mlp-train tutorial

Authors: Fernanda Duarte (fernanda.duartegonzalez@chem.ox.ac.uk)
Veronika Juraskova (veronika.juraskova@chem.ox.ac.uk)
Martin Flerin (martin.flerinmaver@wadham.ox.ac.uk)
Shichen Dong (shichen.dong@wolfson.ox.ac.uk)
Hanwen Zhang (hanwen.zhang1018@gmail.com)

Welcome! In this set of exercises, you will gain hands-on experience in training machine learning interatomic potentials (MLIP) using the Python package `mlp-train`. This code is currently being developed by the Duarte group at the University of Oxford for generating datasets and training MLIP for organic chemical reactions. `mlp-train` supports training of Gaussian Approximation Potential (GAP), linear Atomic Cluster Expansion (ACE) and message-passing neural network-based multi-ACE (MACE) potentials, using reference data computed in Gaussian, Orca and xtb.

1 Installing mlp-train

Installation of `mlp-train` package requires conda or mamba. If you do not have it already installed, you can download it from `https://www.anaconda.com/docs/getting-started/miniconda/install#macos-linux-installation`.

As a first step, clone the `mlp-train` package from `https://github.com/duartegroup/mlp-train`.

```
git clone https://github.com/duartegroup/mlp-train.git
```

In this tutorial, we will use MACE MLIP. We, therefore, need to install only the requirements for the `mace-torch` package. It uses GPU acceleration; the required packages thus need support for CUDA. You can use the following commands:

```
cd mlp-train
CONDA_OVERRIDE_CUDA="11.8" ./install_mace.sh
```

The packages are installed into a new conda environment called mlptrain-mace. To activate it, type:

```
conda activate mlptrain-mace
```

You can now check that the packages are installed with support for CUDA as:

```
conda list | grep pytorch
```

If everything works correctly, you should see something similar to

```
pytorch  2.4.1 cuda118_py39ha48351b_305 conda-forge
```

If the third column does not contain the word **cuda**, you need to install the environment again.

In this tutorial, we will introduce several typical examples for the use of `mlp-train` — namely, training of the MLIP for chemical reactions in the gas phase and explicit solvent. Information about settings and additional examples can be found at `mlp-train` documentation: `https://mlp-train.readthedocs.io/en/latest/index.html`.

2 Active learning in mlp-train

The process of generating data sets and training the MLIP relies on the use of active learning (AL). AL is an algorithm that proactively selects suitable data points from a pool of data , with the goal of improving the performance of a model using a small number of new training points. The version of AL implemented in `mlp-train` is depicted in Fig. 1. During AL, the whole data set for MLIP is generated iteratively, starting from a small number of initial structures (often generated by applying random displacements to the initial structure(s)) labelled with reference energies and forces, for example, at the DFT level of theory. This information is used to train the initial version of the MLP. Subsequently, one structure from this initial training set is chosen as the starting point to propagate the molecular dynamics (MLP-MD) using the first trained version of the MLP.

Several rounds of short MLP-MD simulations are then performed to assess the quality of the potential and generate new training structures. The simulation time is set to $(n^3 + 2)$ fs, where n is the index of the MD run, starting from 0. From each MD trajectory, the last frame is evaluated using a defined selector, which determines whether to add or not this structure to the training set. If the structure is not selected, n is increased, and MLP-MD runs are repeated until either the maximum simulation time or the maximum number of AL iterations is reached. A detailed description of the algorithm can be found in the Refs. 1,2,3.

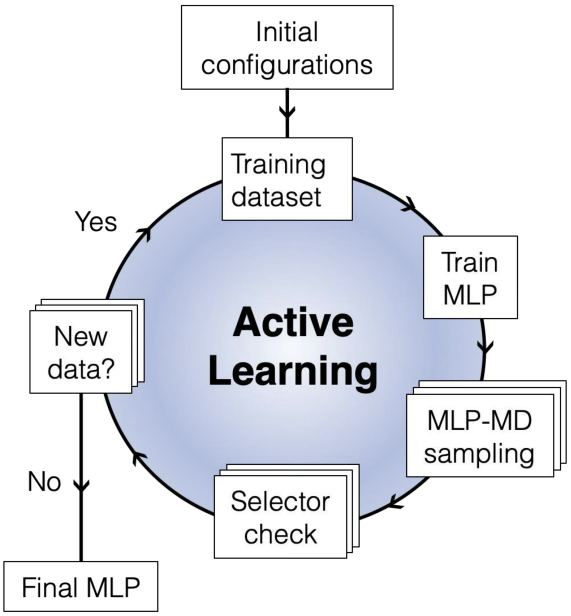


Figure 1: Overview of the active learning loop implemented in `mlp-train`

`mlp-train` package supports three different selectors, depicted in Fig. 2

The simplest and most naive choice of selector is energy. The energy selector directly compares the predicted energy with the reference energy on the data points, and identifies structures where the error in predicted energy exceeds the threshold E_T, satisfying the condition $|E_{DFT} - E_{MLP}| > E_T$. Where E_T is defined by the user. Although reliable, this approach requires reference QM calculations for each selection step, making the AL prohibitively expensive beyond small molecules in the gas phase.

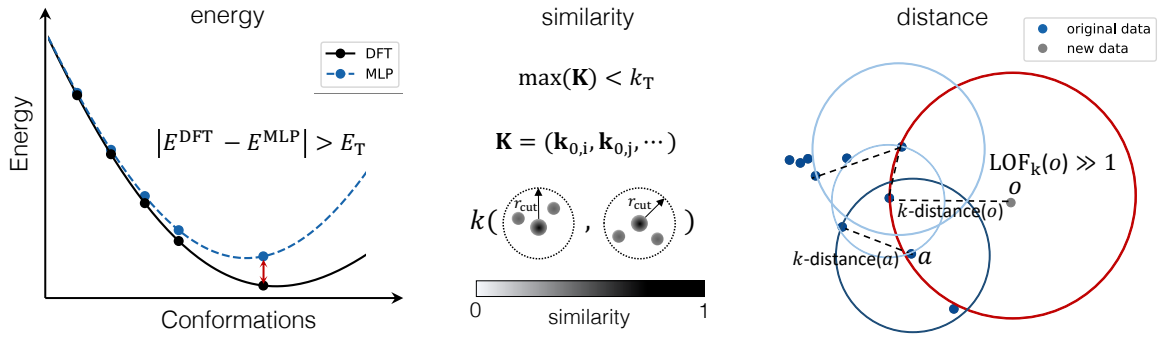


Figure 2: Strategies for structure selection implemented in `mlp-train`: Left: Energy selector based on comparing reference and predicted energy of the newly generated structure. Middle: SOAP similarity selector comparing the similarity in SOAP space between the training set and the new data point. Right: Distance selector, based on the use of the local outlier factor (LOF) method to identify outliers from the original data set in SOAP space.

An alternative approach involves the use of similarity- and distance-based metrics as selectors, referred to as *similarity* and *distance* selectors, respectively. Instead of relying on QM computations, these selectors determine

whether a new data point is added based on its value in the descriptor space. Here, we use Smooth Overlap of Atomic Positions (SOAP) as the molecular descriptor.[4] The *similarity* selector quantifies how closely a new data point p resembles existing configurations $p^{'}$. This is done using the kernel function $k(p \cdot p^{'})$. The similarity vector of the data point is defined as:

$$\mathbf{K} = (|k(\mathbf{p}_0 \cdot \mathbf{p}_\mathrm{i})|^\zeta, |k(\mathbf{p}_0 \cdot \mathbf{p}_\mathrm{j})|^\zeta, \cdots)^T \tag{1}$$

where $\mathbf{p}_0$ is the SOAP vector of the new structure, $\mathbf{p}_\mathrm{i}$ is the SOAP vector of the i-th configuration in the existing set, and ζ is a positive integer that increases the sensitivity of kernel to changes in atomic position. The kernel is computed between the new configuration and all other configurations in the training dataset. The selector adds structure to the training set if the maximum value of its similarity vector, $\mathbf{K}$, is smaller than the threshold k_T, i.e., $\max(\mathbf{K}) < k_\mathrm{T}$. Selecting an appropriate threshold is key, as too low values (e.g., similarity below 0.9) can result in the selection of non-physical structures that fail to converge in the self-consistent field (SCF) computations, while too high values (e.g., too close to 1) do not provide any additional information.

For the *distance* selector, we use the local outlier factor (LOF) method to determine whether the SOAP vector of the new configuration is an outlier compared to the SOAP vectors of the existing training data. LOF is based on the local density of each point, which is calculated by measuring the Euclidean distance between the target point and its k-nearest neighbours (Fig. 2). The local reachability density of an object o, $\mathrm{lrd}_k(o)$, is calculated as:

$$\mathrm{lrd}_\mathrm{k}(o) = \left(\frac{\sum_{i \in N_\mathrm{k}(o)} \mathrm{rd}_\mathrm{k}(o,\, i)}{|N_\mathrm{k}(o)|} \right)^{-1}. \tag{2}$$

$\mathrm{rd}_k(o, a)$ in the equation corresponds to reachability distance defined as

$$\mathrm{rd}_\mathrm{k}(o, a) = \max(k\text{-distance}(a),\ d(o,\, a)). \tag{3}$$

Here, k-distance(a) represents the radius of the smallest circle with its origin in a, which includes the k-nearest neighbours of a (illustrated in dark blue in the Fig.2 right panel), $d(o,\, a)$ is the Euclidean distance between points o and a, where point o is the target point and point a is one of its neighbours. $N_\mathrm{k}(o)$ is a set of k-nearest neighbours of o, which are illustrated by blue points. The local reachability density is thus expressed as the number of neighbours per distance unit. If the local reachability density of the target point is smaller than that of its neighbours, the point is considered an outlier and added to the training dataset. The comparison among the local densities is achieved by computing the ratio of the average local density of neighbours and the local density of the point, as follows:

$$\mathrm{LOF}_\mathrm{k}(o) = \frac{\sum_{i \in N_\mathrm{k}(o)} \frac{\mathrm{lrd}_\mathrm{k}(i)}{\mathrm{lrd}_\mathrm{k}(o)}}{|N_\mathrm{k}(o)|} \tag{4}$$

A LOF value close to 1 indicates that a point is located in a similarly dense region as its neighbours. A LOF value less than 1 represents an inlier, meaning that the point is in a denser region, while a value greater than 1 indicates an outlier. Since there is no definitive rule for selecting an LOF threshold to identify outliers, in this study, we chose the threshold as an LOF value which is larger than 80 % of LOF values for the given training set. This approach ensures that the threshold value varies with the PES exploration during the AL iterations.

