mff Documentation

Release 0.1.0

Aldo, Claudio, Adam

Jan 30, 2019

Table of Contents

MFF (Mapped Force Fields) is a package built to apply machine learning to atomistic simulation within an ASE environment. MFF uses Gaussian process regression to build non-parametric 2- and 3- body force fields from a small dataset of ab-initio simulations. These Gaussian processes are then mapped onto a non-parametric tabulated 2- or 3-body force field that can be used within the ASE environment to run atomistic simulation with the computational speed of a tabulated potential and the chemical accuracy offered by machine learning on ab-initio data. Trajectories or snapshots of the system of interest are used to train the potential, these must contain atomic positions, atomic numbers and forces (and/or total energies), preferrabily calculated via ab-initio methods.

At the moment the package supports single- and two-element atomic environments; we aim to support three-element atomic environments in future versions.

CHAPTER 1

Installation

To install from source, uncompress the source files and, from the directory containing *setup.py*, run the following command:

```
python setup.py install
```

Or, to build in place, run:

```
python setup.py build_ext --inplace
```

If you build in place, you will also need to add your eqtools folder to your PYTHONPATH shell variable:

```
export PYTHONPATH=$PYTHONPATH:/path/to/where/you/put/
```

1.1 Requirements

- Python
- Theano
- Numpy
- Scipy
- Pathos
- ASE
- Asap3

1.2 Usage

Description on how to use the package:

```
import mff
```

CHAPTER 2

Models

The models are the classes used to build, train and test a Gaussian process, and to then build the relative mapped potential. There are six types of models at the moment, each one is used to handle 2-, 3-, or 2+3-body kernels in the case of one or two atomic species. When creating a model, it is therefore necessary to decide a priori the type of Gaussian process and, therefore, the type of mapped potential we want to obtain.

2.1 Building a model

To create a model based on a 2-body kernel for a monoatomic system:

```
from mff import models
mymodel = models.TwoBodySingleSpecies(atomic_number, cutoff_radius, sigma, theta,␣
↪noise)
```

where the parameters refer to the atomic number of the species we are training the GP on, the cutoff radius we want to use, the lengthscale hyperparameter of the Gaussian Process, the hyperparameter governing the exponential decay of the cutoff function, and the noise associated with the output training data. In the case of a 2+3-body kernel for a monoatomic system:

```
from mff import models
mymodel = models.CombinedSingleSpecies(atomic_number, cutoff_radius, sigma_2b, theta_
↪2b, sigma_3b, theta_3b, noise)
```

where we have two additional hyperparameters since the lengthscale value and the cutoff decay ratio of the 2- and 3-body kernels contained inside the combined Gaussian Process are be independent.

When dealing with a two-element system, the syntax is very similar, but the `atomic_number` is instead a list containing the atomic numbers of the two species, in increasing order:

```
from mff import models
mymodel = models.CombinedTwoSpecies(atomic_numbers, cutoff_radius, sigma_2b, theta_2b,
↪ sigma_3b, theta_3b, noise)
```

2.2 Fitting the model

Once a model has been built, we can train it using a dataset of forces, energies, or energies and forces, that has been created using the `configurations` module. If we are training only on forces:

```
mymodel.fit(training_confs, training_forces)
```

training only on energies:

```
mymodel.fit_energy(training_confs, training_energies)
```

training on both forces and energies:

```
mymodel.fit_force_and_energy(training_confs, training_forces, training_energies)
```

Additionaly, the argument `nnodes` can be passed to any fit function in order to run the process on multiple processors:

```
mymodel.fit(training_confs, training_forces, nnodes = 4)
```

2.3 Predicting forces and energies with the GP

Once the Gaussian process has been fitted, it can be used to directly predict forces and energies on test configurations. To predict the force and the energy for a single test configuration:

```
force = mymodel.predict(test_configuration)
energy = mymodel.predict_energy(test_configuration)
```

the boolean variable `return_std` can be passed to the force and energy predict functions in order to obtain also the standard deviation associated with the prediction, default is False:

```
mean_force, std_force = mymodel.predict(test_configuration, return_std = True)
```

2.4 Building a mapped potential

Once the Gaussian process has been fitted, either via force, energy or joint force/energy fit, it can be mapped onto a non-parametric 2- and/or 3-body potential using the `build_grid` function. The `build_grid` function takes as arguments the minimum grid distance (smallest distance between atoms for which the potential will be defined), the number of grid points to use while building the 2-body mapped potential, and the number of points per dimension to use while building the 3-body mapped potential. For a 2-body model:

```
mymodel.build_grid(grid start, num_2b)
```

For a 3-body model:

```
mymodel.build_grid(grid start, num_3b)
```

For a combined model:

```
mymodel.build_grid(grid start, num_2b, num_3b)
```

Additionaly, the argument `nnodes` can be passed to the `build_grid` function for any model in order to run the process on multiple processors:

```
mymodel.build_grid(grid start, num_2b, num_3b, nnodes = 4)
```

2.5 Saving and loading a model

At any stage, a model can be saved using the `save` function that takes a .json filename as the only input:

```
mymodel.save("thismodel.json")
```

the save function will create a .json file containing all of the parameters and hyperparameters of the model, and the paths to the .npy and .npz files containing, respectively, the saved GPs and the saved mapped potentials, which are also created by the save funtion.

To load a previously saved model of a known type (here for example a CombinedSingleSpecies model) simply run:

```
mymodel = models.CombinedSingleSpecies.from_json("thismodel.json")
```

2.6 Model's complete reference

2.7 Two Body Model

Module containing the TwoBodySingleSpecies and TwoBodyTwoSpecies classes, which are used to handle the Gaussian process and the mapping algorithm used to build M-FFs. The model has to be first defined, then the Gaussian process must be trained using training configurations and forces (and/or energies). Once a model has been trained, it can be used to predict forces (and/or energies) on unknonwn atomic configurations. A trained Gaussian process can then be mapped onto a tabulated 2-body potential via the `build grid` function call. A mapped model can be then saved, loaded and used to run molecular dynamics simulations via the calculator module. These mapped potentials retain the accuracy of the GP used to build them, while speeding up the calculations by a factor of 10^4 in typical scenarios.

Example:

```
from mff import models
mymodel = models.TwoBodySingleSpecies(atomic_number, cutoff_radius, sigma, theta,␣
↪noise)
mymodel.fit(training_confs, training_forces)

forces = mymodel.predict(test_configurations)

mymodel.build_grid(grid_start, num_2b)
mymodel.save("thismodel.json")

mymodel = models.TwoBodySingleSpecies.from_json("thismodel.json")
```

class mff.models.twobody.**TwoBodySingleSpeciesModel**(*element*, *r_cut*, *sigma*, *theta*, *noise*, ***kwargs*)

2-body single species model class Class managing the Gaussian process and its mapped counterpart

Parameters

- **element** (*int*) – The atomic number of the element considered
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments

- **sigma** (*float*) – Lengthscale parameter of the Gaussian process
- **theta** (*float*) – decay ratio of the cutoff function in the Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp
method – The 2-body single species Gaussian Process

grid
method – The 2-body single species tabulated potential

grid_start
float – Minimum atomic distance for which the grid is defined (cannot be 0.0)

grid_num
int – number of points used to create the 2-body grid

build_grid(*start*, *num*)
Build the mapped 2-body potential. Calculates the energy predicted by the GP for two atoms at distances that range from start to r_cut, for a total of num points. These energies are stored and a 1D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any couple of atoms. The total force or local energy can then be calculated for any atom by summing the pairwise contributions of every other atom within a cutoff distance r_cut. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 2-body mapped potential
- **num** (*int*) – number of points to use in the grid of the mapped potential

fit(*confs*, *forces*, *nnodes=1*)
Fit the GP to a set of training forces using a 2-body single species force-force kernel

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy(*confs*, *energies*, *nnodes=1*)
Fit the GP to a set of training energies using a 2-body single species energy-energy kernel

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy(*confs*, *forces*, *energies*, *nnodes=1*)
Fit the GP to a set of training forces and energies using 2-body single species force-force, energy-force and energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

classmethod from_json(*path*)
Load the model. Loads the model, the associated GP and the mapped potential, if available.

