..'))

# Get the path to the test data
test_path = os.path.join(spk_path, 'tests/data')

# Load model and structure
model_path = os.path.join(test_path, 'test_md_model.model')
molecule_path = os.path.join(test_path, 'test_molecule.xyz')
```

1.12.2 MD in SchNetPack

In general, a MD code needs to carry out several core tasks during each simulation step. It has to keep track of the positions $\mathbf{R}$ and momenta $\mathbf{p}$ of all nuclei, compute the forces $\mathbf{F}$ acting on the nuclei and use the latter to integrate Newton’s equations of motion.
The overall workflow used in the SchNetPack MD package is sketched in the figure to the left. As can be seen, the various tasks are distributed between different modules.

The **System** class contains all information on the present state of the simulated system (e.g. nuclear positons and momenta). This is a good point to mention, that internally the MD package uses atomic units for all properties.

The **Integrator** computes the positions and momenta of the next step and updates the state of the system accordingly.

In order to carry out this update, the nuclear forces are required. These are computed with a **Calculator**, which takes the positions of atoms and returns the corresponding forces. Typically, the **Calculator** consists of a previously trained machine learning model.

All these modules are linked together in the **Simulator** class, which contains the main MD loop and calls the three previous modules in the correct order.

We will now describe the different components of the MD package in more detail and give an example of how to set up a short MD simulation of an ethanol molecule.

**System**

As stated previously, **System** keeps track of the state of the simulated system and contains the atomic positions $\mathbf{R}$ and momenta $\mathbf{p}$, but also e.g. atom types and computed molecular properties.

A special property of SchNetPack-MD is the use of multidimensional tensors to store the system information (using the `torch.Tensor` class). This makes it possible to make full use of vectorization and e.g. simulate several different molecules as well as different replicas of a molecule in a single step. The general shape of these system tensors is $N_{\text{replicas}} \times N_{\text{molecules}} \times N_{\text{atoms}} \times \ldots$, where the first dimension is the number of replicas of the same molecule (e.g. for ring polymer MD), the second runs over the different molecules simulated (e.g. fragments of different size for sampling) and the third over the maximum number of atoms present in any system.

In order to initialize a **System**, first the number of replicas needs to be given. Here, we want to perform a standard MD and $N_{\text{replicas}} = 1$. In addition, one can specify the device used for the computation. Afterwards, the molecules which should be simulated need to be loaded. These can be read directly from a XYZ-file via the `load_molecules_from_xyz` function. $N_{\text{molecules}}$ is determined automatically based on the number of structures found in this file. In our present case, the loaded files contains the structure of a single ethanol.

```python
[3]: from schnetpack.md import System
    import torch

    # Check if a GPU is available and use a CPU otherwise
    if torch.cuda.is_available():
        md_device = "cuda"
    else:
        md_device = "cpu"

    # Number of molecular replicas
    n_replicas = 1

    # Initialize the system
    md_system = System(n_replicas, device=md_device)

    # Load the structure
    md_system.load_molecules_from_xyz(molecule_path)
```

Right now, all system momenta are set to zero. For practical purposes, one usually wants to draw the momenta from a distribution corresponding to a certain temperature. This can be done via an **Initializer**, which takes the temperature in Kelvin as an input. For this example, we use a Maxwell—Boltzmann initialization:
Here, we have also removed all translational and rotational components of the momenta via the appropriate keyword.

**Integrator**

Having set up the system in such a manner, one needs to specify how the equations of motion should be propagated. Currently, there are two integration schemes implemented in SchNetPack: - a Velocity Verlet integrator which evolves the system in a purely classical manner and - a ring polymer integrator which is able to model a certain degree of nuclear quantum effects.

For demonstration purposes, we will first focus on a purely classical MD using the Velocity Verlet algorithm. An example on how to use ring polymer MD in SchNetPack and potential benefits will be given later in the tutorial. To initialize the integrator, one has to specify the length of the timestep $\Delta t$ used for integration in units of femtoseconds. A common value for classical MD is $\Delta t = 0.5$ fs.

**Calculator**

The only ingredient missing for simulating our system is a **Calculator** to compute molecular forces and other properties. A Calculator can be thought of as an interface between a computation method (e.g. a machine learning model) and the MD code in SchNetPack. SchNetPack comes with several predefined calculators and also offers the possibility to implement custom calculators.

Right now, we are only interested in using a model trained with SchNetPack, hence we use the SchnetPackCalculator. First, we have to load the stored model with Torch and move it to the computation device defined before. To initialize the SchnetPackCalculator, we have to pass it the loaded model. Similar as for the ASE interface in the last tutorial, we have to tell the calculator which properties to compute, how the forces are called in the output. Since the whole SchNetPack-MD package uses atomic units, it is also necessary to specify which units the calculator expects for the positions (position_conversion) and which units it uses for the returned forces (force_conversion).
For the first two points, we can make use of the SchNetPack properties definitions. With regards to units, the current calculator uses $\text{\AA}$ for positions and kcal/mol/ for the forces. The conversion factors can either be given as a number or as a string.

```python
from schnetpack.md.calculators import SchnetPackCalculator
from schnetpack import Properties

# Load the stored model
md_model = torch.load(model_path, map_location=md_device).to(md_device)

# Generate the calculator
md_calculator = SchnetPackCalculator(
    md_model,
    required_properties=[Properties.energy, Properties.forces],
    force_handle=Properties.forces,
    position_conversion='A',
    force_conversion='kcal/mol/A'
)
```

### Simulator (bringing it all together)

With our molecular system, a machine learning calculator for the forces and an integrator at hand, we are almost ready carry out MD simulations. The last step is to pass all these ingredients to a Simulator. The Simulator performs the actual MD simulations, looping over a series of time steps and calling the individual modules in the right order:

```python
from schnetpack.md import Simulator

md_simulator = Simulator(md_system, md_integrator, md_calculator)

To carry out a simulation, one needs to call the simulate function with an integer argument specifying the number of desired simulation steps.

For example, a MD simulation of our ethanol molecule for 100 time steps (50fs) can be done via:

```python
n_steps = 100

md_simulator.simulate(n_steps)
```

Since the Simulator keeps track of the state of the dynamics and the system, we can call it repeatetly to get longer trajectories.

```python
md_simulator.simulate(n_steps)
```

The actual number of steps is stored in the step variable of the Simulator class.

```python
print("Total number of steps:", md_simulator.step)
Total number of steps: 200
```

Although we are now able to run a full-fledged MD simulation, there is one major problem with the current setup: we do not collect any information during the simulation, such as nuclear positions. This means, that we currently have no way of analyzing what happened during the MD trajectory.

This — and many other things — can be done in the SchNetPack-MD package using so-called simulation hooks.
1.12.3 Simulation hooks

Simulation hooks follow the same concept as the hooks used in the SchNetPack Trainer class covered previously. They can be thought of as instructions for the Simulator, which are performed at certain points during each MD step. Simulation hooks can be used to tailor a simulation to one's needs, contributing to the customizability of the SchNetPack-MD package.

The diagram to the left shows how a single MD step of the Simulator is structured in detail and at which points hooks can be applied. Depending on which time they are called and which actions they encode, simulation hooks can achieve a wide range of tasks.

If they are introduced before and after each integration half-step, they can e.g. be used to control the temperature of the system in the form of thermostats.

When acting directly after the computation of the forces done by the Calculator, simulation hooks can be used to control sampling. At this point, enhanced sampling schemes such as metadynamics and accelerated MD can be implemented, which modify the forces and potential energies of the system. It is also possible to introduce active learning for automatically generating machine learning models in this way.

Finally, when called after the second integration step, simulation hooks can be used to collect and store information on the system, which can then be used for analysis.

Multiple hooks can be passed to a simulator at any time, which makes it possible to control a simulation in various manners. In the following, we will demonstrate how to apply a thermostat to the above simulation and how data collection can be done in SchNetPack.

Adding a Thermostat

As mentioned in the force tutorial, thermostats are used to keep the fluctuations of the kinetic energy of a system (temperature) close to a predefined average. Simulations employing thermostats are referred to as canonical ensemble or $NVT$ simulations, since they keep the number of particles $N$, the volume $V$ and the average temperature $T$ constant.

Last time, we used a Langevin thermostat to regulate the temperature of our simulation. This thermostat (and many others) is also available in SchNetPack and can be used via

```
[11]: from schnetpack.md.simulation_hooks import thermostats

    # Set temperature and thermostat constant
    bath_temperature = 300 # K
    time_constant = 100 # fs

    # Initialize the thermostat
    langevin = thermostats.LangevinThermostat(bath_temperature, time_constant)

    INFO:root:Using Langevin thermostat
```

In case of the Langevin thermostat, a bath temperature (in Kelvin) and a time constant (in fs) have to be provided. The first regulates the temperature the system is kept at, the second how fast the thermostat adjusts the temperature. In order to speed up equilibration, we use a more aggressive thermostatting schedule by with a time constant of 100fs.

Finally, we begin collecting the simulation hooks we want to pass to the simulator.

```
[12]: simulation_hooks = [
    langevin
]
```
Collecting Data and storing Checkpoints

The primary way to store simulation data in the SchNetPack-MD package is via the FileLogger class. A FileLogger collects data during the MD and stores it to a database in HDF5 format. The type of data to be collected is specified via so-called DataStreams, which are passed to the FileLogger. The data streams currently available in SchNetPack are: - MoleculeStream: Stores positions and velocities during all simulation steps - PropertyStream: Stores all properties predicted by the calculator - SimulationStream: Collects information on the kinetic energy and system temperature (this can be also done via postprocessing when using the MoleculeStream) By default, the MoleculeStream and PropertyStream are used.

To reduce overhead due to writing to disk, the FileLogger first collects information for a certain number of steps into a buffer, which it then writes to the database at once.

The FileLogger is initialized by specifying the name of the target database, the size of the buffer and which data to store (in form of the respective data streams):

```python
from schnetpack.md.simulation_hooks import logging_hooks

# Path to database
log_file = os.path.join(md_workdir, 'simulation.hdf5')

# Size of the buffer
buffer_size = 100

data_streams = [logging_hooks.MoleculeStream(), logging_hooks.PropertyStream(),]

# Create the file logger
file_logger = logging_hooks.FileLogger(log_file, buffer_size, data_streams=data_streams)

# Update the simulation hooks
simulation_hooks.append(file_logger)
```

In general, it is also a good idea to store checkpoints of the system and simulation state at regular intervals. Should something go wrong with the simulation, these can be used to restart the simulation from the last stored point. In addition, these checkpoints can also be used to only initialize the System. This is e.g. useful for equilibrating simulations with different thermostats.

Storing checkpoints can be done with the Checkpoint hook, which takes a file the data is stored to and the frequency a checkpoint is generated:

```python
# Set the path to the checkpoint file
chk_file = os.path.join(md_workdir, 'simulation.chk')

# Create the checkpoint logger
checkpoint = logging_hooks.Checkpoint(chk_file, every_n_steps=100)

# Update the simulation hooks
simulation_hooks.append(checkpoint)
```
Adding Hooks and Running the Simulation

With all simulation hooks created and collected in `simulation_hooks`, we can finally build our updated simulator. This is done exactly the same way as above, with the difference that now also the hooks are specified.

```python
[15]: md_simulator = Simulator(md_system, md_integrator, md_calculator, simulator_hooks=simulation_hooks)
```

We can now use the simulator to run a MD trajectory of our ethanol. Here, we run for 20000 steps, which are 10ps. This should take approximately 5 minutes on a notebook GPU.

```python
[16]: n_steps = 20000
    md_simulator.simulate(n_steps)
```

The tutorial directory should now contain two files: - `simulation.hdf5`, which holds the collected data and - `simulation.chk` containing the last checkpoint.

### 1.12.4 Reading HDF5 outputs

We will now show, how to access the HDF5 files generated during the simulation. For this purpose, SchNetPack comes with a `HDF5Loader`, which can be used to extract the data by giving the path to the simulation output (`mdtut/simulation.hdf5`).

```python
[17]: from schnetpack.md.utils import HDF5Loader
    data = HDF5Loader(log_file)
```

```plaintext
INFO:root:Loaded properties _atomic_numbers, _positions, velocities, energy and forces from mdtut/simulation.hdf5
```

Extracted data is stored in the `properties` dictionary of the `HDF5Loader` and can be accessed with the `get_property` function. `get_property` requires the name of the property and optionally the index of the molecule and replica for which the data should be extracted. By default, it extracts the first molecule (`mol_idx=0`) and averages over all replicas if more than one are present. Neither is relevant for our current simulation.

Right now, we can access the following entries, all of which should be self-explaining and correspond to the standard SchNetPack `Properties` and `Structure` keys:

```python
[18]: for prop in data.properties:
    print(prop)
    _atomic_numbers
    _positions
    velocities
    energy
    forces
```

We can now e.g. have a look at the potential energies.

```python
[19]: %matplotlib notebook
    import matplotlib.pyplot as plt
    from schnetpack.md.utils import MDUnits
```

```plaintext
# Get potential energies and check the shape
```

(continues on next page)
energies = data.get_property(Properties.energy)
print('Shape:', energies.shape)

# Get the time axis
time_axis = np.arange(data.entries)*data.time_step / MDUnits.fs2atu # in fs

# Plot the energies
plt.figure()
plt.plot(time_axis, energies)
plt.ylabel('E [kcal/mol]')
plt.xlabel('t [fs]')
plt.tight_layout()
plt.show()

Shape: (20000, 1)

The HDF5Loader also offers access to functions for computing some derived properties, such as the kinetic energy (get_kinetic_energy) and the temperature (get_temperature).

[20]: import numpy as np

```python
def plot_temperature(data):
    # Read the temperature
    temperature = data.get_temperature()

    # Compute the cumulative mean
    temperature_mean = np.cumsum(temperature) / (np.arange(data.entries)+1)

    # Get the time axis
    time_axis = np.arange(data.entries)*data.time_step / MDUnits.fs2atu # in fs

    plt.figure(figsize=(8,4))
    plt.plot(time_axis, temperature, label='T')
    plt.plot(time_axis, temperature_mean, label='T (avg.)')
    plt.ylabel('T [K]')
    plt.xlabel('t [fs]')
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_temperature(data)
```

As can be seen, the system requires an initial period for equilibration. This is relevant for simulations, as ensemble properties are typically only computed for fully equilibrated systems.

In SchNetPack, an appropriate analysis can be done in different ways. The checkpoint file of the equilibrated system can be used as a starting point for a production simulation. The easier way, however, is to reject the initial part of the trajectory and only consider the equilibrated system for analysis. This can be done by specifying the number of steps to skip in the HDF5Loader. Here, we skip the first half (25000 steps) of the trajectory. In general, the equilibration period strongly depends on system size and the thermostat settings.
We can easily see, that only the later part of the simulation is now considered by plotting the data once again:

```python
[22]: plot_temperature(equilibrated_data)
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

It should be mentioned at this point, that the HDF5 datafile uses a special convention for units. For internal quantities (e.g. positions, velocities and kinetic energy), atomic units are used. The only exception are temperatures, which are given in units of Kelvin for convenience. For all properties computed by the Calculator the original unit is used, unless a conversion factor is specified during initialization. This means, that the energies and forces collected here have units of kcal/mol and kcal/mol/.

**Vibrational spectra**

While curves of temperatures might be nice to look at, one is usually interested in different quantities when running a MD simulation. One example are vibrational spectra, which give information on which vibrations are active in a molecule. SchNetPack provides the module `schnetpack.md.utils.spectra`, which provides different classes to compute vibrational spectra directly from the HDF5 files. These implementations use several tricks to improve efficiency and accuracy. Currently, power spectra (PowerSpectrum) and infrared (IR) spectra (IRSpectrum) are available.

Here, we will compute the power spectrum from our ethanol simulation:

```python
[23]: from schnetpack.md.utils import PowerSpectrum
```

```python
# Initialize the spectrum
spectrum = PowerSpectrum(equilibrated_data, resolution=4096)

# Compute the spectrum for the first molecule (default)
spectrum.compute_spectrum(molecule_idx=0)
INFO:root:Spectral resolutions: 2.036 [cm^-1]
INFO:root:Spectral range: 33356.410 [cm^-1]
```

The `resolution` keyword specifies, how finely the peaks in the spectrum are resolved. `PowerSpectrum` also computes the effective resolution in inverse centimeters, as well as the spectral range. For molecules, one is usually interested in frequencies up to 4000cm⁻¹, which we will use to restrict the plotting area.

```python
[24]: # Get frequencies and intensities
frequencies, intensities = spectrum.get_spectrum()

# Plot the spectrum
plt.figure()
plt.plot(frequencies, intensities)
plt.xlim(0,4000)
plt.ylim(0,100)
plt.ylabel('I [a.u.]')
plt.xlabel('$\omega$ [cm$^{-1}$]')
plt.show()
```
The spectrum shows several typical vibrational bands for ethanol (which can be checked with experimental tables available online). For example, the peak close to 3700 cm⁻¹ stems from the bond vibrations of the OH-group. The bond vibrations of the CH₃ and CH₂ groups are clustered around 3000 cm⁻¹ and the corresponding bending vibrations can be seen at 1400 cm⁻¹. In general, computed vibrational spectra serve as a good check for the validity of a machine learning potential.

One important fact should be noted at this point: the spectrum computed here is a power spectrum, representing the vibrational density of states. It gives information on all vibrational modes which can potentially be active in an experimental spectrum. Hence, it can help in identifying which motions give rise to which experimental bands. However, depending on the experiment, only a subset of the peaks of the power spectrum can be active and the intensities can vary dramatically. As such, a power spectrum only serves as a poor stand in for simulating e.g. Raman or IR spectra. Using SchNetPack, it is also possible to model IR spectra, by training a model on dipoles in addition to forces. Simulations can then be done in the same manner as above and the corresponding IR spectra can be obtained using IRSpectrum instead of PowerSpectrum.

### 1.12.5 Restarting simulations

In some situations, it is convenient to restart simulations from a previously stored checkpoint, e.g. when the cluster burned down for no apparent reasons.

In SchNetPack, this can be done by loading the checkpoint file with torch and then passing it to a Simulator using the restart_simulation function (here we use the same instance of simulator as before for convenience, in a real setup a new one would be initialized).

```python
[25]: checkpoint = torch.load(chk_file)
    md_simulator.restart_simulation(checkpoint)
```

This restores the full state, including the system state, simulation steps and states of the thermostats.

Sometimes, it can be sufficient to only restore the system state (positions and momenta), for example when starting production level simulations after equilibration. This is achieved by calling load_system_state on the loaded checkpoint:

```python
[26]: md_simulator.load_system_state(checkpoint)
```

### 1.12.6 Ring polymer molecular dynamics with SchNetPack

Above, we have computed a vibrational spectrum of ethanol based on a classical MD simulation using the Velocity Verlet integrator. Unfortunately, this approach completely neglects nuclear quantum effects, such as zero-point energies, etc. One way to recover some of these effects is to use so-called ring polymer molecular dynamics (RPMD). In RPMD, multiple replicas of a system are connected via harmonic springs and propagated simultaneously. This can be thought of a discretization of the path integral formulation of quantum mechanics. The fully quantum solution is then recovered in the limit of an infinite number of replicas, also called beads.

RPMD simulations can easily be carried out using the SchNetPack MD package. Due to need to perform a large number of computations, RPMD profits greatly from the use of machine learning potentials. Moreover, the presence of multiple replicas lends it self to an efficient parallelization of GPUs, which is one reason for the special structure of the system tensors used in SchNetPack-MD.
Here, we will repeat the above simulation for ethanol using RPMD instead of a classical simulation. The main differences in the setup are: - the system needs to be initialized with multiple replicas - a ring polymer integrator needs to be used - special thermostats are required if a canonical ensemble should be simulated

The System can be set up in exactly the same manner as before, only the number of replicas is now set to be greater than one. For demonstration purposes we use n_replicas=4, in general larger numbers are recommended.

```python
n_replicas = 4
rpmd_system = System(n_replicas, device=md_device)
rpmd_system.load_molecules_from_xyz(molecule_path)
rpmd_initializer = MaxwellBoltzmannInit(
    system_temperature,
    remove_translation=True,
    remove_rotation=True)
rpmd_initializer.initialize_system(rpmd_system)
```

Next, we need to change the integrator to the RingPolymer integrator. For RPMD, we need to use a smaller time step, in order to keep the integration numerically stable. In addition, one needs to specify a temperature for the ring polymer, which modulates how strongly the different beads couple. Typically, we use the same temperature as for the thermostat.

```python
rpmd_time_step = 0.2 # fs
rpmd_integrator = RingPolymer(
    n_replicas,
    rpmd_time_step,
    system_temperature,
    device=md_device
)
```

Next, we have to change our thermostat to one suitable for RPMD simulations. Here, we will use the local PILE thermostat, which can be thought of as a RPMD equivalent of the classical Langevin thermostat used above. In general, SchNetPack comes with a wide variety of thermostats for classical and ring polymer simulations (see the thermostats module). For the environment temperature and time constant, the same values as above are used.