3 MACE potential

MACE is a message-passing graph neural network.[5,6] In this formalism, the atoms in a structure are represented through a graph, where the nodes of the graph correspond to different atoms and edges represent the interactions. MACE has different hyper-parameters that control the degree of accuracy and expressivity of the model. Setting these parameters is a trade-off between accuracy and computational cost. In the `mlp-train` package, the default settings for training of the MACE MLIP are defined in `config.py` file. Most of these settings follow the default values recommended in the `mace-torch` package, see `https://github.com/ACEsuit/mace`. However, some parameters are system-dependent and might need to be adjusted for your system.

Some of the important parameters in MACE training are:

- **valid_fraction:** Fraction of data used for training and for validation. 0.1 means that 90 % of structures are used for training and 10 % is used to evaluate the accuracy of the potential.
- **max_num_epochs** number of passes through the data. An epoch is completed when the entire training data has been used once in updating the weights batch by batch.
- **batch_size**: number of configurations evaluated in one batch Number of configs used to compute the gradients for each full update of the network parameters. This training strategy is called stochastic gradient descent because only a subset of the data (batch_size) is used to change the parameters at each update.
- **r_max: The cutoff radius** The distance cutoff is used to create the local ACE environment in each message-passing layer. r_max=5.0 means atoms separated by a distance of more than 5.0 Å do not directly communicate in a single layer. Atoms further than 5.0 Å can still communicate through later messages if intermediate proxy atoms exist. The effective receptive field of the model is num_interactions * r_max. The larger the r_max, the slower the model will be. It is recommended to use values between 4.0 Å and 7.0 Å.
- **correlation: The order of the many-body expansion in ACE** The body order that MACE induces at each layer. Choosing correlation=3 will create basis functions of up to 4-body interactions. If the model has multiple layers, the effective correlation order is higher. For example, a two-layer MACE with correlation=3 has an effective body order of 13.
- **swa: protocol for loss weights** During the training, you will notice that energy errors are at first much higher than force errors because the higher wight of force is assigned in loss function. MACE implements a special protocol that increases the weight on the energy in the loss function (–swa_energy_weight) once the forces are sufficiently accurate. The starting epoch for this special protocol can be controlled by changing –start_swa.
- **max_L: Symmetry of the messages** Determines the symmetry of the messages in message-passing. A value of 0 means MACE will pass only invariant information between neighbourhoods. 1 is the default value; it is a good compromise between speed and accuracy. It is recommended to start with that value for a new project. 2 are the most accurate but slow models.

The complete list of parameters changeable in `mlp-train` is listed below:

```
mace_params = {
        'valid_fraction': 0.1,
        'max_num_epochs': 1200,
        'config_type_weights': '{"Default":1.0}',
        'model': 'MACE',
        'loss': 'weighted',
        'energy_weight': 1.0,
        'forces_weight': 5.0,
        'hidden_irreps': '128x0e + 128x1o',
        'batch_size': 10,
        'r_max': 5.0,
        'correlation': 3,
        'device': mace_device,
        'calc_device': 'cpu',
        'error_table': 'TotalMAE',
        'swa': True,
        'start_swa': 800,
        'ema': True,
        'ema_decay': 0.99,
        'lr': 0.001,
        'patience': 50,
        'scheduler_patience': 20,
        'seed': 345,
        'amsgrad': True,
        'restart_latest': False,
        'save_cpu': True,
        'num_workers': 20,
        'max_L': 1,
```

```
        'dtype': 'float32',
    }
```

To learn more details about MACE, you can follow the MACE tutorials at `https://mace-docs.readthedocs.io/en/latest/examples/tutorials.html`.

4 First example: $[Mg(H_2O)_6]^{2+}$ in gas phase

To test different functionalities of `mlp-train`, we will start with modelling a simple complex of $[Mg(H_2O)_6]^{2+}$ in the gas phase at the GFN2-xTB level of theory.

First, we need an optimised geometry of the $[Mg(H_2O)_6]^{2+}$ complex. The following geometry was optimised at TPSSh-D3BJ/def2-TZVP level of theory.

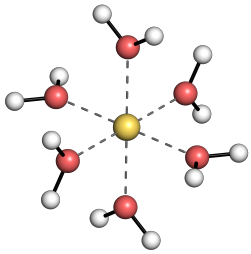


Figure 3: $[Mg(H_2O)_6]^{2+}$ complex

Save the following xyz coordinate into TPSS.xyz file.

```
19

 Mg  0.05323511047051      0.00007794746181     -0.01999176415624
 O  -0.06913864546078      0.00829831435364      2.06353749240888
 H  -0.10791898662379     -0.76569429487583      2.64393909783524
 H  -0.12894733088667      0.78787208216104      2.63464046678206
 O  -2.04009671044481     -0.01999489167871     -0.15412247023908
 H  -2.67360088275904     -0.02861207757230      0.57787041068508
 H  -2.55959759436299     -0.03493870093067     -0.97109840639087
 O   0.17556106448179     -0.00805090939564     -2.10364149570564
 H   0.23525162338492     -0.78744951838374     -2.67499294569908
 H   0.21449023332078      0.76612227735989     -2.68379146704222
 O   2.14658637019409      0.01980124640880      0.11407546538765
 H   2.66605370099103      0.03492169837838      0.93106951431962
 H   2.78012039948876      0.02825492817216     -0.61789296943943
 O   0.07858612844601     -2.09114117892769     -0.01186112862369
 H  -0.68670053895330     -2.68195481616265     -0.06337328664393
 H   0.86271475004895     -2.65605812675173      0.04751042983736
 O   0.02813548729391      2.09125972766537     -0.02827727589276
 H   0.79345932085440      2.68203002745055      0.02319542701950
 H  -0.75595649948378      2.65623326526731     -0.08763609444244
```

One of the advantages of the `mlp-train` package is the data efficiency of the training. In principle, the AL cycle discussed earlier today can start from a small number of structures. We will start the training from one optimised structure of $[Mg(H_2O)_6]^{2+}$, which will be randomly displaced to create an initial dataset of 10 structures.

In the AL part of the code, the structures can be selected by several strategies. First, let's have a look at the *energy* selection. The simple example code to train the MACE potential can look like this.

```
import mlptrain as mlt
import numpy  as np

mlt.Config.mace_params['calc_device'] = 'cuda'

## Beginning of the code
```

```
if __name__ == '__main__':

    system = mlt.System(mlt.Molecule('TPSSH.xyz',charge=2, mult=1),box = None)

    mg = mlt.potentials.MACE('mg_aqua', system = system)

    mg.al_train(method_name = 'xtb', temp=400,
                max_active_iters = 10,
                max_active_time = 3000)

    trajectory_gas = mlt.md.run_mlp_md(configuration=system.random_configuration(),
                                    mlp = mg,
                                    fs=10000,
                                    temp=300,
                                    dt=0.5,
                                    interval=10)

    trajectory_gas.save(filename='mg_aqua_cluster_trajectory.xyz')
    trajectory_gas.compare(mg, 'xtb')
```

Let's go through the input line by line. The line below defines the GPU is used for inference.

```
mlt.Config.mace_params['calc_device'] = 'cuda'
```

Firstly, you need to specify the system for which you will train the MLIP and run the AL loop. The information provided here will be used in electronic structure computations (i.e., the charge and multiplicity).

```
system = mlt.System(mlt.Molecule('TPSSH.xyz',charge=2, mult=1),box = None)
```

Afterwards, you can select the MLIP model to train. Here, we use MACE . The first argument in `mlt.potentials.MACE` corresponds to the name used to save MLIP and related files, such as data sets.

```
mg = mlt.potentials.MACE('mg_aqua', system = system)
```

In the next step, we define the AL loop used to generate the dataset and train the MLIP. The AL generates data through running a short MLIP-MD simulation, followed by testing the last frame of the dynamics using a chosen selector method, to decide whether a new data point will be added. When no selector is defined, the energy selector is used as the default option.

```
mg.al_train(method_name = 'xtb', temp=400,
                max_active_iters = 10,
                max_active_time = 3000)
```

The `method_name` corresponds to the electronic structure code used, `temp` is the temperature in K used in the MLIP-MD run. The two last settings, `max_active_iters` and `max_active_time`, define the condition upon which the AL will stop - the training will be complete when either MLP-MD runs for 3 ps without finding a new structure or when 10 iterations are completed.

After the AL stops, we can test the performance of the MLIP in short dynamics, as:

```
trajectory_gas = mlt.md.run_mlp_md(configuration=system.random_configuration(),
                                    mlp = mg,
                                    fs=10000,
                                    temp=300,
                                    dt=0.5,
                                    interval=10)
trajectory_gas.save(filename='mg_aqua_cluster_trajectory.xyz')
```

The simulation will start from a geometry randomly generated from the optimised structure, `configuration=system.random_configuration()` and will run for 10 ps at 300 K. The snapshots of structures will be printed every 10 steps. At the end, the trajectory will be saved in `mg_aqua_cluster_trajectory.xyz`.

Finally, we can validate the accuracy of the MLIP by comparing the prediction of energy and forces along the generated trajectory.

```
trajectory_gas.compare(mg, 'xtb')
```

Now, the example can be saved as `mg_hexaaqua_mace.py` and run locally as:

```
python mg_hexaaqua_mace.py
```

Alternatively, the script can be evaluated on computational nodes using the corresponding submission file.

The overall AL cycle takes approximately 3h 30 min; we therefore provide the completed files.

When completed, the run produces several files and folders. Files in `.model` format corresponds to MACE models saved after the last AL cycle. The final model is saved in `mg_aqua_stagetwo.model`. The `stage_one` and `stage_two` refer to the part of training when the model was generated - stage two is after the SWA part. The final dataset is saved in extended xyz format in `mg_aqua_al.xyz`, as well as in binary numpy compressed file format (.npz) `mg_aqua_al.npz`. Both these files can be later read by `mlp-train`. The folder `datasets` contains training data saved after each AL loop. These data can be loaded and used as a backup in case the AL loop breaks before completion. Finally, `mg_aqua_cluster_trajectory.xyz` contains the trajectory generated for the MACE validation. The energies and forces computed along the trajectory at the xtb and MLIP levels are saved in `mg_aqua_xtb.npz` file. The comparison between the predicted and computed values is plotted in `mg_aqua_xtb.pdf`.