Parameters **path** (*str*) – path to the .json model file

Returns the model object

Return type model (obj)

load_gp(*filename*)
Loads the GP object, now obsolete

predict(*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

predict_energy(*confs*, *return_std=False*)
Predict the global energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

predict_energy_map(*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save(*path*)
Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potential, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp(*filename*)
Saves the GP object, now obsolete

update_energy(*confs*, *energies*, *nnodes=1*)
Update a fitted GP with a set of energies and using 2-body single species energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force(*confs*, *forces*, *nnodes=1*)
Update a fitted GP with a set of forces and using 2-body single species force-force kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

class mff.models.twobody.**TwoBodyTwoSpeciesModel**(*elements*, *r_cut*, *sigma*, *theta*, *noise*, ***kwargs*)
2-body two species model class Class managing the Gaussian process and its mapped counterpart

Parameters

- **elements** (*list*) – List containing the atomic numbers in increasing order
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments
- **sigma** (*foat*) – Lengthscale parameter of the Gaussian process
- **theta** (*float*) – decay ratio of the cutoff function in the Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp
class – The 2-body two species Gaussian Process

grid
list – Contains the three 2-body two species tabulated potentials, accounting for interactions between two atoms of types 0-0, 0-1, and 1-1.

grid_start
float – Minimum atomic distance for which the grid is defined (cannot be 0)

grid_num
int – number of points used to create the 2-body grids

build_grid(*start*, *num*)
Build the mapped 2-body potential. Calculates the energy predicted by the GP for two atoms at distances that range from start to r_cut, for a total of num points. These energies are stored and a 1D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any couple of atoms. The total force or local energy can then be calculated for any atom by summing the pairwise contributions of every other atom within a cutoff distance r_cut. Three distinct potentials are built for interactions between atoms of type 0 and 0, type 0 and 1, and type 1 and 1. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 2-body mapped potential
- **num** (*int*) – number of points to use in the grid of the mapped potential

fit(*confs*, *forces*, *nnodes=1*)
Fit the GP to a set of training forces using a two body two species force-force kernel

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy(*confs*, *energy*, *nnodes=1*)
Fit the GP to a set of training energies using a two body two species energy-energy kernel

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy(*confs*, *forces*, *energy*, *nnodes=1*)
Fit the GP to a set of training forces and energies using two body two species force-force, energy-force and energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one

- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

classmethod from_json(*path*)
Load the models. Loads the model, the associated GP and the mapped potential, if available.

Parameters **path** (*str*) – path to the .json model file

Returns the model object

Return type model (obj)

load_gp(*filename*)
Loads the GP object, now obsolete

predict(*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

predict_energy(*confs*, *return_std=False*)
Predict the global energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

predict_energy_map(*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one

- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save (*path*)
Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potentials, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp (*filename*)
Saves the GP object, now obsolete

update_energy (*confs*, *energies*, *nnodes=1*)
Update a fitted GP with a set of energies and using 2-body two species energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force (*confs*, *forces*, *nnodes=1*)
Update a fitted GP with a set of forces and using 2-body two species force-force kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

2.8 Three Body Model

Module containing the ThreeBodySingleSpecies and ThreeBodyTwoSpecies classes, which are used to handle the Gaussian process regression and the mapping algorithm used to build M-FFs. The model has to be first defined, then the Gaussian process must be trained using training configurations and forces (and/or local energies). Once a model has been trained, it can be used to predict forces (and/or energies) on unknonwn atomic configurations. A trained Gaussian process can then be mapped onto a tabulated 3-body potential via the `build grid` function call. A mapped model can be then saved, loaded and used to run molecular dynamics simulations via the calculator module. These mapped potentials retain the accuracy of the GP used to build them, while speeding up the calculations by a factor of 10^4 in typical scenarios.

Example:

```
from mff import models
mymodel = models.ThreeBodySingleSpecies(atomic_number, cutoff_radius, sigma, theta,␣
↪noise)
mymodel.fit(training_confs, training_forces)
forces = mymodel.predict(test_configurations)
mymodel.build_grid(grid_start, num_3b)
mymodel.save("thismodel.json")
mymodel = models.CombinedSingleSpecies.from_json("thismodel.json")
```

class mff.models.threebody.**ThreeBodySingleSpeciesModel**(*element*, *r_cut*, *sigma*, *theta*, *noise*, ***kwargs*)

3-body single species model class Class managing the Gaussian process and its mapped counterpart

Parameters

- **element** (*int*) – The atomic number of the element considered
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments
- **sigma** (*foat*) – Lengthscale parameter of the Gaussian process
- **theta** (*float*) – decay ratio of the cutoff function in the Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp
method – The 3-body single species Gaussian Process

grid
method – The 3-body single species tabulated potential

grid_start
float – Minimum atomic distance for which the grid is defined (cannot be 0.0)

grid_num
int – number of points per side used to create the 3-body grid. This is a 3-dimensional grid, therefore the total number of grid points will be grid_num^3.

build_grid(*start*, *num*, *nnodes=1*)
Build the mapped 3-body potential. Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The grid building exploits all the permutation invariances to reduce the number of energy calculations needed to fill the grid. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 3-body mapped potential
- **num** (*int*) – number of points to use to generate the list of distances used to generate the triplets of atoms for the mapped potential
- **nnodes** (*int*) – number of CPUs to use to calculate the energy predictions

fit(*confs*, *forces*, *nnodes=1*)
Fit the GP to a set of training forces using a 3-body single species force-force kernel function

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy(*confs*, *energies*, *nnodes=1*)

Fit the GP to a set of training energies using a 3-body single species energy-energy kernel function

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy(*confs*, *forces*, *energies*, *nnodes=1*)

Fit the GP to a set of training forces and energies using 3-body single species force-force, energy-force and energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

static generate_triplets(*dists*)

Generate a list of all valid triplets using perutational invariance. Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The grid building exploits all the permutation invariances to reduce the number of energy calculations needed to fill the grid. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)

Returns

array of booleans indicating which triplets (three distance values) need to be evaluated to fill the 3D grid of energy values.

r_ij_x (array): array containing the x coordinate of the second atom j w.r.t. the central atom i
r_ki_x (array): array containing the x coordinate of the third atom k w.r.t. the central atom i
r_ki_y (array): array containing the y coordinate of the third atom k w.r.t. the central atom i

Return type inds (array)

load_gp (*filename*)
Loads the GP object, now obsolete

predict (*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

predict_energy (*confs*, *return_std=False*)
Predict the global energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

predict_energy_map (*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save(*path*)
Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potential, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp(*filename*)
Saves the GP object, now obsolete

update_energy(*confs*, *energies*, *nnodes=1*)
Update a fitted GP with a set of energies and using 3-body single species energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force(*confs*, *forces*, *nnodes=1*)
Update a fitted GP with a set of forces and using 3-body single species force-force kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

class mff.models.threebody.**ThreeBodyTwoSpeciesModel**(*elements*, *r_cut*, *sigma*, *theta*, *noise*, ***kwargs*)
3-body two species model class Class managing the Gaussian process and its mapped counterpart

Parameters

- **elements** (*list*) – List containing the atomic numbers in increasing order
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments
- **sigma** (*foat*) – Lengthscale parameter of the Gaussian process
- **theta** (*float*) – decay ratio of the cutoff function in the Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp
class – The 3-body two species Gaussian Process

grid
list – Contains the three 3-body two species tabulated potentials, accounting for interactions between three atoms of types 0-0-0, 0-0-1, 0-1-1, and 1-1-1.

grid_start
float – Minimum atomic distance for which the grid is defined (cannot be 0)

grid_num
int – number of points per side used to create the 3-body grids. These are 3-dimensional grids, therefore the total number of grid points will be grid_num^3.

build_grid(*start*, *num*, *nnodes=1*)
Function used to create the four different 3-body energy grids for atoms of elements 0-0-0, 0-0-1, 0-1-1, and 1-1-1. The function calls the `build_grid_3b` function for each of those combinations of elements.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 3-body mapped potential
- **num** (*int*) – number of points to use to generate the list of distances used to generate the triplets of atoms for the mapped potential
- **nnodes** (*int*) – number of CPUs to use to calculate the energy predictions

build_grid_3b(*dists*, *element_i*, *element_j*, *element_k*, *nnodes*)
Build a mapped 3-body potential. Calculates the energy predicted by the GP for three atoms of elements element_i, element_j, element_k, at all possible combinations of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality), found by calling the `generate_triplets_with_permutation_invariance` function. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters

- **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)
- **element_i** (*int*) – atomic number of the central atom i in a triplet
- **element_j** (*int*) – atomic number of the second atom j in a triplet
- **element_k** (*int*) – atomic number of the third atom k in a triplet
- **nnodes** (*int*) – number of CPUs to use when computing the triplet local energies

Returns

a 3D spline object that can be used to predict the energy and the force associated to the central atom of a triplet.

Return type spline3D (obj)

fit(*confs*, *forces*, *nnodes=1*)
Fit the GP to a set of training forces using a 3-body two species force-force kernel function

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy(*confs*, *forces*, *nnodes=1*)
Fit the GP to a set of training energies using a 3-body two species energy-energy kernel function

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy(*confs*, *forces*, *energies*, *nnodes=1*)

Fit the GP to a set of training forces and energies using 3-body two species force-force, energy-force and energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

classmethod from_json(*path*)

Load the model. Loads the model, the associated GP and the mapped potential, if available.