```python
pile = thermostats.PILELocalThermostat(bath_temperature, time_constant)
INFO:root:Using Langevin thermostat
```

The hooks are generated in exactly the same way as before.
# Logging

```python
rpmd_log_file = os.path.join(md_workdir, 'rpmd_simulation.hdf5')
rpmd_data_streams = [
    logging_hooks.MoleculeStream(),
    logging_hooks.PropertyStream(),
]
rpmd_file_logger = logging_hooks.FileLogger(
    rpmd_log_file,
    buffer_size,
    data_streams=rpmd_data_streams
)
```

# Checkpoints

```python
rpmd_chk_file = os.path.join(md_workdir, 'rpmd_simulation.chk')
rpmd_checkpoint = logging_hooks.Checkpoint(rpmd_chk_file, every_n_steps=100)
```

# Assemble the hooks:

```python
rpmd_hooks = [
    pile,
    rpmd_file_logger,
    rpmd_checkpoint
]
```

And so is the simulator:

```python
# Assemble the simulator
rpmd_simulator = Simulator(rpmd_system, rpmd_integrator, md_calculator, simulator_hooks=rpmd_hooks)
```

Bow we can carry out the simulations. Since our time step is shorter, we will run for longer in order to cover the same time scale as the classical simulation (runs approximately 13 minutes on a notebook GPU):

```python
n_steps = 50000

rpmd_simulator.simulate(n_steps)
```

100% || 50000/50000 [12:59<00:00, 64.16it/s]

Loading of the data with the `HDF5Loader` works exactly the same as before. When loading properties for RPMD datafiles, the `HDF5Loader` default of using centroid properties (meaning an average over all beads) becomes active. This is usually what one wants to analyze. If a specific replica should be used, it can be specified via `replica_idx` in the `get_property` function. Here, we immediately skip the a part of the trajectory to only load the equilibrated system:

```python
rpmd_data = HDF5Loader(rpmd_log_file, skip_initial=0)
plot_temperature(rpmd_data)
```

INFO:root:Loaded properties _atomic_numbers, _positions, velocities, energy and _forces from mdtut/rpmd_simulation.hdf5

Finally, we can compute the power spectrum and compare it to its classical counterpart:

```python
# Initialize the spectrum
rpmd_spectrum = PowerSpectrum(rpmd_data, resolution=8192)
```

(continues on next page)
# Compute the spectrum for the first molecule (default)
rpmd_spectrum.compute_spectrum(molecule_idx=0)

# Get frequencies and intensities
rpmd_freqencies, rpmd_intensities = rpmd_spectrum.get_spectrum()

# Plot the spectrum
plt.figure(figsize=(8,4))
plt.plot(freqencies, intensities, label='MD')
plt.plot(rpmd_freqencies, rpmd_intensities, label='RPMD')
plt.xlim(0,4000)
plt.ylim(0,100)
plt.ylabel('I [a.u.]')
plt.xlabel('$\omega$ [cm$^{-1}$]')
plt.legend()
plt.show()

INFO:root:Spectral resolutions: 2.545 [cm$^{-1}$]
INFO:root:Spectral range: 83391.024 [cm$^{-1}$]

One problem of purely classical simulations can be observed in the high frequency regions of the MD spectrum. Peaks are shifted towards higher wave numbers compared to the expected experimental values, e.g. 3100cm$^{-1}$ vs. 2900cm$^{-1}$ for the CH vibrations. The inclusion of effects like zero point energy shifts these bands towards lower frequencies, leading to an improved agreement with experiment.

1.12.7 Quick setup with input files

Although encoding simulations in the way above can be useful when testing and developing new approaches, it has limited use for routine simulations. Because of this, the MD package also provides the script `spk_md.py`, which can be used to run a simulation according to the instructions given in an input file.

The input file uses yaml format and is structured in a similar way as the main modules introduced above. In the following, we will construct an input file for repeating the classical MD simulation above.

**Input file format**

The first few lines of the input files contain general instructions (e.g. device, random seed, simulation directory):

```yaml
device: cuda
simulation_dir: mdtut
seed: 662524648
overwrite: false
```

If no seed is specified, a new one is generated. The `overwrite` flag specifies, whether an old simulation should be overwritten.

Instructions for the calculator are specified in the `calculator` block:

```yaml
calculator:
  type: schnet
  model_file: PATH/TO/MODEL
```

(continues on next page)
required_properties:
  - energy
  - forces

force_handle: forces
position_conversion: Angstrom
force_conversion: kcal/mol/Angstrom

Here, the type of calculator is specified and path to the model has to be set accordingly (model_file). Conversion units are given in the same manner as above.

The system is specified in the system block of the input file, which controls the number of replicas, the file the structure is loaded from, as well as initialization routines:

```
system:
  molecule_file: PATH/TO/STRUCTURE.xyz
  n_replicas: 1
  initializer:
    type: maxwell-boltzmann
    temperature: 300
    remove_translation: true
    remove_rotation: true
```

The control of the dynamics simulation itself is handled in the dynamics section:

```
dynamics:
  n_steps: 20000
  integrator:
    type: verlet
    time_step: 0.50
  thermostat:
    type: langevin
    temperature: 300
    time_constant: 100
```

Thermostats are invoked via thermostat, where the type is specified via a string. The additional arguments vary according to the thermostat used. The integrator is given over the integrator subblock. Units are the same as used in the examples above.

Finally, logging to a file is handled via logging:

```
file_logger:
  buffer_size: 100
  streams:
    - molecules
    - properties
  write_checkpoints: 100
```

The logging file name is set to simulation_dir/simulation.hdf5 by default, while the checkpoint file uses simulation_dir/simulation.chk. Data streams for the file logger are passed via a list, where molecules corresponds to the MoleculeStream and properties to the PropertiesStream.

**Performing a Simulation**

Once an input file has been generated, it can be called via
SchNetPack also stores a config file in yaml format into the `simulation_dir` containing the detailed settings of the MD.

**Available Options**

There will be a more general documentation on the options available for the input file in the future. For now, the most important base settings can be found in the `schnetpack.md.parsers.md_options` module, which contains the initializers for the input blocks.

These initializers are: - `ThermostatInit` for thermostats - `IntegratorInit` for the integrators - `InitialConditionsInit` for controlling initial conditions - `CalculatorInit` for setting up the calculator

All of these classes have the utility function `print_options`. When this function is called, it prints as short summary on the available options and which inputs (and input types) are required as a minimum. In case of the thermostats, the following is obtained:

```
[35]: from schnetpack.md.parsers.md_options import *

ThermostatInit.print_options()

Available basic options:

berendsen
---------
  temperature (float)
  time_constant (float)

langevin
--------
  temperature (float)
  time_constant (float)

gle
---
  temperature (float)
  gle_input (str)

pile-l
------
  temperature (float)
  time_constant (float)

pile-g
------
  temperature (float)
  time_constant (float)

piglet
------
  temperature (float)
  gle_input (str)

nhc
---
  temperature (float)
```

(continues on next page)
In addition, every block in the input file can also be passed additional input options of the basic modules. An example is the calculator block:

```
calculator:
  type: schnet
  model_file: PATH/TO/MODEL
  required_properties:
    - energy
    - forces
  force_handle: forces
  position_conversion: Angstrom
  force_conversion: kcal/mol/Angstrom
```

Compared to the required input options for the SchNet calculator (model, required_properties and force_handle)

```
[36]: CalculatorInit.print_options()
```

Available basic options:

```
schnet
  -----
    model (model)
    required_properties (list)
    force_handle (str)

orca
  ----
    required_properties (list)
    force_handle (str)
    compdir (str)
    qm_executable (str)
    orca_template (str)
```

(continues on next page)
it is also possible to pass additional keyword arguments (here `position_conversion` and `force_conversion`). These correspond directly to the input parameters of the target class (see `calculators`).

Example input files

In the following, we provide two example input files for the classical MD and RPMD simulations performed above. Only the paths (`simulation_dir`, `model_file` and `molecule_file`) and the `device` need to be adjusted accordingly.

Classical MD

display

```python
device: cpu
simulation_dir: mdtut_md
overwrite: false
calculator:
  type: schnet
  model_file: PATH/TO/MODEL
  required_properties:
  - energy
  - forces
  force_handle: forces
  position_conversion: Angstrom
  force_conversion: kcal/mol/Angstrom
system:
  molecule_file: PATH/TO/STRUCTURE.xyz
  n_replicas: 1
initializer:
  type: maxwell-boltzmann
  temperature: 300
  remove_translation: true
  remove_rotation: true
dynamics:
  n_steps: 20000
  integrator:
    type: verlet
    time_step: 0.50
  thermostat:
    type: langevin
    temperature: 300
    time_constant: 100
logging:
  file_logger:
    buffer_size: 100
    streams:
      - molecules
      - properties
    write_checkpoints: 100
```
1.12.8 Summary

In this tutorial, we have given a basic introduction to the structure and functionality of the MD package in SchNetPack. After setting up a standard MD simulation, we have explored how to use simulation hooks to control simulations in a modular way. We have shown how to extract data from the HDF5 files generated during MD and how available functions can be used to compute dynamic quantities, such as power spectra. This was followed by a short example of using more advanced simulation techniques in the form of RPMD. Finally, a short introduction to the `spk_md.py` and its input file structure was given.

Future tutorials will cover the use of advanced sampling techniques (e.g. metadynamics) and how to write custom calculators and hooks for performing your own simulations.

SchNetpack consists of the following modules:
1.13 schnetpack

class schnetpack.Properties
    Keys to access structure properties in schnetpack.data.AtomsData

1.14 schnetpack.data

1.14.1 Data Loading

class schnetpack.data.AtomsData(dbpath, subset=None, available_properties=None, load_only=None, units=None, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>, collect_triples=False, center_positions=True)

PyTorch dataset for atomistic data. The raw data is stored in the specified ASE database. Use together with schnetpack.data.AtomsLoader to feed data to your model.

To improve the performance, the data is not stored in string format, as usual in the ASE database. Instead, it is encoded as binary before being written to the database. Reading work both with binary-encoded as well as standard ASE files.

Parameters

- dbpath (str) – path to directory containing database.
- subset (list, optional) – indices to subset. Set to None for entire database.
- available_properties (list, optional) – complete set of physical properties that are contained in the database.
- load_only (list, optional) – reduced set of properties to be loaded
- units (list, optional) – definition of units for all available properties
- collect_triples (bool, optional) – Set to True if angular features are needed.
- center_positions (bool, optional) – subtract center of mass from all positions (default=True)

add_system (atoms, **properties)
Add atoms data to the dataset.

Parameters

- atoms (ase.Atoms) – system composition and geometry
- **properties – properties as key-value pairs. Keys have to match the available_properties of the dataset.

add_systems (atoms_list, property_list)
Add atoms data to the dataset.

Parameters

- atoms_list (list of ase.Atoms) – system composition and geometry
- property_list (list) – Properties as list of key-value pairs in the same order as corresponding list of atoms. Keys have to match the available_properties of the dataset.
create_subset(idx)
Returns a new dataset that only consists of provided indices.
:param idx: subset indices :type idx: numpy.ndarray

Returns dataset with subset of original data
Return type schnetpack.data.AtomsData

get_atomref(properties)
Return multiple single atom reference values as a dictionary.

Parameters
properties (list or str) – Desired properties for which the atomrefs are calculated.

Returns atomic references
Return type dict

get_atoms(idx)
Return atoms of provided index.

Parameters
idx (int) – atoms index

Returns atoms data
Return type ase.Atoms

get_available_properties(available_properties)
Get available properties from argument or database.

Parameters
available_properties (list or None) – all properties of the dataset

Returns all properties of the dataset
Return type (list)

get_metadata(key=None)
Returns an entry from the metadata dictionary of the ASE db.

Parameters
key – Name of metadata entry. Return full dict if None.

Returns Value of metadata entry or full metadata dict, if key is None.
Return type value

get_properties(idx)
Return property dictionary at given index.

Parameters
idx (int) – data index

Returns:

set_metadata(metadata=None, **kwargs)
Sets the metadata dictionary of the ASE db.

Parameters
• metadata (dict) – dictionary of metadata for the ASE db
• kwargs – further key-value pairs for convenience

class schnetpack.data.AtomsLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=<function _collate_aseatoms>, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None)
Specialized `torch.data.DataLoader` which uses the correct `collate_fn` for `AtomsData` and provides functionality for calculating mean and stddev.

**Parameters**

- `dataset (Dataset)` – dataset from which to load the data.
- `batch_size (int, optional)` – how many samples per batch to load (default: 1).
- `shuffle (bool, optional)` – set to True to have the data reshuffled at every epoch (default: False).
- `sampler (Sampler, optional)` – defines the strategy to draw samples from the dataset. If specified, shuffle must be False.
- `batch_sampler (Sampler, optional)` – like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last.
- `num_workers (int, optional)` – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- `collate_fn (callable, optional)` – merges a list of samples to form a mini-batch (default: `collate_atoms`).
- `pin_memory (bool, optional)` – If True, the data loader will copy tensors into CUDA pinned memory before returning them.
- `drop_last (bool, optional)` – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: False)
- `timeout (numeric, optional)` – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
- `worker_init_fn (callable, optional)` – If not None, this will be called on each worker subprocess with the worker id (an int in [0, `num_workers` - 1]) as input, after seeding and before data loading. (default: None)

**get_statistics (property_names, divide_by_atoms=False, single_atom_ref=None)**

Compute mean and variance of a property. Uses the incremental Welford algorithm implemented in StatisticsAccumulator

**Parameters**

- `property_names (str or list)` – Name of the property for which the mean and standard deviation should be computed
- `divide_by_atoms (dict or bool)` – divide mean by number of atoms if True (default: False)
- `single_atom_ref (dict or bool)` – reference values for single atoms (default: None)

**Returns** Mean value stddev: Standard deviation

**Return type** mean

### 1.14.2 Predefined Datasets

Classes wrapping various standard benchmark datasets.
Molecule datasets

class schnetpack.datasets.QM9(dbpath, download=True, subset=None, load_only=None, collect_triples=False, remove_uncharacterized=False, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>, **kwargs)

QM9 benchmark database for organic molecules.

The QM9 database contains small organic molecules with up to nine non-hydrogen atoms from including C, O, N, F. This class adds convenient functions to download QM9 from figshare and load the data into pytorch.

Parameters

- **dbpath** (str) – path to directory containing database.
- **download** (bool, optional) – enable downloading if database does not exists.
- **subset** (list, optional) – indices to subset. Set to None for entire database.
- **load_only** (list, optional) – reduced set of properties to be loaded
- **collect_triples** (bool, optional) – Set to True if angular features are needed.
- **remove_uncharacterized** (bool, optional) – remove uncharacterized molecules.
- **environment_provider** (spk.environment.BaseEnvironmentProvider) – define how neighborhood is calculated (default=spk.environment.SimpleEnvironmentProvider).

References

create_subset(idx)

Returns a new dataset that only consists of provided indices. :param idx: subset indices :type idx: numpy.ndarray

Returns dataset with subset of original data

Return type schnetpack.data.AtomsData

class schnetpack.datasets.MD17(dbpath, molecule=None, subset=None, download=True, collect_triples=False, load_only=None, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>)

MD17 benchmark data set for molecular dynamics of small molecules containing molecular forces.

Parameters

- **dbpath** (str) – path to database
- **molecule** (str) – Name of molecule to load into database. Allowed are: aspirin benzene ethanol malonaldehyde naphthalene salicylic_acid toluene uracil
- **subset** (list) – indices of subset. Set to None for entire dataset (default: None)
- **download** (bool) – set true if dataset should be downloaded (default: True)
- **collect_triples** (bool) – set true if triples for angular functions should be computed (default: False)
- **load_only** (list, optional) – reduced set of properties to be loaded
- **environment_provider** (spk.environment.BaseEnvironmentProvider) – define how neighborhood is calculated (default=spk.environment.SimpleEnvironmentProvider).
create_subset(idx)
Returns a new dataset that only consists of provided indices. :param idx: subset indices :type idx: numpy.ndarray

Returns dataset with subset of original data

Return type schnetpack.data.AtomsData

class schnetpack.datasets.ISO17(datapath, fold, download=True, load_only=None, subset=None, collect_triples=False, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>)
ISO17 benchmark data set for molecular dynamics of C7O2H10 isomers containing molecular forces.

Parameters

• datapath (str) – Path to database directory
• fold (str) – Fold of data to load. Allowed are: reference - 80% of steps of 80% of MD trajectories reference_eq - equilibrium conformations of those molecules
  test_within - remaining 20% unseen steps of reference trajectories test_other - remaining 20% unseen MD trajectories test_eq - equilibrium conformations of test trajectories
• subset (list) – indices of subset. Set to None for entire dataset (default: None)
• load_only (list, optional) – reduced set of properties to be loaded
• download (bool) – set to true if dataset should be downloaded. (default: True)
• collect_triples (false) – set to true to compute triples for angular functions (default: true)
• environment_provider (spk.environment.BaseEnvironmentProvider) – define how neighborhood is calculated (default=spk.environment.SimpleEnvironmentProvider).

create_subset(idx)
Returns a new dataset that only consists of provided indices. :param idx: subset indices :type idx: numpy.ndarray

Returns dataset with subset of original data

Return type schnetpack.data.AtomsData

class schnetpack.datasets.ANI1(dbpath, download=True, load_only=None, subset=None, collect_triples=False, num_heavy_atoms=8, high_energies=False, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>)
ANI1 benchmark database.

This class adds convenience functions to download ANI1 from figshare and load the data into pytorch.

Parameters

• dbpath (str) – path to directory containing database.
• download (bool, optional) – enable downloading if database does not exists.
• subset (list, optional) – indices to subset. Set to None for entire database.
• **load_only** *(list, optional)* – reduced set of properties to be loaded

• **collect_triples** *(bool, optional)* – Set to True if angular features are needed.

• **num_heavy_atoms** *(int, optional)* – number of heavy atoms. (See ‘Table 1’ in Ref.²)

• **high_energies** *(bool, optional)* – add high energy conformations. (See ‘Technical Validation’ of Ref.²)

• **environment_provider** *(spk.environment.BaseEnvironmentProvider)*
  – define how neighborhood is calculated (default=spk.environment.SimpleEnvironmentProvider).

### References

**create_subset** *(idx)*

Return a new database that only consists of provided indices.

**Parameters**

- **idx** *(numpy.ndarray)* – indices to subset.

**Returns**

- [dataset with subset of original data](https://arxiv.org/abs/1708.04987)

**Return type** *schnetpack.data.AtomsData*

### Materials datasets

**class schnetpack.datasets.MaterialsProject** *(dbpath, apikey=None, download=True, subset=None, load_only=None, collect_triples=False, environment_provider=schnetpack.environment.SimpleEnvironmentProvider)*

Materials Project (MP) database of bulk crystals.

This class adds convenient functions to download Materials Project data into pytorch.

**Parameters**

- **dbpath** *(str)* – path to directory containing database.
- **cutoff** *(float)* – cutoff for bulk interactions.
- **apikey** *(str, optional)* – materials project key needed to download the data.
- **download** *(bool, optional)* – enable downloading if database does not exist.
- **subset** *(list, optional)* – indices to subset. Set to None for entire database.
- **load_only** *(list, optional)* – reduced set of properties to be loaded
- **collect_triples** *(bool, optional)* – Set to True if angular features are needed.
- **environment_provider** *(spk.environment.BaseEnvironmentProvider)*
  – define how neighborhood is calculated (default=spk.environment.SimpleEnvironmentProvider).

**create_subset** *(idx)*

Returns a new dataset that only consists of provided indices.

- [dataset with subset of original data](https://arxiv.org/abs/1708.04987)

**Returns**

- [dataset with subset of original data](https://arxiv.org/abs/1708.04987)

**Return type** *schnetpack.data.AtomsData*
class schnetpack.datasets.OrganicMaterialsDatabase(path, download=True, subset=None, load_only=None, collect_triples=False, environment_provider=<schnetpack.environment.SimpleEnvironmentProvider object>)

Organic Materials Database (OMDB) of bulk organic crystals.

Registration to the OMDB is free for academic users. This database contains DFT (PBE) band gap (OMDB-GAP1 database) for 12500 non-magnetic materials.

Parameters

- **path** (*str*) – path to directory containing database.
- **cutoff** (*float*) – cutoff for bulk interactions.
- **download** (*bool, optional*) – enable downloading if database does not exists.
- **subset** (*list*) – indices to subset. Set to None for entire database.
- **load_only** (*list, optional*) – reduced set of properties to be loaded
- **collect_triples** (*bool, optional*) – Set to True if angular features are needed.

References


create_subset(idx)

Returns a new dataset that only consists of provided indices. :param idx: subset indices :type idx: numpy.ndarray

Returns dataset with subset of original data

Return type schnetpack.data.AtomsData

1.15 schnetpack.datasets

1.16 schnetpack.environment

1.16.1 Environment providers

class schnetpack.environment.BaseEnvironmentProvider

Environment Providers are supposed to collect neighboring atoms within local, atom-centered environments. All environment providers should inherit from this class.

get_environment(atoms)

Returns the neighbor indices and offsets

Parameters atoms (*ase.Atoms*) – atomistic system

Returns

indices of the neighbors with shape n_atoms x n_max_neighbors
offset (np.ndarray): offset in lattice coordinates for periodic systems (otherwise zero matrix) of shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \times 3 \)

Return type neighborhood_idx (np.ndarray)

class schnetpack.environment.SimpleEnvironmentProvider
A simple environment provider for small molecules where all atoms are each other’s neighbors. It calculates full distance matrices and does not support cutoffs or periodic boundary conditions.

def get_environment(atoms, grid=None)
Returns the neighbor indices and offsets

Parameters atoms (ase.Atoms) – atomistic system

Returns

indices of the neighbors with shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \)

offset (np.ndarray): offset in lattice coordinates for periodic systems (otherwise zero matrix) of shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \times 3 \)

Return type neighborhood_idx (np.ndarray)

class schnetpack.environment.AseEnvironmentProvider(cutoff)
Environment provider making use of ASE neighbor lists. Supports cutoffs and PBCs.

def get_environment(atoms, grid=None)
Returns the neighbor indices and offsets

Parameters atoms (ase.Atoms) – atomistic system

Returns

indices of the neighbors with shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \)

offset (np.ndarray): offset in lattice coordinates for periodic systems (otherwise zero matrix) of shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \times 3 \)

Return type neighborhood_idx (np.ndarray)

class schnetpack.environment.TorchEnvironmentProvider(cutoff, device)
Environment provider making use of neighbor lists as implemented in TorchAni (https://github.com/aiqm/torchani/blob/master/torchani/aev.py). Supports cutoffs, PBCs and can be performed on either CPU or GPU.

def get_environment(atoms)
Returns the neighbor indices and offsets

Parameters atoms (ase.Atoms) – atomistic system

Returns

indices of the neighbors with shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \)

offset (np.ndarray): offset in lattice coordinates for periodic systems (otherwise zero matrix) of shape \( n_{\text{atoms}} \times n_{\text{max_neighbors}} \times 3 \)

Return type neighborhood_idx (np.ndarray)

1.16.2 Functions

schnetpack.environment.collect_atom_triples(nbh_idx)
Collect all valid triples of atoms by rearranging neighbor indices obtained from an environment provider.

Parameters nbh_idx (numpy.ndarray) – neighbor indices
Returns triple indices offset_idx_j, offset_idx_k (numpy.ndarray): offset indices for PBC

Return type nbh_idx_j, nbh_idx_k (numpy.ndarray)

1.17 schnetpack.nn

Basic building blocks of SchNetPack models. Contains various basic and specialized network layers, layers for cutoff functions, as well as several auxiliary layers and functions.

1.17.1 Basic layers

class schnetpack.nn.Dense(in_features, out_features, bias=True, activation=None, weight_init=<sphinx.ext.autodoc.importer._MockObject object>, bias_init=functools.partial(<sphinx.ext.autodoc.importer._MockObject object>, val=0.0))

Fully connected linear layer with activation function.

\[ y = \text{activation}(xW^T + b) \]

Parameters

- **in_features** (int) – number of input feature \( x \).
- **out_features** (int) – number of output features \( y \).
- **bias** (bool, optional) – if False, the layer will not adapt bias \( b \).
- **activation** (callable, optional) – if None, no activation function is used.
- **weight_init** (callable, optional) – weight initializer from current weight.
- **bias_init** (callable, optional) – bias initializer from current bias.

forward(inputs)

Compute layer output.

Parameters **inputs** (dict of torch.Tensor) – batch of input values.

Returns layer output.

Return type torch.Tensor

reset_parameters()

Reinitialize model weight and bias values.

class schnetpack.nn.GetItem(key)

Extraction layer to get an item from SchNetPack dictionary of input tensors.

Parameters **key** (str) – Property to be extracted from SchNetPack input tensors.

forward(inputs)

Compute layer output.

Parameters **inputs** (dict of torch.Tensor) – SchNetPack dictionary of input tensors.

Returns layer output.

Return type torch.Tensor
class schnetpack.nn.ScaleShift(mean, stddev)
    Scale and shift layer for standardization.
    
    $$y = x \times \sigma + \mu$$

    Parameters
    
    * mean (torch.Tensor) – mean value $$\mu$$.
    * stddev (torch.Tensor) – standard deviation value $$\sigma$$.

    forward(input)
        Compute layer output.
        Parameters
        * input (torch.Tensor) – input data.
        Returns
        * layer output.
        Return type torch.Tensor

class schnetpack.nn.Standardize(mean, stddev, eps=1e-09)
    Standardize layer for shifting and scaling.
    
    $$y = \frac{x - \mu}{\sigma}$$

    Parameters
    
    * mean (torch.Tensor) – mean value $$\mu$$.
    * stddev (torch.Tensor) – standard deviation value $$\sigma$$.
    * eps (float, optional) – small offset value to avoid zero division.

    forward(input)
        Compute layer output.
        Parameters
        * input (torch.Tensor) – input data.
        Returns
        * layer output.
        Return type torch.Tensor

class schnetpack.nn.Aggregate(axis, mean=False, keepdim=True)
    Pooling layer based on sum or average with optional masking.

    Parameters
    
    * axis (int) – axis along which pooling is done.
    * mean (bool, optional) – if True, use average instead for sum pooling.
    * keepdim (bool, optional) – whether the output tensor has dim retained or not.

    forward(input, mask=None)
        Compute layer output.
        Parameters
        * input (torch.Tensor) – input data.
        * mask (torch.Tensor, optional) – mask to be applied; e.g. neighbors mask.
        Returns
        * layer output.
        Return type torch.Tensor
1.17.2 Blocks

class schnetpack.nn.MLP(n_in, n_out, n_hidden=None, n_layers=2, activation=<function shifted_softplus>)

Multiple layer fully connected perceptron neural network.

Parameters

• n_in (int) – number of input nodes.
• n_out (int) – number of output nodes.
• n_hidden (list of int or int, optional) – number hidden layer nodes. If an integer, same number of node is used for all hidden layers resulting in a rectangular network. If None, the number of neurons is divided by two after each layer starting n_in resulting in a pyramidal network.
• n_layers (int, optional) – number of layers.
• activation (callable, optional) – activation function. All hidden layers would the same activation function except the output layer that does not apply any activation function.

forward(inputs)
   Compute neural network output.

   Parameters inputs (torch.Tensor) – network input.

   Returns network output.

   Return type torch.Tensor

class schnetpack.nn.TiledMultiLayerNN(n_in, n_out, n_tiles, n_hidden=50, n_layers=3, activation=<function shifted_softplus>)

Tiled multilayer networks which are applied to the input and produce n_tiles different outputs. These outputs are then stacked and returned. Used e.g. to construct element-dependent prediction networks of the Behler-Parrinello type.

Parameters

• n_in (int) – number of input nodes
• n_out (int) – number of output nodes
• n_tiles (int) – number of networks to be tiled
• n_hidden (int) – number of nodes in hidden nn (default 50)
• n_layers (int) – number of layers (default: 3)

forward(inputs)
   Parameters inputs (torch.Tensor) – Network inputs.

   Returns Tiled network outputs.

   Return type torch.Tensor

class schnetpack.nn.ElementalGate(elements, onehot=True, trainable=False)

Produces a Nbatch x Natoms x Nelem mask depending on the nuclear charges passed as an argument. If onehot is set, mask is one-hot mask, else a random embedding is used. If the trainable flag is set to true, the gate values can be adapted during training.

Parameters

• elements (set of int) – Set of atomic number present in the data
• **onehot** (*bool*) – Use one hit encoding for elemental gate. If set to False, random embedding is used instead.

• **trainable** (*bool*) – If set to true, gate can be learned during training (default False)

