pacemaker

Contents

`pacemaker` is a tool for fitting of interatomic potentials in a general nonlinear Atomic Cluster Expansion (ACE) form.

1 Installation

1.1 (optional) Creating a conda environment

It is common practice creating a separate `conda environment` to avoid dependencies mixing. You can create the new environment named `ace` with minimal amount of required packages with the following command:

```
conda env create -n ace python<3.9
```

Then, activate the environment with `source activate ace` or `conda activate ace`. To deactivate the environment, use `deactivate` command

1.2 Installation of `tensorpotential`

`tensorpotential` allows for the GPU accelerated optimization of the ACE potential using TensorFlow. However, it is recommended to use it even if you don't have a GPU available. Install it using the following commands:

```
pip install tensorflow==2.5.0 # newer version should be also compatible
cd tensorpotential
pip install --upgrade .
```

1.3 Installation of `pyace`

Finally, `pyace` could be installed with

```
cd pyace
pip install --upgrade .
```

2 Quick start

Running a fit with `pacemaker` requires at least two components: fitting dataset and configurational input file. Fitting dataset contains structural information as well as corresponding energies and forces that are subject to fitting with ACE. Input file contains details about desired ACE potential configuration and various parameters influencing optimization process.
In this section we will describe the format of the fitting dataset, we will run a fit with an example dataset and overview the output produced by `pacemaker`. Input parameters are detailed in the section below.

2.1 Fitting dataset preparation

In order to use your data for fitting with `pacemaker` one would need to provide it in the form of `pandas` DataFrame. An example DataFrame can be red as:

```
import pandas as pd
df = pd.read_pickle("../data/exmpl_df.pckl.gzip", compression="gzip", protocol=4)
```

And it contains the following entries:

	energy	forces	ase_atoms	energy_corrected
0	-3.69679	[[0.0, 0.0, 0.0]]	Atoms(symbols='Al', pbc=True, cell=[[0.0, 1.949947, 1.949947], [1.949947, 0.0, 1.949947], [1.949947, 1.949947, 0.0]])	-3.69679
1	-3.71569	[[0.0, 0.0, 0.0]]	Atoms(symbols='Al', pbc=True, cell=[[0.0, 1.964285, 1.964285], [1.964285, 0.0, 1.964285], [1.964285, 1.964285, 0.0]])	-3.71569
2	-3.72955	[[0.0, 0.0, 0.0]]	Atoms(symbols='Al', pbc=True, cell=[[0.0, 1.978417, 1.978417], [1.978417, 0.0, 1.978417], [1.978417, 1.978417, 0.0]])	-3.72955
3	-3.7389	[[0.0, 0.0, 0.0]]	Atoms(symbols='Al', pbc=True, cell=[[0.0, 1.99235, 1.99235], [1.99235, 0.0, 1.99235], [1.99235, 1.99235, 0.0]])	-3.7389
4	-3.74421	[[0.0, 0.0, 0.0]]	Atoms(symbols='Al', pbc=True, cell=[[0.0, 2.006091, 2.006091], [2.006091, 0.0, 2.006091], [2.006091, 2.006091, 0.0]])	-3.74421

- Columns have the following meaning:
  - `ase_atoms`: is the instance of the ASE Atoms class. This is the main form of storing structural information that `pacemkaer` relies on. It must contain information about atomic positions, corresponding atom types, pbc and lattice vectors.
  - `energy`: total energy of the corresponding `ase_atoms` structure (in eV).
  - `forces`: corresponding atomic forces in the form of 2D array with dimensions [NumberOfAtoms, 3] (in eV/A).
  - `energy_corrected`: total energy of a structure minus a reference energy.

  Reference energy might be different depending on the dataset at hand. In general, one would prefer to reference `energy` against the free atom energies. In this case `energy_corrected` corresponds to the cohesive energy. If the free atom energies are not available, reference energy might be any constant shift or 0. In this example `energy` is already the cohesive energy.
  NOTE: regardless how `energy_corrected` is produced, *this is the energy that will be used for fitting.*