You can open the XYZ dataset in your favourite visualization program, e.g., `VMD`.

```
vmd mg_aqua_al.xyz
```

In this run, AL overall yielded 41 structures. The first 10 structures in the data set are very similar to each other - these are the first 10 data points generated by the random displacement from the initial optimised structure. After these 10 structures, the water molecules around the central Mg_2^+ atom get more distorted, as these are extracted from increasingly long MD simulations.

We can plot a histogram of the distance between the Mg_2^+ and the water molecules in the data set.

```
import numpy as np
import matplotlib.pyplot as plt
import ase.io as aio

data = aio.read("mg_aqua_al.xyz", index=":")

distances = []

for structure in data:
    distances.append(structure.get_distances(0, np.arange(1, 19, 3)))


plt.hist(np.concatenate(distances), bins=30, color="skyblue", edgecolor="black")


plt.xlabel("Distance (Å)")
plt.ylabel("Frequency")
plt.title("Mg-O distance")

plt.savefig("mg_o_dataset.pdf", bbox_inches="tight")
```

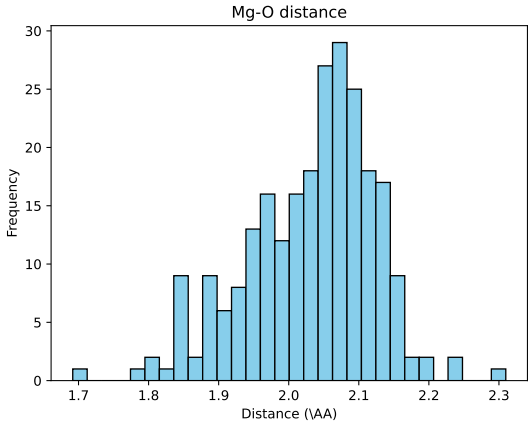


Figure 4: Histogram of the Mg-O distance in the training data

The histogram demonstrates that all the generated data corresponds to the bounded hexaaqua complex, with all water molecules within 2.5 Å from the central Mg_2^+ cation without any changes in coordination number.

The performance of the MLIP over the 10 ps trajectory is assessed in `mg_aqua_xtb.pdf`.

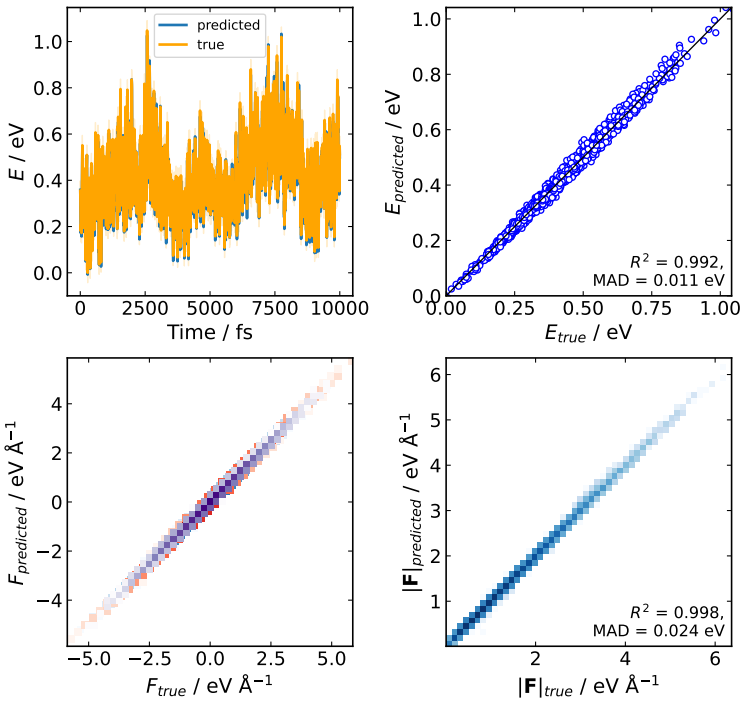


Figure 5: Performance of the MACE potential over 10 ps $[Mg(H_2O)_6]_2^+$ trajectory.

The mean absolute deviation (MAD) for the energy prediction is 11 meV, which corresponds to 0.58 meV/atom or 0.25 kcal/mol. The force error is 24 meV/Å(0.55 kcal /mol·Å). These errors demonstrate good accuracy of the MACE across the tested trajectory.

When we look at the structures in this simulation, the histogram Mg-O distances (Fig. 6) corresponds to the distances covered in the training data set. The MACE potential thus shows good performance in the data which were covered in the training dataset.

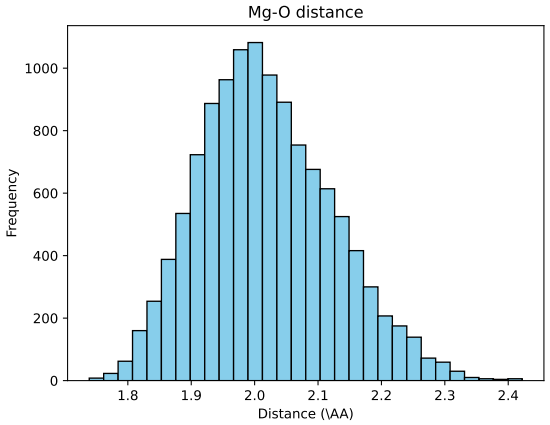


Figure 6: Histogram of the Mg-O distances over 10 ps $[Mg(H_2O)_6]^{2+}$ trajectory.

Let's assess the performance of MACE outside of the region of training data. Due to large stability of the $[Mg(H_2O)_6]^{2+}$ complex, our training and validation sets contain solely the bonded structure close to its equilibrium position. To create distorted structures, we will use the MACE potential and run dynamics in which we will pull one of the water molecules far from the central Mg^{2+} atom.

For the pulling, we will use moving restraint as implemented in Plumed.[7,8]. More information on the settings can be founds on Plumed website: `https://www.plumed.org/doc-v2.9/user-doc/html/_m_o_v_i_n_g_r_e_s_t_r_a_i_n_t.html`

To be able to use the restraint, we will first prepare `plumed.in` file, which will be read by `mlp-train`. The content of the file is:

```
UNITS LENGTH=A ENERGY=kcal/mol
atoms: GROUP ATOMS=@allatoms REMOVE=1
d: DISTANCE ATOMS=1,17
restraint: MOVINGRESTRAINT ARG=d STEP0=0 AT0=1.9 KAPPA0=10.0 STEP1=2000 AT1=5.0
PRINT ARG=d,restraint.bias FILE=COLVAR
```

Then, we run the pulling trajectory in `mlp-train`. The code is similar as for equilibrium sampling:

```
import mlptrain as mlt

mlt.Config.mace_params["calc_device"] = "cuda"

if __name__ == "__main__":
    system = mlt.System(mlt.Molecule("TPSSH.xyz", charge=2, mult=1), box=None)
    mg = mlt.potentials.MACE("mg_aqua_stagetwo", system=system)
    al_bias = mlt.PlumedBias(filename="plumed.dat")

    trajectory_gas = mlt.md.run_mlp_md(
        configuration=system.random_configuration(),
        mlp=mg,
        fs=500,
        temp=300,
        dt=0.5,
        interval=25,
        bias=al_bias,
    )

    trajectory_gas.save(filename="mg_aqua_pulling_trajectory.xyz")
    trajectory_gas.compare(mg, "xtb")
```

The only difference is the definition of biasing potential, in our case read from the file:

```
al_bias = mlt.PlumedBias(filename="plumed.dat")
```

The bias also needs to be added to the MLP-MD:

```
trajectory_gas = mlt.md.run_mlp_md(
        configuration=system.random_configuration(),
        mlp=mg,
        fs=500,
        temp=300,
        dt=0.5,
        interval=25,
        bias=al_bias,
    )
```

We can now plot the Mg-O distances in the resulting trajectory:

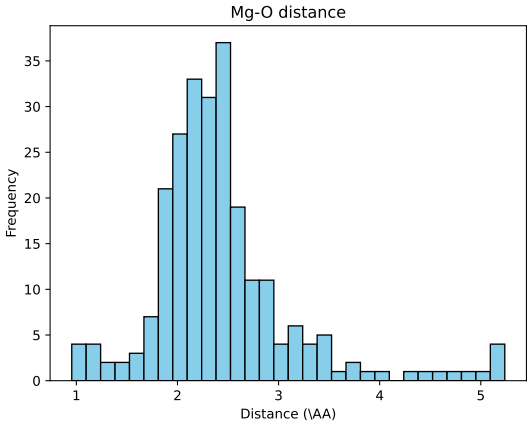


Figure 7: Histogram of the Mg-O distances over the pulling trajectory.

It is clear, that the Mg-O sample much broader range of values, many of the significantly outside of the training data. Consequently, the performance of the MACE potential is significantly worse:

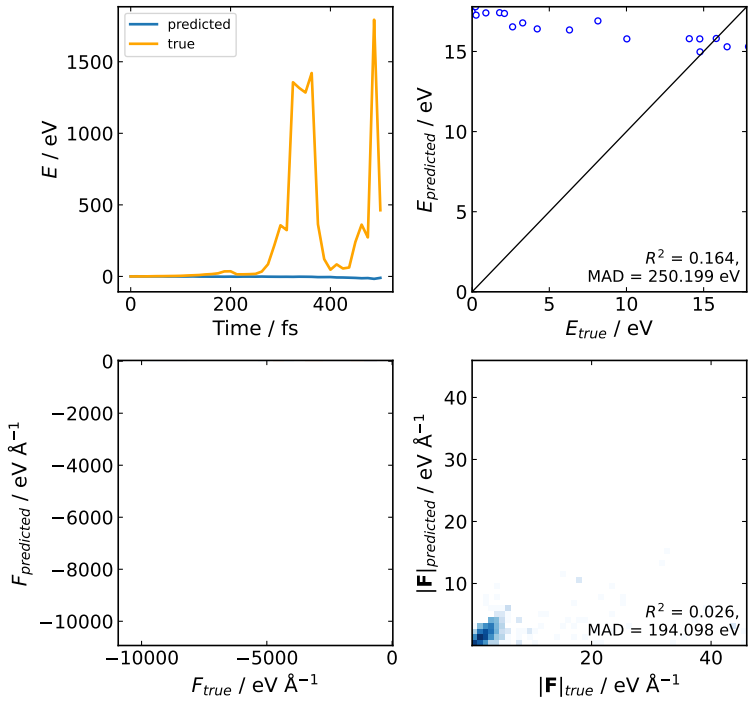


Figure 8: Validation of MACE potential over the pulling trajectory.