```python
def forward(atomic_numbers)
    Parameters atomic_numbers (*torch.Tensor*) – Tensor containing atomic numbers of each atom.
    Returns One-hot vector which is one at the position of the element and zero otherwise.
    Return type *torch.Tensor*
```

**class schnetpack.nn.GatedNetwork** (*nin*, *nout*, *elements*, *n_hidden=50*, *n_layers=3*, *trainable=False*, *onehot=True*, *activation=<function shifted_softplus>*)

Combines the TiledMultiLayerNN with the elemental gate to obtain element specific atomistic networks as in typical Behler–Parrinello networks¹.

Parameters

• **nin** (*int*) – number of input nodes

• **nout** (*int*) – number of output nodes

• **nnodes** (*int*) – number of nodes in hidden nn (default 50)

• **nlayers** (*int*) – number of layers (default 3)

• **elements** (*set of ints*) – Set of atomic number present in the data

• **onehot** (*bool*) – Use one hit encoding for elemental gate. If set to False, random embedding is used instead.

• **trainable** (*bool*) – If set to true, gate can be learned during training (default False)

• **activation** (*callable*) – activation function

**References**

```
def forward(inputs)
    Returns Output of the gated network.
    Return type *torch.Tensor*
```

### 1.17.3 Convolutions

**class schnetpack.nn.CFConv** (*n_in*, *n_filters*, *n_out*, *filter_network*, *cutoff_network=None*, *activation=None*, *normalize_filter=False*, *axis=2*)

Continuous-filter convolution block used in SchNet module.

Parameters

• **n_in** (*int*) – number of input (i.e. atomic embedding) dimensions.

• **n_filters** (*int*) – number of filter dimensions.

• **n_out** (*int*) – number of output dimensions.
• **filter_network** (*nn.Module*) – filter block.
• **cutoff_network** (*nn.Module, optional*) – if None, no cut off function is used.
• **activation** (*callable, optional*) – if None, no activation function is used.
• **normalize_filter** (*bool, optional*) – If True, normalize filter to the number of neighbors when aggregating.
• **axis** (*int, optional*) – axis over which convolution should be applied.

### forward(x, r_ij, neighbors, pairwise_mask, f_ij=None)
Compute convolution block.

**Parameters**

• **x** (*torch.Tensor*) – input representation/embedding of atomic environments with \((N_b, N_a, n_in)\) shape.
• **r_ij** (*torch.Tensor*) – interatomic distances of \((N_b, N_a, N_nbh)\) shape.
• **neighbors** (*torch.Tensor*) – indices of neighbors of \((N_b, N_a, N_nbh)\) shape.
• **pairwise_mask** (*torch.Tensor*) – mask to filter out non-existing neighbors introduced via padding.
• **f_ij** (*torch.Tensor, optional*) – expanded interatomic distances in a basis. If None, \(r_ij.unsqueeze(-1)\) is used.

**Returns** block output with \((N_b, N_a, n_out)\) shape.

**Return type** *torch.Tensor*

---

### 1.17.4 Cutoff

#### class schnetpack.nn.CosineCutoff(*cutoff=5.0*)
Class of Behler cosine cutoff.

\[ f(r) = \begin{cases} 
0.5 \times \left[ 1 + \cos \left( \frac{\pi r}{r_{cutoff}} \right) \right] & r < r_{cutoff} \\
0 & r \geq r_{cutoff} 
\end{cases} \]

**Parameters**

• **cutoff** (*float, optional*) – cutoff radius.

**forward**(*distances*)
Compute cutoff.

**Parameters**

• **distances** (*torch.Tensor*) – values of interatomic distances.

**Returns** values of cutoff function.

**Return type** *torch.Tensor*

#### class schnetpack.nn.MollifierCutoff(*cutoff=5.0, eps=1e-07*)
Class for mollifier cutoff scaled to have a value of 1 at \(r = 0\).

\[ f(r) = \begin{cases} 
\exp \left( 1 - \frac{1}{1 - \left( \frac{r}{r_{cutoff}} \right)^2} \right) & r < r_{cutoff} \\
0 & r \geq r_{cutoff} 
\end{cases} \]

**Parameters**
• **cutoff** *(float, optional)* – Cutoff radius.
• **eps** *(float, optional)* – offset added to distances for numerical stability.

**forward** *(distances)*
Compute cutoff.

**Parameters**
• **distances** *(torch.Tensor)* – values of interatomic distances.

**Returns**
values of cutoff function.

**Return type**
torch.Tensor

```python
class schnetpack.nn.HardCutoff(cutoff=5.0)
```
Class of hard cutoff.

\[
f(r) = \begin{cases} 1 & r \leq r_{\text{cutoff}} \\ 0 & r > r_{\text{cutoff}} \end{cases}
\]

**Parameters**
• **cutoff** *(float)* – cutoff radius.

**forward** *(distances)*
Compute cutoff.

**Parameters**
• **distances** *(torch.Tensor)* – values of interatomic distances.

**Returns**
values of cutoff function.

**Return type**
torch.Tensor

### 1.17.5 Neighbors

```python
class schnetpack.nn.AtomDistances(return_directions=False)
```
Layer for computing distance of every atom to its neighbors.

**Parameters**
• **return_directions** *(bool, optional)* – if True, the **forward** method also returns normalized direction vectors.

**forward** *(positions, neighbors, cell=None, cell_offsets=None, neighbor_mask=None)*
Compute distance of every atom to its neighbors.

**Parameters**
• **positions** *(torch.Tensor)* – atomic Cartesian coordinates with \((N_b \times N_at \times 3)\) shape.
• **neighbors** *(torch.Tensor)* – indices of neighboring atoms to consider with \((N_b \times N_at \times N_nbh)\) shape.
• **cell** *(torch.tensor, optional)* – periodic cell of \((N_b \times 3 \times 3)\) shape.
• **cell_offsets** *(torch.Tensor, optional)* – offset of atom in cell coordinates with \((N_b \times N_at \times N_nbh \times 3)\) shape.
• **neighbor_mask** *(torch.Tensor, optional)* – boolean mask for neighbor positions. Required for the stable computation of forces in molecules with different sizes.

**Returns**
layer output of \((N_b \times N_at \times N_nbh)\) shape.

**Return type**
torch.Tensor
schnetpack.nn.atom_distances(positions, neighbors, cell=None, cell_offsets=None, return_vecs=False, normalize_vecs=False, neighbor_mask=None)

Compute distance of every atom to its neighbors.

This function uses advanced torch indexing to compute differentiable distances of every central atom to its relevant neighbors.

Parameters

- **positions** (torch.Tensor) – atomic Cartesian coordinates with (N_b x N_at x 3) shape
- **neighbors** (torch.Tensor) – indices of neighboring atoms to consider with (N_b x N_at x N_nbh) shape
- **cell** (torch.tensor, optional) – periodic cell of (N_b x 3 x 3) shape
- **cell_offsets** (torch.Tensor, optional) – offset of atom in cell coordinates with (N_b x N_at x N_nbh x 3) shape
- **return_vecs** (bool, optional) – if True, also returns direction vectors.
- **normalize_vecs** (bool, optional) – if True, normalize direction vectors.
- **neighbor_mask** (torch.Tensor, optional) – boolean mask for neighbor positions.

Returns

- **distances**: distance of every atom to its neighbors with (N_b x N_at x N_nbh) shape.
- **dist_vec**: direction cosines of every atom to its neighbors with (N_b x N_at x N_nbh x 3) shape (optional).

Return type (torch.Tensor, torch.Tensor)

class schnetpack.nn.TriplesDistances

Layer that gets all distances between atoms forming a triangle with the central atoms. Required e.g. for angular symmetry functions.

forward(positions, neighbors_j, neighbors_k)

Parameters

- **positions** (torch.Tensor) – Atomic positions
- **neighbors_j** (torch.Tensor) – Indices of first neighbor in triangle
- **neighbors_k** (torch.Tensor) – Indices of second neighbor in triangle

Returns Distance between central atom and neighbor j torch.Tensor: Distance between central atom and neighbor k torch.Tensor: Distance between neighbors

Return type torch.Tensor

class schnetpack.nn.NeighborElements

Layer to obtain the atomic numbers associated with the neighboring atoms.

forward(atomic_numbers, neighbors)

Parameters

- **atomic_numbers** (torch.Tensor) – Atomic numbers (Nbatch x Nat x 1)
- **neighbors** (torch.Tensor) – Neighbor indices (Nbatch x Nat x Nneigh)

Returns Atomic numbers of neighbors (Nbatch x Nat x Nneigh)
**SchNetPack Documentation, Release 0.2rc**

**Return type**  torch.Tensor

### 1.17.6 ACSF

class schnetpack.nn.AngularDistribution (radial_filter, angular_filter, cutoff_functions=<class 'schnetpack.nn.cutoff.CosineCutoff'>, crossterms=False, pairwise_elements=False)

Routine used to compute angular type symmetry functions between all atoms i-j-k, where i is the central atom.

**Parameters**

- **radial_filter** (*callable*) – Function used to expand distances (e.g. Gaussians)
- **angular_filter** (*callable*) – Function used to expand angles between triples of atoms (e.g. BehlerAngular)
- **cutoff_functions** (*callable*) – Cutoff function
- **crossterms** (*bool*) – Include radial contributions of the distances r_jk
- **pairwise_elements** (*bool*) – Recombine elemental embedding vectors via an outer product. If e.g. one-hot encoding is used for the elements, this is equivalent to standard Behler functions (default=False).

**forward**(r_ij, r_ik, r_jk, triple_masks=None, elemental_weights=None)

**Parameters**

- **r_ij** (*torch.Tensor*) – Distances to neighbor j
- **r_ik** (*torch.Tensor*) – Distances to neighbor k
- **r_jk** (*torch.Tensor*) – Distances between neighbor j and k
- **triple_masks** (*torch.Tensor*) – Tensor mask for non-counted pairs (e.g. due to cutoff)
- **elemental_weights** (*tuple of two torch.Tensor*) – Weighting functions for neighboring elements, first is for neighbors j, second for k

**Returns**  Angular distribution functions

**Return type**  torch.Tensor

class schnetpack.nn.BehlerAngular (zetas={1})

Compute Behler type angular contribution of the angle spanned by three atoms:

$$2^{(1-ζ)}(1 + \lambda \cos(\theta_{ijk}))^ζ$$

Sets of zetas with lambdas of -1 and +1 are generated automatically.

**Parameters** zetas (*set of int*) – Set of exponents used to compute angular Behler term (default={1})

**forward**(cos_theta)

**Parameters** cos_theta (*torch.Tensor*) – Cosines between all pairs of neighbors of the central atom.

**Returns**  Tensor containing values of the angular filters.

**Return type**  torch.Tensor
class schnetpack.nn.GaussianSmearing(start=0.0, stop=5.0, n_gaussians=50, centered=False, trainable=False)

Smear layer using a set of Gaussian functions.

Parameters

• **start** *(float, optional)* – center of first Gaussian function, \(\mu_0\).
• **stop** *(float, optional)* – center of last Gaussian function, \(\mu_N\).
• **n_gaussians** *(int, optional)* – total number of Gaussian functions, \(N_g\).
• **centered** *(bool, optional)* – If True, Gaussians are centered at the origin and the offsets are used as their widths (used e.g. for angular functions).
• **trainable** *(bool, optional)* – If True, widths and offset of Gaussian functions are adjusted during training process.

```python
forward(distances)
```
Compute smeared-gaussian distance values.

Parameters

• **distances** *(torch.Tensor)* – interatomic distance values of (N_b x N_at x N_nbh) shape.

Returns

layer output of (N_b x N_at x N_nbh x N_g) shape.

Return type
torch.Tensor

class schnetpack.nn.RadialDistribution(radial_filter, cutoff_function=<class 'schnetpack.nn.cutoff.CosineCutoff'>)

Radial distribution function used e.g. to compute Behler type radial symmetry functions.

Parameters

• **radial_filter** *(callable)* – Function used to expand distances (e.g. Gaussians)
• **cutoff_function** *(callable)* – Cutoff function

```python
forward(r_ij, elemental_weights=None, neighbor_mask=None)
```

Parameters

• **r_ij** *(torch.Tensor)* – Interatomic distances
• **elemental_weights** *(torch.Tensor)* – Element-specific weights for distance functions
• **neighbor_mask** *(torch.Tensor)* – Mask to identify positions of neighboring atoms

Returns

Nbatch x Nats x Nfilter tensor containing radial distribution functions.

Return type
torch.Tensor

1.17.7 Activation functions

schnetpack.nn.shifted_softplus(x)

Compute shifted soft-plus activation function.

\[
y = \ln \left( 1 + e^{-x} \right) - \ln(2)
\]

Parameters

• **x** *(torch.Tensor)* – input tensor.

Returns

shifted soft-plus of input.

Return type
torch.Tensor
1.18 schnetpack.representation

Classes for constructing the different representations available in SchnetPack. This encompasses SchNet\(^1\), Behler-type atom centered symmetry functions (ACSF)\(^2\) and a weighted variant thereof (wACSF)\(^3\).

References

1.18.1 SchNet

class schnetpack.representation.SchNet(n_atom_basis=128, n_filters=128, n_interactions=3, cutoff=5.0, n_gaussians=25, normalize_filter=False, coupled_interactions=False, return_intermediate=False, max_z=100, cutoff_network=<class 'schnetpack.nn.cutoff.HardCutoff'>, trainable_gaussians=False, distance_expansion=None, charged_systems=False)

SchNet architecture for learning representations of atomistic systems.

Parameters

- `n_atom_basis (int, optional)` – number of features to describe atomic environments. This determines the size of each embedding vector; i.e. embeddings_dim.
- `n_filters(int, optional)` – number of filters used in continuous-filter convolution
- `n_interactions(int, optional)` – number of interaction blocks.
- `cutoff(float, optional)` – cutoff radius.
- `n_gaussians(int, optional)` – number of Gaussian functions used to expand atomic distances.
- `normalize_filter(bool, optional)` – if True, divide aggregated filter by number of neighbors over which convolution is applied.
- `coupled_interactions(bool, optional)` – if True, share the weights across interaction blocks and filter-generating networks.
- `return_intermediate(bool, optional)` – if True, forward method also returns intermediate atomic representations after each interaction block is applied.
- `max_z(int, optional)` – maximum nuclear charge allowed in database. This determines the size of the dictionary of embedding; i.e. num_embeddings.
- `cutoff_network(nn.Module, optional)` – cutoff layer.
- `trainable_gaussians(bool, optional)` – If True, widths and offset of Gaussian functions are adjusted during training process.
- `distance_expansion(nn.Module, optional)` – layer for expanding interatomic distances in a basis.
- `charged_systems(bool, optional)` –

---


References: .. [schnet1] Schütt, Arbabzadah, Chmiela, Müller, Tkatchenko:

**forward** *(inputs)*
Compute atomic representations/embeddings.

- **Parameters** `inputs` *(dict of torch.Tensor)* – SchNetPack dictionary of input tensors.
- **Returns** atom-wise representation. list of torch.Tensor: intermediate atom-wise representations, if return_intermediate=True was used.
- **Return type** torch.Tensor