One could create such DataFrame from raw data following this example:

```
import pandas as pd
from ase import Atoms

# Collect raw data for the first structure
# Positions
pos1 = [[2.04748516, 2.04748516, 0.        ],
       [0.        , 0.        , 0.        ],
       [2.04748516, 0.        , 1.44281847],
       [0.        , 2.04748516, 1.44475745]]
# Matrix of lattice vectors
lattice1 = [[4.09497 , 0.      , 0.      ],
       [0.      , 4.09497 , 0.      ],
       [0.      , 0.      , 2.887576]]
# Atomic symbols
symbls1 = ['Al', 'Al', 'Ni', 'Ni']
# energy
e1 = -21.07723361
# Forces
f1 = [[0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0],
     [0.0, 0.0, 0.00725587],
     [0.0, 0.0, -0.00725587]]
# create ASE atoms
at1 = Atoms(symbols=symbls1, positions=pos1, cell=lattice1, pbc=True)

#Collect raw data for the second structure
pos2  = [[0., 0., 0.]]
lattice2 = [[0.      , 1.781758, 1.781758],
           [1.781758, 0.      , 1.781758],
           [1.781758, 1.781758, 0.      ]]
symbls2 = ['Ni']
e2 = -5.45708644
f2 = [[0.0, 0.0, 0.0]]
at2 = Atoms(symbols=symbls2, positions=pos2, cell=lattice2, pbc=True)

# set reference energy to 0
reference_energy = 0
# collect all the data into a dictionary
data = {'energy': [e1, e2],
        'forces': [f1, f2],
        'ase_atoms': [at1, at2],
        'energy_corrected': [e1 - reference_energy, e2 - reference_energy]}
# create a DataFrame
df = pd.DataFrame(data)
# and save it
df.to_pickle('my_data.pckl.gzip', compression='gzip', protocol=4)
```

The resulting dataframe can be used for fitting with `pacemaker`.

2.2 Creating an input file

In order to fit an ACE potential to the data prepared following the previous section, one need to create a configurational file with relevant settings. `pacemaker` utilizes `.yaml` format for configurations. An input file template can be created by running `pacemaker --template` (or `pacemaker -t`). Doing so will produce an `input.yaml` file with the most general settings that can be adjusted for a particular task. Detailed overview of the input file parameters can be found in the section below.
In this example we will use template as it is, however one would need to provide a path to the example dataset `exmpl_df.pckl.gzip`. This can be done by changing `filename` parameter in the `data` section of the `input.yaml`:

```
...
data:
    filename: /path/to/the/pyace/data/exmpl_df.pckl.gzip
...
```

2.3 Run fitting

Running a fit is as easy as executing the command:

```
pacemaker input.yaml
```

or to run the fitting process in the background:

```
nohup pacemaker input.yaml &
```

For more `pacemaker` command options see the corresponding section.

Default behavior of pacemaker is to utilize a GPU accelerated fitting of ACE using `tensorpotential`. However, GPU parallelization is not supported at the moment. Therefore, if your machine has a multi GPU setup one would need to select a single one before running `pacemaker`. This can be done by executing `export CUDA_VISIBLE_DEVICES=ind` in the shell replacing `ind` with the GPU index (i.e. 0, 1, . . .) or `-1` to disable GPU usage.
Note, that `tensorpotential` can be used without a GPU as well.

2.4 Analysis

During and after the fitting `pacemaker` produces several outputs, including:

- `interim_potential_X.yaml`: current state of the potential at each iteration of fit cycle (i.g. X=0, 1, . . .)
- `interim_potential_best_cycle.yaml`: best out of X interim potentials
- `log.txt`: log file containing all current information including summary of the optimization steps.
- `report`: folder containing figures displaying various error statistics and distributions.
- `output_potential.yaml`: final fitted potential.

There are two main types of the information in the log file:

- optimization step log:

```
Iteration   #999  (1052 evals):     Loss: 0.000192 | RMSE Energy(low): 17.95 (16.79) meV/at | Forces(low):
```

where `Iteration` is the index of the optimization step performed by the optimizer (number in parentheses shows the number of function evaluation calls done by optimizaer), `Loss` is the current value of the loss function, `RMSE Energy/Forces` is the current root mean-squared error for energy/forces wrt. training dataset (numbers in paretheses show corresponding values for the structures which energy is not greater than `e_min + 1 eV`, where `e_min` is the lowest energy in the training set). `Time/eval` shows the computational time spent on evaluating loss function and it's gradient for the training dataset averaged across evaluations and divided by the number of atoms. This timing doesn't include optimization step itself.