You can clearly see that after 200 fs, the structure of the complex collapsed and xtb energy starts to be

significantly large. However, this change is not reflected in MACE, showing strong extrapolation in this region, with MACE strongly overestimates the distorted structures of the complex.

It is thus necessary to always properly validate the performance of the MACE. Here, we can clearly see that the model performs accurately only on the equilibrium structure of $[Mg(H_2O)_6]^{2+}$, which is represented in the data set.

When you visualise the pulling trajectory, you can further see the changes in the complex, which becomes completely destroyed as a result of failing MACE potential.

5 Modelling reactions in gas phase

In the previous example, we generate data for metal complex in the equilibrium state, without considering chemical changes. Here, we will look into organic chemical reactions in the gas phase. As a first example, we will model the Diels-Alder (DA) reaction between cyclopentadiene (CP) and methyl vinyl ketone (MVK). The scheme of the reaction is depicted in Fig. 9.

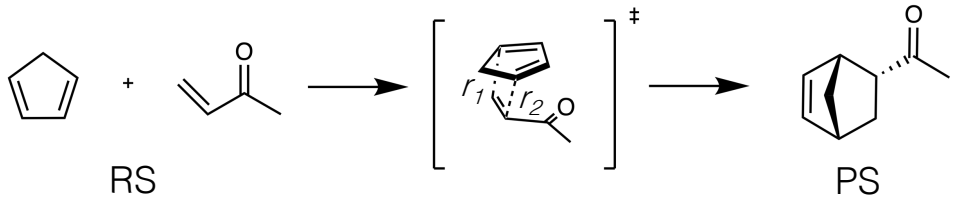


Figure 9: Scheme of the Diels-Alder reaction between cyclopentadiene and methyl vinyl ketone.

5.1 Downhill sampling

In contrast to the previous example, our aim is to develop MLIP, which will be able to describe reactants, products and the reaction pathway between the point crossing the transition state (TS) region. One of the strategies how to ensure that whole reaction path is covered in the training set is to start the AL from the TS. The subsequent dynamics will ensure that the structure will follow the gradient downhill on the potential energy surface (PES) forward towards products or back to reactants.

We will start from the TS optimised at PBE0/def2-svp level of theory.

```
22

 C   -2.45625042994264     4.22697136718373    -0.69050566209957
 C   -3.28702195379458     3.16527993919028    -0.02209098677282
 C   -2.41028766314595     2.14322126716790     0.34022614745311
 H   -4.18631817798479     3.39651295844350     0.53629598217147
 C   -1.20817318676536     2.29480857602016    -0.36982113539345
 H   -2.66413749952356     1.29424573410959     0.95978388172974
 C   -1.28146913102924     3.42909805269927    -1.13984450034102
 H   -0.40867019996620     1.56728893158943    -0.38835776572309
 H   -0.50683361420779     3.80598439935262    -1.79369703691900
 H   -2.11400085795616     4.92541367601641     0.08909189132240
 H   -2.95734970838890     4.81263532467316    -1.45923411982257
 C   -3.10205484705813     2.37368598258547    -2.68346342635020
 C   -4.11637859827911     2.44256568361059    -1.74418593887422
 H   -4.85151124593427     3.23644849500096    -1.82006001153415
 H   -4.47736922293900     1.50457689648386    -1.33925710572378
 C   -2.36389642614390     1.13176739960370    -2.83295879682348
 H   -2.91472881437345     3.18240648333903    -3.37940748993174
 O   -2.50989176905760     0.20606095908980    -2.04760734145413
 C   -1.42166214759852     1.00292166731278    -4.00365141692310
 H   -1.03881265683699     1.96596828196554    -4.34659501197343
 H   -1.96765811314529     0.54168812578948    -4.83288296604129
 H   -0.59713790022863     0.34139508057279    -3.73891706567561
```

For the AL, we will use similar code as in the previous case:

```python
import mlptrain as mlt

mlt.Config.n_cores = 10
mlt.Config.orca_keywords = ['PBE0', 'def2-SVP', 'EnGrad']
mlt.Config.mace_params['calc_device'] = 'cuda'

if __name__ == '__main__':
    system = mlt.System(
        mlt.Molecule('cis_endo_TS_PBE0.xyz', charge=0, mult=1), box=None
    )

    mace = mlt.potentials.MACE('endo_ace', system=system)

    mace.al_train(
        method_name='orca',
        temp=500,
        max_active_time=1000,
        fix_init_config=True,
        keep_al_trajs=True,
    )

    # Run some dynamics with the potential
    trajectory = mlt.md.run_mlp_md(
        configuration=system.configuration,
        mlp=mace,
        fs=200,
        temp=300,
        dt=0.5,
        interval=10,
    )

    # and compare, plotting a parity diagram and E_true, $\Delta$ E and $\Delta$ F
    trajectory.compare(mace, 'orca')
```

In comparison to the previous example, there are several differences. Firstly, we are going to describe the reaction using Density Functional Theory (DFT), which provides more accurate results than GFN2-xtb. We thus need to change the code used for the electronic structure computations from xtb to Orca. There is no default settings for the DFT functional used; we thus need to specify what level of theory we would like to use. In this case, we use PBE0/def2-SVP. This level is typically not sufficient for the accurate description of chemical reactions, and it is used solely for illustration.

```
mlt.Config.orca_keywords = ['PBE0', 'def2-SVP', 'EnGrad']
```

Another modification we need to make is in the AL cycle. Firstly, we set the method used in the training to 'orca'. We then fix the initial configuration in the AL (`fix_init_config=True`). This setting ensures that each AL cycle will start from the TS structure, i.e., the downhill sampling will be used in every run. Finally, we will set `keep_al_trajs=True`, to save the trajectories sampled during each AL for future reference.

```
mace.al_train(
        method_name='orca',
        temp=500,
        max_active_time=1000,
        fix_init_config=True,
        keep_al_trajs=True,
    )
```

Apart from the datasets, models and validation of the trajectory, the code now also creates `al_trajectories/` folder, containing trajectories from each AL iteration. The trajectories are saved as `traj_iter{n}_{m}.xyz`, where n is the AL iteration and m is the index of the trajectory. You can visualise some of these trajectories to see how different iterations led to reactants or products sampling.

The data set generated during the AL contains 128 structures. To see the coverage of the reaction space, we plot the data based on the collective variable, defined as $\frac{r_1+r_2}{2}$, where r_1 and r_2 are the two bonds formed during the Diels-Alder reaction.

```python
import numpy as np
from matplotlib import rc
```

```
import matplotlib.pyplot as plt
import ase.io as aio

rc("text", usetex=True)

data = aio.read("endo_ace_al.xyz", index=":")


collective_variable = []

for structure in data:
    r1 = structure.get_distance(1, 12)
    r2 = structure.get_distance(6, 11)
    collective_variable.append(0.5 * (r1 + r2))


x = np.arange(0, len(collective_variable))

clas = np.where(
    np.array(collective_variable) < 1.8,
    "PS",
    np.where(np.array(collective_variable) > 2.8, "RS", "TS"),
)

cdict = {"RS": "red", "TS": "blue", "PS": "black"}

fig, ax = plt.subplots()
for c in np.unique(clas):
    ix = np.where(clas == c)
    ax.scatter(x[ix], np.array(collective_variable)[ix], c=cdict[c], label=c, s=10)

plt.xlabel("Index")
plt.ylabel(r"$\frac{(r_1+r_2)}{2}$")
plt.title("Data points")
plt.legend()

plt.savefig("r12_dataset.pdf", bbox_inches="tight")
```

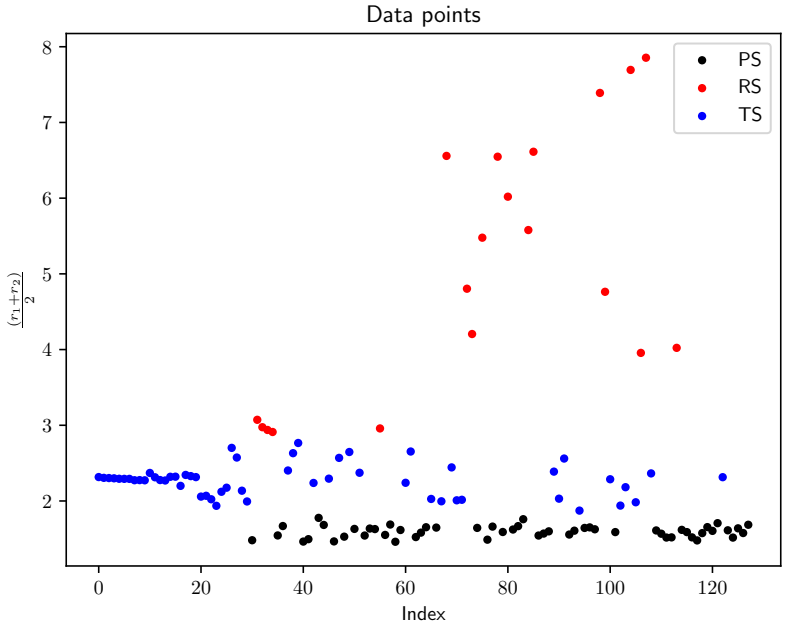


Figure 10: Classification of the structures from the dataset

You can again see that the first 10 data points are very similar to each other - these correspond to 10 distorted TS structures. Afterwards, the downhill sampling in AL generates structures of both reactants and products.

We can now check the performance of the MACE over a short 200-fs validation trajectory.