Parameters **path** (*str*) – path to the .json model file

Returns the model object

Return type model (obj)

static generate_triplets_all(*dists*)

Generate a list of all valid triplets Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)

Returns

array of booleans indicating which triplets (three distance values) need to be evaluated to fill the 3D grid of energy values.

r_ij_x (array): array containing the x coordinate of the second atom j w.r.t. the central atom i
r_ki_x (array): array containing the x coordinate of the third atom k w.r.t. the central atom i
r_ki_y (array): array containing the y coordinate of the third atom k w.r.t. the central atom i

Return type inds (array)

static generate_triplets_with_permutation_invariance(*dists*)

Generate a list of all valid triplets using perutational invariance. Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The grid building exploits all the permutation invariances to reduce

the number of energy calculations needed to fill the grid. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters **`dists`** (*`array`*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)

Returns

array of booleans indicating which triplets (three distance values) need to be evaluated to fill the 3D grid of energy values.

r_ij_x (array): array containing the x coordinate of the second atom j w.r.t. the central atom i
r_ki_x (array): array containing the x coordinate of the third atom k w.r.t. the central atom i
r_ki_y (array): array containing the y coordinate of the third atom k w.r.t. the central atom i

Return type inds (array)

`load_gp` (*filename*)
Loads the GP object, now obsolete

`predict` (*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using a GP

Parameters

- **`confs`** (*`list`*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **`return_std`** (*`bool`*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

`predict_energy` (*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using a GP

Parameters

- **`confs`** (*`list`*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **`return_std`** (*`bool`*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

predict_energy_map(*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using a GP

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GP energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save(*path*)
Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potentials, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp(*filename*)
Saves the GP object, now obsolete

update_energy(*confs*, *energies*, *nnodes=1*)
Update a fitted GP with a set of energies and using 3-body two species energy-energy kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force(*confs*, *forces*, *nnodes=1*)
Update a fitted GP with a set of forces and using 3-body twp species force-force kernels

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

2.9 Combined Model

Module that uses 2- and 3-body kernels to do Guassian process regression, and to build 2- and 3-body mapped potentials. The model has to be first defined, then the Gaussian processes must be trained using training configurations and forces (and/or energies). Once a model has been trained, it can be used to predict forces (and/or energies) on unknonwn atomic configurations. A trained Gaussian process can then be mapped onto a tabulated 2-body potential and a tabultaed 3-body potential via the `build grid` function call. A mapped model can be thensaved, loaded and

used to run molecular dynamics simulations via the calculator module. These mapped potentials retain the accuracy of the GP used to build them, while speeding up the calculations by a factor of 10^4 in typical scenarios.

Example:

```
from mff import models
mymodel = models.CombinedSingleSpecies(atomic_number, cutoff_radius,
                      sigma_2b, sigma_3b, sigma_2b, theta_3b, noise)
mymodel.fit(training_confs, training_forces)
forces = mymodel.predict(test_configurations)
mymodel.build_grid(grid_start, num_2b)
mymodel.save("thismodel.json")
mymodel = models.CombinedSingleSpecies.from_json("thismodel.json")
```

class mff.models.combined.**CombinedSingleSpeciesModel**(*element*, *r_cut*, *sigma_2b*, *sigma_3b*, *theta_2b*, *theta_3b*, *noise*, ***kwargs*)

2- and 3-body single species model class Class managing the Gaussian processes and their mapped counterparts

Parameters

- **element** (*int*) – The atomic number of the element considered
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments
- **sigma_2b** (*foat*) – Lengthscale parameter of the 2-body Gaussian process
- **sigma_3b** (*foat*) – Lengthscale parameter of the 2-body Gaussian process
- **theta_2b** (*float*) – decay ratio of the cutoff function in the 2-body Gaussian Process
- **theta_3b** (*float*) – decay ratio of the cutoff function in the 3-body Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp_2b
method – The 2-body single species Gaussian Process

gp_3b
method – The 3-body single species Gaussian Process

grid_2b
method – The 2-body single species tabulated potential

grid_3b
method – The 3-body single species tabulated potential

grid_start
float – Minimum atomic distance for which the grids are defined (cannot be 0.0)

grid_num
int – number of points per side used to create the 2- and 3-body grid. The 3-body grid is 3-dimensional, therefore its total number of grid points will be grid_num^3

build_grid(*start*, *num_2b*, *num_3b*, *nnodes=1*)
Build the mapped 2- and 3-body potentials. Calculates the energy predicted by the GP for two and three atoms at all possible combination of num distances ranging from start to r_cut. The energy for the 3-body mapped grid is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The grid building exploits all the permutation invariances to reduce the number of energy calculations needed to fill the grid. The computed 2-body energies are stored in an array of values, and a 1D spline interpolation is created. The computed 3-body energies are stored in a 3D cube of values, and a 3D spline interpolation is created. The total force or local energy can then be calculated for any atom by summing the pairwise and triplet contributions of every

valid couple and triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module, which is built to work within the ase python package.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 3-body mapped potential
- **num_2b** (*int*) – number of points to use in the grid of the 2-body mapped potential
- **num_3b** (*int*) – number of points to use to generate the list of distances used to generate the triplets of atoms for the 2-body mapped potential
- **nnodes** (*int*) – number of CPUs to use to calculate the energy predictions

fit (*confs*, *forces*, *nnodes=1*)

Fit the GP to a set of training forces using a 2- and 3-body single species force-force kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of force on the training configurations

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy (*confs*, *energies*, *nnodes=1*)

Fit the GP to a set of training energies using a 2- and 3-body single species energy-energy kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training energies and the 2-body predictions of energies on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy (*confs*, *forces*, *energies*, *nnodes=1*)

Fit the GP to a set of training energies using a 2- and 3-body single species force-force, energy-energy, and energy-forces kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training energies (and forces) and the 2-body predictions of energies (and forces) on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

classmethod from_json(*path*)
Load the model. Loads the model, the associated GPs and the mapped potentials, if available.

Parameters **path** (*str*) – path to the .json model file

Returns the model object

Return type model (obj)

static generate_triplets(*dists*)
Generate a list of all valid triplets using perutational invariance. Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The grid building exploits all the permutation invariances to reduce the number of energy calculations needed to fill the grid. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)

Returns

array of booleans indicating which triplets (three distance values) need to be evaluated to fill the 3D grid of energy values.

r_ij_x (array): array containing the x coordinate of the second atom j w.r.t. the central atom i
r_ki_x (array): array containing the x coordinate of the third atom k w.r.t. the central atom i
r_ki_y (array): array containing the y coordinate of the third atom k w.r.t. the central atom i

Return type inds (array)

load_gp(*filename_2b*, *filename_3b*)
Loads the GP objects, now obsolete

predict(*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using the 2- and 3-body GPs. The total force is the sum of the two predictions.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GPs forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

predict_energy(*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using the 2- and 3-body GPs. The total force is the sum of the two predictions.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GPs energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save_combined(*path*)
Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potentials, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp(*filename_2b*, *filename_3b*)
Saves the GP objects, now obsolete

update_energy(*confs*, *energies*, *nnodes=1*)
Update a fitted GP with a set of training energies using a 2- and 3-body single species force-force kernel functions. The 2-body Gaussian process is first updated, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of forces on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force(*confs*, *forces*, *nnodes=1*)
Update a fitted GP with a set of training energies using a 2- and 3-body single species force-force kernel functions. The 2-body Gaussian process is first updated, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of forces on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

class mff.models.combined.**CombinedTwoSpeciesModel**(*elements*, *r_cut*, *sigma_2b*, *sigma_3b*, *theta_2b*, *theta_3b*, *noise*, ***kwargs*)
2- and 3-body two species model class Class managing the Gaussian processes and their mapped counterparts

Parameters

- **elements** (*list*) – List containing the atomic numbers in increasing order
- **r_cut** (*foat*) – The cutoff radius used to carve the atomic environments
- **sigma_2b** (*foat*) – Lengthscale parameter of the 2-body Gaussian process

- **sigma_3b** (*foat*) – Lengthscale parameter of the 2-body Gaussian process
- **theta_2b** (*float*) – decay ratio of the cutoff function in the 2-body Gaussian Process
- **theta_3b** (*float*) – decay ratio of the cutoff function in the 3-body Gaussian Process
- **noise** (*float*) – noise value associated with the training output data

gp_2b
method – The 2-body single species Gaussian Process

gp_3b
method – The 3-body single species Gaussian Process

grid_2b
list – Contains the three 2-body two species tabulated potentials, accounting for interactions between two atoms of types 0-0, 0-1, and 1-1.

grid_2b
list – Contains the three 3-body two species tabulated potentials, accounting for interactions between three atoms of types 0-0-0, 0-0-1, 0-1-1, and 1-1-1.

grid_start
float – Minimum atomic distance for which the grids are defined (cannot be 0.0)

grid_num_2b
int – number of points to use in the grid of the 2-body mapped potential

grid_num_3b
int – number of points to use to generate the list of distances used to generate the triplets of atoms for the 2-body mapped potential

build_grid(*start*, *num_2b*, *num_3b*, *nnodes=1*)
Function used to create the three different 2-body energy grids for atoms of elements 0-0, 0-1, and 1-1, and the four different 3-body energy grids for atoms of elements 0-0-0, 0-0-1, 0-1-1, and 1-1-1. The function calls the `build_grid_3b` function for each of the 3-body grids to build.

Parameters

- **start** (*float*) – smallest interatomic distance for which the energy is predicted by the GP and stored inn the 3-body mapped potential
- **num** (*int*) – number of points to use in the grid of the 2-body mapped potentials
- **num_3b** (*int*) – number of points to use to generate the list of distances used to generate the triplets of atoms for the 3-body mapped potentials
- **nnodes** (*int*) – number of CPUs to use to calculate the energy predictions

build_grid_3b(*dists*, *element_k*, *element_i*, *element_j*, *nnodes*)
Build a mapped 3-body potential. Calculates the energy predicted by the GP for three atoms of elements element_i, element_j, element_k, at all possible combinations of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality), found by calling the `generate_triplets_with_permutation_invariance` function. The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters

- **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)
- **element_i** (*int*) – atomic number of the central atom i in a triplet
- **element_j** (*int*) – atomic number of the second atom j in a triplet
- **element_k** (*int*) – atomic number of the third atom k in a triplet
- **nnodes** (*int*) – number of CPUs to use when computing the triplet local energies

Returns

a 3D spline object that can be used to predict the energy and the force associated to the central atom of a triplet.