- fit statistics:

```
--------------------------------------------FIT STATS--------------------------------------------
Iteration:  #1000Loss:    Total:  1.9159e-04 (100%)
                         Energy:  1.6074e-04 ( 84%)
                         Force:  3.0859e-05 ( 16%)
                             L1:  0.0000e+00 (  0%)
                             L2:  0.0000e+00 (  0%)
Number of params./funcs:    232/86                                  Avg. time:     526.93 mcs/at
-------------------------------------------------------------------------------------------------
            Energy/at, meV/at   Energy_low/at, meV/at      Force, meV/A        Force_low, meV/A
    RMSE:          17.93                 16.73                 7.86                     7.06
    MAE:           12.22                 11.11                 5.31                     3.30
    MAX_AE:        53.19                 38.30                35.19                    20.32
-------------------------------------------------------------------------------------------------
```

Every display_step the summary of fit statistics is printed out. It displays the total loss function value and contributions to it from energy, forces and other regularizations parameters. In addition to RMSE, mean-absolute error (MAE) and maximum absolute error (MAX_AE) are also printed.

2.5 Using fitted potential

Fitted potential can be used for calculations both within python/ASE as well as LAMMPS.

2.5.1 ASE

Python interface of the ACE potential is realized via ASE calculator:

```
from ase import Atoms
from pyace import PyACECalculator

# use the example of the Atoms from the first section
# Positions
pos1 = [[2.04748516, 2.04748516, 0.        ],
       [0.        , 0.        , 0.        ],
       [2.04748516, 0.        , 1.44281847],
       [0.        , 2.04748516, 1.44475745]]
# Matrix of lattice vectors
lattice1 = [[4.09497 , 0.      , 0.      ],
       [0.      , 4.09497 , 0.      ],
       [0.      , 0.      , 2.887576]]
# Atomic symbols
symbls1 = ['Al', 'Al', 'Ni', 'Ni']
# create ASE atoms
at1 = Atoms(symbols=symbls1, positions=pos1, cell=lattice1, pbc=True)

# Create calculator
calc = PyACECalculator('output_potential.yaml')
# Attach it to the Atmos
at1.set_calculator(calc)
# Evaluate properties
energy = at1.get_potential_energy()
forces = at1.get_forces()
```

2.5.2 LAMMPS

Using potential with LAMMPS requires its conversion into **YACE** format with command

```
pace_yaml2yace output_potential.yaml
```

that will generate `output_potential.yace` file, which you could use in LAMMPS input file

```
## in.lammps

pair_style  pace
pair_coeff  * * output_potential.yace Al Ni
```

2.5.2.1 LAMMPS compilation: You could get the supported version of LAMMPS from GitHub repository

2.5.2.1.1 Build with `make`

Follow LAMMPS installation instructions

1. Go to `lammps/src` folder
2. Compile the ML-PACE library by running `make lib-pace args="-b"`
3. Include `ML-PACE` in the compilation by running `make yes-ml-pace`
4. Compile lammps as usual, i.e. `make serial` or `make mpi`.

2.5.2.1.2 Build with `cmake`

1. Create build directory and go there with

```
cd lammps
mkdir build
cd build
```

2. Configure the lammps build with

```
cmake -DCMAKE_BUILD_TYPE=Release -DPKG_ML-PACE=ON ../cmake
```

or

```
cmake -DCMAKE_BUILD_TYPE=Release -D BUILD_MPI=ON -DPKG_ML-PACE=ON ../cmake
```

For more information see here.

3. Build LAMMPS using `cmake --build .` or `make`

---

3 `pacemaker` workflow

The `pacemaker` workflow is described in the following and summarized in the figure above.

- `pacemaker` starts by constructing the potential according to the user specified basis configuration (ν-order, n_{max}, l_{max}, etc.) or loads it from an available potential file. Then the B-basis functions are constructed, to this end generalized Clebsch-Gordan coefficients are set up for generating product basis functions that are invariant with respect to rotation and inversion.
- Then `pacemaker` constructs the neighborlist for all structures in the dataframe. The neighborlist can be added to the reference dataframe for a fast restart of future parameterization runs.
- Next the weights for each structure and atom as required by the loss function are set up. `pacemaker` provides different weighting schemes. The weights are then added to the reference dataframe. Weights may also be added directly to the reference dataframe, so that the user has full control over the weights for each structure and force.
- `pacemaker` splits the dataset for training and for testing.
- The further specification of L_1, L_2 and radial smoothness w_0, w_1, w_2 regularization contributions and the relative weight κ of energy and force errors enables `pacemaker` to set up the loss function.
- The hierarchical basis extension is setup as ladder fitting scheme if requested by the user.
- The optimization of the loss function can be carried out with different optimizers and optimization strategies. For each optimization step `pacemaker` stores the current potential and computes error metrics for energies and forces. In addition, external Python code can be called to perform specific calculations for advanced on-the-fly validation.
- If requested, optimization is repeated with intermediate randomization of the training parameter.
- During and at the end of loss function optimization `pacemaker` provides outputs for assessing the quality and convergence of a parameterization.
- `pacemaker` stores (and loads) the ACE potentials in a transparent YAML file format.