```python
class schnetpack.representation.SchNetInteraction(n_atom_basis, n_spatial_basis, n_filters, cutoff, cutoff_network=<class 'schnetpack.nn.cutoff.HardCutoff'>, normalize_filter=False)
```

SchNet interaction block for modeling interactions of atomistic systems.

**Parameters**
- `n_atom_basis` *(int)* – number of features to describe atomic environments.
- `n_spatial_basis` *(int)* – number of input features of filter-generating networks.
- `n_filters` *(int)* – number of filters used in continuous-filter convolution.
- `cutoff` *(float)* – cutoff radius.
- `normalize_filter` *(bool, optional)* – if True, divide aggregated filter by number of neighbors over which convolution is applied.

**forward** *(x, r_ij, neighbors, neighbor_mask, f_ij=None)*
Compute interaction output.

- **Parameters**
  - `x` *(torch.Tensor)* – input representation/embedding of atomic environments with (N_b, N_a, n_atom_basis) shape.
  - `r_ij` *(torch.Tensor)* – interatomic distances of (N_b, N_a, N_nbh) shape.
  - `neighbors` *(torch.Tensor)* – indices of neighbors of (N_b, N_a, N_nbh) shape.
  - `neighbor_mask` *(torch.Tensor)* – mask to filter out non-existing neighbors introduced via padding.
  - `f_ij` *(torch.Tensor, optional)* – expanded interatomic distances in a basis. If None, r_ij.unsqueeze(-1) is used.

- **Returns** block output with (N_b, N_a, n_atom_basis) shape.
- **Return type** torch.Tensor
1.18.2 Atom-centered symmetry functions

```python
class schnetpack.representation.SymmetryFunctions(n_radial=22, n_angular=5, zetas={1},
cutoff=<class 'schnetpack.nn.cutoff.CosineCutoff'>,
cutoff_radius=5.0, centered=False, crossterms=False,
elements=frozenset({1, 6, 7, 8, 9}), sharez=True,
trainz=False, initz='weighted', len_embedding=5,
pairwise_elements=False)
```

Compute atom-centered symmetry functions and weighted variant thereof as described in Reference. By default, the atomic number is used as element depended weight. However, by specifying the trainz=True, a more general elemental embedding is learned instead.

**Parameters**

- `n_radial` (*int*) – Number of radial functions
- `n_angular` (*int*) – Number of angular functions
- `zetas` (*set of int*) – Set of exponents used to compute the angular term, default is zetas={1}
- `cutoff` (*callable*) – Cutoff function, default is the typical cosine cutoff function
- `cutoff_radius` (*float*) – Cutoff radius, default are 5 Angstrom
- `centered` (*bool*) – Whether centered Gaussians should be used for radial functions. Angular functions use centered Gaussians by default.
- `crossterms` (*bool*) – Whether cutoff and exponential terms of the distance r_{jk} between both neighbors should be included in the angular functions. Default is False
- `elements` (*set of int*) – Nuclear charge present in the molecules, default is {1,6,7,8,9} (H,C,N,O and F).
- `sharez` (*bool*) – Whether angular and radial functions should use the same elemental weighting. The default is true.
- `trainz` (*bool*) – If set to true, elemental weights are initialized at random and learned during training. (default is False)
- `initz` (*str*) – How elemental weights are initialized. Allowed are (default='weighted'):
  - weighted: Weigh symmetry functions with nuclear charges (wACSF) onehot: Represent elements by onehot vectors. This can be used in combination with pairwise_elements
  - embedding: Use random embeddings, which can e.g. be trained. Embedding length is modified via
    - `len_embedding` (*int*) – Number of elemental weights, default is 1. If more are used, embedding vectors similar to SchNet can be obtained.

---


• **pairwise_elements** *(bool)* – Recombine elemental embedding vectors in the angular functions via an outer product. If e.g. one-hot encoding is used for the elements, this is equivalent to standard Behler functions (default=False).

**References**

**forward** *(inputs)*

Parameters **inputs** *(dict of torch.Tensor)* – SchNetPack format dictionary of input tensors.

Returns Nbatch x Natoms x Nsymmetry_functions Tensor containing ACSFs or wACSFs.

Return type torch.Tensor

**initz** *(mode, elements)*

Subroutine to initialize the element dependent weights.

Parameters

- **mode** *(str)* – Manner in which the weights are initialized. Possible are: weighted: Weigh symmetry functions with nuclear charges (wACSF) onehot: Represent elements by onehot vectors. This can be used in combination with pairwise_elements in order to emulate the behavior of classic Behler symmetry functions.
- **elements** *(set of int)* – List of elements present in the molecule.

Returns Embedding layer of the initialized elemental weights.

Return type torch.nn.Embedding

class schnetpack.representation.StandardizeSF *(SFBlock, data_loader=None, cuda=False)*

Compute mean and standard deviation of all symmetry functions computed for the molecules in the data loader and use them to standardize the descriptor vectors,

Parameters

- **SFBlock** *(callable)* – Object for computing the descriptor vectors
- **data_loader** *(object)* – DataLoader containing the molecules used for computing the statistics. If None, dummy vectors are generated instead
- **cuda** *(bool)* – Cuda flag

**forward** *(inputs)*

Parameters **inputs** *(dict of torch.Tensor)* – SchNetPack format dictionary of input tensors.

Returns Standardized representations.

Return type torch.Tensor
1.19 schnetpack.atomistic

1.19.1 Atomistic Models

class schnetpack.atomistic.AtomisticModel(representation, output_modules)
Join a representation model with output modules.

Parameters

• representation (torch.nn.Module) – Representation block of the model.

• output_modules (list or nn.ModuleList or spk.output_modules.Atomwise) – Output block of the model. Needed for predicting properties.

Returns property predictions

Return type dict

forward(inputs)
Forward representation output through output modules.

1.19.2 General output modules

class schnetpack.atomistic.Atomwise(n_in, n_out=1, aggregation_mode='sum', n_layers=2, n_neurons=None, activation=<function shifted_softplus>, property='y', contributions=None, derivative=None, negative_dr=False, create_graph=True, mean=None, stddev=None, atomref=None, outnet=None)
Predicts atom-wise contributions and accumulates global prediction, e.g. for the energy.

Parameters

• n_in (int) – input dimension of representation

• n_out (int) – output dimension of target property (default: 1)

• aggregation_mode (str) – one of {sum, avg} (default: sum)

• n_layers (int) – number of nn in output network (default: 2)

• n_neurons (list of int or None) – number of neurons in each layer of the output network. If None, divide neurons by 2 in each layer. (default: None)

• activation (function) – activation function for hidden nn (default: spk.nn.activations.shifted_softplus)

• property (str) – name of the output property (default: “y”)

• contributions (str or None) – Name of property contributions in return dict. No contributions returned if None. (default: None)

• derivative (str or None) – Name of property derivative. No derivative returned if None. (default: None)

• negative_dr (bool) – Multiply the derivative with -1 if True. (default: False)

• create_graph (bool) – If False, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of create_graph. (default: False)

• mean (torch.Tensor or None) – mean of property
• **stddev** *(torch.Tensor or None)* – standard deviation of property (default: None)

• **atomref** *(torch.Tensor or None)* – reference single-atom properties. Expects an 
  \((\max_z + 1) \times 1\) array where \(\text{atomref}[Z]\) corresponds to the reference property of element \(Z\). The value of \(\text{atomref}[0]\) must be zero, as this corresponds to the reference property for for “mask” atoms. (default: None)

• **outnet** *(callable)* – Network used for atomistic outputs. Takes schnetpack input dictionary as input. Output is not normalized. If set to None, a pyramidal network is generated automatically. (default: None)

**Returns**

prediction for property

If contributions is not None additionally returns atom-wise contributions.

If derivative is not None additionally returns derivative w.r.t. atom positions.

**Return type** *tuple*

**forward** *(inputs)*

predicts atomwise property

```python
class schnetpack.atomistic.ElementalAtomwise(n_in=128, n_out=1, aggregation_mode='sum', n_layers=3, property='y', derivative=None, negative_dr=False, contributions=None, create_graph=True, elements=frozenset({1, 6, 7, 8, 9}), n_hidden=50, activation=<function shifted_softplus>, mean=None, stddev=None, atomref=None)
```

Predicts properties in atom-wise fashion using a separate network for every chemical element of the central atom. Particularly useful for networks of Behler-Parrinello type.

**Parameters**

• **n_in** *(int)* – input dimension of representation (default: 128)

• **n_out** *(int)* – output dimension of target property (default: 1)

• **aggregation_mode** *(str)* – one of \{sum, avg\} (default: sum)

• **n_layers** *(int)* – number of nn in output network (default: 3)

• **property** *(str)* – name of the output property (default: “y”)

• **derivative** *(str or None)* – Name of property derivative. No derivative returned if None. (default: None)

• **negative_dr** *(bool)* – Multiply the derivative with -1 if True. (default: False)

• **contributions** *(str or None)* – Name of property contributions in return dict. No contributions returned if None. (default: None)

• **create_graph** *(bool)* – If False, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to True is not neee ded and often can be worked around in a much more efficient way. Defaults to the value of create_graph. (default: False)

• **elements** *(set of int)* – List of atomic numbers present in the training set \{1,6,7,8,9\} for QM9. (default: frozenset(1,6,7,8,9))

• **n_hidden** *(int)* – number of neurons in each layer of the output network. (default: 50)
• **activation** *(function)* – activation function for hidden nn (default: spk.nn.activations.shifted_softplus)

• **mean** *(torch.Tensor or None)* – mean of property

• **stddev** *(torch.Tensor or None)* – standard deviation of property (default: None)

• **atomref** *(torch.Tensor or None)* – reference single-atom properties. Expects an \((\text{max}_z + 1) \times 1\) array where \(\text{atomref}[Z]\) corresponds to the reference property of element \(Z\). The value of \(\text{atomref}[0]\) must be zero, as this corresponds to the reference property for for “mask” atoms. (default: None)

### 1.19.3 Property-specific output modules

```python
class schnetpack.atomistic.DipoleMoment(n_in, n_layers=2, n_neurons=None, activation=schnetpack.nn.activations.shifted_softplus, property='y', contributions=None, predict_magnitude=False, mean=None, stddev=None, outnet=None)
```

Predicts latent partial charges and calculates dipole moment.

**Parameters**

- **n_in** *(int)* – input dimension of representation
- **n_layers** *(int)* – number of layers in output network (default: 2)
- **n_neurons** *(list of int or None)* – number of neurons in each layer of the output network. If None, divide neurons by 2 in each layer. (default: None)
- **activation** *(torch.Function)* – activation function for hidden nn (default: schnetpack.nn.activations.shifted_softplus)
- **property** *(str)* – name of the output property (default: “y”)
- **contributions** *(str or None)* – Name of property contributions in return dict. No contributions returned if None. (default: None)
- **predict_magnitude** *(bool)* – if True, predict the magnitude of the dipole moment instead of the vector (default: False)
- **mean** *(torch.FloatTensor or None)* – mean of dipole (default: None)
- **stddev** *(torch.FloatTensor or None)* – stddev of dipole (default: None)

**Returns**

- **vector for the dipole moment**

  If `predict_magnitude` is True returns the magnitude of the dipole moment instead of the vector.

  If contributions is not None latent atomic charges are added to the output dictionary.

**Return type** *dict*

```python
forward(inputs)
```

predicts dipole moment

```python
class schnetpack.atomistic.ElementalDipoleMoment(n_in, n_out=1, n_layers=3, contributions=False, property='y', predict_magnitude=False, elements=frozenset({1, 6, 7, 8, 9}), n_hidden=50, activation=schnetpack.nn.activations.shifted_softplus, mean=None, stddev=None)
```

Chapter 1. Contents
Predicts partial charges and computes dipole moment using separate NNs for every different element. Particularly useful for networks of Behler-Parrinello type.

**Parameters**

- `n_in (int)` – input dimension of representation
- `n_out (int)` – output dimension of representation (default: 1)
- `n_layers (int)` – number of layers in output network (default: 3)
- `predict_magnitude (bool)` – if True, predict the magnitude of the dipole moment instead of the vector (default: False)
- `elements (set of int)` – List of atomic numbers present in the training set {1,6,7,8,9} for QM9. (default: frozenset(1,6,7,8,9))
- `n_hidden (int)` – number of neurons in each layer of the output network. (default: 50)
- `activation (function)` – activation function for hidden nn (default: schnetpack.nn.activations.shifted_softplus)
- `activation` – activation function for hidden nn
- `mean (torch.FloatTensor)` – mean of energy
- `stddev (torch.FloatTensor)` – standard deviation of energy

```python
class schnetpack.atomistic.Polarizability(n_in=128, aggregation_mode='sum', n_layers=2, n_neurons=None, activation=<function shifted_softplus>, property='y', isotropic=None, create_graph=True, outnet=None, cutoff_network=None)
```

Predicts polarizability of input molecules.

**Parameters**

- `n_in (int)` – input dimension of representation (default: 128)
- `aggregation_mode (str)` – one of {sum, avg} (default: sum)
- `n_layers (int)` – number of nn in output network (default: 2)
- `n_neurons (list of int or None)` – number of neurons in each layer of the output network. If `None`, divide neurons by 2 in each layer. (default: None)
- `activation (function)` – activation function for hidden nn (default: schnetpack.nn.activations.shifted_softplus)
- `property (str)` – name of the output property (default: “y”)
- `isotropic (str or None)` – Name of isotropic polarizability property in output dict. Only calculated if not None. (default: None)
- `create_graph (bool)` – If False, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of create_graph. (default: False)
- `outnet (callable)` – Network used for atomistic outputs. Takes schnetpack input dictionary as input. Output is not normalized. If set to None, a pyramidal network is generated automatically. (default: None)
- `cutoff_network (nn.Module)` – cutoff network (default: None)
Returns

Polarizability of molecules

Adds isotropic polarizability if isotropic is not None.

Return type  dict

forward(inputs)

predicts atomwise property

1.20 schnetpack.train

Classes to manage the training process.

schnetpack.train.Trainer encapsulates the training loop. It also can automatically monitor the performance on the validation set and contains logic for checkpointing. The training process can be customized using Hooks which derive from schnetpack.train.Hooks.

1.20.1 Trainer

class schnetpack.train.Trainer(model_path, model, loss_fn, optimizer, train_loader, validation_loader, keep_n_checkpoints=3, checkpoint_interval=10, validation_interval=1, hooks=[], loss_is_normalized=True)

Class to train a model.

This contains an internal training loop which takes care of validation and can be extended with custom functionality using hooks.

Parameters

• model_path(str) – path to the model directory.
• model(torch.Module) – model to be trained.
• loss_fn(callable) – training loss function.
• optimizer(torch.optim.optimizer.Optimizer) – training optimizer.
• train_loader(torch.utils.data.DataLoader) – data loader for training set.
• validation_loader(torch.utils.data.DataLoader) – data loader for validation set.
• keep_n_checkpoints(int, optional) – number of saved checkpoints.
• checkpoint_interval(int, optional) – intervals after which checkpoints is saved.
• hooks(list, optional) – hooks to customize training process.
• loss_is_normalized(bool, optional) – if True, the loss per data point will be reported. Otherwise, the accumulated loss is reported.

train(device, n_epochs=9223372036854775807)

Train the model for the given number of epochs on a specified device.

Parameters

• device(torch.torch.Device) – device on which training takes place.
• n_epochs(int) – number of training epochs.
Note: Depending on the `hooks`, training can stop earlier than `n_epochs`.

### 1.20.2 Hooks

#### `class schnetpack.train.LoggingHook(log_path, metrics, log_train_loss=True, log_validation_loss=True, log_learning_rate=True)`

Base class for logging hooks.

**Parameters**

- `log_path` *(str)* – path to directory in which log files will be stored.
- `metrics` *(list)* – metrics to log; each metric has to be a subclass of spk.Metric.
- `log_train_loss` *(bool, optional)* – enable logging of training loss.
- `log_validation_loss` *(bool, optional)* – enable logging of validation loss.
- `log_learning_rate` *(bool, optional)* – enable logging of current learning rate.

**on_epoch_begin(trainer)**

Log at the beginning of train epoch.

**Parameters**

- `trainer` *(Trainer)* – instance of schnetpack.train.trainer.Trainer class.

#### `class schnetpack.train.TensorboardHook(log_path, metrics, log_train_loss=True, log_validation_loss=True, log_learning_rate=True, every_n_epochs=1, img_every_n_epochs=10, log_histogram=False)`

Hook for logging training process to tensorboard.

**Parameters**

- `log_path` *(str)* – path to directory in which log files will be stored.
- `metrics` *(list)* – metrics to log; each metric has to be a subclass of spk.Metric.
- `log_train_loss` *(bool, optional)* – enable logging of training loss.
- `log_validation_loss` *(bool, optional)* – enable logging of validation loss.
- `log_learning_rate` *(bool, optional)* – enable logging of current learning rate.
- `every_n_epochs` *(int, optional)* – epochs after which logging takes place.
- `img_every_n_epochs` *(int, optional)* –
- `log_histogram` *(bool, optional)* –

#### `class schnetpack.train.CSVHook(log_path, metrics, log_train_loss=True, log_validation_loss=True, log_learning_rate=True, every_n_epochs=1)`

Hook for logging training process to CSV files.

**Parameters**

- `log_path` *(str)* – path to directory in which log files will be stored.
- `metrics` *(list)* – metrics to log; each metric has to be a subclass of spk.Metric.
- `log_train_loss` *(bool, optional)* – enable logging of training loss.
- `log_validation_loss` *(bool, optional)* – enable logging of validation loss.
- `log_learning_rate` *(bool, optional)* – enable logging of current learning rate.
- `every_n_epochs` *(int, optional)* – epochs after which logging takes place.
class schnetpack.train.EarlyStoppingHook(patience, threshold_ratio=0.0001)
Hook to stop training if validation loss fails to improve.

Parameters
- **patience** (*int*) – number of epochs which can pass without improvement of validation loss before training ends.
- **threshold_ratio** (*float, optional*) – counter increases if curr_val_loss > (1-threshold_ratio) * best_loss

class schnetpack.train.MaxEpochHook(max_epochs)
Hook to stop training when a maximum number of epochs is reached.

Parameters **max_epochs** (*int*) – maximal number of epochs.

on_epoch_begin(trainer)
Log at the beginning of train epoch.

Parameters **trainer** (*Trainer*) – instance of schnetpack.train.trainer.Trainer class.

class schnetpack.train.MaxStepHook(max_steps)
Hook to stop training when a maximum number of steps is reached.

Parameters **max_steps** (*int*) – maximum number of steps.

on_batch_begin(trainer, train_batch)
Log at the beginning of train batch.

Parameters
- **trainer** (*Trainer*) – instance of schnetpack.train.trainer.Trainer class.
- **train_batch** (*dict of torch.Tensor*) – SchNetPack dictionary of input tensors.

class schnetpack.train.LRScheduleHook(scheduler, each_step=False)
Base class for learning rate scheduling hooks.

This class provides a thin wrapper around torch.optim.lr_schedule._LRScheduler.

Parameters
- **scheduler** (*torch.optim.lr_schedule._LRScheduler*) – scheduler.
- **each_step** (*bool, optional*) – if set to True scheduler.step() is called every step, otherwise every epoch.

on_batch_begin(trainer, train_batch)
Log at the beginning of train batch.

Parameters
- **trainer** (*Trainer*) – instance of schnetpack.train.trainer.Trainer class.
- **train_batch** (*dict of torch.Tensor*) – SchNetPack dictionary of input tensors.

on_epoch_begin(trainer)
Log at the beginning of train epoch.

Parameters **trainer** (*Trainer*) – instance of schnetpack.train.trainer.Trainer class.

class schnetpack.train.ReduceLROnPlateauHook(optimizer, patience=25, factor=0.5, min_lr=1e-06, window_length=1, stop_after_min=False)
Hook for reduce plateau learning rate scheduling.
This class provides a thin wrapper around torch.optim.lr_schedule.ReduceLROnPlateau. It takes the parameters of ReduceLROnPlateau as arguments and creates a scheduler from it whose step() function will be called every epoch.

Parameters

- **patience** *(int, optional)* – number of epochs with no improvement after which learning rate will be reduced. For example, if patience = 2, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn’t improved then.

- **factor** *(float, optional)* – factor by which the learning rate will be reduced. new_lr = lr * factor.

- **min_lr** *(float or list, optional)* – scalar or list of scalars. A lower bound on the learning rate of all param groups or each group respectively.

- **window_length** *(int, optional)* – window over which the accumulated loss will be averaged.

- **stop_after_min** *(bool, optional)* – if enabled stops after minimal learning rate is reached.

class schnetpack.train.ExponentialDecayHook(optimizer, gamma=0.96, step_size=100000)

Hook for reduce plateau learning rate scheduling.

This class provides a thin wrapper around torch.optim.lr_schedule.StepLR. It takes the parameters of StepLR as arguments and creates a scheduler from it whose step() function will be called every step.

Parameters

- **gamma** *(float)* – Factor by which the learning rate will be reduced. new_lr = lr * gamma

- **step_size** *(int)* – Period of learning rate decay.

1.20.3 Metrics

class schnetpack.train.Metric(target, model_output=None, name=None, element_wise=False)

Base class for all metrics.

Metrics measure the performance during the training and evaluation.

Parameters

- **target** *(str)* – name of target property

- **model_output** *(int, str)* – index or key, in case of multiple outputs (Default: None)

- **name** *(str)* – name used in logging for this metric. If set to None, MSE_[target] will be used (Default: None)

- **element_wise** *(bool)* – set to True if the model output is an element-wise property (forces, positions, . . . )

add_batch(batch, result)

Add a batch to calculate the metric on

aggregate()

Aggregate metric over all previously added batches.

reset()

Reset metric attributes after aggregation to collect new batches.
class schnetpack.train.ModelBias(target,  
model_output=None,  
name=None,  
element_wise=False)
Calculates the bias of the model. For non-scalar quantities, the mean of all components is taken.

Parameters

• **target (str)** – name of target property

• **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)

• **name (str)** – name used in logging for this metric. If set to None, MSE_[target] will be used (Default: None)

• **element_wise (bool)** – set to True if the model output is an element-wise property (forces, positions, . . .)

add_batch (batch, result)
Add a batch to calculate the metric on

aggregate ()
Aggregate metric over all previously added batches.

reset ()
Reset metric attributes after aggregation to collect new batches.

class schnetpack.train.MeanSquaredError(target,  
model_output=None,  
bias_correction=None,  
name=None,  
element_wise=False)
Metric for mean square error. For non-scalar quantities, the mean of all components is taken.

Parameters

• **target (str)** – name of target property

• **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)

• **name (str)** – name used in logging for this metric. If set to None, MSE_[target] will be used (Default: None)

• **element_wise (bool)** – set to True if the model output is an element-wise property (forces, positions, . . .)

add_batch (batch, result)
Add a batch to calculate the metric on

aggregate ()
Aggregate metric over all previously added batches.

reset ()
Reset metric attributes after aggregation to collect new batches.

class schnetpack.train.RootMeanSquaredError(target,  
model_output=None,  
bias_correction=None,  
name=None,  
element_wise=False)
Metric for root mean square error. For non-scalar quantities, the mean of all components is taken.

Parameters

• **target (str)** – name of target property

• **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)

• **name (str)** – name used in logging for this metric. If set to None, RMSE_[target] will be used (Default: None)
- **element_wise** *(bool)* – set to True if the model output is an element-wise property (forces, positions, ...)