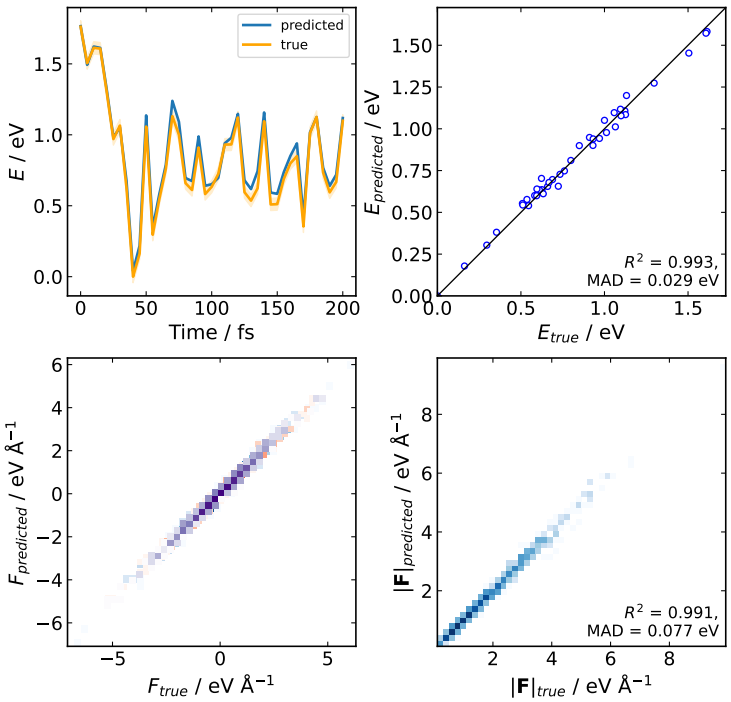


Figure 11: Validation of MACE over 200 fs dynamics

The MAD in energy is 29 meV, corresponding to 1.32 meV/atom. MAD in forces is 77 meV/Å. These errors are higher than in the previous case, suggesting that we might need to set the time in the AL longer than in the current settings. Indeed, the 1 ps sampling time is rather short. When plotting the validation trajectory, we see that the dynamics correspond to the reaction to products. This demonstrates that despite the errors, our MLIP is capable of describing the formation of the product state. To provide a better estimate of the performance, it might be beneficial to repeat the validation on trajectory which covers all relevant structures, including different orientations of reactants. For simplicity, we will skip this extended validation.

The resulting potential can now be used for other simulations. For instance, we can run an umbrella sampling (US) simulation to compute the free energy barrier of the reaction.

`mlp-train` includes its own US implementation, which automatically defines the windows.

The example script can look like this:

```
import mlptrain as mlt
import numpy as np
from mlptrain.box import Box
from mlptrain.log import logger

mlt.Config.mace_params["calc_device"] = "cuda"

if __name__ == "__main__":
    us = mlt.UmbrellaSampling(zeta_func=mlt.AverageDistance((1, 12), (6, 11)), kappa=10)

    irc = mlt.ConfigurationSet()
    irc.load_xyz(filename="irc_IRC_Full_trj.xyz", charge=0, mult=1)

    for config in irc:
        config.box = Box([100, 100, 100])

    irc.reverse()

    TS_mol = mlt.Molecule(name="cis_endo_TS_PBE0.xyz", charge=0, mult=1, box=None)

```

```
    system = mlt.System(TS_mol, box=Box([100, 100, 100]))

    endo = mlt.potentials.MACE("endo_ace_stagetwo", system)

    us.run_umbrella_sampling(
        irc,
        mlp=endo,
        temp=300,
        interval=5,
        dt=0.5,
        n_windows=15,
        init_ref=1.55,
        final_ref=4,
        ps=10,
    )
    us.save("wide_US")

    # Run a second, narrower US with a higher force constant
    us.kappa = 20
    us.run_umbrella_sampling(
        irc,
        mlp=endo,
        temp=300,
        interval=5,
        dt=0.5,
        n_windows=15,
        init_ref=1.7,
        final_ref=2.5,
        ps=10,
    )

    us.save("narrow_US")

    total_us = mlt.UmbrellaSampling.from_folders("wide_US", "narrow_US", temp=temp)
    total_us.wham()
```

The final windows are saved in `fitted_data.pdf` file and the reconstructed free energy profile is `umbrella_free_energy.pdf`

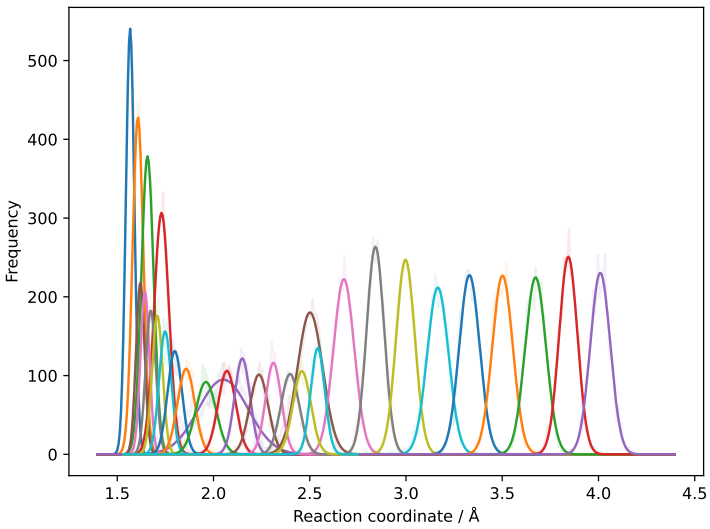


Figure 12: Windows sampled during the umbrella sampling.

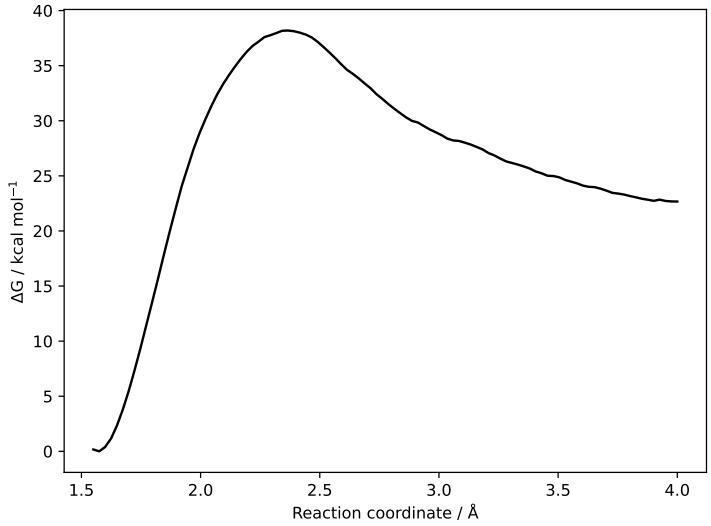


Figure 13: Free energy profile for the Diels-Alder reaction in the gas phase

5.2 Metadynamics

In the previous example, we adopted the AL + downhill sampling methodology to generate the training sets, where the directions of the MLP-MD trajectories are determined by the randomly assigned initial velocities from the starting configuration (in this case, from the TS toward either the RS or PS). While this strategy is straightforward to use, it has several limitations. First of all, we need to be able to identify a suitable TS structure, which is not always the case. For instance, in many metal complexes and systems in solution, identifying the TS is still an open challenge. Furthermore, in cases where the reaction can undergo competing pathways, one needs to provide the TS for each of them separately. To overcome these limitations and improve the efficiency of the training, we integrated well-tempered metadynamics (WTMetaD) sampling into the AL.

Metadynamics is an enhanced sampling technique which accelerates the exploration of the free energy surface (FES) by adding Gaussian-shaped biasing potentials into previously visited parts of the FES. The accumulated potential pushes the structures away from the minima, over the barriers on the FES, thereby accelerating the sampling of chemical reactions. The well-tempered version of metadynamics introduces a scaling factor, which decreases the height of the deposited Gaussians and improves convergence of the sampling. More information on the metadynamics and well-tempered metadynamics algorithms can be found in Ref. [9].

In our approach, we perform WTMetaD simulations within the AL process. This enables sampling from any point on the potential energy surface (PES), preferably reactants, without the need to know the corresponding TS structure. Moreover, we introduce an inherited bias scheme, where the bias potential generated in the previous AL iteration is carried over to the current one, preventing repeated depositing of bias into already sampled regions. The starting point for MLP-MD in each iteration is updated as well, starting from the configuration with the lowest energy on the biased surface, further prioritising unexplored regions. We demonstrated that this continuous update of both the bias potential and starting configurations significantly improves sampling efficiency compared to stand-alone WTMetaD. [3] The differences among the three introduced sampling methods so far have been depicted in Fig. 14.