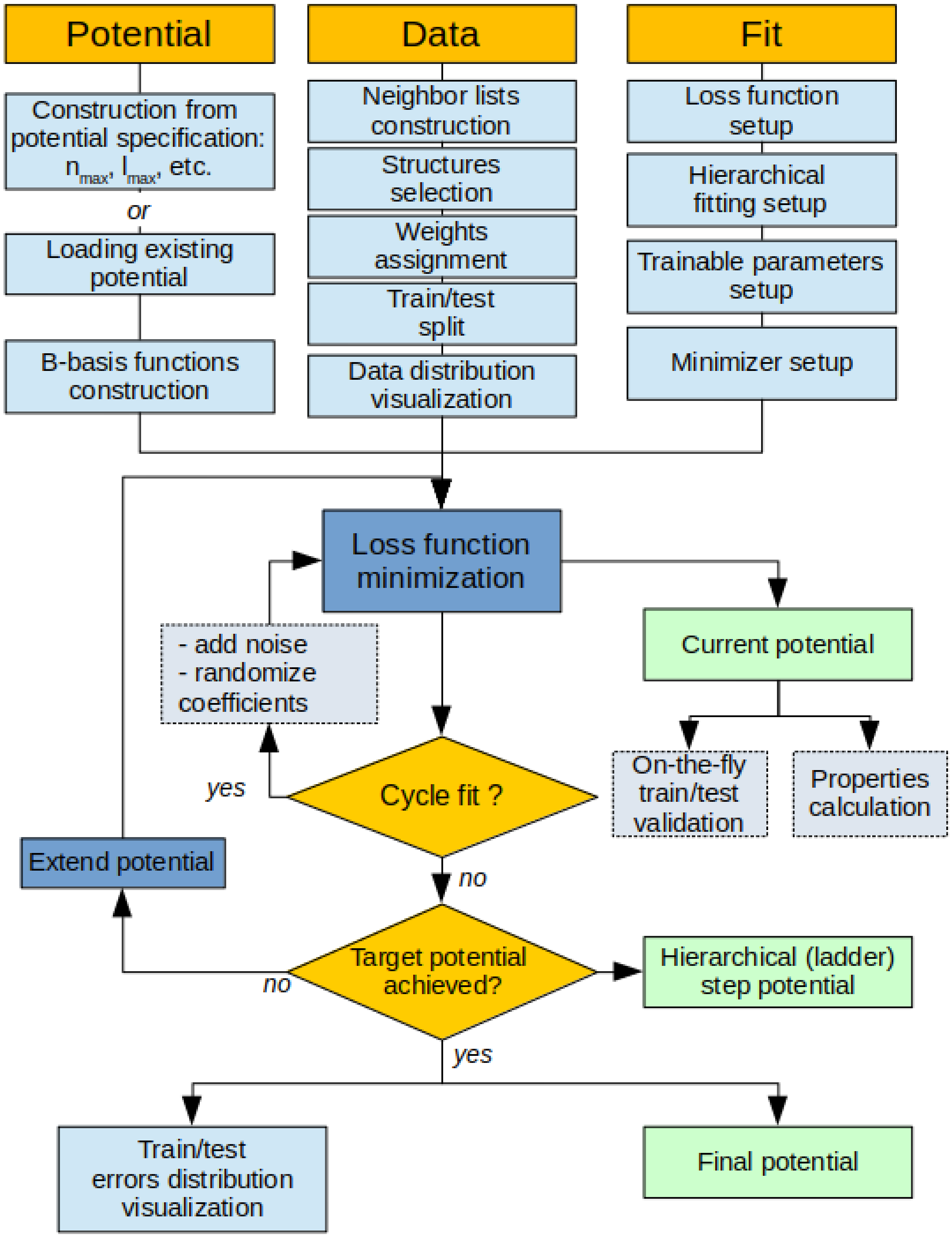
Potential
Data
Fit
Construction from potential specification: n_{max}, l_{max}, etc.
or
Loading existing potential
B-basis functions construction
Neighbor lists construction
Structures selection
Weights assignment
Train/test split
Data distribution visualization
Loss function setup
Hierarchical fitting setup
Trainable parameters setup
Minimizer setup
Loss function minimization
- add noise
- randomize coefficients
Current potential
On-the-fly train/test validation
Properties calculation
yes
Cycle fit ?
no
Extend potential
no
Target potential achieved?
Hierarchical (ladder) step potential
yes
Train/test errors distribution visualization
Final potential

Figure 1: `pacemaker` workflow scheme

4 Input file

The required settings are provided by input YAML file. This file consists of several sections devoted to setting up particular settings of `pacemaker`. The sections are listed below.