```python
aggregate()
```
Aggregate metric over all previously added batches.

```python
class schnetpack.train.MeanAbsoluteError(target, model_output=None, bias_correction=None, name=None, element_wise=False)
```
Metric for mean absolute error. For non-scalar quantities, the mean of all components is taken.

**Parameters**

- **target** *(str)* – name of target property
- **model_output** *(int, str)* – index or key, in case of multiple outputs (Default: None)
- **name** *(str)* – name used in logging for this metric. If set to None, MAE_[target] will be used (Default: None)
- **element_wise** *(bool)* – set to True if the model output is an element-wise property (forces, positions, ...)

```python
add_batch(batch, result)
```
Add a batch to calculate the metric on

```python
aggregate()
```
Aggregate metric over all previously added batches.

```python
reset()
```
Reset metric attributes after aggregation to collect new batches.

```python
class schnetpack.train.HeatmapMAE(target, model_output=None, name=None, element_wise=False)
```
Metric for heatmap of component-wise mean square error of non-scalar quantities.

**Parameters**

- **target** *(str)* – name of target property
- **model_output** *(int, str)* – index or key, in case of multiple outputs (Default: None)
- **name** *(str)* – name used in logging for this metric. If set to None, HeatmapMAE_[target] will be used (Default: None)
- **element_wise** *(bool)* – set to True if the model output is an element-wise property (forces, positions, ...)

```python
add_batch(batch, result)
```
Add a batch to calculate the metric on

```python
aggregate()
```
Aggregate metric over all previously added batches.

```python
class schnetpack.train.SumMAE(target, model_output=None, axis=1, name=None, element_wise=False)
```
Metric for mean absolute error of length.

**Parameters**

- **target** *(str)* – name of target property
- **model_output** *(int, str)* – index or key, in case of multiple outputs (Default: None)
- **name** *(str)* – name used in logging for this metric. If set to None, LengthMAE_[target] will be used (Default: None)
class schnetpack.train.LengthMSE(target, model_output=None, name=None, element_wise=False)

Metric for mean square error of length.

Parameters

- **target (str)** – name of target property
- **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)
- **name (str)** – name used in logging for this metric. If set to None, LengthMSE_[target] will be used (Default: None)
- **element_wise (bool)** – set to True if the model output is an element-wise property (forces, positions, ...)

class schnetpack.train.LengthMAE(target, model_output=None, name=None, element_wise=False)

Metric for mean absolute error of length.

Parameters

- **target (str)** – name of target property
- **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)
- **name (str)** – name used in logging for this metric. If set to None, LengthMAE_[target] will be used (Default: None)

class schnetpack.train.LengthRMSE(target, model_output=None, name=None, element_wise=False)

Metric for root mean square error of length.

Parameters

- **target (str)** – name of target property
- **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)
- **name (str)** – name used in logging for this metric. If set to None, LengthRMSE_[target] will be used (Default: None)
- **element_wise (bool)** – set to True if the model output is an element-wise property (forces, positions, ...)

class schnetpack.train.AngleMSE(target, model_output=None, name=None)

Metric for mean square error of angles.

Parameters

- **target (str)** – name of target property
- **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)
- **name (str)** – name used in logging for this metric. If set to None, AngleMSE_[target] will be used (Default: None)

add_batch(batch, result)

Add a batch to calculate the metric on

class schnetpack.train.AngleMAE(target, model_output=None, name=None)

Metric for mean absolute error of angles.

Parameters

- **target (str)** – name of target property
- **model_output (int, str)** – index or key, in case of multiple outputs (Default: None)
• name (str) – name used in logging for this metric. If set to None, AngleMAE_[target] will be used (Default: None)

add_batch (batch, result)
Add a batch to calculate the metric on

class schnetpack.train.AngleRMSE (target, model_output=None, name=None)
Metric for root mean square error of angles.

Parameters
• target (str) – name of target property
• model_output (int, str) – index or key, in case of multiple outputs (Default: None)
• name (str) – name used in logging for this metric. If set to None, AngleRMSE_[target] will be used (Default: None)

add_batch (batch, result)
Add a batch to calculate the metric on

1.21 schnetpack.md

This module contains all functionality for performing various molecular dynamics simulations using SchNetPack.

1.21.1 Calculators

Calculators for molecular dynamics simulations in SchNetPack. These calculators take the current structures in the schnetpack.md.System class and compute the associated forces. Other properties can be returned, but molecular forces are the bare minimum for driving a simulations. All calculators should be derived from the base classes MDCalculator (if no external code is called) or QMCalculator (if external codes are required).

Currently implemented machine learning calculators include the SchnetPackCalculator for all models generated with SchNetPack and the SGDMLCalculator for sGDML models1. In addition, an OrcaCalculator class can be used to carry out molecular dynamics using the ORCA electronic structure code2.

References

class schnetpack.md.calculators.MDCalculator (required_properties, force_handle, position_conversion=1.0, force_conversion=1.0, property_conversion={}, detach=True)
Base MDCalculator class for computing and updating the forces of the simulated system, as well as other properties predicted by the model.

Parameters
• required_properties (list) – List of the property names which will be passed to the simulator
• force_handle (str) – Name of the property corresponding to the forces.

2 Neese: The ORCA program system. WIREs Comput Mol Sci, 2 (1), 73-78. 2012.
• **position_conversion** *(float)* – Unit conversion for the length used in the model computing all properties. E.g. if the model needs Angstrom, one has to provide the conversion factor converting from the atomic units used internally (Bohr) to Angstrom: 0.529177...

• **force_conversion** *(float)* – Conversion factor converting the forces returned by the used model back to atomic units (Hartree/Bohr).

• **property_conversion** *(dict(float))* – Optional dictionary of conversion factors for other properties predicted by the model. Only changes the units used for logging the various outputs.

• **detach** *(bool)* – Detach property computation graph after every calculator call. Enabled by default. Should only be disabled if one wants to e.g. compute derivatives over short trajectory snippets.

**calculate**(system)
Main calculator routine, which needs to be implemented individually. This routine should take the current system state, perform the appropriate computations to get the forces and use them to update the system forces stored in system.forces

To this end, results should be stored in the dictionary self.results using the keys contained in self.required_properties. Afterwards, the routine self._update_system(system) can be used to update the system state.

**Parameters**
- **system**(schnetpack.md.System) – System object containing current state of the simulation.

**class** schnetpack.md.calculators.QMCalculator(required_properties, force_handle, compdir, qm_executable, position_conversion=0.5291772105638411, force_conversion=1.0, property_conversion={}, adaptive=False)

Basic calculator for interfacing quantum chemistry codes with SchNetPack molecular dynamics.

Calculator for interfacing the ORCA code package with SchNetPack molecular dynamics. Requires ORCA to be installed and an input file template. This template is a standard ORCA input file, with everything past the specification of coordinate format, charge and multiplicity removed (coordinates and final *). If desired, a Queuer can be give, which will attempt to send all jobs to a grid engine queue.

In general, the calculator will take the current System to generate inputs, perform the calculation with ORCA, extract data from the output file (using the OrcaParser class) and update the System.

**Parameters**
- **required_properties** *(list)* – List of properties which should be extracted from output.

- **force_handle** *(str)* – Indicator for molecular forces.

- **compdir** *(str)* – Directory in which computations are performed.

- **qm_executable** *(str)* – Path to the ORCA executable.

- **position_conversion** *(str/float, optional)* – Conversion of positions from atomic units.

- **force_conversion** *(str/float, optional)* – Conversion of forces to atomic units.

- **property_conversion** *(dict, optional)* – Convert properties to requested units. If left empty, no conversion is performed.
adaptive (bool, optional) – Specify, whether the calculator should be used for adaptive sampling.

calculate (system, samples=None)
Perform the calculation with a quantum chemistry code. If samples is given, only a subset of molecules is selected.

Parameters

• system (schnetpack.md.System) – System from the molecular dynamics simulation.

• samples (np.array, optional) – Integer array specifying whether only particular replicas and molecules in the system should be used for computations. Only works with adaptive sampling.

Returns

atom_buffer: List of ASE atoms objects of every computed molecule. Only returned if adaptive sampling is activated.

property_buffer: List of property dictionaries for every computation. Only returned if adaptive sampling is activated.

Return type (list, list)

class schnetpack.md.calculators.SchnetPackCalculator (model, required_properties, force_handle, position_conversion=0.5291772105638411, force_conversion=0.01944690395740863, property_conversion={}, detach=True)

MD calculator for schnetpack models.

Parameters

• model (object) – Loaded schnetpack model.

• required_properties (list) – List of properties to be computed by the calculator

• force_handle (str) – String indicating the entry corresponding to the molecular forces

• position_conversion (float) – Unit conversion for the length used in the model computing all properties. E.g. if the model needs Angstrom, one has to provide the conversion factor converting from the atomic units used internally (Bohr) to Angstrom: 0.529177...

• force_conversion (float) – Conversion factor converting the forces returned by the used model back to atomic units (Hartree/Bohr).

• property_conversion (dict (float)) – Optional dictionary of conversion factors for other properties predicted by the model. Only changes the units used for logging the various outputs.

• detach (bool) – Detach property computation graph after every calculator call. Enabled by default. Should only be disabled if one wants to e.g. compute derivatives over short trajectory snippets.

calculate (system)
Main routine, generates a properly formatted input for the schnetpack model from the system, performs the computation and uses the results to update the system state.

Parameters system (schnetpack.md.System) – System object containing current state of the simulation.
class schnetpack.md.calculators.OrcaCalculator(required_properties, force_handle, compdir, qm_executable, orca_template, orca_parser=schnetpack.md.parsers.orca_parser.OrcaMainFileParser(), position_conversion=0.5291772105638411, force_conversion=1.0, property_conversion={}, queuer=None, adaptive=False, basename='input')

Calculator for interfacing the ORCA code package with SchNetPack molecular dynamics. Requires ORCA to be installed and an input file template. This template is a standard ORCA input file, with everything past the specification of coordinate format, charge and multiplicity removed (coordinates and final *). If desired, a Queuer can be given, which will attempt to send all jobs to a grid engine queue.

In general, the calculator will take the current System to generate inputs, perform the calculation with ORCA, extract data from the output file (using the OrcaParser class) and update the System.

Parameters

- **required_properties (list)** – List of properties which should be extracted from output.
- **force_handle (str)** – Indicator for molecular forces.
- **compdir (str)** – Directory in which computations are performed.
- **qm_executable (str)** – Path to the ORCA executable.
- **orca_template (str)** – Path to an ORCA template which will be used to generate input files.
- **orca_parser (schnetpack.md.parsers.OrcaParser, optional)** – Parser used to extract data from output files.
- **position_conversion (str/float, optional)** – Conversion of positions from atomic units.
- **force_conversion (str/float, optional)** – Conversion of forces to atomic units.
- **property_conversion (dict, optional)** – Convert properties to requested units. If left empty, no conversion is performed.
- **queuer (schnetpack.md.calculator.Queuer, optional)** – If given, jobs will be submitted to a grid engine queue.
- **adaptive (bool, optional)** – Specify, whether the calculator should be used for adaptive sampling.
- **basename (str, optional)** – Basename of the generated input files.

---

3 Neese: The ORCA program system. WIREs Comput Mol Sci, 2 (1), 73-78. 2012.
References

class schnetpack.md.calculators.SGMLCalculator(model, required_properties=['energy', 'forces'], force_handle='forces', position_conversion=0.5291772105638411, force_conversion=0.0008432975637539252, property_conversion={}, detach=True)

Calculator for the sGDML model published in\(^4\) and\(^5\). This model predicts energies and forces and currently only is available for molecules of the same size. In order to use the calculator, the sgdml code package available online at https://github.com/stefanch/sGDML and described in\(^6\) is required.

Parameters

- **model** (torch.nn.module) – Loaded sGDML model.
- **required_properties** (list) – Properties to predict with sGDML, available are energies and forces, which are default.
- **force_handle** (str) – Name of forces in output. Default is ‘forces’.
- **force_conversion** (float) – Convert forces from sGDML to atomic units. sGDML used kcal/mol/Angstrom, so this default is used here.
- **position_conversion** (float) – Unit conversion for the length used in the model computing all properties. E.g. if the model needs Angstrom, one has to provide the conversion factor converting from the atomic units used internally (Bohr) to Angstrom: 0.529177. Since sGDML uses Angstrom and the MD algorithm atomic units, this factor is used as default here
- **property_conversion** (dict(float)) – Optional dictionary of conversion factors for other properties predicted by the model. Only changes the units used for logging the various outputs.
- **detach** (bool) – Detach property computation graph after every calculator call. Enabled by default. Should only be disabled if one wants to e.g. compute derivatives over short trajectory snippets.

References

calculate(system)

Main routine, extract the positions from the current system, use the sGDML model to predict the energies and forces and construct the results dictionary, which is then used to update the system.

Parameters **system** (schnetpack.md.System) – System object containing current state of the simulation.

1.21.2 Integrators

Integrators are used to propagate the simulated system in time. SchNetPack provides two basic types of integrators. The Velocity Verlet integrator is a standard integrator for a purely classical simulations of the nuclei. The ring polymer


molecular dynamics integrator simulates multiple replicas of the system coupled by harmonic springs and recovers a certain extent of nuclear quantum effects (e.g. tunneling).

```python
class schnetpack.md.VelocityVerlet(time_step, device='cuda')
    Standard velocity Verlet integrator for non ring-polymer simulations.
```

**Parameters**
- `time_step (float)` – Integration time step in femto seconds.

```python
class schnetpack.md.RingPolymer(n_beads, time_step, temperature, transformation=<class 'schnetpack.md.utils.normal_mode_transformation.NormalModeTransformer'>, device='cuda')
```

Integrator for ring polymer molecular dynamics, as e.g. described in\(^7\)

During the main step, ring polymer positions and momenta are transformed from bead to normal mode representation, propagated deterministically and then transformed back. Needs the number of beads and the ring polymer temperature in order to initialize the propagator matrix. The integrator reverts to standard velocity Verlet integration if only one bead is used.

Uses atomic units of time internally.

**Parameters**
- `n_beads (int)` – Number of beads in the ring polymer.
- `time_step (float)` – Time step in femto seconds.
- `temperature (float)` – Ring polymer temperature in Kelvin.
- `transformation (object)` – Normal mode transformer class.
- `device (str)` – Device used for computations, default is GPU (`'cuda'`)

**References**

1.21.3 System

This module is used to store all information on the simulated atomistic systems. It includes functionality for loading molecules from files. All this functionality is encoded in the `schnetpack.md.System` class.