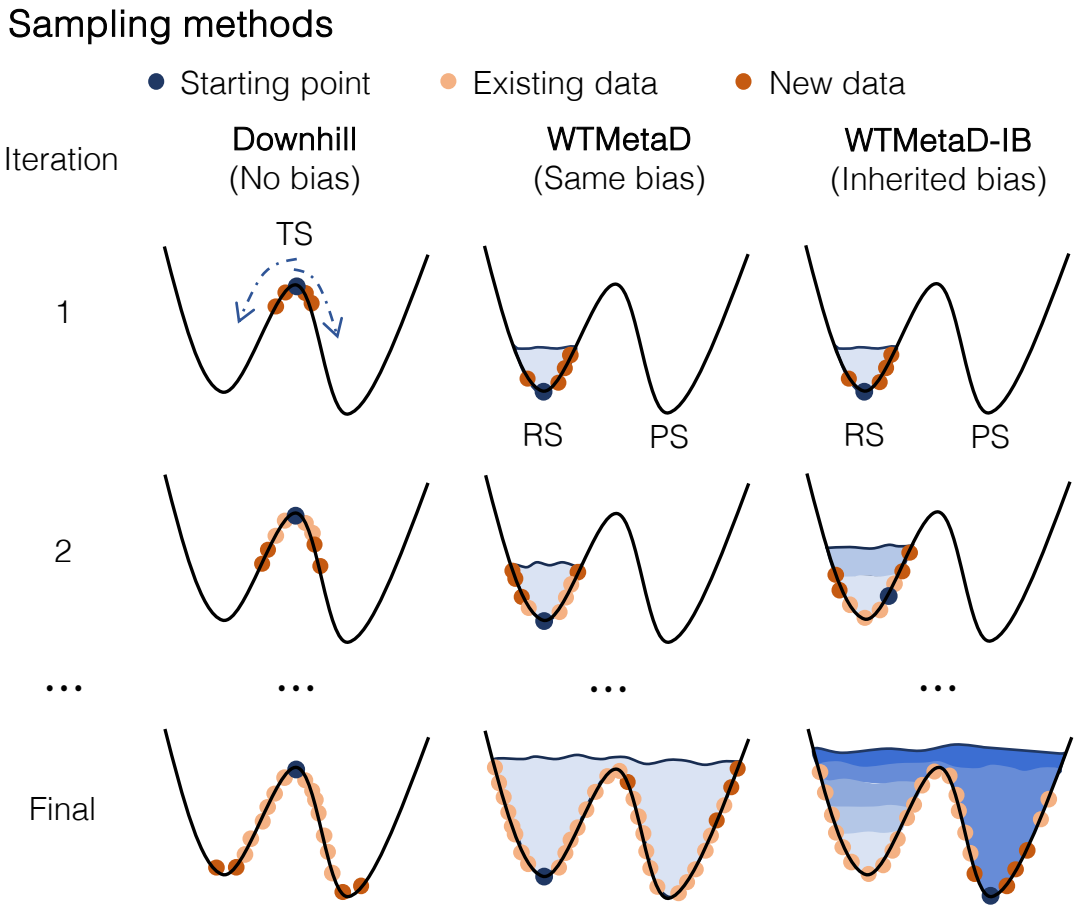


Figure 14: Schematic representations of sampling methods in the AL: MLP-MD sampling can be performed using either downhill, well-tempered metadynamics (WTMetaD) or WTMetaD with inherited bias (WTMetaD-IB). Dark blue points denote the starting points in MLP-MD, yellow points indicate the existing training data and dark orange points denote the training points selected in each iteration. The different blue shades shown in WTMetaD-IB indicate that the biases are updated after each iteration, while in WTMetaD, the same bias is maintained.

In this example, we will illustrate the use WTMetaD in modelling the S_N2 reaction between fluoride and chloromethane as the model system, as shown in Figure 15, with CPCM(water)-PBE0-D3BJ/def2-SVP level of theory as reference.

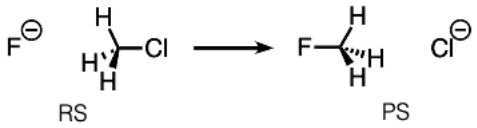


Figure 15: Scheme of S_N2 reaction between fluoride and chloromethane in implicit water.

We will start the training from the non-optimised reactant structure, saved in `ch3cl_f.xyz`:

```
6

C          0.11150        0.23091        0.58288
F         -0.88850        1.32101       -2.02704
Cl         0.64512       -0.39439        2.08233
H          1.02099        0.50449       -0.09172
```

```
H         -0.45901        1.13911        0.74042
H         -0.45901       -0.53348        0.04182
```

Next, it is important to define the collective variables (CVs) which will be biased in the metadynamics simulations. Due to the simplicity of the studied process, we define only one CV to describe the reaction – the difference between the C–Cl and C–F bond distances, $(r_{\mathrm{Cl}} - r_{\mathrm{F}})$.

Furthermore, we will define the second auxiliary CV, the average distance of the C–Cl and C–F bonds, $\frac{1}{2}(r_{\mathrm{Cl}} + r_{\mathrm{F}})$. While this CV will not be directly biased during the WTMetad, we will attach to it an upper-wall restraint to limit its range - it will act as a soft barrier around the reactants, ensuring that the reactive atoms do not drift too far apart.

The complete code is written below. We will highlight important parts separately:

```python
import mlptrain as mlt

mlt.Config.n_cores = 10
mlt.Config.mace_params['calc_device'] = 'cuda'
mlt.Config.orca_keywords = ["PBE0", "def2-SVP", "EnGrad", "CPCM(Water)"]

if __name__ == "__main__":
    system = mlt.System(mlt.Molecule("ch3cl_f.xyz", charge=-1, mult=1), box=None)

    # Define CV for WTMetaD AL (r_cl - r_f)

    diff_r = mlt.PlumedDifferenceCV(name="diff_r", atom_groups=((0, 2), (0, 1)))

    # Define auxiliary CV and attach an upper wall bias
    avg_r = mlt.PlumedAverageCV(name="avg_r", atom_groups=((0, 1), (0, 2)))
    avg_r.attach_upper_wall(location=2.5, kappa=1000)

    # Initialise PlumedBias for WTMetaD AL
    bias = mlt.PlumedBias(cvs=(diff_r, avg_r))
    bias.initialise_for_metad_al(width=0.05, cvs=diff_r, biasfactor=100)

    # Define the potential and train using WTMetaD AL

    mace = mlt.potentials.MACE("r1_wtmetad", system=system)
    mace.al_train(
        method_name="orca",
        temp=300,
        n_init_configs=5,
        n_configs_iter=5,
        max_active_iters=50,
        min_active_iters=30,
        inherit_metad_bias=True,
        bias=bias,
    )
```

The difference between the distances can be defined directly using `PlumedDifferenceCV` function, corresponding to $r_{CCl} - r_{CF}$.

```python
    diff_r = mlt.PlumedDifferenceCV(name="diff_r", atom_groups=((0, 2), (0, 1)))
```

Afterwards, we define the auxiliary CV as an average of the r_{CCl} and r_{CF}. The attached upper wall bias is located at 2.5 Å, ensuring that when the average bond length passes this value, it is softly pushed back by a harmonic restraint with a force constant of 1000 eV Å^{-2}.

```python
    avg_r = mlt.PlumedAverageCV(name="avg_r", atom_groups=((0, 1), (0, 2)))
    avg_r.attach_upper_wall(location=2.5, kappa=1000)
```

The collective variables (CVs) are then passed to `PlumedBias` class, which is used to generate input files for Plumed.

```python
    # Initialise PlumedBias for WTMetaD AL
    bias = mlt.PlumedBias(cvs=(avg_r, diff_r))
```

Finally, we specify the CV biased during the WTMetaD simulation, together with the parameters of the deposited Gaussian, i.e., their width and bias factor.

```
    bias.initialise_for_metad_al(width=0.05, cvs=diff_r, biasfactor=100)
```

The MACE model and AL loop are set up the same way as described previously, with the additional parameters for WTMetaD: `bias=bias, inherit_metad_bias=True`, which ensures that the bias from previous iterations is brought to the following iterations, and `bias=bias` included in the `al_train` function to initiate the WTMetaD-IB AL training.

```

    # Define the potential and train using WTMetaD AL (inherit_metad_bias=True)
    mace = mlt.potentials.MACE('r1_wtmetad', system=system)
    mace.al_train(
        method_name='orca',
        temp=300,
        n_init_configs=5,
        n_configs_iter=5,
        max_active_iters=50,
        min_active_iters=30,
        bias=bias,
        inherit_metad_bias=True,
    )
```

The discussed scripts yields 177 data points, saved in `r1_wtmetad_al.xyz`. You can visualise them and check how the different parts of the PES are explored.

Once the model is trained, it can be used again in molecular dynamics. Apart from the umbrella sampling simulation mentioned before, `mlp-train` also includes metadynamics implementation.

The example code for running WTMetad can look like this:

```
import mlptrain as mlt

mlt.Config.mace_params['calc_device'] = 'cuda'

if __name__ == "__main__":
    system = mlt.System(mlt.Molecule("ch3cl_f.xyz", charge=-1, mult=1), box=None)

    # Define CV and attach an upper wall
    r_cl = mlt.PlumedAverageCV(name="r_cl", atom_groups=(0, 2))
    r_cl.attach_upper_wall(location=5, kappa=100)

    r_f = mlt.PlumedAverageCV(name="r_f", atom_groups=(0, 1))
    r_f.attach_upper_wall(location=5, kappa=100)

    mace = mlt.potentials.MACE("r1_wtmetad_stagetwo_compiled", system=system)

    # Attach CVs to the metadynamics object
    metad = mlt.Metadynamics(cvs=(r_cl,r_f))
    # Generate a starting metadynamics configuration
    config = system.random_configuration()
    # run the optional function to help choosing parameters
    width = metad.estimate_width(configurations=config, mlp=mace, plot=True)

    # for width estimation

    metad.run_metadynamics(
        configuration=config,
        mlp=mace,
        temp=300,
        dt=0.5,
        ps=100,
        interval=10,
        n_runs=3,
        width=width,
        height=0.01,
        biasfactor=10,
    )
```

```

    metad.plot_fes()

    metad.plot_fes_convergence(stride=10, n_surfaces=5)
```

In this simulation, we use two CVs to run the WTMetad, i.e., the C-Cl and C-F distances. To make sure that the F and Cl atoms stay in a reasonable distance from the carbon atoms, we also set the upper wall bias to them in a distance of 5 Å.

```
r_cl = mlt.PlumedAverageCV(name="r_cl", atom_groups=(0, 2))
r_cl.attach_upper_wall(location=5, kappa=100)

r_f = mlt.PlumedAverageCV(name="r_f", atom_groups=(0, 1))
r_f.attach_upper_wall(location=5, kappa=100)
```

We then attach these CVs to the Metadynamics object:

```
    # Attach CVs to the metadynamics object
    metad = mlt.Metadynamics(cvs=(r_cl,r_f))
```

For the metadynamics run, we can estimate the suitable values for the Gaussian width and bias factor using the functions `estimate_width` and `try_multiple_biasfactors`, respectively. The appropriate width of the deposited Gaussian potential is estimated from short NVT simulations by calculating the standard deviation of the selected CV. The minimum standard deviation across these simulations is selected as the optimal width. For bias factor selection, multiple well-tempered metadynamics runs are executed in parallel using different bias factors, and the resulting trajectories are analysed to determine the most appropriate value.

```

  width = metad.estimate_width(configurations=config, mlp=mace, plot=True)

   metad.try_multiple_biasfactors(
       configuration=config,
       mlp=mace,
       width=width
       biasfactor=(5,10,15),
       plotted_cvs=dih,
    )
```