4.1 Cutoff and (optional) metadata

- Global cutoff for the neighborlist constructor is setup as:

```
cutoff: 10.0
```

- Metadata (optional)

This is arbitrary key (string)-value (string) pairs that would be added to the potential YAML file:

```
metadata:
  info: some info
  comment: some comment
  purpose: some purpose
```

Moreover, `starttime` and `user` fields would be added automatically

4.2 Dataset specification section

This section is denoted by the key

```
data:
  ...
```

Fitting dataset could be queried automatically from `structdb` (if corresponding `structdborm` package is installed and connection to database is configured, see `structdb.ini` file in home folder). Alternatively, dataset could be saved into file as a pickled `pandas` dataframe with special names for columns.

Example:

```
data: # dataset specification section
  # data configuration section
  config:
    element: Al                     # element name
    calculator: FHI-aims/PBE/tight # calculator type from `structdb`
    # ref_energy: -1.234            # single atom reference energy
                                    # if not specified, then it will be queried from database

  # seed: 42                        # random seed for shuffling the data
  # query_limit: 1000               # limiting number of entries to query from `structdb`
                                    # ignored if reading from cache

  # cache_ref_df: True              # whether to store the queried or modified dataset into file, default -
  # filename: some.pckl.gzip        # force to read reference pickled dataframe from given file
  # ignore_weights: False           # whether to ignore energy and force weighting columns in dataframe
  # datapath: ../data               # path to folder with cache files with pickled dataframes
```

Alternatively, instead of `data::config` section, one can specify just the cache file with pickled dataframe as `data::filename`:

```
data:
  filename: small_df_tf_atoms.pckl
  datapath: ../tests/
```

`data:datapath` option, if not provided, could be replaced with *environment variable* **PACEMAKERDATAPATH**

Example of creating the **subselection of fitting dataframe** and saving it is given in `notebooks/data_preprocess.ipynb`

Example of generating **custom energy/forces weights** is given in `notebooks/data_custom_weights.ipynb`

4.2.1 Querying data

You can just query and preprocess data, without running potential fitting. Here is the minimalistic input YAML:

```yaml
# input.yaml file

cutoff: 10.0  # use larger cutoff to have excess neighbour list
data: # dataset specification section
  config:
    element: Al                    # element name
    calculator: FHI-aims/PBE/tight # calculator type from `structdb`
  seed: 42
  datapath: ../data                # path to the directory with cache files
  # query_limit: 100               # number of entries to query
```

4.2.2 Preparing the data / constructing neighbor list

You can use existing `.pckl.gzip` dataset and generate all necessary columns for that, including neighbourlists Here is the minimalistic input YAML:

```yaml
# input.yaml file

cutoff: 10.

data:
  filename: my_dataset.pckl.gzip

backend:
  evaluator: tensorpot  # pyace, tensorpot
```

Then execute `pacemaker --prepare-data input.yaml` Generation of the `my_dataset.pckl.gzip` from, for example, *pyiron* is shown in `notebooks/convert-pyiron-to-pacemaker.ipynb`

4.2.3 Test set

You could provide test set either as a fraction or certain number of samples from the train set (option `test_size`) or as a separate pckl.gzip file (option `test_filename`)

```yaml
data:
  test_filename: my_test_dataset.pckl.gzip
```

or

```yaml
data:
  test_size: 100 # would take 100 samples randomly from train/fit set
  # test_size: 0.1 #  if <1 - would take given fraction of samples randomly from train/fit set
```