Return type spline3D (obj)

fit(*confs*, *forces*, *nnodes=1*)

Fit the GP to a set of training forces using a 2- and 3-body two species force-force kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of force on the training configurations

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_energy(*confs*, *energies*, *nnodes=1*)

Fit the GP to a set of training energies using a 2- and 3-body two species energy-energy kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training energies and the 2-body predictions of energies on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

fit_force_and_energy(*confs*, *forces*, *energies*, *nnodes=1*)

Fit the GP to a set of training energies using a 2- and 3-body two species force-force, energy-energy, and energy-forces kernel functions. The 2-body Gaussian process is first fitted, then the 3-body GP is fitted to the difference between the training energies (and forces) and the 2-body predictions of energies (and forces) on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **energies** (*array*) – Array containing the scalar local energies of the central atoms of the training configurations

- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

classmethod from_json(*path*)
Load the model. Loads the model, the associated GPs and the mapped potentials, if available.

Parameters **path** (*str*) – path to the .json model file

Returns the model object

Return type model (obj)

static generate_triplets_all(*dists*)
Generate a list of all valid triplets. Calculates the energy predicted by the GP for three atoms at all possible combination of num distances ranging from start to r_cut. The energy is calculated only for `valid` triplets of atoms, i.e. sets of three distances which form a triangle (this is checked via the triangle inequality). The computed energies are stored in a 3D cube of values, and a 3D spline interpolation is created, which can be used to predict the energy and, through its analytic derivative, the force associated to any triplet of atoms. The total force or local energy can then be calculated for any atom by summing the triplet contributions of every valid triplet of atoms of which one is always the central one. The prediction is done by the `calculator` module which is built to work within the ase python package.

Parameters **dists** (*array*) – array of floats containing all of the distances which can be used to build triplets of atoms. This array is created by calling np.linspace(start, r_cut, num)

Returns

array of booleans indicating which triplets (three distance values) need to be evaluated to fill the 3D grid of energy values.

r_ij_x (array): array containing the x coordinate of the second atom j w.r.t. the central atom i
r_ki_x (array): array containing the x coordinate of the third atom k w.r.t. the central atom i
r_ki_y (array): array containing the y coordinate of the third atom k w.r.t. the central atom i

Return type inds (array)

load_gp(*filename_2b*, *filename_3b*)
Loads the GP objects, now obsolete

predict(*confs*, *return_std=False*)
Predict the forces acting on the central atoms of confs using the 2- and 3-body GPs. The total force is the sum of the two predictions.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GPs forces_errors (array): errors associated to the force predictions,

returned only if return_std is True

Return type forces (array)

predict_energy(*confs*, *return_std=False*)
Predict the local energies of the central atoms of confs using the 2- and 3-body GPs. The total force is the sum of the two predictions.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **return_std** (*bool*) – if True, returns the standard deviation associated to predictions according to the GP framework

Returns

array of force vectors predicted by the GPs energies_errors (array): errors associated to the energies predictions,

returned only if return_std is True

Return type energies (array)

save_combined(*path*)

Save the model. This creates a .json file containing the parameters of the model and the paths to the GP objects and the mapped potentials, which are saved as separate .gpy and .gpz files, respectively.

Parameters **path** (*str*) – path to the file

save_gp(*filename_2b*, *filename_3b*)

Saves the GP objects, now obsolete

update_energy(*confs*, *energies*, *nnodes=1*)

Update a fitted GP with a set of training energies using a 2- and 3-body two species force-force kernel functions. The 2-body Gaussian process is first updated, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of forces on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

update_force(*confs*, *forces*, *nnodes=1*)

Update a fitted GP with a set of training energies using a 2- and 3-body two species force-force kernel functions. The 2-body Gaussian process is first updated, then the 3-body GP is fitted to the difference between the training forces and the 2-body predictions of forces on the training configurations.

Parameters

- **confs** (*list*) – List of M x 5 arrays containing coordinates and atomic numbers of atoms within a cutoff from the central one
- **forces** (*array*) – Array containing the vector forces on the central atoms of the training configurations
- **nnodes** (*int*) – number of CPUs to use for the gram matrix evaluation

CHAPTER 3

Configurations

The MFF package uses training and testing data extracted from .xyz files. The `mff.configurations` module contains the function `carve_confs` which is used to save .npy files containing local atomic environments, the forces acting on the central atoms of these local atomic environments and, if present, the energy associated with the snapshot the local environment has been extracted from. To extract local atomic environments, forces, energies and a list of all the elements contained in an ase `atoms` object:

```
from ase.io import read
from mff.configurations import carve_confs
traj = read(filename, format='extxyz')
elements, confs, forces, energies = carve_confs(traj, r_cut, n_data)
```

where `r_cut` specifies the cutoff radius that will be applied to extract local atomic environments containing all atomis within r_cut from the central one, and n_data specifies the total number of local atomic environments to extract.

class mff.configurations.**Configurations**(*confs=None*)
Configurations can represent a list of configurations

class mff.configurations.**ConfsTwoBodySingleForces**(*r_cut*)

class mff.configurations.**ConfsTwoForces**(*r_cut*)

class mff.configurations.**Energies**(*confs=None*, *energy=None*)

class mff.configurations.**Forces**(*confs=None*, *forces=None*)

exception mff.configurations.**MissingData**

mff.configurations.**carve_2body_confs**(*atoms*, *r_cut*, *nbins=100*, *forces_label=None*, *energy_label=None*)
Extract atomic configurations, the forces acting on the central atoms os said configurations, and the local energy values associeated. This function is optimised to get configurations that contain a diverse set of interatomic distances.

Parameters

- **atoms** (*ase atoms object*) – Ase trajectory file, opened with ase.io.read
- **r_cut** (*float*) – Cutoff to use when carving out atomic environments

- **nbins** (*int*) – number of bins used to sample the distance values. An atomic configuration is selected only if it contains at least a bond which length falls in an unoccupied bin, so that the final database will contain the most diverse set of bond lengths possible.
- **forces_label** (*str*) – Name of the force label in the trajectory file, if None default is "forces"
- **energy_label** (*str*) – Name of the energy label in the trajectory file, if None default is "energy"

Returns

Array of atomic numbers in increasing order confs (list of arrays): List of M by 5 numpy arrays, where M is the number of atoms within

r_cut from the central one. The first 3 components are positions w.r.t the central atom in Angstroms, the fourth is the atomic number of the central atom, the fifth the atomic number of each atom.

forces (array): x,y,z components of the force acting on the central atom in eV/Angstrom energies (array): value of the local atomic energy in eV

Return type elements (array)

mff.configurations.**carve_3body_confs**(*atoms*, *r_cut*, *nbins=50*, *forces_label=None*, *energy_label=None*)

Extract atomic configurations, the forces acting on the central atoms os said configurations, and the local energy values associeated. This function is optimised to get configurations that contain a diverse set of interatomic distances and angles.

Parameters

- **atoms** (*ase atoms object*) – Ase trajectory file, opened with ase.io.read
- **r_cut** (*float*) – Cutoff to use when carving out atomic environments
- **nbins** (*int*) – number of bins used to sample the distance values. An atomic configuration is selected only if it contains at least a triplet three distances fall in an unoccupied bin, so that the final database will contain the most diverse set of triplets possible.
- **forces_label** (*str*) – Name of the force label in the trajectory file, if None default is "forces"
- **energy_label** (*str*) – Name of the energy label in the trajectory file, if None default is "energy"

Returns

Array of atomic numbers in increasing order confs (list of arrays): List of M by 5 numpy arrays, where M is the number of atoms within

r_cut from the central one. The first 3 components are positions w.r.t the central atom in Angstroms, the fourth is the atomic number of the central atom, the fifth the atomic number of each atom.

forces (array): x,y,z components of the force acting on the central atom in eV/Angstrom energies (array): value of the local atomic energy in eV

Return type elements (array)

mff.configurations.**carve_confs**(*atoms*, *r_cut*, *n_data*, *forces_label=None*, *energy_label=None*, *boundaries=None*)

Extract atomic configurations, the forces acting on the central atoms os said configurations, and the local energy values associeated.

Parameters

- **atoms** (*ase atoms object*) – Ase trajectory file, opened with ase.io.read
- **r_cut** (*float*) – Cutoff to use when carving out atomic environments
- **n_data** (*int*) – Total number of atomic configurations to extract from the trajectory
- **forces_label** (*str*) – Name of the force label in the trajectory file, if None default is "forces"
- **energy_label** (*str*) – Name of the energy label in the trajectory file, if None default is "energy"
- **boundaries** (*list*) – List containing three lists for the three cartesian coordinates. Each of them contains a list of tuples indicating every interval that must be used to sample central atoms from. Example: boundaries = [[], [], [[-10.0, +5.3]]] Default is None, and all of the snapshot is used.