Finally, you can run metadynamics simulation using `run_metadynamics` function and specify the associated parameters, such as width, height (**in eV**) and biasfactor.

```
  metad.run_metadynamics(
      configuration=config,
      mlp=mace,
      temp=300,
      width=width,
      height=0.01,
      biasfactor=10
  )
```

The free energy profile can also be plotted using

```
    metad.plot_fes()
    metad.plot_fes_convergence(stride=10, n_surfaces=5)
```

After 100 ps, the resulting free energy surface (FES) looks like this:

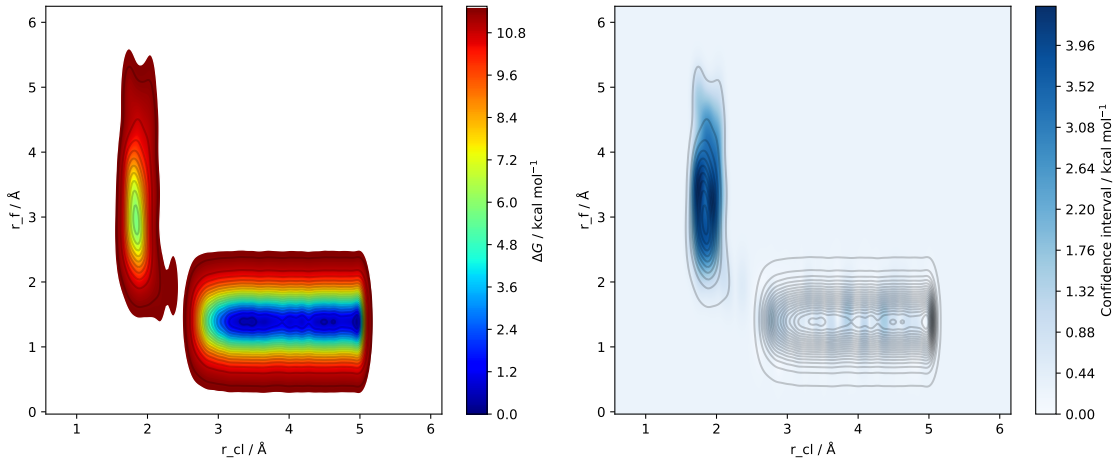


Figure 16: Free energy surface of SN_2 reaction after 100 ps of metadynamics

6 Solvation function

In the previous examples, we discussed chemical processes in the gas phase and in solution, described via implicit solvent. However, in many cases, we need to model the solvent explicitly. The solvation functionality in `mlp-train` allows users to create explicit solvent environments around molecular systems. This is particularly valuable for studying chemical reactions and molecular behavior in realistic solvated conditions, where solvent effects can significantly influence reaction mechanisms, barriers, and product distributions.

The solvation algorithm uses a k-d tree approach with periodic boundary conditions to efficiently place solvent molecules around a solute while avoiding atomic overlaps. The method supports both predefined solvents from an internal database and custom solvent molecules with user-specified densities.

6.1 Example: Solvating cyclopentadiene in different solvents

To demonstrate the solvation capabilities, we will solvate cyclopentadiene (one of the reactants from the Diels-Alder reaction) in different solvent environments. This example shows how solvent choice can be systematically varied to study environmental effects on chemical systems.

First, let's create a cyclopentadiene molecule and convert it to a Configuration object:

```
import mlptrain as mlt
import autode as ade

# Create cyclopentadiene from SMILES
cp_smiles = "C1=CC=CC1"  # cyclopentadiene
cp_mol = ade.Molecule(smiles=cp_smiles)

# Optimize the geometry
cp_mol.optimise(method=ade.methods.XTB())

# Convert to MLPtrain Configuration
cp_config = mlt.Configuration(atoms=cp_mol.atoms, charge=0, mult=1)
```

6.1.1 Method 1: Using solvent database

The simplest approach is to specify the solvent by name from the built-in database. `mlp-train` includes densities for over 80 common solvents:

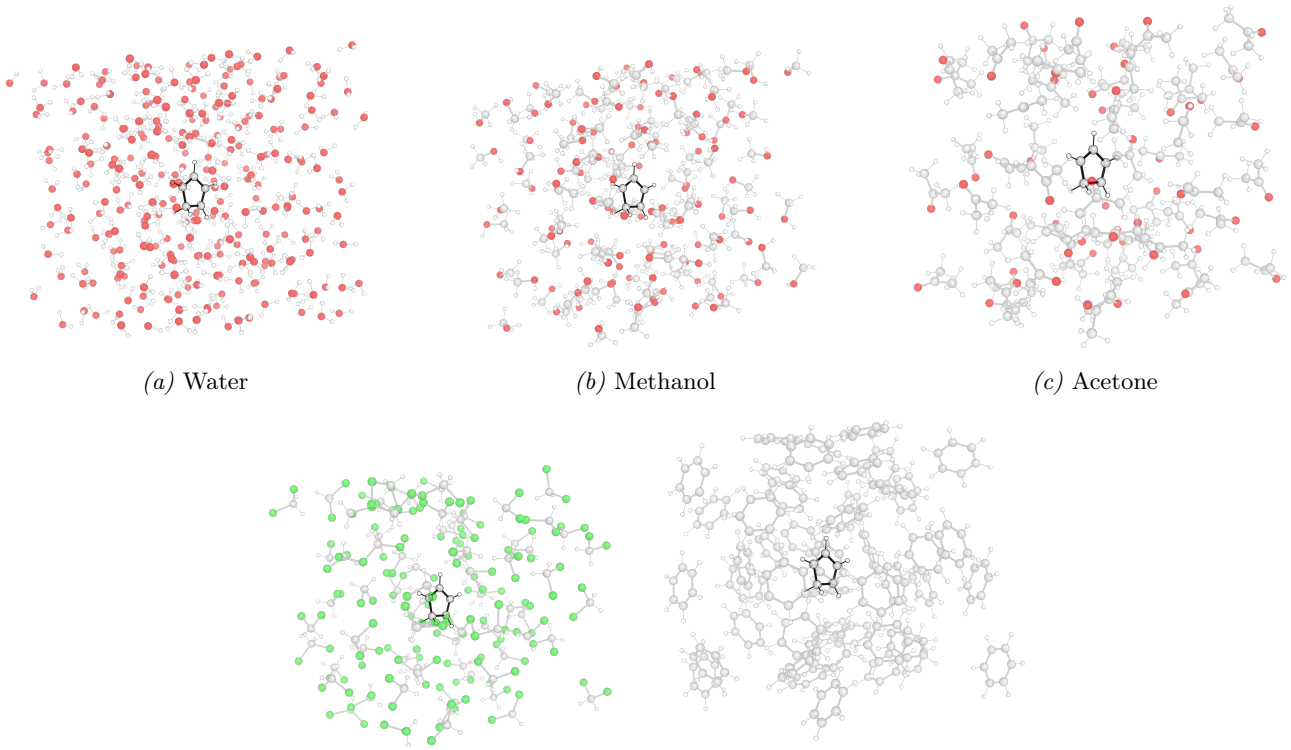
```
# Solvate in water with default parameters
cp_water = cp_config.copy()
cp_water.solvate(solvent_name='water')

# Save the solvated structure
cp_water.save_xyz('cyclopentadiene_water.xyz')
```

```
print(f"Solvated system contains {len(cp_water.atoms)} atoms")
print(f"Box size: {cp_water.box.size[0]:.2f} Å")
```

We can also solvate in different solvents to study solvent effects:

```
# Solvate in different polar and non-polar solvents
solvents = ['water', 'methanol', 'acetone', 'dichloromethane', 'benzene']

for solvent in solvents:
    cp_solvated = cp_config.copy()
    cp_solvated.solvate(
        solvent_name=solvent,
        box_size=20.0,              # Fixed box size for comparison
        contact_threshold=1.8,       # Standard contact distance
        random_seed=42              # Reproducible placement
    )

    filename = f'cyclopentadiene_{solvent}.xyz'
    cp_solvated.save_xyz(filename)

    # Count solvent molecules
    n_solvent = len(cp_solvated.atoms) - len(cp_config.atoms)

    # Assuming solvent molecules are defined in mol_dict and that there is only one type of solvent molecule
    # Get the number of atoms in the solvent molecule
    n_atoms_in_solvent = cp_solvated.mol_dict[solvent][0]['end'] - cp_solvated.mol_dict[solvent][0]['start']

    # Calculate the number of solvent molecules
    n_molecules = n_solvent // n_atoms_in_solvent

    print(f"{solvent}: {n_molecules} molecules, {n_solvent} atoms")
```

The resulting solvated structures for cyclopentadiene in different solvent environments are shown in Fig. 17. These visualizations demonstrate how the molecular packing and local environment vary significantly between polar and non-polar solvents.



(a) Water *(b)* Methanol *(c)* Acetone

(d) Dichloromethane *(e)* Benzene

Figure 17: Cyclopentadiene solvated in different solvent environments: (a) water, (b) methanol, (c) acetone, (d) dichloromethane, and (e) benzene. The visualizations show how solvent polarity and molecular size affect the local solvation structure around the cyclopentadiene molecule.