4.3 Interatomic potential (or B-basis) configuration

4.3.1 Basis configuration

In order to specify the B-basis potential, you have to provide four main components (aka **basis shape**): `elements`, `embeddings` for each element, `bonds` for each possible pairs of elements and `functions` for each possible combination of elements (unary, binary, ternary, etc.) as follows:

```
potential:
  deltaSplineBins: 0.001
  elements: [Al, Ni]  # list of all element

  # Embeddings are specified for each individual elements,
  # all parameters could be distinct for different species
  embeddings: ## possible keywords: ALL, UNARY, elements: Al, Ni
    Al: {
      npot: 'FinnisSinclairShiftedScaled',
      fs_parameters: [1, 1, 1, 0.5], ## non-linear embedding function: 1*rho_1^1 + 1*rho_2^0.5
      ndensity: 2,

      # core repulsion parameters
      rho_core_cut: 200000,
      drho_core_cut: 250
    }

    Ni: {
      npot: 'FinnisSinclairShiftedScaled', ## linear embedding function: 1*rho_1^1
      fs_parameters: [1, 1],
      ndensity: 1,

      # core repulsion parameters
      rho_core_cut: 3000,
      drho_core_cut: 150
    }

  ## Bonds are specified for each possible pairs of elements
  ## One could use keywords: ALL (Al,Ni, AlNi, NiAl)
  bonds: ## possible keywords: ALL, UNARY, BINARY, elements pairs as AlAl, AlNi, NiAl, etc...
    ALL: {
        radbase: ChebExpCos,
        radparameters: [5.25],

        ## outer cutoff, applied in a range [rcut - dcut, rcut]
        rcut: 5,
        dcut: 0.01,

        ## inner cutoff, applied in a range [r_in, r_in + delta_in]
        r_in: 1.0,
        delta_in: 0.5,

        ## core-repulsion parameters `prefactor` and `lambda` in
        ## prefactor*exp(-lambda*r^2)/r, >0 only r<r_in+delta_in
        core-repulsion: [0.0, 5.0],
    }

    ## BINARY overwrites ALL settings when they are repeated
    BINARY: {
        radbase: ChebPow,
        radparameters: [6.25],

        ## cutoff may vary for different bonds
        rcut: 5.5,
        dcut: 0.01,

        ## inner cutoff, applied in a range [r_in, r_in + delta_in]
        r_in: 1.0,
        delta_in: 0.5,

        ## core-repulsion parameters `prefactor` and `lambda` in
        ## prefactor*exp(-lambda*r^2)/r, >0 only r<r_in+delta_in
        core-repulsion: [0.0, 5.0],
    }

  ## possible keywords: ALL, UNARY, BINARY, TERNARY, QUATERNARY, QUINARY,
  ##  element combinations as (Al,Al), (Al, Ni), (Al, Ni, Zn), etc...
  functions:
    UNARY: {
      nradmax_by_orders: [15, 3, 2, 2, 1],
      lmax_by_orders: [ 0, 2, 2, 1, 1],
      # coefs_init: zero # initialization of functions coefficients: zero (default) or random
    }

    BINARY: {
      nradmax_by_orders: [15, 2, 2, 2],
      lmax_by_orders: [ 0, 2, 2, 1],
      # coefs_init: zero # initialization of functions coefficients: zero (default) or random
    }
```

In sections `embeddings`, `bonds` and `functions` one could use keywords (ALL, UNARY, BINARY, TERNARY, QUATERNARY, QUINARY). The settings provided by more specific keyword will override those from less specific keyword, i.e. ALL < UNARY < BINARY < (‘Al’,‘Ni’)

4.3.2 Upfitting

If you want to continue the fitting of the existing potential from `potential.yaml` file, then specify

```
potential: potential yaml
```

alternatively, one could use `pacemaker ... -p potential.yaml` option.

For specifying both initial and target potential from the file one could provide:

```
potential:
  filename: potential.yaml

  ## in "ladder" fitting scheme, potential from with to start fit
  # initial_potential: initial_potential.yaml

  ## reset potential from potential.yaml, i.e. set radial coefficients to delta_nk and func coeffs=[0..]
  # reset: true
```

or alternatively, one could use `pacemaker ... -p potential.yaml -ip initial_potential.yaml` options.