```python
class schnetpack.md.System(n_replicas, device='cuda', neighborlist=<class 'schnetpack.md.neighbor_lists.SimpleNeighborList'>, initializer=None)
```

Container for all properties associated with the simulated molecular system (masses, positions, momenta, ...). Uses atomic units internally.

In order to simulate multiple systems efficiently dynamics properties (positions, momenta, forces) are torch tensors with the following dimensions:

\[
\text{n\_replicas x n\_molecules x n\_atoms x 3}
\]

Here `n\_replicas` is the number of copies for every molecule. In a normal simulation, these are treated as independent molecules e.g. for sampling purposes. In the case of ring polymer molecular dynamics (using the RingPolymer integrator), these replicas correspond to the beads of the polymer. `n\_molecules` is the number of different molecules constituting the system, these can e.g. be different initial configurations of the same system (once again for sampling) or completely different molecules. In the latter case, the maximum number of atoms `n\_atoms` (3rd dimension) is determined automatically and all arrays padded with zeros.

Static properties (`n\_atoms`, masses, atom\_types and atom\_masks) are stored in tensors of the shape:

\[
\text{n\_atoms}
\]

---

\(^7\) Ceriotti, Parrinello, Markland, Manolopoulos: Efficient stochastic thermostatting of path integral molecular dynamics. The Journal of Chemical Physics, 133, 124105. 2010.
n_atoms : 1 x n_molecules (the same for all replicas) masses : 1 x n_molecules x n_atoms x 1 (the same for all replicas) atom_types : n_replicas x n_molecules x n_atoms x 1 (are brought to this shape in order to avoid reshapes during every calculator call)

atom_masks [n_replicas x n_molecules x n_atoms x 1 (can change if neighbor lists change for the replicas)

n_atoms contains the number of atoms present in every molecule, masses and atom_types contain the molecular masses and nuclear charges. atom_masks is a binary array used to mask superfluous entries introduced by the zero-padding for differently sized molecules.

Finally a dictionary properties stores the results of every calculator call for easy access of e.g. energies and dipole moments.

Parameters
- n_replicas (int) – Number of replicas generated for each molecule.
- device (str) – Computation device (default='cuda').
- neighborlist (object) – Routine for generating the neighbor list used in the calculator (default is SimpleNeighborList).

center_of_mass
Compute the center of mass for each replica and molecule

Returns

n_replicas x n_molecules x 1 x 3 tensor holding the center of mass.

Return type torch.Tensor

centroid_kinetic_energy
Convenience property for computing the kinetic energy associated with the centroid of each molecule. Only sensible in the context of ring polymer molecular dynamics.

Returns

Tensor of the centroid kinetic energies (in Hartree) with the shape 1 x n_molecules

Return type torch.Tensor

centroid_momenta
Convenience property to access the centroid momenta during ring polymer molecular dynamics. Does not make sense during a standard dynamics setup.

Returns

Tensor of the shape 1 x n_molecules x n_atoms x 3 holding the centroid momenta.

Return type torch.Tensor

centroid_positions
Convenience property to access the positions of the centroid during ring polymer molecular dynamics. Does not make sense during a standard dynamics setup.

Returns

Tensor of the shape 1 x n_molecules x n_atoms x 3 holding the centroid positions.

Return type torch.Tensor

centroid_temperature
Convenience property for accessing the instantaneous temperatures of the centroid of each molecule. Only makes sense in the context of ring polymer molecular dynamics.
Returns

Tensor of the instantaneous centroid temperatures (in Kelvin) with the shape 1 x n_molecules

Return type torch.Tensor

centroid_velocities
Convenience property to access the velocities of the centroid during ring polymer molecular dynamics (e.g. for computing power spectra). Does not make sense during a standard dynamics setup.

Returns Tensor of the shape 1 x n_molecules x n_atoms x 3 holding the centroid velocities.

Return type torch.Tensor

kinetic_energy
Convenience property for computing the kinetic energy associated with each replica and molecule.

Returns

Tensor of the kinetic energies (in Hartree) with the shape n_replicas x n_molecules

Return type torch.Tensor

load_molecules(molecules)
Initializes all required variables and tensors based on a list of ASE atoms objects.

Parameters

• list(molecules) – List of ASE atoms objects containing structures and chemical elements.

load_molecules_from_xyz(path_to_file)
Wrapper for loading molecules from .xyz file

Parameters path_to_file(str) – path to data-file

remove_com()
Move all structures to their respective center of mass.

remove_com_rotation(detach=True)
Remove all components in the current momenta associated with rotational motion using Eckart conditions.

Parameters detach(bool) – Whether computational graph should be detached in order to accelerated the simulation (default=True).

remove_com_translation()
Remove all components in the current momenta associated with translational motion.

state_dict
State dict for storing the system state.

Returns

Dictionary containing all properties for restoring the current state of the system during simulation.

Return type dict

temperature
Convenience property for accessing the instantaneous temperatures of each replica and molecule.

Returns

Tensor of the instantaneous temperatures (in Kelvin) with the shape n_replicas x n_molecules
Return type torch.Tensor
velocities
Convenience property to access molecular velocities instead of the momenta (e.g for power spectra)
Returns Velocity tensor with the same shape as the momenta.
Return type torch.Tensor

1.21.4 Initial Conditions

Module for setting up the initial conditions of the molecules in schnetpack.md.System. This entails sampling the momenta from random distributions corresponding to certain temperatures.

class schnetpack.md.Initializer(temperature)
    Basic initializer class template. Initializes the systems momenta to correspond to a certain temperature.

    Parameters temperature (float) – Target initialization temperature in Kelvin.

    initialize_system(system)
    Initialize the system according to the instructions given in _setup_momenta.

    Parameters system (object) – System class containing all molecules and their replicas.

class schnetpack.md.MaxwellBoltzmannInit(temperature, remove_translation=False, remove_rotation=False)
    Initializes the system momenta according to a Maxwell–Boltzmann distribution at the given temperature.

    Parameters
    • temperature (float) – Target temperature in Kelvin.
    • remove_translation (bool) – Remove the translational components of the momenta after initialization. Will stop molecular drift for NVE simulations and NVT simulations with deterministic thermostats (default=False).
    • remove_rotation (bool) – Remove the rotational components of the momenta after initialization. Will reduce molecular rotation for NVE simulations and NVT simulations with deterministic thermostats (default=False).

1.21.5 Neighbor Lists

Neighbor lists are used to obtain the indices of neighbors surrounding an atom for the schnetpack.md.calculators.SchnetPackCalculator. Currently only a primitive version of a neighbor list is implemented, which cannot deal with periodic boundary conditions and does not possess optimal scaling for large systems.

class schnetpack.md.MDNeighborList(system, cutoff)
    Basic neighbor list template for molecular dynamics simulations required for the calculator. This is used to obtain the indices of the atoms adjacent to a central atom and e.g. used to compute the molecular interactions. The neighbor mask is zero for interactions which should not be counted and one otherwise.

    Parameters
    • system (object) – System class containing all molecules and their replicas.
    • cutoff (float) – Cutoff radius used for neighbor list construction.

    get_neighbors()
    Convenience routine to obtain the neighbor list and neighbor mask in one step.

    Returns Contains the neighbor list and neighbor mask tensors.
Return type  tuple

update_neighbors()
Recompute the neighbor list (e.g. during MD simulations).

class schnetpack.md.SimpleNeighborList(system, cutoff=None)
Basic implementation of a neighbor list. Simply enumerates the neighbors of all atoms in the molecule after
eliminating self interactions. This work fine for small to medium sized systems, but should not be used for
extended molecules, etc. The cutoff fulfills no purpose in this basic implementation.

Parameters

- system (object) – System class containing all molecules and their replicas.
- cutoff (float) – Cutoff radius used for neighbor list construction, not used in the present
  implementation.

update_neighbors()
Simply rebuilds the neighbor list in this naive implementation.

1.21.6 Simulator

All molecular dynamics in SchNetPack is performed using the schnetpack.md.Simulator class. This class collects the atomistic system (schnetpack.md.System), calculators (schnetpack.md.calculators), integrators (schnetpack.md.integrators) and various simulation hooks (schnetpack.md.simulation_hooks) and performs the time integration.

class schnetpack.md.simulator.Simulator (system, integrator, calculator, simulator_hooks=[], step=0, restart=False)
Main driver of the molecular dynamics simulation. Uses an integrator to propagate the molecular system defined in the system class according to the forces yielded by a provided calculator.

In addition, hooks can be applied at five different stages of each simulation step:

- Start of the simulation (e.g. for initializing thermostats)
- Before first integrator half step (e.g. thermostats)
- After computation of the forces and before main integrator step (e.g. for accelerated MD)
- After second integrator half step (e.g. thermostats, output routines)
- At the end of the simulation (e.g. general wrap up of file writes, etc.)

This routine has a state dict which can be used to restart a previous simulation.

Parameters

- system (object) – Instance of the system class defined in molecular_dynamics.system
  holding the structures, masses, atom type, momenta, forces and properties of all molecules
  and their replicas
- integrator (object) – Integrator for propagating the molecular dynamics simulation,
  defined in schnetpack.md.integrators
- calculator (object) – Calculator class used to compute molecular forces for propagation
  and (if requested) various other properties.
- simulator_hooks (list (object)) – List of different hooks to be applied during
  simulations. Examples would be file loggers and thermostats.
**step** (*int*) – Index of the initial simulation step.

**restart** (*bool*) – Indicates, whether the simulation is restarted. E.g. if set to True, the simulator tries to continue logging in the previously created dataset. (default=False) This is set automatically by the `restart_simulation` function. Enabling it without the function currently only makes sense if independent simulations should be written to the same file.

`load_system_state` (*state_dict*)
Routine for only loading the system state of previous simulations. This can e.g. be used for production runs, where an equilibrated system is loaded, but the thermostat is changed.

Parameters **state_dict** (*dict*) – State dict of the current simulation

`restart_simulation` (*state_dict, soft=False*)
Routine for restarting a simulation. Reads the current step, as well as system state from the provided state dict. In case of the simulation hooks, only the states of the thermostat hooks are restored, as all other hooks do not depend on previous simulations.

If the soft option is chosen, only restores states of thermostats if they are present in the current simulation and the state dict. Otherwise, all thermostats found in the state dict are required to be present in the current simulation.

Parameters

- **state_dict** (*dict*) – State dict of the current simulation
- **soft** (*bool*) – Flag to toggle hard/soft thermostat restarts (default=False)

`simulate` (*n_steps=10000*)
Main simulation function. Propagates the system for a certain number of steps.

Parameters **n_steps** (*int*) – Number of simulation steps to be performed (default=10000)

**state_dict**
State dict used to restart the simulation. Generates a dictionary with the following entries:

- **step**: current simulation step
- **systems**: state dict of the system holding current positions, momenta, forces, etc…
- **simulator_hooks**: dict of state dicts of the various hooks used during simulation using their basic class name as keys.

Returns

State dict containing the current step, the system parameters (positions, momenta, etc.) and all simulator_hook state dicts

Return type *dict*

### 1.21.7 Simulation Hooks

**class schnetpack.md.simulation_hooks.SimulationHook**
Basic class for simulator hooks

**class schnetpack.md.simulation_hooks.RemoveCOMMotion** (*every_n_steps=10, remove_rotation=True*)
Periodically remove motions of the center of mass from the system.

Parameters

- **every_n_steps** (*int*) – Frequency with which motions are removed.
• **remove_rotation** *(bool)* – Also remove rotations *(default=False)*.

### Logging

All logging operations in SchNetPack molecular dynamics simulations are performed via simulation hooks. This includes the generation of checkpoint files. The main tool to store simulation data is the `schnetpack.md.simulation_hooks.FileLogger`, which uses data streams to collect information (positions, velocities, properties, ...) during a simulation and store it to specially formatted HDF5 files. These files can then be read using the `schnetpack.md.utils.HDF5Loader`.

```python
class schnetpack.md.simulation_hooks.logging_hooks.Checkpoint (checkpoint_file, every_n_steps=1000)
```

Hook for writing out checkpoint files containing the state_dict of the simulator. Used to restart the simulation from a previous step of previous system configuration.

**Parameters**

- `checkpoint_file` *(str)* – Name of the file used to store the state_dict periodically.
- `every_n_steps` *(int)* – Frequency with which checkpoint files are written.

```python
on_simulation_end (simulator)
```

Store state_dict at the end of the simulation.

**Parameters**

- `simulator` *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

```python
on_step_end (simulator)
```

Store state_dict at specified intervals.

**Parameters**

- `simulator` *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

```python
class schnetpack.md.simulation_hooks.logging_hooks.TemperatureLogger (log_file, every_n_steps=100)
```

TensorBoard logging hook for the temperatures of the replicas, as well as of the corresponding centroids for each molecule in the system container.

**Parameters**

- `log_file` *(str)* – Path to the TensorBoard file.
- `every_n_steps` *(int)* – Frequency with which data is logged to TensorBoard.

```python
on_step_end (simulator)
```

Log the systems temperatures at the given intervals.

**Parameters**

- `simulator` *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

```python
class schnetpack.md.simulation_hooks.logging_hooks.FileLogger (filename, buffer_size, data_streams=[<schnetpack.md.simulation_hooks.logging_hooks.MoleculeStream object>, <schnetpack.md.simulation_hooks.logging_hooks.PropertyStream object>], every_n_steps=1)
```

Class for monitoring the simulation and storing the resulting data to a hdf5 dataset. The properties to monitor are given via instances of the DataStream class. Uses buffers of a given size, which are accumulated and flushed.
to the main file in regular intervals in order to reduce I/O overhead. All arrays are initialized for the full number of requested simulation steps, the current positions in each data group is handled via the ‘entries’ attribute.

**Parameters**

- **filename** *(str)* – Path to the hdf5 database file.
- **buffer_size** *(int)* – Size of the buffer, once full, data is stored to the hdf5 dataset.
- **list** *(data_streams)* – List of DataStreams used to collect and log information to the main hdf5 dataset, default are properties and molecules.
- **every_n_steps** *(int)* – Frequency with which the buffer is updated.

**on_simulation_end** *(simulator)*

Perform one final flush of the buffers and close the file upon the end of the simulation.

**Parameters**

- **simulator** *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

**on_simulation_start** *(simulator)*

Initializes all present data streams (creating groups, determining buffer shapes, storing metadata, etc.). In addition, the ‘entries’ attribute of each data stream is read from the existing data set upon restart.

**Parameters**

- **simulator** *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

**on_step_end** *(simulator)*

Update the buffer of each stream after each specified interval and flush the buffer to the main file if full.

**Parameters**

- **simulator** *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

**class** schnetpack.md.simulation_hooks.logging_hooks.MoleculeStream

DataStream for logging atom types, positions and velocities to the group ‘molecules’ of the main hdf5 dataset. Positions and velocities are stored in a n_steps x n_replicas x n_molecules x 6 array, where n_steps is the number of simulation steps, n_replicas and n_molecules are the number of simulation replicas and different molecules. The first 3 of the final 6 components are the Cartesian positions and the last 3 the velocities in atomic units. Atom types, the numbers of replicas, molecules and atoms, as well as the length of the time step in atomic units (for spectra) are stored in the group attributes.

**update_buffer** *(buffer_position, simulator)*

Routine for updating the buffer.

**Parameters**

- **buffer_position** *(int)* – Current position in the buffer.
- **simulator** *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

**class** schnetpack.md.simulation_hooks.logging_hooks.DataStream*(group_name)*

Basic DataStream class to be used with the FileLogger. Creates data groups in the main hdf5 file, accumulates the associated information and flushes them to the file periodically.

**Parameters**

- **group_name** *(str)* – Name of the data group entry.

**flush_buffer** *(file_position, buffer_position)*

Write data contained in buffer into the main hdf5 file.

**Parameters**

- **file_position** *(int)* – Current position in the main dataset file.
• **buffer_position** (*int*) – Most recent entry in the buffer. Used to ensure no buffer entries are written to the main file.

**init_data_stream**(*simulator, main_dataset, buffer_size, restart=False, every_n_steps=1*)

Wrapper for initializing the data containers based on the instructions provided in the current simulator. For every data stream, the current number of valid entries is stored, which is updated periodically. This is necessary if a simulation is e.g. restarted or data is extracted during a running simulations, as all arrays are initially constructed taking the full length of the simulation into account.

**Parameters**

- **simulator** (*schnetpack.simulation_hooks.Simulator*) – Simulator class used in the molecular dynamics simulation.
- **main_dataset** (*h5py.File*) – Main h5py dataset object.
- **buffer_size** (*int*) – Size of the buffer, once full, data is stored to the hdf5 dataset.
- **restart** (*bool*) – If the simulation is restarted, continue logging in the previously created dataset. (default=False)
- **every_n_steps** (*int*) – How often simulation steps are logged. Used e.g. to determine overall time step in MoleculeStream.

**update_buffer**(*buffer_position, simulator*)

Instructions for updating the buffer. Needs to take into account reformatting of data, etc.

**Parameters**

- **buffer_position** (*int*) – Current position in the buffer.
- **simulator** (*schnetpack.simulation_hooks.Simulator*) – Simulator class used in the molecular dynamics simulation.

**class schnetpack.md.simulation_hooks.logging_hooks.PropertyStream(target_properties=None)**

Main routine for logging the properties predicted by the calculator to the group ‘properties’ of hdf5 dataset. Stores properties in a flattened array and writes names, shapes and positions to the group data section. Since this routine determines property shapes based on the system.properties dictionary, at least one computations needs to be performed beforehand. Properties are stored in an array of the shape n_steps x n_replicas x n_molecules x n_properties, where n_steps is the number of simulation steps, n_replicas and n_molecules is the number of simulation replicas and different molecules and n_properties is the length of the flattened property array.

**Parameters** **target_properties**(*list*) – List of properties to be written to the hdf5 database.

If no list is given, defaults to None, which means all properties are stored.

**update_buffer**(*buffer_position, simulator*)

Routine for updating the property buffer.

**Parameters**

- **buffer_position** (*int*) – Current position in the buffer.
- **simulator** (*schnetpack.simulation_hooks.Simulator*) – Simulator class used in the molecular dynamics simulation.

**class schnetpack.md.simulation_hooks.logging_hooks.SimulationStream**

DataStream for dynamic system properties, such as kinetic energy and temperatures of the individual replicas, as well as centroids. Adds and updates the group ‘simulation’. Stores properties in a flattened array and writes names, shapes and positions to the group data section. Properties are stored in an array of the shape n_steps x n_replicas x n_molecules x n_properties, where n_steps is the number of simulation steps, n_replicas and n_molecules is the number of simulation replicas and different molecules and n_properties is the length of the flattened property array.
**update_buffer** *(buffer_position, simulator)*  
Routine for updating the buffer.

**Parameters**

- **buffer_position** *(int)* – Current position in the buffer.
- **simulator** *(schnetpack.simulation_hooks.Simulator)* – Simulator class used in the molecular dynamics simulation.

**Sampling**

This module contains all routines for influencing the sampling during a molecular dynamics run. In general, these routines are derived from `schnetpack.md.simulation_hooks.SimulationHook` and act on the middle part of each simulator step. Currently, accelerated molecular dynamics and metadynamics are implemented.

```python
class schnetpack.md.simulation_hooks.sampling.BiasPotential
    Placeholder class for bias potentials used for adaptive/accelerated sampling.
	on_step_end(simulator)
    Needs to be implemented. This applies a bias potential after computing the forces and properties.

Parameters

- **simulator** *(schnetpack.md.Simulator)* – Main simulator used for driving the dynamics
```

```python
class schnetpack.md.simulation_hooks.sampling.AcceleratedMD(energy_threshold, acceleration_factor, energy_handle='energy', energy_conversion=1.0)
```

Hook for performing accelerated molecular dynamics\(^8\). This method distorts the potential energy surface in order to make deep valleys smoother. This smoothing is applied to everything below an energy threshold and its strength is regulated via a acceleration factor. Care should be taken on choosing the right conventions for the energy conversion. Currently, it is assumed that everything uses atomic units.

**Parameters**

- **energy_threshold** *(float)* – Energy threshold in units of energy used by the calculator.
- **acceleration_factor** *(float)* – Acceleration factor steering the smoothness of the bias potential
- **energy_handle** *(str, optional)* – Identifier for the potential energies.
- **energy_conversion** *(str, optional)* – Conversion factor for the calculator energies.

**References**

Parameters `simulator` (*schnetpack.md.Simulator*) – Main simulator used for driving the dynamics

```python
class schnetpack.md.simulation_hooks.sampling.MetaDyn(collective_variables,
            frequency=200,
            weight=0.0015936026415557386,
            store_potential=True)
```

Perform a metadynamics simulation, where Gaussian potentials are deposited along collective variables in order to steer the sampling of a molecular dynamics trajectory.\(^9\)

Parameters

- **collective_variables** (*list*) – List of collective variables to be sampled (*schnetpack.md.simulation_hooks.CollectiveVariable*).
- **frequency** (*int, optional*) – Frequency with which Gaussians are deposited (every \(n\) simulation steps).
- **weight** (*float, optional*) – Weight of each Gaussian in units of energy (Hartee).
- **store_potential** (*bool, optional*) – Whether centers and widths of the placed Gaussians should be stored.

References

**on_simulation_start** (*simulator*)

Initialize the tensor holding the Gaussian centers and widths. These will be populated during the simulation.

Parameters `simulator` (*schnetpack.md.Simulator*) – Main simulator used for driving the dynamics

**on_step_middle** (*simulator*)

Based on the current structure, compute the collective variables and the associated Gaussian potentials. If multiple collective variables are given, a product potential is formed. `torch.autograd` is used to compute the forces resulting from the potential, which are then in turn used to update the system forces. A new Gaussian is added after a certain number of steps.

Parameters `simulator` (*schnetpack.md.Simulator*) – Main simulator used for driving the dynamics

```python
class schnetpack.md.simulation_hooks.sampling.CollectiveVariable(width)
```

Basic collective variable to be used in combination with the `MetaDyn` hook. The `_colvar_function` needs to be implemented based on the desired collective variable.

Parameters **width** (*float*) – Parameter regulating the standard deviation of the Gaussians.

**get_colvar** (*structure*)

Compute the collective variable.

Parameters **structure** (*torch.Tensor*) – Atoms positions taken from the system in the `schnetpack.md.Simulator`.

Returns Collective variable computed for the structure.

ReturnType `torch.Tensor`