6.1.2 Method 2: Custom solvent molecules

For specialised solvents or non-standard conditions, you can provide custom solvent molecules with explicit densities:

```
# Create a custom THF molecule
thf_mol = ade.Molecule(smiles="C1CCOC1")  # tetrahydrofuran
thf_mol.optimise(method=ade.methods.XTB())

# Solvate using custom molecule and density
cp_thf = cp_config.copy()
cp_thf.solvate(
    solvent_molecule=thf_mol,
    solvent_density=0.889,          # THF density in g/cm^3
    box_size=18.0,
    contact_threshold=2.0
)

cp_thf.save_xyz('cyclopentadiene_custom_thf.xyz')
```

6.1.3 Understanding contact threshold effects

The `contact_threshold` parameter critically affects the solvation density and molecular packing. This parameter defines the minimum allowed distance between any two atoms during solvent placement:

```
# Study effect of contact threshold on solvation
thresholds = [1.5, 1.8, 2.0, 2.5]

for threshold in thresholds:
    cp_solvated = cp_config.copy()
    cp_solvated.solvate(
        solvent_name='water',
        box_size=15.0,
```

```
        contact_threshold=threshold,
        random_seed=42
    )

    n_water_atoms = len(cp_solvated.atoms) - len(cp_config.atoms)
    n_water_molecules = n_water_atoms // 3

    print(f"Threshold {threshold} Å: {n_water_molecules} water molecules")
```

The output with the above settings should give the following amounts of water molecules.

```
Threshold 1.5 Å: 112 water molecules
Threshold 1.8 Å: 111 water molecules
Threshold 2.0 Å: 93 water molecules
Threshold 2.5 Å: 56 water molecules
```

The algorithm will attempt to add the number of water molecules that would be required to achieve the density of water (1 g/cm^3). Because the solute also displaces some volume, it is expected that the number of solvent molecules differs from what would be expected in a simulation box of the same size but without the solute. For the 2.5 threshold setting, this is reported as:

```
INFO Attempting to add 112 solvent molecules with the formula H2O to a cubic box with a side length of 15.00 Å
INFO Inserted 56 solvent molecules into the box
```

6.1.4 Analysis and validation

After solvation, you can analyze the molecular distribution and validate the structure, as is seen in Fig. 18:

```
import numpy as np
import matplotlib.pyplot as plt

# Analyze radial distribution around the solute
def analyze_solvation_shell(config, solute_atoms=15):
    """Calculate distances from solute center of mass to solvent molecules"""

    # Get solute center of mass
    solute_coords = config.coordinates[:solute_atoms]
    solute_com = np.mean(solute_coords, axis=0)

    # Get solvent coordinates (oxygen atoms for water)
    solvent_coords = config.coordinates[solute_atoms::3]  # Every 3rd atom (O)

    # Calculate distances
    distances = np.linalg.norm(solvent_coords - solute_com, axis=1)

    return distances

# Load and analyze the solvated structure
cp_water = mlt.Configuration.from_xyz('cyclopentadiene_water.xyz')
distances = analyze_solvation_shell(cp_water)

# Plot radial distribution
plt.hist(distances, bins=20, alpha=0.7, edgecolor='black')
plt.xlabel('Distance from solute COM (Å)')
plt.ylabel('Number of water molecules')
plt.title('Radial distribution of water around cyclopentadiene')
plt.savefig('solvation_analysis.pdf', bbox_inches='tight')
```

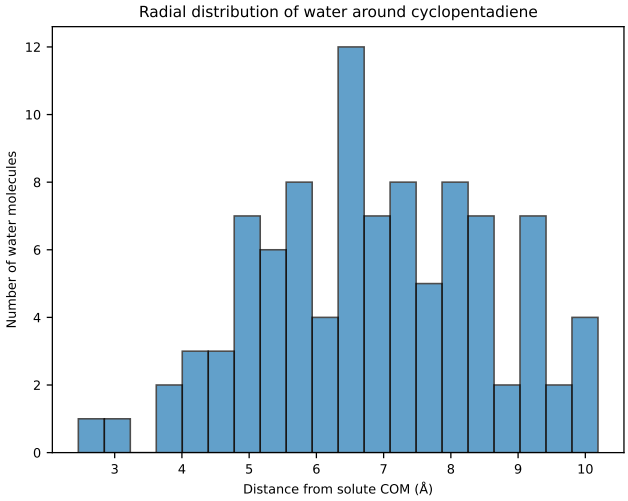


Figure 18: The histogram showing the radial distribution of water molecules around the CP molecule following use of the solvation function.

6.1.5 Integration with ML potential training

The solvated configurations can be directly used as starting points for active learning or MD simulations. As the size of the increases with the added solvent molecules, using *energy* selector in the AL becomes extremely costly. Instead, we illustrate here the use of structure selection based on *similarity* selector.

```
import mlptrain as mlt
from mlptrain.training.selection import AtomicEnvSimilarity
from mlptrain.descriptor import SoapDescriptor

mlt.Config.orca_keywords = ['PBE0', 'D3BJ','def2-SVP', 'EnGrad']
mlt.Config.mace_params['calc_device'] = 'cuda'

# Use solvated structure for ML potential training
system = mlt.System(mlt.Molecule('cyclopentadiene_water.xyz'), box=cp_water.box)

descriptor = SoapDescriptor(average='outer', r_cut=6.0, n_max=6, l_max=6)
selector = AtomicEnvSimilarity(descriptor=descriptor, threshold=0.9995)

# Train MACE potential on solvated system
mace = mlt.potentials.MACE('cp_water_mace', system=system)

mace.al_train(
    method_name='orca',
    temp=300,
    max_active_time=1000,
    selection_method=selector,
    fix_init_config=False  # Allow exploration from solvated structure
)

# Run solvated MD simulation
trajectory = mlt.md.run_mlp_md(
    configuration=cp_water,
    mlp=mace,
    fs=5000,                    # 5 ps simulation
    temp=300,
    dt=0.5,
    interval=10
)

trajectory.save('cp_water_md.xyz')
```

To be able to use the similarity selector, we first define the descriptor used in the computation:

```
descriptor = SoapDescriptor(average='outer', r_cut=6.0, n_max=6, l_max=6)
```

The solvation functionality provides a robust foundation for explicit solvent studies, enabling systematic investigation of environmental effects on chemical systems. The k-d tree algorithm ensures efficient and realistic solvent placement, while the flexible parameter system allows fine-tuning for different molecular systems and research objectives.

7 Application of Foundation models

Foundation models in MLIPs are gaining significant attention within the fields of computational chemistry and materials science. These models, which are trained on extensive datasets such as SPICE and the Materials Project (MP), are large-scale, pre-trained neural networks proficient in predicting atomic interactions across a wide range of chemical systems. Their capabilities facilitate high-fidelity simulations of molecular dynamics, material properties, and chemical reactions.

The main promise of the foundation model is to provide stable dynamics out-of-the-box, i.e., without the need to develop system-specific data sets and train the model. The `mlp-train` package supports the use of pre-trained foundation models (FM) based on the MACE architecture. These include for example MACE-MP-0 models,[10] targeting the chemistry of materials and MACE-OFF models,[11] focusing on the description of organic systems, including the condensed phase. These FMs are also available in different model sizes, i.e., small, medium and large models, which differs in number of parameters and accuracy. The list of existing MACE FM can be found at `https://github.com/ACEsuit/mace-foundations`.

In the following example, we will illustrate the use of the small MACE-OFF(23) model in MD simulations of cyclopentadiene in randomly placed water molecules, which were produced using the solvation function.

```
import mlptrain as mlt

# Load solvated structure from *xyz file

cp_water = mlt.Configuration.from_xyz('cyclopentadiene_water.xyz')

system = mlt.System(mlt.Molecule('cyclopentadiene_water.xyz'), box=cp_water.box)

# Load the pre-train MACE-OFF model
mace = mlt.potentials.MACE('MACE-OFF23_small', system=system)

# Run solvated MD simulation
trajectory = mlt.md.run_mlp_md(
    configuration=cp_water,
    mlp=mace,
    fs=5000,                  # 5 ps simulation
    temp=300,
    dt=0.5,
    interval=10
)

trajectory.save('cp_water_md_foundation.xyz')
```

You can compare the trajectory generated via MLIP trained from scratch with the one produced by a foundation model. While the foundation model is capable of yielding chemically reasonable results without the need for additional training, the more accurate modelling of specific systems requires additional re-training of the system, so-called fine-tuning.[12]

References

[1] Young, T. A.; Johnston-Wood, T.; Zhang, H.; Duarte, F. Reaction dynamics of Diels–Alder reactions from machine learned potentials. *Phys. Chem. Chem. Phys.* **2022**, *24*, 20820–20827.

[2] Zhang, H.; Juraskova, V.; Duarte, F. Modelling chemical processes in explicit solvents with machine learning potentials. *Nat. Commun* **2024**, *15*, 6114.

[3] Vitartas, V.; Zhang, H.; Juraskova, V.; Johnston-Wood, T.; Duarte, F. Active learning meets metadynamics: Automated workflow for reactive machine learning potentials. 2025.

[4] Bartok, A. P.; Kondor, R.; Csanyi, G. On representing chemical environments. *Phys. Rev. B* **2013**, *87*, 184115.

[5] Batatia, I.; Péter Kovács, D.; Simm, G. N. C.; Ortner, C.; Csányi, G. MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields. Advances in Neural Information Processing Systems. 2022; pp 11423–11436.

[6] Batatia, I.; Batzner, S.; Kovács, D. P.; Musaelian, A.; Simm, G. N. C.; Drautz, R.; Ortner, C.; Kozinsky, B.; Csányi, G. The design space of E(3)-equivariant atom-centred interatomic potentials. *Nat. Mach. Intell.* **2025**, *7*, 56–67.

[7] Bonomi, M.; Branduardi, D.; Bussi, G.; Camilloni, C.; Provasi, D.; Raiteri, P.; Donadio, D.; Marinelli, F.; Pietrucci, F.; Broglia, R. A.; Parrinello, M. PLUMED: A portable plugin for free-energy calculations with molecular dynamics. *Comput. Phys. Commun.* **2009**, *180*, 1961–1972.

[8] Tribello, G. A.; Bonomi, M.; Branduardi, D.; Camilloni, C.; Bussi, G. PLUMED 2: New feathers for an old bird. *Comput. Phys. Commun.* **2014**, *185*, 604–613.

[9] Bussi, G.; Laio, A. Using metadynamics to explore complex free-energy landscapes. *Nat. Rev. Phys.* **2020**, *2*, 200–212.

[10] Batatia, I. *et al.* A foundation model for atomistic materials chemistry. 2024; `https://arxiv.org/abs/2401.00096`.

[11] Kovács, D. P.; Moore, J. H.; Browning, N. J.; Batatia, I.; Horton, J. T.; Pu, Y.; Kapil, V.; Witt, W. C.; Magdău, I.-B.; Cole, D. J.; Csányi, G. MACE-OFF: Short-Range Transferable Machine Learning Force Fields for Organic Molecules. *J. Am. Chem. Soc.* **2025**, *147*, 17598–17611.

[12] Kaur, H.; Della Pia, F.; Batatia, I.; Advincula, X. R.; Shi, B. X.; Lan, J.; Csányi, G.; Michaelides, A.; Kapil, V. Data-efficient fine-tuning of foundational models for first-principles quality sublimation enthalpies. *Faraday Discuss.* **2025**, *256*, 120–138.