Returns

Array of atomic numbers in increasing order confs (list of arrays): List of M by 5 numpy arrays, where M is the number of atoms within

r_cut from the central one. The first 3 components are positions w.r.t the central atom in Angstroms, the fourth is the atomic number of the central atom, the fifth the atomic number of each atom.

forces (array): x,y,z components of the force acting on the central atom in eV/Angstrom energies (array): value of the local atomic energy in eV

Return type elements (array)

mff.configurations.**carve_from_snapshot**(*atoms*, *atoms_ind*, *r_cut*, *forces_label=None*, *energy_label=None*)

Extract atomic configurations, the forces acting on the central atoms os said configurations, and the local energy values associated to a single atoms object.

Parameters

- **atoms** (*ase atoms object*) – Ase atoms file, opened with ase.io.read
- **atoms_ind** (*list*) – indexes of the atoms for which a conf is created
- **r_cut** (*float*) – Cutoff to use when carving out atomic environments
- **forces_label** (*str*) – Name of the force label in the trajectory file, if None default is "forces"
- **energy_label** (*str*) – Name of the energy label in the trajectory file, if None default is "energy"

Returns

List of M by 5 numpy arrays, where M is the number of atoms within r_cut from the central one. The first 3 components are positions w.r.t the central atom in Angstroms, the fourth is the atomic number of the central atom, the fifth the atomic number of each atom.

forces (array): x,y,z components of the force acting on the central atom in eV/Angstrom energies (array): value of the local atomic energy in eV

Return type confs (list of arrays)

CHAPTER 4

Gaussian Processes

Gaussian process regression module suited to learn and predict energies and forces

Example:

```
gp = GaussianProcess(kernel, noise)
gp.fit(train_configurations, train_forces)
gp.predict(test_configurations)
```

class mff.gp.**GaussianProcess**(*kernel=None*, *noise=1e-10*, *optimizer=None*, *n_restarts_optimizer=0*)

Gaussian process class Class of GP regression of QM energies and forces

Parameters

- **kernel** (*obj*) – A kernel object (typically a two or three body)
- **noise** (*float*) – The regularising noise level (typically named sigma_n^2)
- **optimizer** (*str*) – The kind of optimization of marginal likelihood (not implemented yet)

X_train_
list – The configurations used for training

alpha_
array – The coefficients obtained during training

L_
array – The lower triangular matrix from cholesky decomposition of gram matrix

K
array – The kernel gram matrix

calc_gram_ee(*X*)
Calculate the force-force kernel gram matrix

Parameters **X** (*list*) – list of N training configurations, which are M x 5 matrices

Returns The energy energy gram matrix, has dimensions N x N

Return type K (matrix)

calc_gram_ff(*X*)
Calculate the force-force kernel gram matrix

Parameters **X** (*list*) – list of N training configurations, which are M x 5 matrices

Returns The force-force gram matrix, has dimensions 3N x 3N

Return type K (matrix)

fit(*X*, *y*, *nnodes=1*)
Fit a Gaussian process regression model on training forces

Parameters

- **X** (*list*) – training configurations
- **y** (*np.ndarray*) – training forces
- **nnodes** (*int*) – number of CPU workers to use, default is 1

fit_energy(*X*, *y*, *nnodes=1*)
Fit a Gaussian process regression model using local energies.

Parameters

- **X** (*list*) – training configurations
- **y** (*np.ndarray*) – training energies
- **nnodes** (*int*) – number of CPU workers to use, default is 1
- **energy of each configuration is E/N where E is the total** (*The*) –
- **energy and N the atoms in that snapshot** (*snapshot*) –

fit_force_and_energy(*X*, *y_force*, *y_energy*, *nnodes=1*)
Fit a Gaussian process regression model using forces and energies

Parameters

- **X** (*list*) – training configurations
- **y_force** (*np.ndarray*) – training forces
- **y_energy** (*np.ndarray*) – training local energies
- **nnodes** (*int*) – number of CPU workers to use, default is 1

fit_update(*X2_up*, *y2_up*, *nnodes=1*)
Update an existing energy-energy gram matrix with a list of new datapoints

Parameters

- **X2_up** (*list*) – training configurations
- **y2_up** (*np.ndarray*) – training forces
- **nnodes** (*int*) – number of CPU workers to use, default is 1

fit_update_energy(*X2_up*, *y2_up*, *nnodes=1*)
Update an existing energy-energy gram matrix with a list of new datapoints

Parameters

- **X2_up** (*list*) – training configurations

- **y2_up** (*np.ndarray*) – training energies
- **nnodes** (*int*) – number of CPU workers to use, default is 1

fit_update_single (*X2_up*, *y2_up*, *nnodes=1*)
Update an existing force-force gram matrix with a single new datapoint

fit_update_single_energy (*X2_up*, *y2_up*, *nnodes=1*)
Update an existing energy-energy gram matrix with a single new datapoint

load (*filename*)
Load a saved GP model

Parameters **filename** (*str*) – name of the file where the GP is saved

log_marginal_likelihood (*theta=None*, *eval_gradient=False*)
Returns log-marginal likelihood of theta for training data.

Parameters

- **theta** – array-like, shape = (n_kernel_params,) or None Kernel hyperparameters for which the log-marginal likelihood is evaluated. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.
- **eval_gradient** – bool, default: False If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. If True, theta must not be None.

Returns

float Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

Return type log_likelihood

predict (*X*, *return_std=False*)
Predict forces using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True)

Parameters

- **X** (*list*) – Target configurations where the GP is evaluated
- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

predict_energy (*X*, *return_std=False*)
Predict energies using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True)

Parameters

- **X** (*list*) – Target configurations where the GP is evaluated
- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

predict_energy_map(*X*, *return_std=False*)

Predict energies using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True)

Parameters

- **X** (*list*) – Target configurations where the GP is evaluated
- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

predict_energy_single(*X*, *return_std=False*)

Predict energies from forces only using the Gaussian process regression model

This function evaluates the GP energies for a set of test configurations.

Parameters

- **X** (*np.ndarray*) – Target configurations where the GP is evaluated
- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

predict_energy_single_map(*X*, *return_std=False*)

Predict energies from forces only using the Gaussian process regression model

This function evaluates the GP energies for a set of test configurations.

Parameters

- **X** (*np.ndarray*) – Target configurations where the GP is evaluated

- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

predict_single(*X*, *return_std=False*)

Predict forces using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True)

Parameters

- **X** (*np.ndarray*) – Target configuration where the GP is evaluated
- **return_std** (*bool*) – If True, the standard-deviation of the predictive distribution of the target configurations is returned along with the mean.

Returns

Mean of predictive distribution at target configurations. y_std (np.ndarray): Standard deviation of predictive distribution at target

configurations. Only returned when return_std is True.

Return type y_mean (np.ndarray)

pseudo_log_likelihood()

Returns pseudo log-likelihood of the training data.

Parameters

- **theta** – array-like, shape = (n_kernel_params,) or None Kernel hyperparameters for which the log-marginal likelihood is evaluated. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.
- **eval_gradient** – bool, default: False If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. If True, theta must not be None.

Returns

float Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

Return type log_likelihood

save(*filename*)

Dump the current GP model for later use

Parameters **filename** (*str*) – name of the file where to save the GP

class mff.gp.**ThreeBodySingleSpeciesGP**(*theta*, *noise=1e-10*, *optimizer=None*, *n_restarts_optimizer=0*)

build_grid(*dists*, *element1*)
Function that builds and predicts energies on a cube of values

class mff.gp.**TwoBodySingleSpeciesGP**(*theta*, *noise=1e-10*, *optimizer=None*, *n_restarts_optimizer=0*)

4.1 Two Body Kernel

Module that contains the expressions for the 2-body single-species and multi-species kernel. The module uses the Theano package to create the energy-energy, force-energy and force-force kernels through automatic differentiation of the energy-energy kernel. The module is used to calculate the energy-energy, energy-force and force-force gram matrices for the Gaussian processes, and supports multi processing. The module is called by the gp.py script.

Example:

```
from twobodykernel import TwoBodySingleSpeciesKernel
kernel = kernels.TwoBodySingleSpeciesKernel(theta=[sigma, theta, r_cut])
ee_gram_matrix = kernel.calc_gram_e(training_configurations, number_nodes)
```

class mff.kernels.twobodykernel.**BaseTwoBody**(*kernel_name*, *theta*, *bounds*)
Two body kernel class Handles the functions common to the single-species and multi-species two-body kernels.