```python
class schnetpack.md.simulation_hooks.sampling.BondColvar(idx_a, idx_b, width)
```

Collective variable acting on bonds between atoms. `idx_a` indicates the index of the first atom in the structure tensor, `idx_b` the second. Counting starts at zero.

Parameters

- `idx_a` *(int)* – Index of the first atom in the positions tensor provided by the simulator system.
- `idx_b` *(int)* – Index of the second atom.
- `width` *(float)* – Width of the Gaussians applied to this collective variable. For bonds, units of Bohr are used.

Thermostats

This module contains various thermostats for regulating the temperature of the system during molecular dynamics simulations. Apart from standard thermostats for conventional simulations, a series of special thermostats developed for ring polymer molecular dynamics is also provided.

class schnetpack.md.simulation_hooks.thermostats.ThermostatHook(
    temperature_bath,  
    nm_transformation=None,  
    detach=True
)

Basic thermostat hook for simulator class. This class is initialized based on the simulator and system specifications during the first MD step. Thermostats are applied before and after each MD step.

Parameters

- `temperature_bath` *(float)* – Temperature of the heat bath in Kelvin.
- `nm_transformation` *(schnetpack.md.utils.NormalModeTransformer)* – Module used to transform between beads and normal mode representation in ring polymer dynamics.
- `detach` *(bool)* – Whether the computational graph should be detached after each simulation step. Default is true, should be changed if differentiable MD is desired. TODO: Make detach frequency instead

on_simulation_start *(simulator)*

Routine to initialize the thermostat based on the current state of the simulator. Reads the device to be used, as well as the number of molecular replicas present in simulator.system. Furthermore, the normal mode transformer is initialized during ring polymer simulations. In addition, a flag is set so that the thermostat is not reinitialized upon continuation of the MD.

Main function is the _init_thermostat routine, which takes the simulator as input and must be provided for every new thermostat.

Parameters simulator *(schnetpack.simulation_hooks.simulator.Simulator)* – Main simulator class containing information on the time step, system, etc.

on_step_begin *(simulator)*

First application of the thermostat before the first half step of the dynamics. Regulates temperature and applies a mask to the system momenta in order to avoid problems of e.g. thermal noise added to the zero padded tensors. The detach is carried out here.

Main function is the _apply_thermostat routine, which takes the simulator as input and must be provided for every new thermostat.

Parameters simulator *(schnetpack.simulation_hooks.simulator.Simulator)* – Main simulator class containing information on the time step, system, etc.

on_step_end *(simulator)*

First application of the thermostat before the first half step of the dynamics. Regulates temperature and
applies a mask to the system momenta in order to avoid problems of e.g. thermal noise added to the zero padded tensors. The detach is carried out here.

Main function is the _apply_thermostat routine, which takes the simulator as input and must be provided for every new thermostat.

**Parameters**

- **simulator** *(schnetpack.simulation_hooks.simulator.Simulator)* – Main simulator class containing information on the time step, system, etc.

**class schnetpack.md.simulation_hooks.thermostats.BerendsenThermostat**(temperature_bath, time_constant)

Berendsen velocity rescaling thermostat, as described in\(^{10}\). Simple thermostat for e.g. equilibrating the system, does not sample the canonical ensemble.

**Parameters**

- **temperature_bath** *(float)* – Temperature of the external heat bath in Kelvin.
- **time_constant** *(float)* – Thermostat time constant in fs

**References**

**class schnetpack.md.simulation_hooks.thermostats.GLEThermostat**(temperature_bath, gle_file, nm_transformation=None)

Stochastic generalized Langevin colored noise thermostat by Ceriotti et. al. as described in\(^ {11}\). This thermostat requires specially parametrized matrices, which can be obtained online from: http://gle4md.org/index.html?page=matrix

The additional degrees of freedom added to the system are defined via the matrix dimensions. This could in principle be used for ring polymer dynamics by providing a normal mode transformation.

**Parameters**

- **temperature_bath** *(float)* – Temperature of the external heat bath in Kelvin.
- **gle_file** *(str)* – File containing the GLE matrices
- **nm_transformation** *(schnetpack.md.utils.NormalModeTransformer)* – Module use dto transform between beads and normal model representation in ring polymer dynamics.

**References**

**class schnetpack.md.simulation_hooks.thermostats.PIGLETThermostat**(temperature_bath, gle_file, nm_transformation=None)

Efficient generalized Langevin equation stochastic thermostat for ring polymer dynamics simulations, see\(^ {12}\) for a detailed description. In contrast to the standard GLE thermostat, every normal mode of the ring polymer is thermostated seperately.

---

\(^{10}\) Berendsen, Postma, van Gunsteren, DiNola, Haak: Molecular dynamics with coupling to an external bath. The Journal of Chemical Physics, 81 (8), 3684-3690. 1984.


Parameters

- `temperature_bath (float)` – Temperature of the external heat bath in Kelvin.
- `gle_file (str)` – File containing the GLE matrices
- `nm_transformation (schnetpack.md.utils.NormalModeTransformer)` – Module use dto transform between beads and normal model representation in ring polymer dynamics.

References

class schnetpack.md.simulation_hooks.thermostats.LangevinThermostat (temperature_bath, time_constant, nm_transformation=None)

Basic stochastic Langevin thermostat, see e.g.\(^{13}\) for more details.

Parameters

- `temperature_bath (float)` – Temperature of the external heat bath in Kelvin.
- `time_constant (float)` – Thermostat time constant in fs
- `nm_transformation (schnetpack.md.utils.NormalModeTransformer)` – Module use dto transform between beads and normal model representation in ring polymer dynamics.

References

class schnetpack.md.simulation_hooks.thermostats.PILELocalThermostat (temperature_bath, time_constant, nm_transformation=<class 'schnetpack.md.utils.normal_mode_transformation.NormalModeTransformer'>, thermostat_centroid=True, damping=None)

Langevin thermostat for ring polymer molecular dynamics as introduced in\(^{14}\). Applies specially initialized Langevin thermostats to the beads of the ring polymer in normal mode representation.

Parameters

- `temperature_bath (float)` – Temperature of the external heat bath in Kelvin.
- `time_constant (float)` – Thermostat time constant in fs
- `nm_transformation (schnetpack.md.utils.NormalModeTransformer)` – Module use dto transform between beads and normal model representation in ring polymer dynamics.
- `thermostat_centroid (bool)` – Whether a thermostat should be applied to the centroid of the ring polymer in normal mode representation (relevant e.g. for TRPMD, default is True)


\(^{14}\) Ceriotti, Parrinello, Markland, Manolopoulos: Efficient stochastic thermostatting of path integral molecular dynamics. The Journal of Chemical Physics, 133 (12), 124104. 2010.
• **damping** (*float*) – If specified, damping factor is applied to the current momenta of the system (used in TRPMD, default is no damping).

**References**

class schnetpack.md.simulation_hooks.thermostats.PILEGlobalThermostat (*temperature_bath*, *time_constant*, *nm_transformation*=
`<class 'schnetpack.md.utils.normal_mode_transformation.NormalModeTransformer'>`)

Global variant of the ring polymer Langevin thermostat as suggested in\(^{15}\). This thermostat applies a stochastic velocity rescaling thermostat\(^{16}\) to the ring polymer centroid in normal mode representation.

**Parameters**

- **temperature_bath** (*float*) – Temperature of the external heat bath in Kelvin.
- **time_constant** (*float*) – Thermostat time constant in fs
- **nm_transformation** (*schnetpack.md.utils.NormalModeTransformer*) – Module use dto transform between beads and normal model representation in ring polymer dynamics.

**References**

class schnetpack.md.simulation_hooks.thermostats.NHCThermostat (*temperature_bath*, *time_constant*, *chain_length*=3, *massive=False*, *nm_transformation=None*, *multi_step=2*, *integration_order=3*)

Nose-Hover chain thermostat, which links the system to a chain of deterministic Nose-Hoover thermostats first introduced in\(^{17}\) and described in great detail in\(^{18}\). Advantage of the NHC thermostat is, that it does not apply random perturbations to the system and is hence fully deterministic. However, this comes at an increased numerical cost compared to e.g. the stochastic thermostats described above.

**Parameters**

- **temperature_bath** (*float*) – Temperature of the external heat bath in Kelvin.
- **time_constant** (*float*) – Thermostat time constant in fs
- **chain_length** (*int*) – Number of Nose-Hoover thermostats applied in the chain.
- **massive** (*bool*) – If set to true, an individual thermostat is applied to each degree of freedom in the system. Can e.g. be used for thermostatting (default=False).
- **nm_transformation** (*schnetpack.md.utils.NormalModeTransformer*) – Module used to transform between beads and normal model representation in ring polymer dynamics.

\(^{15}\) Ceriotti, Parrinello, Markland, Manolopoulos: Efficient stochastic thermostatting of path integral molecular dynamics. The Journal of Chemical Physics, 133 (12), 124104. 2010.

\(^{16}\) Bussi, Donadio, Parrinello: Canonical sampling through velocity rescaling. The Journal of chemical physics, 126(1), 014101. 2007.


- **multi_step (int)** – Number of steps used for integrating the NH equations of motion (default=2)

- **integration_order (int)** – Order of the Yoshida-Suzuki integrator used for propagating the thermostat (default=3).

References

class schnetpack.md.simulation_hooks.thermostats.NHCRingPolymerThermostat (temperature_bath, time_constant, chain_length=3, local=True, nm_transformation=schnetpack.md.utils.normal_mode_transformation.NormalModeTransformer, multi_step=2, integration_order=3)

Nose-Hoover chain thermostat for ring polymer molecular dynamics simulations as e.g. described in\(^\text{19}\). This is based on the massive setting of the standard NHC thermostat but operates in the normal mode representation and uses specially initialized thermostat masses.

Parameters

- **temperature_bath (float)** – Temperature of the external heat bath in Kelvin.
- **time_constant (float)** – Thermostat time constant in fs
- **chain_length (int)** – Number of Nose-Hoover thermostats applied in the chain.
- **local (bool)** – If set to true, an individual thermostat is applied to each degree of freedom in the system. Can e.g. be used for thermostatting (default=False).
- **nm_transformation (schnetpack.md.utils.NormalModeTransformer)** – Module used to transform between beads and normal model representation in ring polymer dynamics.
- **multi_step (int)** – Number of steps used for integrating the NH equations of motion (default=2)
- **integration_order (int)** – Order of the Yoshida-Suzuki integrator used for propagating the thermostat (default=3).

References

class schnetpack.md.simulation_hooks.thermostats.TRPMDThermostat (temperature_bath, damping, nm_transformation=schnetpack.md.utils.normal_mode_transformation.NormalModeTransformer)

Thermostatted ring polymer molecular dynamics thermostat variant of the local PILE thermostat as introduced

\(^{19}\) Ceriotti, Parrinello, Markland, Manolopoulos: Efficient stochastic thermostatting of path integral molecular dynamics. The Journal of Chemical Physics, 133 (12), 124104. 2010.
in\textsuperscript{20}. Here, no thermostat is applied to the centroid and the dynamics of the system are damped via a given damping factor.

**Parameters**

- `temperature_bath (float)` – Temperature of the external heat bath in Kelvin.
- `damping (float)` – Damping factor of the thermostat.
- `nm_transformation (schnetpack.md.utils.NormalModeTransformer)` – Module used to transform between beads and normal mode representation in ring polymer dynamics.

**References**

### 1.21.8 Utils

This module contains various utility functions and classes used in combination with schnetpack.md. This includes e.g. unit conversion, loading of molecular dynamics data and the computation of spectra.

**class schnetpack.md.utils.MDUnits**

Basic conversion factors to atomic units used internally:
- fs2atu (time): femtoseconds to atomic time units
- eV2Ha (energy): electron Volt to Hartree
- d2amu (mass): Dalton to atomic mass units
- angs2bohr (length): Angstrom to Bohr
- auforces2aseforces (forces): Hartree per Bohr to electron Volt per Angstrom

**Definitions for constants:**
- kB: Boltzmann constant in units of Hartree per Kelvin.
- hbar: Reduced Planck constant in atomic units.

**static parse_mdunit (unit)**

Auxiliary functions, used to determine the conversion factor of position and force units between MD propagation and the provided ML Calculator. Allowed units are:
- mol, kcal, eV, Bohr, Angstrom, Hartree and all combinations using ‘/’ thereof (e.g. kcal/mol/Angstrom).

**Parameters unit (str/float)** – Unit to be used to convert forces from Calculator units to atomic units used in the MD. Can be a str, which is converted to the corresponding numerical value or a float, which is returned.

**Returns** Factor used for conversion in the Calculator.

**Return type** float

**class schnetpack.md.utils.NormalModeTransformer (n_beads, device='cuda')**

Class for transforming between bead and normal mode representation of the ring polymer, used e.g. in propagating the ring polymer during simulation. An in depth description of the transformation can be found e.g. in\textsuperscript{21}. Here, a simple matrix multiplication is used instead of a Fourier transformation, which can be more performant in certain cases. On the GPU however, no significant performance gains were observed when using a FT based transformation over the matrix version.

This transformation operates on the first dimension of the property tensors (e.g. positions, momenta) defined in the system class. Hence, the transformation can be carried out for several molecules at the same time.

**Parameters**

\textsuperscript{20} Rossi, Ceriotti, Manolopoulos: How to remove the spurious resonances from ring polymer molecular dynamics. The Journal of Chemical Physics, 140(23), 234116. 2014.

\textsuperscript{21} Ceriotti, Parrinello, Markland, Manolopoulos: Efficient stochastic thermostatting of path integral molecular dynamics. The Journal of Chemical Physics, 133, 124105. 2010.
- **n_beads (int)** – Number of beads in the ring polymer.
- **device (str)** – Computation device (default='cuda').

**References**

**beads2normal (x_beads)**

Transform a system tensor (e.g. momenta, positions) from the bead representation to normal mode representation.

**Parameters**

- **x_beads (torch.Tensor)** – System tensor in bead representation with the general shape n_beads x n_molecules x ...

**Returns** System tensor in normal mode representation with the same shape as the input tensor.

**Return type** torch.Tensor

**normal2beads (x_normal)**

Transform a system tensor (e.g. momenta, positions) in normal mode representation back to bead representation.

**Parameters**

- **x_normal (torch.Tensor)** – System tensor in normal mode representation with the general shape n_beads x n_molecules x ...

**Returns** System tensor in bead representation with the same shape as the input tensor.

**Return type** torch.Tensor

**HDF5-Loader**

Class for extracting information from the HDF5 files generated during simulation by the schnetpack.md.
simulation_hooks.logging_hooks.FileLogger. In addition to loading structures, velocities, etc., various postprocessing functions are available.

**Class** schnetpack.md.utils.HDF5Loader (hdf5_database, skip_initial=0, load_properties=True)

Class for loading HDF5 datasets written by the FileLogger. By default, this requires at least a MoleculeStream to be present. PropertyData is also read by default, but can be disabled.

**Parameters**

- **hdf5_database (str)** – Path to the database file.
- **skip_initial (int)** – Skip the initial N configurations in the trajectory, e.g. to account for equilibration (default=0).
- **load_properties (bool)** – Extract and reconstruct the property data stored by a PropertyStream (e.g. forces, energies, etc.), enabled by default.

**get_kinetic_energy (mol_idx=0, replica_idx=None)**

Auxiliary routine for computing the kinetic energy of every configuration based on it's velocities.

**Parameters**

- **mol_idx (int)** – Index of the molecule to extract, by default uses the first molecule (mol_idx=0)
- **replica_idx (int)** – Replica of the molecule to extract (e.g. for ring polymer molecular dynamics). If replica_idx is set to None (default), the centroid is returned if multiple replicas are present.

**Returns** N_steps array containing the kinetic energy of every configuration in atomic units.
Return type np.array

get_positions(mol_idx=0, replica_idx=None)
Auxiliary routine for getting the positions of specific molecules and replicas.

Parameters

• mol_idx (int) – Index of the molecule to extract, by default uses the first molecule (mol_idx=0)
• replica_idx (int) – Replica of the molecule to extract (e.g. for ring polymer molecular dynamics). If replica_idx is set to None (default), the centroid is returned if multiple replicas are present.

Returns N_steps x N_atoms x 3 array containing the atom positions of the simulation in atomic units.

Return type np.array

get_property(property_name, mol_idx=0, replica_idx=None, atomistic=False)
Extract property from dataset.

Parameters

• property_name (str) – Name of the property as contained in the self.properties dictionary.
• mol_idx (int) – Index of the molecule to extract, by default uses the first molecule (mol_idx=0)
• replica_idx (int) – Replica of the molecule to extract (e.g. for ring polymer molecular dynamics). If replica_idx is set to None (default), the centroid is returned if multiple replicas are present.
• atomistic (bool) – Whether the property is atomistic (e.g. forces) or defined for the whole molecule (e.g. energies, dipole moments). If set to True, the array is masked according to the number of atoms for the requested molecule to counteract potential zero-padding. (default=False)

Returns N_steps x property dimensions array containing the requested property collected during the simulation.

Return type np.array

get_temperature(mol_idx=0, replica_idx=None)
Auxiliary routine for computing the instantaneous temperature of every configuration.

Parameters

• mol_idx (int) – Index of the molecule to extract, by default uses the first molecule (mol_idx=0)
• replica_idx (int) – Replica of the molecule to extract (e.g. for ring polymer molecular dynamics). If replica_idx is set to None (default), the centroid is returned if multiple replicas are present.

Returns N_steps array containing the temperature of every configuration in Kelvin.

Return type np.array

get_velocities(mol_idx=0, replica_idx=None)
Auxiliary routine for getting the velocities of specific molecules and replicas.
Parameters

- **mol_idx** (*int*) – Index of the molecule to extract, by default uses the first molecule (mol_idx=0)

- **replica_idx** (*int*) – Replica of the molecule to extract (e.g. for ring polymer molecular dynamics). If replica_idx is set to None (default), the centroid is returned if multiple replicas are present.

**Returns**

N_steps x N_atoms x 3 array containing the atom velocities of the simulation in atomic units.

**Return type** np.array

Spectra

Collection of utilities for computing autocorrelation functions and molecular spectra from HDF5 files generated during molecular dynamics. For a good overview on how to compute spectra from molecular dynamics simulations and details on the techniques used, we recommend \(^\text{22}\).

References

```python
class schnetpack.md.utils.VibrationalSpectrum(data, resolution=4096, window=<function cosine_sq_window>)
```

Base class for computing vibrational spectra from HDF5 datasets using autocorrelation functions and fast fourier transforms.

**Parameters**

- **data** (*schnetpack.md.utils.HDF5Loader*) – Loaded dataset.

- **resolution** (*int*) – Resolution used when computing the spectrum. Indicates how many time lags are considered in the autocorrelation function is used.

- **window** (*function, optional*) – Window function used for computing the spectrum.

```python
def compute_spectrum(molecule_idx=0)
```

Main routine for computing spectra. First the relevant data is read, then autocorrelations are computed and processed. Based on the processed autocorrelations, spectra are computed and, if requested, subjected to additional postprocessing.

**Parameters**

- **molecule_idx** (*int*) – Index of the molecule for which the spectrum should be computed. Uses the same conventions as schnetpack.md.System.

```python
def get_spectrum()
```

Returns all computed spectra in the form of a list of tuples of frequencies and intensities.

**Returns**

List of tuples of frequencies and intensities of all computed spectra.

**Return type** list

```python
class schnetpack.md.utils.PowerSpectrum(data, resolution=4096)
```

Compute power spectra from a molecular dynamics HDF5 dataset.

**Parameters**

- **data** (*schnetpack.md.utils.HDF5Loader*) – Loaded dataset.

• **resolution** (*int, optional*) – Resolution used when computing the spectrum. Indicates how many time lags are considered in the autocorrelation function is used.

```python
class schnetpack.md.utils.IRSpectrum(data, resolution=4096, dipole_moment_handle='dipole_moment')
```

Compute infrared spectra from a molecular dynamics HDF5 dataset. This class requires the dipole moments to be present in the HDF5 dataset.

**Parameters**

- **data** (*schnetpack.md.utils.HDF5Loader*) – Loaded dataset.
- **resolution** (*int, optional*) – Resolution used when computing the spectrum. Indicates how many time lags are considered in the autocorrelation function is used.
- **dipole_moment_handle** (*str, optional*) – Identifier used for extracting dipole data.

### 1.21.9 Parsers

This module contains several parsers. This includes utilities for reading and converting molecular dynamics input files to instructions for the `schnetpack.md.simulator.Simulator`. In addition, there is a full package for parsing ORCA output files.

#### ORCA Parsers

Module for extracting information from ORCA output files. Can be used to read arbitrary information from the output files, as well as gradient and hessian files generated by the ORCA code package. Contains several predefined parsers.

```python
class schnetpack.md.parsers.orca_parser.OrcaParserException
    Exception for OrcaParser class.

class schnetpack.md.parsers.orca_parser.OrcaParser(dbpath, properties, filter=None, mask_charges=False)
```

Main parsing utility for ORCA output files. Runs over a list of output files, extracts the data and stores it into a formatted ASE database in SchNetPack format (`schnetpack.data.AtomsData`). This class makes use of the `OrcaMainFileParser` and `OrcaHessianFileParser` defined below.

**Parameters**

- **dbpath** (*str*) – Path to the target database.
- **properties** (*list*) – List of properties to extract from ORCA files.
- **filter** (*dict, optional*) – Dictionary giving the name of a property and a threshold value. Entries in the output files with values exceeding the threshold in magnitude are discarded. This can be used to e.g. screen for numerical noise in implicit solvent computations, etc.
- **mask_charges** (*bool, optional*) – If the ORCA calculation used external charges, these are removed from the positions and atom types read by the parser.

```python
parse_data(data_files, buffer_size=10)
```

Reads in a list of ORCA output files, extracts the data, performs reformattting and then stores structures and properties to an ASE database.

**Parameters**

- **data_files** (*list*) – List of the paths to the ORCA output files.
• **buffer_size** (*int, optional*) – Collects a certain number of molecules before writing them to the database.

class schnetpack.md.parsers.orca_parser.OrcaParser(dbpath, properties, filter=None, mask_charges=False)

Main parsing utility for ORCA output files. Runs over a list of output files, extracts the data and stores it into a formatted ASE database in SchNetPack format (*schnetpack.data.AtomsData*). This class makes use of the OrcaMainFileParser and OrcaHessianFileParser defined below.

**Parameters**

• **dbpath** (*str*) – Path to the target database.

• **properties** (*list*) – List of properties to extract from ORCA files.

• **filter** (*dict, optional*) – Dictionary giving the name of a property and a threshold value. Entries in in the output files with values exceeding the threshold in magnitude are discarded. This can be used to e.g. screen for numerical noise in implicit solvent computations, etc.

• **mask_charges** (*bool, optional*) – If the ORCA calculation used external charges, these are removed from the positions and atom types read by the parser.

def parse_data(data_files, buffer_size=10)

Reads in a list of ORCA output files, extracts the data, performs reformating and then stores structures and properties to an ASE database.

**Parameters**

• **data_files** (*list*) – List of the paths to the ORCA output files.

• **buffer_size** (*int, optional*) – Collects a certain number of molecules before writing them to the database.

class schnetpack.md.parsers.orca_parser.OrcaOutputParser(parsers)

Basic ORCA output parser class. Parses an Orca output file according to the parsers specified in the ‘parsers’ dictionary. Parsed data is stored in an dictionary, using the same keys as the parsers. If a list of formatters is provided to a parser, a list of the parsed entries is stored in the output dictionary.

**Parameters**

**parsers** (*dict[str->callable]*) – dictionary of OrcaPropertyParser, each with their own OrcaFormatter.

def get_parsed()

Auxiliary routine to collect the data from the parser.

**Returns** Dictionary of data entries according to parser keys.

**Return type** dict[str->list]

def parse_file(path)

Open the file and iterate over its lines, applying all parsers. In the end, all data is collected in a dictionary.

**Parameters**

**path** (*str*) – path to Orca output file.

class schnetpack.md.parsers.orca_parser.OrcaPropertyParser(start, stop, formatters=None)

Basic property parser for ORCA output files. Takes a start flag and a stop flag/list of stop flags and collects the data entries in between. If a OrcaFormatter is provided, the data is formatted accordingly upon retrieval. Operates in a line-wise fashion.

**Parameters**

• **start** (*str*) – begins to collect data starting from this string

• **stop** (*str/list(str]*) – stops data collection if any of these strings is encountered
• **formatters** (OrcaFormatter) – OrcaFormatter to convert collected data

`get_parsed()`
Returns data, if formatters are specified in the corresponding format.

**Returns** Formatted data.

**Return type** numpy.array

`parse_line(line)`
 Parses a line in the output file and updates the main container.

**Parameters**
- `line` (str) – line of Orca output file

`reset()`
Reset state of the parser.