4.4 Fitting settings

Example of `fit` section is:

```
fit:
    ## LOSS FUNCTION OPTIONS ##
    loss: {
      ## [0..1] or auto, relative force weight,
      ## kappa = 0 - energies-only fit,
      ## kappa = 1 - forces-only fit
      ## auto - determined from dataset based on variance of energies and forces
      kappa: 0,
      ## L1-regularization coefficient
      L1_coeffs: 0,
      ## L2-regularization coefficient
      L2_coeffs: 0,
      ## w0 radial smoothness regularization coefficient
      w0_rad: 0,
      ## w1 radial smoothness regularization coefficient
      w1_rad: 0,
      ## w2 radial smoothness regularization coefficient
      w2_rad: 0
    }

    ## DATA WEIGHTING OPTIONS ##
    weighting: {
        ## weights for the structures energies/forces are associated according to the distance to E_min:
        ## convex hull ( energy: convex_hull) or minimal energy per atom (energy: cohesive)
        type: EnergyBasedWeightingPolicy,
        ## number of structures to randomly select from the initial dataset
        nfit: 10000,
        ## only the structures with energy up to E_min + DEup will be selected
        DEup: 10.0,  ## eV, upper energy range (E_min + DElow, E_min + DEup)
        ## only the structures with maximal force on atom  up to DFup will be selected
        DFup: 50.0, ## eV/A
        ## lower energy range (E_min, E_min + DElow)
        DElow: 1.0,  ## eV
        ## delta_E  shift for weights, see paper
        DE: 1.0,
        ## delta_F  shift for weights, see paper
        DF: 1.0,
        ## 0<wlow<1 or None: if provided, the renormalization weights of the structures on lower energy ra
        wlow: 0.75,
        ##  "convex_hull" or "cohesive" : method to compute the E_min
        energy: convex_hull,
        ## structures types: all (default), bulk or cluster
        reftype: all,
        ## random number seed
        seed: 42
    }

    ## Custom weights:  corresponding to main dataset index and `w_energy` and `w_forces` columns should
    ## be provided in pckl.gzip file
    #weighting: {type: ExternalWeightingPolicy, filename: custom_weights_only.pckl.gzip}

    ## OPTIMIZATION OPTIONS ##
    optimizer: BFGS # BFGS, L-BFGS-B, Nelder-Mead, etc. : scipy minimization algorithm
    ## additional options for scipy.minimize(..., options={...}, ...)
    #options: {maxcor: 100}
    maxiter: 1000 # maximum number of iteration for EACH scipy minimization round

    ## EXTRA OPTIONS ##
    repulsion: auto            # set inner cutoff based on the minimal distance in the dataset

    #trainable_parameters: ALL  # ALL, UNARY, BINARY, ..., radial, func, {"AlNi": "func"}, {"AlNi": {"func

    ##(optional) number of consequentive runs of fitting algorithm (for each ladder step), that helps conv
    #fit_cycles: 1

    ## starting from second fit_cycle:

    ## applies Gaussian noise with specified relative sigma/mean ratio to all potential trainable coeffici
    #noise_relative_sigma: 1e-3

    ## applies Gaussian noise with specified absolute sigma to all potential trainable coefficients
    #noise_absolute_sigma: 1e-3

    # reset the function coefficients according to Gaussian distribution with given sigma; enable ensemble
    #randomize_func_coeffs: 1e-3

    ## LADDER SCHEME (i.e. hierarchical fitting) ##
    ## enables hierarchical fitting (LADDER SCHEME), that sequentially add specified number of B-functions
    #ladder_step: [10, 0.02]
    ##      - integer >= 1 - number of basis functions to add in ladder scheme,
    ##      - float between 0 and 1 - relative ladder step size wrt. current basis step
    ##      - list of both above values - select maximum between two possibilities on each iteration
    ##     see. Ladder scheme fitting for more info


    ## Possible values:
    ## body_order  -  new basis functions are added according to the body-order, i.e., a function with hig
    ##                will not be added until the list of functions of the previous body-order is exhauste
    ## power_order -  the order of adding new basis functions is defined by the "power rank" p of a functi
    ##                p = len(ns) + sum(ns) + sum(ls). Functions with the smallest p are added first
    #ladder_type: body_order


    ## callbacks during the fitting. Module quick_validation.py should be available for import
    ## see example/pacemaker_with_callback for more details and examples
    #callbacks:
    #  - quick_validation.test_fcc_potential_callback
```

If not specified, then *uniform weight* and *energy-only* fit (kappa=0), *fit_cycles*=1, *noise_relative_sigma* = 0 settings will be used.

If ladder fitting scheme is used, then intermediate version of the potential after each ladder step will be saved into `interim_potential_ladder_step_{LADDER_STEP}.yaml`.

4.5 Backend specification

```
backend:
  evaluator: tensorpot  # pyace, tensorpot

  ## for `tensorpot` evaluator, following options are available:
```

```
  # batch_size: 10            # batch size for loss function evaluation, default is 10
  # batch_size_reduction: True # automatic batch_size reduction if not enough memory (default - True)
  # batch_size_reduction_factor: 1.618  # batch size reduction factor
  # display_step: 20          # frequency of detailed metric calculation and printing

  ## for `pyace` evaluator, following options are available:
  # parallel_mode: process    # process, serial  - parallelization mode for `pyace` evaluator
  # n_workers: 4              # number of parallel workers for `process` parallelization mode
```