Parameters

- **kernel_name** (*str*) – To choose between single- and two-species kernel
- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius
- **bounds** (*list*) – bounds of the kernel function.

k2_ee
object – Energy-energy kernel function

k2_ef
object – Energy-force kernel function

k2_ef_loc
object – Local Energy-force kernel function

k2_ff
object – Force-force kernel function

calc(*X1*, *X2*)
Calculate the force-force kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1*3 x N2*3 matrix of the matrix-valued kernels

Return type K (matrix)

calc_ee(*X1*, *X2*)
Calculate the energy-energy kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2 matrix of the scalar-valued kernels

Return type K (matrix)

calc_ef(*X1*, *X2*)
Calculate the energy-force kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2*3 matrix of the vector-valued kernels

Return type K (matrix)

calc_ef_loc(*X1*, *X2*)
Calculate the local energy-force kernel between two sets of configurations. Used only during mapping since it is faster than calc_ef and equivalent in that case.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2*3 matrix of the vector-valued kernels

Return type K (matrix)

calc_gram(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the force-force gram matrix for a set of configurations X.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N*3 x N*3 gram matrix of the matrix-valued kernels

Return type gram (matrix)

calc_gram_e(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the energy-energy gram matrix for a set of configurations X.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N x N gram matrix of the scalar-valued kernels

Return type gram (matrix)

calc_gram_ef(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the energy-force gram matrix for a set of configurations X. This returns a non-symmetric matrix which is equal to the transpose of the force-energy gram matrix.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N x N*3 gram matrix of the vector-valued kernels

Return type gram (matrix)

class mff.kernels.twobodykernel.**TwoBodySingleSpeciesKernel**(*theta=(1.0, 1.0, 1.0), bounds=((0.01, 100.0), (0.01, 100.0), (0.01, 100.0))*)

Two body single species kernel.

Parameters

- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius

static compile_theano()

This function generates theano compiled kernels for global energy and force learning

The position of the atoms relative to the central one, and their chemical species are defined by a matrix of dimension Mx5 here called r1 and r2.

Returns energy-energy kernel k2_ef (func): energy-force kernel k2_ff (func): force-force kernel

Return type k2_ee (func)

class mff.kernels.twobodykernel.**TwoBodyTwoSpeciesKernel**(*theta=(1.0, 1.0, 1.0), bounds=((0.01, 100.0), (0.01, 100.0), (0.01, 100.0))*)

Two body two species kernel.

Parameters

- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius

static compile_theano()

This function generates theano compiled kernels for global energy and force learning

The position of the atoms relative to the central one, and their chemical species are defined by a matrix of dimension Mx5 here called r1 and r2.

Returns energy-energy kernel k2_ef (func): energy-force kernel k2_ff (func): force-force kernel

Return type k2_ee (func)

4.2 Three Body Kernel

Module that contains the expressions for the 3-body single-species and multi-species kernel. The module uses the Theano package to create the energy-energy, force-energy and force-force kernels through automatic differentiation

of the energy-energy kernel. The module is used to calculate the energy-energy, energy-force and force-force gram matrices for the Gaussian processes, and supports multi processing. The module is called by the gp.py script.

Example:

```
from threebodykernel import ThreeBodySingleSpeciesKernel
kernel = kernels.ThreeBodySingleSpeciesKernel(theta=[sigma, theta, r_cut])
ee_gram_matrix = kernel.calc_gram_e(training_configurations, number_nodes)
```

class mff.kernels.threebodykernel.**BaseThreeBody**(*kernel_name*, *theta*, *bounds*)

Three body kernel class Handles the functions common to the single-species and multi-species three-body kernels.

Parameters

- **kernel_name** (*str*) – To choose between single- and two-species kernel
- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius
- **bounds** (*list*) – bounds of the kernel function.

k3_ee

object – Energy-energy kernel function

k3_ef

object – Energy-force kernel function

k3_ef_loc

object – Local Energy-force kernel function

k3_ff

object – Force-force kernel function

calc(*X1*, *X2*)

Calculate the force-force kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1*3 x N2*3 matrix of the matrix-valued kernels

Return type K (matrix)

calc_ee(*X1*, *X2*)

Calculate the energy-energy kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2 matrix of the scalar-valued kernels

Return type K (matrix)

calc_ef(*X1*, *X2*)

Calculate the energy-force kernel between two sets of configurations.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2*3 matrix of the vector-valued kernels

Return type K (matrix)

calc_ef_loc(*X1*, *X2*)
Calculate the local energy-force kernel between two sets of configurations. Used only during mapping since it is faster than calc_ef and equivalent in that case.

Parameters

- **X1** (*list*) – list of N1 Mx5 arrays containing xyz coordinates and atomic species
- **X2** (*list*) – list of N2 Mx5 arrays containing xyz coordinates and atomic species

Returns N1 x N2*3 matrix of the vector-valued kernels

Return type K (matrix)

calc_gram(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the force-force gram matrix for a set of configurations X.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N*3 x N*3 gram matrix of the matrix-valued kernels

Return type gram (matrix)

calc_gram_e(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the energy-energy gram matrix for a set of configurations X.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N x N gram matrix of the scalar-valued kernels

Return type gram (matrix)

calc_gram_ef(*X*, *nnodes=1*, *eval_gradient=False*)
Calculate the energy-force gram matrix for a set of configurations X. This returns a non-symmetric matrix which is equal to the transpose of the force-energy gram matrix.

Parameters

- **X** (*list*) – list of N Mx5 arrays containing xyz coordinates and atomic species
- **nnodes** (*int*) – Number of CPU nodes to use for multiprocessing (default is 1)
- **eval_gradient** (*bool*) – if True, evaluate the gradient of the gram matrix

Returns N x N*3 gram matrix of the vector-valued kernels

Return type gram (matrix)

class mff.kernels.threebodykernel.**ThreeBodySingleSpeciesKernel**(*theta=(1.0, 1.0, 1.0), bounds=((0.01, 100.0), (0.01, 100.0), (0.01, 100.0))*)

Three body two species kernel.

Parameters

- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius

static compile_theano()

This function generates theano compiled kernels for energy and force learning ker_jkmn_withcutoff = ker_jkmn #* cutoff_ikmn

The position of the atoms relative to the centrla one, and their chemical species are defined by a matrix of dimension Mx5

Returns energy-energy kernel k3_ef (func): energy-force kernel k3_ff (func): force-force kernel

Return type k3_ee (func)

class mff.kernels.threebodykernel.**ThreeBodyTwoSpeciesKernel**(*theta=(1.0, 1.0, 1.0), bounds=((0.01, 100.0), (0.01, 100.0), (0.01, 100.0))*)

Three body two species kernel.

Parameters

- **theta[0]** (*float*) – lengthscale of the kernel
- **theta[1]** (*float*) – decay rate of the cutoff function
- **theta[2]** (*float*) – cutoff radius

static compile_theano()

This function generates theano compiled kernels for energy and force learning ker_jkmn_withcutoff = ker_jkmn #* cutoff_ikmn

The position of the atoms relative to the centrla one, and their chemical species are defined by a matrix of dimension Mx5

Returns energy-energy kernel k3_ef (func): energy-force kernel k3_ff (func): force-force kernel

Return type k3_ee (func)

CHAPTER 5

Calculators

A mapped potential is a tabulated 2- or 3-body interatomic potential created using Gaussian process regresssion and a 2- or 3-body kernel. To use a mapped potential created with this python package within the ASE environment, it is necessary to setup a calculator using the `mff.calculators` class.

5.1 Theory/Introduction

The `model.build_grid()` function builds a taulated 2- or 3- body potential. The calculator method allows to exploit the ASE functionalities to run molecular dynamics simulations. For a 2-body potential, the calculator class computes the energy and force contributions to a central atom for each other atom in its neighbourhood. These contributions depend only on the interatomic pairwise distance, and are computed trhough 1D spline interpolation of the stored values of the pairwise energy. The magnitude and verse of the pairwise force contributions are computed using the analytic derivitive of this 1D spline, while the direction of the force contribution must be the line that connects the central atom and its neighbour, for symmetry. When a 3-body potential is used, the local energy and force acting on an atom are a sum of triplet contributions which contain the central atom and two other atoms within a cutoff distance. The triplet energy contributions are computed using a 3D spline interpolation on the stored values of triplet energy which have been calculated using `model.build_grid()`. The local force contributions are obtained through analytic derivative of the 3D spline interpolation used to calculate triplet energies. The calculator behaves like a tabulated potential, and its speed scales linearly (2-body) or quadratically (3-body) with the number of atoms within a cutoff distance, and is completely independent of the number of training points used for the Gaussian process regression. The force field obtained is also analytically energy conserving, since the force is the opposite of the analytic derivative of the local energy. When using a 2- or 2+3-body force field, the `rep_alpha` parameter allows the user to include a Lennard-Jones like repulsive term that adds a 2-body repulsion: $$E_{rep}(r) = 0.5 (text{rep_alpha}/r)^{12}. $$ This introduces a repulsive term that impedes atomic collisions when the interatomic distances fall under the region where data is available. This is especially useful for high tempoerature simulations. Default `rep_alpha` is zero. In the case of a multi-species force field, the `rep_alpha` parameter is common to every pair of elements in the current version of the code.

WARNING: The atoms in the `atoms` object must be ordered in increasing atomic number for the calculator to work correctly. To do so, simply run the following line of code on the `atoms` object before the calculator is assigned:

```
atoms = atoms[np.argsort(atoms.get_atomic_numbers())]
```

5.2 Example

Assuming we already trained a model named `model` and built the relative mapped force field, we can assign an ASE calculator based on such force field to an ASE atoms object.