```python
class schnetpack.md.parsers.orca_parser.OrcaFormatter(position, stop=None, datatype=‘vector’, converter=<class ‘float’>, skip_first=None, unit=None, default=None)
```
Format raw ORCA data collected by an OrcaPropertyParser. Behavior is determined by the datatype option. This is e.g. used to extract the correct gradient values from the associated text block taken from the ORCA output and convert it to a properly formatted force array.

**Parameters**
- `position` (int) – Position to start formatting. If no stop is provided returns only value at position, otherwise all values between position and stop are returned. (Only used for vector mode)
- `stop` (int, optional) – Stop value for range. (Only used for vector mode)
- `datatype` (str, optional) – Change formatting behavior. The possible options are:
  - `vector`: Formats data between position and stop argument, if provided converting it to the type given in the converter.
  - `matrix`: Formats collected matrix data into the shape of a square, symmetric numpy.ndarray. Ignores other options.
- `converter` (type, optional) – Convert data to type. (Only used for ‘vector’ mode)
- `default` (float, optional) – Default value to be returned if nothing is parsed (e.g. 1.0 for vacuum in case of dielectric constant.
- `skip_first` (int, optional) – If not None, skip the first N lines (default=None).

`format(parsed)`
Format the raw parsed data according to the given instructions.

**Parameters**
- `parsed` (list) – List of raw parsed lines taken from the output file.

**Returns** Formatted numpy array holding the processed properties.

**Return type** numpy.array

```python
class schnetpack.md.parsers.orca_parser.OrcaMainFileParser(properties=None)
```
Predefined OrcaOutputParser for extracting data from the main ORCA output file. Can read and format structure and atom types, as well as the energies, forces, dipole moments, polarizabilities and nuclear shielding tensors.

```python
class schnetpack.md.parsers.orca_parser.OrcaHessianFileParser(properties=None)
```
Predefined OrcaOutputParser for extracting data from the hessian output file written by ORCA if some
higher order derivatives are requested. Can read and format Hessians, as well as Cartesian dipole moment and polarizability derivatives.

### 1.22 schnetpack.utils

#### schnetpack.utils.set_random_seed(seed)
This function sets the random seed (if given) or creates one for torch and numpy random state initialization

**Parameters**

- **seed** *(int, optional)* — if seed not present, it is generated based on time
# Python Module Index

## S

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>schnetpack</td>
<td>49</td>
</tr>
<tr>
<td>schnetpack.atomistic</td>
<td>70</td>
</tr>
<tr>
<td>schnetpack.data</td>
<td>49</td>
</tr>
<tr>
<td>schnetpack.datasets</td>
<td>51</td>
</tr>
<tr>
<td>schnetpack.environment</td>
<td>55</td>
</tr>
<tr>
<td>schnetpack.md</td>
<td>81</td>
</tr>
<tr>
<td>schnetpack.md.calculators</td>
<td>81</td>
</tr>
<tr>
<td>schnetpack.md.initial_conditions</td>
<td>89</td>
</tr>
<tr>
<td>schnetpack.md.integrators</td>
<td>85</td>
</tr>
<tr>
<td>schnetpack.md.neighbor_lists</td>
<td>89</td>
</tr>
<tr>
<td>schnetpack.md.parsers</td>
<td>106</td>
</tr>
<tr>
<td>schnetpack.md.parsers.orca_parser</td>
<td>106</td>
</tr>
<tr>
<td>schnetpack.md.simulation_hooks</td>
<td>91</td>
</tr>
<tr>
<td>schnetpack.md.simulation_hooks.logging_hooks</td>
<td>92</td>
</tr>
<tr>
<td>schnetpack.md.simulation_hooks.sampling</td>
<td>95</td>
</tr>
<tr>
<td>schnetpack.md.simulation_hooks.thermostats</td>
<td>97</td>
</tr>
<tr>
<td>schnetpack.md.simulator</td>
<td>90</td>
</tr>
<tr>
<td>schnetpack.md.system</td>
<td>86</td>
</tr>
<tr>
<td>schnetpack.md.utils</td>
<td>102</td>
</tr>
<tr>
<td>schnetpack.md.utils.hdf5_data</td>
<td>103</td>
</tr>
<tr>
<td>schnetpack.md.utils.spectra</td>
<td>105</td>
</tr>
<tr>
<td>schnetpack.nn</td>
<td>57</td>
</tr>
<tr>
<td>schnetpack.nn.acsf</td>
<td>64</td>
</tr>
<tr>
<td>schnetpack.nn.activations</td>
<td>65</td>
</tr>
<tr>
<td>schnetpack.nn.base</td>
<td>57</td>
</tr>
<tr>
<td>schnetpack.nn.blocks</td>
<td>59</td>
</tr>
<tr>
<td>schnetpack.nn.cfconv</td>
<td>60</td>
</tr>
<tr>
<td>schnetpack.nn.cutoff</td>
<td>61</td>
</tr>
<tr>
<td>schnetpack.nn.neighbors</td>
<td>62</td>
</tr>
<tr>
<td>schnetpack.representation</td>
<td>66</td>
</tr>
<tr>
<td>schnetpack.representation.hdnn</td>
<td>68</td>
</tr>
<tr>
<td>schnetpack.train</td>
<td>74</td>
</tr>
<tr>
<td>schnetpack.train.hooks</td>
<td>75</td>
</tr>
<tr>
<td>schnetpack.train.trainer</td>
<td>74</td>
</tr>
</tbody>
</table>

schnetpack.utils, 109
Index

A
AcceleratedMD (class in schnetpack.md.simulation_hooks.sampling), 95
add_batch() (schnetpack.train.AngleMAE method), 81
add_batch() (schnetpack.train.AngleMSE method), 80
add_batch() (schnetpack.train.AngleRMSE method), 81
add_batch() (schnetpack.train.HeatmapMAE method), 79
add_batch() (schnetpack.train.MeanAbsoluteError method), 79
add_batch() (schnetpack.train.MeanSquaredError method), 78
add_batch() (schnetpack.train.ModelBias method), 78
add_system() (schnetpack.data.AtomsData method), 49
add_systems() (schnetpack.data.AtomsData method), 49
Aggregate (class in schnetpack.md), 58
aggregate() (schnetpack.train.HeatmapMAE method), 79
aggregate() (schnetpack.train.MeanAbsoluteError method), 79
aggregate() (schnetpack.train.MeanSquaredError method), 78
aggregate() (schnetpack.train.ModelBias method), 77
aggregate() (schnetpack.train.RootMeanSquaredError method), 79
AngleMAE (class in schnetpack.train), 80
AngleMSE (class in schnetpack.train), 80
AngleRMSE (class in schnetpack.train), 81
AngularDistribution (class in schnetpack.md), 64

ANI1 (class in schnetpack.datasets), 53
AseEnvironmentProvider (class in schnetpack.environment), 56
atom_distances() (module schnetpack.md), 62
AtomDistances (class in schnetpack.md), 49
AtomisticModel (class in schnetpack.atomistic), 70
AtomsData (class in schnetpack.data), 49
AtomsLoader (class in schnetpack.data), 50
Atomwise (class in schnetpack.atomistic), 70

B
BaseEnvironmentProvider (class in schnetpack.environment), 55
beads2normal() (schnetpack.md.utils.NormalModeTransformer method), 103
BehlerAngular (class in schnetpack.md), 64
BerendsenThermostat (class in schnetpack.md.simulation_hooks.thermostats), 98
BiasPotential (class in schnetpack.md.simulation_hooks.sampling), 95
BondColvar (class in schnetpack.md.simulation_hooks.sampling), 96

calculate() (schnetpack.md.calculators.MDCalculator method), 82
calculate() (schnetpack.md.calculators.QMCalculator method), 83
calculate() (schnetpack.md.calculators.SchnetPackCalculator method), 83

calculate() (schnetpack.md.calculators.SGDMLCalculator method), 85
center_of_mass (schnetpack.md.System attribute), 87
<table>
<thead>
<tr>
<th>Class/Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>centroid_kinetic_energy</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>centroid_momenta</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>centroid_positions</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>centroid_temperature</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>centroid_velocities</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>centroid_velocities</td>
<td>schnetpack.md.System attribute</td>
<td>87</td>
</tr>
<tr>
<td>CFConv</td>
<td>class in schnetpack.nn</td>
<td>60</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>class in schnetpack.md.simulation_hooks.logging_hooks</td>
<td>92</td>
</tr>
<tr>
<td>collect_atom_triples()</td>
<td>(in module schnetpack.environment)</td>
<td>56</td>
</tr>
<tr>
<td>CollectiveVariable</td>
<td>class in schnetpack.md.simulation_hooks.sampling</td>
<td>96</td>
</tr>
<tr>
<td>compute_spectrum()</td>
<td>schnetpack.md.utils.VibrationalSpectrum method</td>
<td>105</td>
</tr>
<tr>
<td>CosineCutoff</td>
<td>class in schnetpack.nn</td>
<td>61</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.data.AtomsData method</td>
<td>49</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.ANI1 method</td>
<td>54</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.ISO17 method</td>
<td>53</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.MaterialsProject method</td>
<td>54</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.MD17 method</td>
<td>53</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.OrganicMaterialsDatabase method</td>
<td>55</td>
</tr>
<tr>
<td>create_subset()</td>
<td>schnetpack.datasets.QM9 method</td>
<td>52</td>
</tr>
<tr>
<td>CSVHook</td>
<td>class in schnetpack.train</td>
<td>75</td>
</tr>
<tr>
<td>DataStream</td>
<td>class in schnetpack.md.simulation_hooks.logging_hooks</td>
<td>93</td>
</tr>
<tr>
<td>Dense</td>
<td>class in schnetpack.nn</td>
<td>57</td>
</tr>
<tr>
<td>DipoleMoment</td>
<td>class in schnetpack.atomistic</td>
<td>72</td>
</tr>
<tr>
<td>ExponentialDecayHook</td>
<td>class in schnetpack.train</td>
<td>77</td>
</tr>
<tr>
<td>FileLogger</td>
<td>class in schnetpack.md.simulation_hooks.logging_hooks</td>
<td>92</td>
</tr>
<tr>
<td>flush_buffer()</td>
<td>class in schnetpack.md.simulation_hooks.logging_hooks.DataStream method</td>
<td>93</td>
</tr>
<tr>
<td>format()</td>
<td>schnetpack.parsers.orca_parser.OrcaFormatter method</td>
<td>108</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.atomistic.AtomisticModel method</td>
<td>70</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.atomistic.Atomwise method</td>
<td>71</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.atomistic.DipoleMoment method</td>
<td>72</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.atomistic.Polarizability method</td>
<td>74</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.Aggregate method</td>
<td>58</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.AngularDistribution method</td>
<td>64</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.AtomDistances method</td>
<td>62</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.BehlerAngular method</td>
<td>64</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.CFConv method</td>
<td>61</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.CosineCutoff method</td>
<td>61</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.Dense method</td>
<td>57</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.ElementalGate method</td>
<td>60</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.GatedNetwork method</td>
<td>60</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.GaussianSmearing method</td>
<td>65</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.GetItem method</td>
<td>57</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.HardCutoff method</td>
<td>62</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.MLP method</td>
<td>59</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.MollifierCutoff method</td>
<td>62</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.NeighborElements method</td>
<td>63</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.RadialDistribution method</td>
<td>65</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.ScaleShift method</td>
<td>58</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.Standardize method</td>
<td>58</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.TiledMultiLayerNN method</td>
<td>59</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.nn.TriplesDistances method</td>
<td>63</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.representation.SchNet method</td>
<td>67</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.representation.SchNetInteraction method</td>
<td>67</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.representation.StandardizeSF method</td>
<td>69</td>
</tr>
<tr>
<td>forward()</td>
<td>schnetpack.representation.SymmetryFunctions method</td>
<td>69</td>
</tr>
</tbody>
</table>
G

GatedNetwork (class in schnetpack.nn), 60
GaussianSmearing (class in schnetpack.nn), 64
get_atomref () (schnetpack.data.AtomsData method), 50
get_atoms () (schnetpack.data.AtomsData method), 50
get_available_properties () (schnetpack.data.AtomsData method), 50
get_colvar () (schnetpack.md.simulation_hooks.sampling.CollectiveVariable method), 96
get_environment () (schnetpack.environment.AseEnvironmentProvider method), 56
get_environment () (schnetpack.environment.BaseEnvironmentProvider method), 55
get_environment () (schnetpack.environment.SimpleEnvironmentProvider method), 56
get_environment () (schnetpack.environment.TorchEnvironmentProvider method), 56
get_kinetic_energy () (schnetpack.md.utils.HDF5Loader method), 103
get_metadata () (schnetpack.data.AtomsData method), 50
get_neighbors () (schnetpack.md.MDNeighborList method), 89
get_parsed () (schnetpack.md.parsers.orca_parser.OrcaOutputParser method), 107
get_parsed () (schnetpack.md.parsers.orca_parser.OrcaPropertyParser method), 108
get_positions () (schnetpack.md.utils.HDF5Loader method), 104
get_properties () (schnetpack.data.AtomsData method), 50
get_property () (schnetpack.md.utils.HDF5Loader method), 104
get_spectrum () (schnetpack.md.utils.VibrationalSpectrum method), 105
get_statistics () (schnetpack.data.AtomsLoader method), 51
get_temperature () (schnetpack.md.utils.HDF5Loader method), 104
get_velocities () (schnetpack.md.utils.HDF5Loader method), 104
GetItem (class in schnetpack.nn), 57
GLEThermostat (class in schnetpack.md.simulation_hooks.thermostats), 98

H

HardCutoff (class in schnetpack.nn), 62
HDF5Loader (class in schnetpack.md.utils), 103
HeatmapMAE (class in schnetpack.train), 79
init_data_stream () (schnetpack.md.simulation_hooks.logging_hooks.DataStream method), 94
initialize_system () (schnetpack.md.Initializer method), 89
Initializer (class in schnetpack.md), 89
initz () (schnetpack.representation.SymmetryFunctions method), 69
IRSSpectrum (class in schnetpack.md.utils), 106
ISO17 (class in schnetpack.datasets), 53

K

kinetic_energy (schnetpack.md.System attribute), 88

L

LangevinThermostat (class in schnetpack.md.simulation_hooks.thermostats), 99
LengthMAE (class in schnetpack.train), 80
LengthMSE (class in schnetpack.train), 80
LengthRMSE (class in schnetpack.train), 80
load_molecules () (schnetpack.md.System method), 88
load_molecules_from_xyz () (schnetpack.md.System method), 88
load_system_state () (schnetpack.md.simulator.Simulator method), 91
LoggingHook (class in schnetpack.train), 75
LRScheduleHook (class in schnetpack.train), 76

M

MaterialsProject (class in schnetpack.datasets), 54
MaxEpochHook (class in schnetpack.train), 76
MaxStepHook (class in schnetpack.train), 76
MaxwellBoltzmannInit (class in schnetpack.md), 89
MD17 (class in schnetpack.datasets), 52
MDCalculator (class in schnetpack.md.calculators), 81
MDNeighborList (class in schnetpack.md), 89
MDUnits (class in schnetpack.md.utils), 102
MeanAbsoluteError (class in schnetpack.train), 79
MeanSquaredError (class in schnetpack.train), 78

Index

115
MetaDyn (class in schnetpack.md.simulation_hooks.sampling), 96
Metric (class in schnetpack.train), 77
MLP (class in schnetpack.nn), 59
ModelBias (class in schnetpack.train), 77
MoleculeStream (class in schnetpack.md.simulation_hooks.logging_hooks), 93
MollifierCutoff (class in schnetpack.nn), 61

N
NeighborElements (class in schnetpack.nn), 63
NHCRingPolymerThermostat (class in schnetpack.md.simulation_hooks.thermostats), 101
NHCThermostat (class in schnetpack.md.simulation_hooks.thermostats), 100

O
normal2beads () (schnetpack.md.utils.NormalModeTransformer method), 103
NormalModeTransformer (class in schnetpack.md.utils), 102

P
parse_data () (schnetpack.md.parsers.orca_parser.OrcaParser method), 108
parse_file () (schnetpack.md.parsers.orca_parser.OrcaOutputParser method), 106
parse_line () (schnetpack.md.parsers.orca_parser.OrcaPropertyParser method), 106
parse_mdunit () (schnetpack.md.utils.MDUnits static method), 102
PIGLETThermostat (class in schnetpack.md.simulation_hooks.thermostats), 98

Index
FILEGlobalThermostat (class in schnetpack.md.simulation_hooks.thermostats), 100
FILELocalThermostat (class in schnetpack.md.simulation_hooks.thermostats), 99
Polarizability (class in schnetpack.atomistic), 73
PowerSpectrum (class in schnetpack.md.utils), 105
Properties (class in schnetpack), 49
PropertyStream (class in schnetpack.md.simulation_hooks.logging_hooks), 94

Q
QM9 (class in schnetpack.datasets), 52
QMCalculator (class in schnetpack.md.calculators), 82

R
RadialDistribution (class in schnetpack.nn), 65
ReduceLROnPlateauHook (class in schnetpack.train), 76
remove_com() (schnetpack.md.System method), 88
remove_com_rotation() (schnetpack.md.System method), 88
remove_com_translation() (schnetpack.md.System method), 88
RemoveCOMMotion (class in schnetpack.md.simulation_hooks), 91
reset() (schnetpack.md.parsers.orca_parser.OrcaPropertyParser method), 108
reset() (schnetpack.train.MeanAbsoluteError method), 79
reset() (schnetpack.train.MeanSquaredError method), 78
reset() (schnetpack.train.Metric method), 77
reset() (schnetpack.train.ModelBias method), 78
reset_parameters() (schnetpack.nn.Dense method), 57
restart_simulation() (schnetpack.md.simulator.Simulator method), 91
RingPolymer (class in schnetpack.md), 86
RootMeanSquaredError (class in schnetpack.train), 78

S
ScaleShift (class in schnetpack.nn), 57
SchNet (class in schnetpack.representation), 66
SchNetInteraction (class in schnetpack.representation), 67
schnetpack (module), 49
schnetpack.atomistic (module), 70
schnetpack.data (module), 49
schnetpack.datasets (module), 51
schnetpack.environment (module), 55
schnetpack.md (module), 81
schnetpack.md.calculators (module), 81
schnetpack.md.initial_conditions (module), 89
schnetpack.md.integrators (module), 85
schnetpack.md.neighbor_lists (module), 89
schnetpack.md.parsers (module), 106
schnetpack.md.parsers.orca_parser (module), 106
schnetpack.md.simulation_hooks (module), 91
schnetpack.md.simulation_hooks.logging_hooks (module), 92
schnetpack.md.simulation_hooks.sampling (module), 95
schnetpack.md.simulation_hooks.thermostats (module), 97
schnetpack.md.simulator (module), 90
schnetpack.md.system (module), 86
schnetpack.md.utils (module), 102
schnetpack.md.utils.hdf5_data (module), 103
schnetpack.md.utils.spectra (module), 105
schnetpack.nn (module), 57
schnetpack.nn.acsf (module), 64
schnetpack.nn.activations (module), 65
schnetpack.nn.base (module), 57
schnetpack.nn.blocks (module), 59
schnetpack.nn.cfconv (module), 60
schnetpack.nn.cutoff (module), 61
schnetpack.nn.neighbors (module), 62
schnetpack.representation (module), 66
schnetpack.representation.schnet (module), 68
schnetpack.train (module), 74
schnetpack.train.hooks (module), 75
schnetpack.train.trainer (module), 74
schnetpack.train.utils (module), 109
SchnetPackCalculator (class in schnetpack.md.calculators), 83
set_metadata() (schnetpack.data.AtomsData method), 50
set_random_seed() (in module schnetpack.utils), 109
SGDMLCalculator (class in schnetpack.md.calculators), 85
shifted_softplus() (in module schnetpack.nn), 65
SimpleEnvironmentProvider (class in schnetpack.environment), 56
SimpleNeighborList (class in schnetpack.md), 90
simulate() (schnetpack.md.simulator.Simulator method), 91
SimulationHook (class in schnetpack.md.simulation_hooks), 91
SimulationStream (class in schnetpack.md.simulation_hooks.logging_hooks), 94
Simulator (class in schnetpack.md.simulator), 90
Standardize (class in schnetpack.nn), 58
StandardizeSF (class in schnetpack.representation), 69
state_dict (schnetpack.md.simulator.Simulator attribute), 91
state_dict (schnetpack.md.System attribute), 88
SumMAE (class in schnetpack.train), 79
SymmetryFunctions (class in schnetpack.representation), 68
System (class in schnetpack.md), 86

temperature (schnetpack.md.System attribute), 88
TemperatureLogger (class in schnetpack.md.simulation_hooks.logging_hooks), 92
TensorboardHook (class in schnetpack.train), 75
ThermostatHook (class in schnetpack.md.simulation_hooks.thermostats), 97
TiledMultiLayerNN (class in schnetpack.nn), 59
TorchEnvironmentProvider (class in schnetpack.environment), 56
train() (schnetpack.train.Trainer method), 74
Trainer (class in schnetpack.train), 74
TriplesDistances (class in schnetpack.nn), 63
TRPMDThermostat (class in schnetpack.md.simulation_hooks.thermostats), 101

update_buffer() (schnetpack.md.simulation_hooks.logging_hooks.DataStream method), 94
update_buffer() (schnetpack.md.simulation_hooks.logging_hooks.MoleculeStream method), 93
update_buffer() (schnetpack.md.simulation_hooks.logging_hooks.PropertyStream method), 94
update_buffer() (schnetpack.md.simulation_hooks.logging_hooks.SimulationStream method), 94
update_neighbors() (schnetpack.md.MDNeighborList method), 90

velocities (schnetpack.md.System attribute), 89
VelocityVerlet (class in schnetpack.md), 86
VibrationalSpectrum (class in schnetpack.md.utils), 105