For a 2-body single species model:

```
from mff.calculators import TwoBodySingleSpecies
calc = TwoBodySingleSpecies(r_cut, model.grid, rep_alpha = 1.5)
atoms = atoms[np.argsort(atoms.get_atomic_numbers())]
atoms.set_calculator(calc)
```

For a 3-body model:

```
from mff.calculators import ThreeBodySingleSpecies
calc = ThreeBodySingleSpecies(r_cut, model.grid)
atoms = atoms[np.argsort(atoms.get_atomic_numbers())]
atoms.set_calculator(calc)
```

For a combined (2+3-body) model:

```
from mff.calculators import CombinedSingleSpecies
calc = CombinedSingleSpecies(r_cut, model.grid_2b, model.grid_3b, rep_alpha = 1.5)
atoms = atoms[np.argsort(atoms.get_atomic_numbers())]
atoms.set_calculator(calc)
```

class mff.calculators.**CombinedSingleSpecies**(*r_cut*, *grid_2b*, *grid_3b*, *rep_alpha=0.0*, ***kwargs*)

class mff.calculators.**CombinedTwoSpecies**(*r_cut*, *element0*, *element1*, *grids_2b*, *grids_3b*, *rep_alpha=0.0*, ***kwargs*)

class mff.calculators.**MappedPotential**(*r_cut*, ***kwargs*)

calculate(*atoms=None*, *properties=('energy', 'forces')*, *system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)

Do the calculation.

properties: list of str List of what needs to be calculated. Can be any combination of 'energy', 'forces', 'stress', 'dipole', 'charges', 'magmom' and 'magmoms'.

system_changes: list of str List of what has changed since last calculation. Can be any combination of these six: 'positions', 'numbers', 'cell', 'pbc', 'initial_charges' and 'initial_magmoms'.

Subclasses need to implement this, but can ignore properties and system_changes if they want. Calculated properties should be inserted into results dictionary like shown in this dummy example:

```
self.results = {'energy': 0.0,
                'forces': np.zeros((len(atoms), 3)),
                'stress': np.zeros(6),
                'dipole': np.zeros(3),
                'charges': np.zeros(len(atoms)),
                'magmom': 0.0,
                'magmoms': np.zeros(len(atoms))}
```

The subclass implementation should first call this implementation to set the atoms attribute.

set(***kwargs*)
Set parameters like set(key1=value1, key2=value2, ...).

A dictionary containing the parameters that have been changed is returned.

Subclasses must implement a set() method that will look at the chaneged parameters and decide if a call to reset() is needed. If the changed parameters are harmless, like a change in verbosity, then there is no need to call reset().

The special keyword 'parameters' can be used to read parameters from a file.

exception mff.calculators.**SingleSpecies**

class mff.calculators.**ThreeBodySingleSpecies**(*r_cut*, *grid_3b*, ***kwargs*)
A mapped 3-body calculator for ase

grid_3b
object – 3D Spline interpolator for the 3-body mapped grid

results
dict – energy and forces calculated on the atoms object

calculate(*atoms=None, properties=('energy', 'forces'), system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Do the calculation.

find_triplets()
Function that efficiently finds all of the valid triplets of atoms in the atoms object.

Returns

array containing the indices of atoms belonging to any valid triplet. Has shape T by 3 where T is the number of valid triplets in the atoms object

distances (array): array containing the relative distances of every triplet of atoms. Has shape T by 3 where T is the number of valid triplets in the atoms object

positions (dictionary): versor of position w.r.t. the central atom of every atom indexed in indices. Has shape T by 3 where T is the number of valid triplets in the atoms object

Return type indices (array)

class mff.calculators.**ThreeBodyTwoSpecies**(*r_cut*, *element0*, *element1*, *grids_3b*, ***kwargs*)
A mapped 3-body 2-species calculator for ase

elements
list – List of ordered atomic numbers of the mapped two species system.

grids_3b
dict – contains the four 3D Spline interpolators relative to the 3-body mapped grids for element0-element0-element0, element0-element0-element1, element0-element1-element1 and element1-element1-element1 interactions.

results
dict – energy and forces calculated on the atoms object

calculate(*atoms=None, properties=('energy', 'forces'), system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Do the calculation.

find_triplets(*atoms*)
Function that efficiently finds all of the valid triplets of atoms in the atoms object.

Returns

array containing the indices of atoms belonging to any valid triplet. Has shape T by 3 where T is the number of valid triplets in the atoms object

distances (array): array containing the relative distances of every triplet of atoms. Has shape T by 3 where T is the number of valid triplets in the atoms object

positions (dictionary): versor of position w.r.t. the central atom of every atom indexed in indices. Has shape T by 3 where T is the number of valid triplets in the atoms object

Return type indices (array)

class mff.calculators.**TwoBodySingleSpecies**(*r_cut*, *grid_2b*, *rep_alpha=0.0*, ***kwargs*)

A mapped 2-body calculator for ase

grid_2b
object – 1D Spline interpolator for the 2-body mapped grid

rep_alpha
float – Repulsion parameter, used when no data for very close atoms are available in order to avoid collisions during MD. The parameter governs a repulsion force added to the computed one.

results
dict – energy and forces calculated on the atoms object

calculate(*atoms=None, properties=('energy', 'forces'), system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Do the calculation.

class mff.calculators.**TwoBodyTwoSpecies**(*r_cut*, *element0*, *element1*, *grids_2b*, *rep_alpha=0.0*, ***kwargs*)

A mapped 2-body 2-species calculator for ase

elements
list – List of ordered atomic numbers of the mapped two species system.

grids_2b
dict – contains the three 1D Spline interpolators relative to the 2-body mapped grids for element0-element0, element0-element1 and element1-element1 interactions

rep_alpha
float – Repulsion parameter, used when no data for very close atoms are available in order to avoid collisions during MD. The parameter governs a repulsion force added to the computed one.

results
dict – energy and forces calculated on the atoms object

calculate(*atoms=None, properties=('energy', 'forces'), system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Do the calculation.

class mff.calculators.**TwoSpeciesMappedPotential**(*r_cut*, *element0*, *element1*, ***kwargs*)

calculate(*atoms=None, properties=('energy', 'forces'), system_changes=['positions', 'numbers', 'cell', 'pbc', 'initial_charges', 'initial_magmoms']*)
Do the calculation.

properties: list of str List of what needs to be calculated. Can be any combination of 'energy', 'forces', 'stress', 'dipole', 'charges', 'magmom' and 'magmoms'.

system_changes: list of str List of what has changed since last calculation. Can be any combination of these six: 'positions', 'numbers', 'cell', 'pbc', 'initial_charges' and 'initial_magmoms'.

Subclasses need to implement this, but can ignore properties and system_changes if they want. Calculated properties should be inserted into results dictionary like shown in this dummy example:

```
self.results = {'energy': 0.0,
                'forces': np.zeros((len(atoms), 3)),
                'stress': np.zeros(6),
                'dipole': np.zeros(3),
                'charges': np.zeros(len(atoms)),
                'magmom': 0.0,
                'magmoms': np.zeros(len(atoms))}
```

The subclass implementation should first call this implementation to set the atoms attribute.

set (***kwargs*)

Set parameters like set(key1=value1, key2=value2, ...).

A dictionary containing the parameters that have been changed is returned.

Subclasses must implement a set() method that will look at the chaneged parameters and decide if a call to reset() is needed. If the changed parameters are harmless, like a change in verbosity, then there is no need to call reset().

The special keyword 'parameters' can be used to read parameters from a file.

CHAPTER 6

Advanced Sampling

This module contains functions that can be used in order to subsample from very large datasets.

6.1 Theory/Introduction

6.2 Example

Assuming we already extracted all of the configurations, forces (and possibly local energies) from a .xyz file, we can apply one of the methods contained in advanced_sampling in order to subsample a meaningful and representative training set.

We first load the configurations and forces previously extracted from the .xyz file:

```
confs = np.load(configurations_file)
forces = np.load(configurations_file)
```

We then initialize the sampling class and separate ntest configurations for the test set:

```
s = Sampling(confs=confs,forces=forces, sigma_2b = 0.05, sigma_3b = 0.1, sigma_mb = 0.
↪2, noise = 0.001, r_cut = 8.5, theta = 0.5)
s.train_test_split(confs=confs, forces = forces, ntest = 200)
```

Now we can subsample a training set using our preferred method, for example importance vector machine sampling on the variance of force predicion:

```
MAE, STD, RMSE, index, time = s.ivm_f(method = '2b', ntrain = ntr, batchsize = 1000)
```

or importance vector machine sampling on the measured error of force predicion for a 3-body kernel:

```
MAE, STD, RMSE, index, time = s.ivm_f(method = '3b', ntrain = ntr, batchsize = 1000,␣
↪use_pred_error = False)
```

Other methods include a sampling based on the interatomic distance values present in every configuration:

```
MAE, STD, RMSE, index, time = s.grid(method = '2b', nbins = 1000)
```

Or a sampling based on the interatomic distance values present in every configuration:

```
MAE, STD, RMSE, index, time = s.grid(method = '2b', nbins = 1000)
```

class mff.advanced_sampling.**Sampling**(*confs=None*, *energies=None*, *forces=None*, *sigma_2b=0.05*, *sigma_3b=0.1*, *sigma_mb=0.2*, *noise=0.001*, *r_cut=8.5*, *theta=0.5*)

Sampling methods class Class containing sampling methods to optimize the trainng database selection. The class is currently set in order to work with local atomic energies, and is therefore made to be used in confined systems (nanoclusters, molecules). Some of the mothods used can be applied to force training too (ivm, random), or are independent to the training outputs (grid). These methods can be used on systems with PBCs where a local energy is not well defined. The class also initializes two GP objects to use in some of its methods.

Parameters

- **confs** (*list of arrays*) – List of the configurations as M*5 arrays
- **energies** (*array*) – Local atomic energies, one per configuration
- **forces** (*array*) – Forces acting on the central atoms of confs, one per configuration
- **sigma_2b** (*float*) – Lengthscale parameter of the 2-body kernels in Amstrongs
- **sigma_3b** (*float*) – Lengthscale parameter of the 3-body kernels in Amstrongs
- **sigma_mb** (*float*) – Lengthscale parameter of the many-body kernel in Amstrongs
- **noise** (*float*) – Regularization parameter of the Gaussian process
- **r_cut** (*float*) – Cutoff function for the Gaussian process
- **theta** (*float*) – Decay lengthscale of the cutoff function for the Gaussian process

elements
list – List of the atomic number of the atoms present in the system

natoms
int – Number of atoms in the system, used for nanoclusters

K2
array – Gram matrix for the energy-energy 2-body kernel using the full reduced dataset

K3
array – Gram matrix for the energy-energy 3-body kernel using the full reduced dataset

clean_dataset(*randomized=True*, *shuffling=True*)
Function used to subsample from a complete trajectory only one atomic environment per snapshot. This is necessary when training on energies of nanoclusters in order to assign an unique energy value to every configuration and to avoid using redundant information in the form of local atomic environments centered around different atoms in the same snapshot.

Parameters

- **randomized** (*bool*) – If True, an atom at random is chosen every snapshot, if false always the first atom in the configurations will be chosen to represent said snapshot.
- **shuffling** (*bool*) – if True, once the dataset is created, it is shuffled randomly in order to avoid any bias during incremental training set optimization methods (e.g. rvm, cur, ivm).

cur (*method='2b'*, *ntrain=1000*, *batchsize=1000*, *error_metric='energy'*)
Sampling using the CUR decomposition technique. The complete dataset is first divided into batches, then the energy-energy Gram matrix is calculated for each batch. An svd decomposition is subsequently applied to each gram matrix, and a number of entries (columns) is selected based on their importance score. The method is calibrated so that the final number of training points selected is roughly equal to the input parameter ntrain.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **ntrain** (*int*) – Number of training points to be selected from the whole dataset
- **batchsize** (*int*) – Number of data points to be used for each calculation of the gram matrix. Lower values make the computation faster but the error might be higher.
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

grid (*method='2b'*, *nbins=100*, *error_metric='energy'*, *return_error=True*)
Grid sampling, based either on interatomic distances (2b) or on triplets of interatomic distances (3b). Training configurations are shuffled and are then included in the final database only if they contain a distance value (or a triplet of distance values) which is not yet present in the binned histogram of distance values (or triplets of distance values) of the final database. This method is very fast since it does not evaluate kernel functions nor gram matrices.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **nbins** (*int*) – Number of bins to use when building an histogram of interatomic distances. If method is 2b, this will specify the value only for distances from the central atom, if method is 3b, this will specify the value for triplets of distances.
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces
- **return_error** (*bool*) – if true, error on test set using sampled database is returned

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

ivm_e (*method='2b'*, *ntrain=500*, *batchsize=1000*, *use_pred_error=True*, *error_metric='energy'*)
Importance vector machine sampling for energies. This method uses a 2- or 2-body energy kernel and trains it on the energies of the partitioned training dataset. The algortihm starts from two configurations chosen at random. At each iteration, the predicted variance or on the observed error calculated on batchsize

configurations from the training set is calculated, and the configuration with the highest value is included in the final set. The method finishes when ntrain configurations are included in the final set.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **ntrain** (*int*) – Number of training points to extract from the training dataset
- **batchsize** (*int*) – number of training points to use in each iteration of the error prediction
- **use_pred_error** (*bool*) – if true, the predicted variance is used as a metric of the ivm, if false the observed error is used instead
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

ivm_f(*method='2b'*, *ntrain=500*, *batchsize=1000*, *use_pred_error=True*, *error_metric='energy'*)

Importance vector machine sampling for forces. This method uses a 2- or 2-body energy kernel and trains it on the energies of the partitioned training dataset. The algortihm starts from two configurations chosen at random. At each iteration, the predicted variance or on the observed error calculated on batchsize configurations from the training set is calculated, and the configuration with the highest value is included in the final set. The method finishes when ntrain configurations are included in the final set.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **ntrain** (*int*) – Number of training points to extract from the training dataset
- **batchsize** (*int*) – number of training points to use in each iteration of the error prediction
- **use_pred_error** (*bool*) – if true, the predicted variance is used as a metric of the ivm, if false the observed error is used instead
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

random(*method='2b'*, *ntrain=500*, *error_metric='energy'*, *return_error=True*)

Random subsampling of training points from the larger training dataset.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **ntrain** (*int*) – Number of points to include in the final dataset.
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces
- **return_error** (*bool*) – if True, train a GP and run a test

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

rvm(*method='2b'*, *batchsize=1000*)

Relevance vector machine sampling. This method trains a 2-, 3- or many-body kernel on the energies of the partitioned training dataset. The algortihm starts from a dataset containing a batchsize number of training configurations extracted from the whole dataset at random. Subsequently, a rvm method is called and a variable number of configurations is selected. These are then included in the next batch, and the operation is repeated until every point in the training dataset was included at least once. The function then returns the indexes of the points returned by the last call of the rvm method.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **batchsize** (*int*) – number of training points to include in each iteration of the gram matrix calculation

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

test_forces(*index*, *method='2b'*, *sig_2b=0.2*, *sig_3b=0.8*, *noise=0.001*)

Random subsampling of training points from the larger training dataset.

Parameters

- **method** (*str*) – 2b or 3b, speciefies which energy kernel to use to calculate the gram matrix
- **ntrain** (*int*) – Number of points to include in the final dataset.
- **errror_metric** (*str*) – specifies whether the final error is calculated on energies or on forces

Returns Mean absolute error made by the final iteration of the method on the test set SMAE (float):Standard deviation of the absolute error made by the final iteration of the method on the test set RMSE (float): Root mean squared error made by the final iteration of the method on the test set index (list): List containing the indexes of all the selected training points total_time (float): Excecution time in seconds

Return type MAE (float)

train_test_split(*confs*, *forces=None*, *energies=None*, *ntest=10*)

Function used to subsample a training and a test set: the test set is extracted at random and the remaining dataset is trated as a training set (from which we then subsample using the various methods).

Parameters

- **confs** (*array or list*) – List of the configurations as M*5 arrays
- **energies** (*array*) – Local atomic energies, one per configuration
- **forces** (*array*) – Forces acting on the central atoms of confs, one per configuration
- **ntest** (*int*) – Number of test points, if None, every point that is not a training point will be used as a test point

CHAPTER 7

API Reference

7.1 The “models” module

mff.models.twobody

mff.models.twobody.TwoBodyTwoSpeciesModel

mff.models.threebody.ThreeBodySingleSpeciesModel

mff.models.threebody.ThreeBodyTwoSpeciesModel

mff.models.combined.CombinedSingleSpeciesModel

mff.models.combined.CombinedTwoSpeciesModel

7.2 The “gp” module

mff.gp.GaussianProcess

7.3 The “configurations” module

mff.configurations.Configurations

7.4 The “calculators” module

mff.calculators.TwoBodySingleSpecies

mff.calculators.TwoBodyTwoSpecies

mff.calculators.ThreeBodySingleSpecies

mff.calculators.ThreeBodyTwoSpecies

mff.calculators.CombinedSingleSpecies

mff.calculators.CombinedTwoSpecies

7.5 The “kernels” module

mff.kernels.twobodykernel.TwoBodySingleSpeciesKernel

mff.kernels.twobodykernel.TwoBodyTwoSpeciesKernel

mff.kernels.threebodykernel.ThreeBodySingleSpeciesKernel

mff.kernels.threebodykernel.ThreeBodyTwoSpeciesKernel

7.6 The “advanced_sampling” module

mff.advanced_sampling

CHAPTER 8

Index

CHAPTER 9

Maintainers

- Claudio Zeni (claudio.zeni@kcl.ac.uk),
- Aldo Glielmo (aldo.glielmo@kcl.ac.uk),
- Ádám Fekete (adam.fekete@kcl.ac.uk).

CHAPTER 10

References

[1] A. Glielmo, C. Zeni, A. De Vita, *Efficient non-parametric n-body force fields from machine learning* (https://arxiv.org/abs/1801.04823)

[2] C .Zeni, K. Rossi, A. Glielmo, N. Gaston, F. Baletto, A. De Vita *Building machine learning force fields for nanoclusters* (https://arxiv.org/abs/1802.01417)

CHAPTER 11

Indices and tables

- Index
- modindex

Python Module Index

.

Index

E

F

G

K

L

M

N

P

R

S

T

U

X