Alternatively, backend could be selected as `pacemaker ... -b tensorpot`

4.6 Ladder (hiererchical) basis extension

In a ladder scheme potential extension happens by adding new portion of basis functions step-by-step, to form a "ladder" from *initial potential* to *final potential*. Following settings should be added to the input YAML file:

- Specify *final potential* shape by providing `potential` section:

```
potential:
  deltaSplineBins: 0.001
  element: Al
  ...
```

- Specify *initial potential* by providing `initial_potential` option in `potential` section:

```
potential:
    ...
    initial_potential: some_start_or_interim_potential.yaml    # potential to start fit from
```

If *initial potential* is not specified, then the fit will start from empty potential. Alternatively, you can specify *initial potential* by command-line option

```
pacemaker ... -ip some_start_or_interim_potential.yaml
```

- Specify `ladder_step` in `fit` section:

```
fit:
    ...

    ladder_step: [10, 0.02]
## Possible values:
##  - integer >= 1 - number of basis functions to add in ladder scheme,
##  - float between 0 and 1 - relative ladder step size wrt. current basis step
##  - list of both above values - select maximum between two possibilities on each iteration
```

See `example/ladder_fit_pyace.yaml` and `example/ladder_fit_tensorpot.yaml` example input files

5 pacemaker command line interface

`pacemaker` is an utility for fitting the atomic cluster expansion potential. Usage:

```
usage: pacemaker [-h] [-c] [-o OUTPUT] [-p POTENTIAL] [-ip INITIAL_POTENTIAL]
                 [-b BACKEND] [-d DATA] [--query-data] [--prepare-data]
                 [--rebuild] [-l LOG] [-dr] [-t]
                 [input]

Fitting utility for atomic cluster expansion potentials

positional arguments:
  input                 input YAML file, default: input.yaml
```

```
optional arguments:
  -h, --help            show this help message and exit
  -c, --clean           Remove all generated data
  -o OUTPUT, --output OUTPUT
                        output B-basis YAML file name, default:
                        output_potential.yaml
  -p POTENTIAL, --potential POTENTIAL
                        input potential YAML file name, will override input
                        file 'potential' section
  -ip INITIAL_POTENTIAL, --initial-potential INITIAL_POTENTIAL
                        initial potential YAML file name, will override input
                        file 'potential::initial_potential' section
  -b BACKEND, --backend BACKEND
                        backend evaluator, will override section
                        'backend::evaluator' from input file
  -d DATA, --data DATA  data file, will override section 'YAML:fit:filename'
                        from input file
  --query-data          query the training data from database, prepare and
                        save them
  --prepare-data        prepare and save training data only
  --rebuild             force to rebuild necessary neighbour lists
  -l LOG, --log LOG     log filename, default: log.txt
  -dr, --dry-run        Dry run: performs all preprocessing and analysis, but
                        do not do the fitting
  -t, --template        Create a template 'input.yaml' file
  -v, --version         Show version info
  --no-fit              Do not fit the potential
  --no-predict          Do not compute and save the predictions
  --verbose-tf          Make tensorflow more verbose (off by defeault)
```

6 Utilities

6.1 Potential conversion

There are **two** basic formats for ACE potentials:

1. **B-basis set** - YAML format, i.e. 'Al.pbe.yaml'. This is an internal *complete* format for potential fitting.
2. **Ctilde-basis set** - YACE (special form of YAML) format, i.e. 'Al.pbe.yace'. This format is *irreversibly* converted from *B-basis set* for public potentials distribution and for using in LAMMPS simulations.

Please see [pacemaker paper] for more details about **B-basis** and **Ctilde-basis sets**

To convert potential you can use following utility, that is installed together with `pyace` package into you executable paths: * `YAML` to `yace` : `pace_yaml2yace`. Usage:

```
  usage: pace_yaml2yace [-h] [-o OUTPUT] input [input ...]

Conversion utility from B-basis (.yaml file) to new-style Ctilde-basis (.yace
file)

positional arguments:
  input                 input B-basis file name (.yaml)

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
```

```
                        output Ctilde-basis file name (.yace)
```

6.2 YAML potential timing

Utility to run the single-CPU timing test for PACE (.yaml) potential. Usage:

```
pace_timing [-h] potential_file
```

6.3 YAML potential info

Utility to show the basic information (type of embedding, cutoff, radial functions, n-max, l-max etc.) for PACE (.yaml) potential. Usage:

```
pace_info [-h] potential_file
```

---

7 Frequently asked questions (FAQ)

7.1 What is a good value for batch_size?

In order to achieve better fitting performance large *batch_size* (i.e. 100 or 1000) is recommended. If `batch_size_reduction=True` (default option), then automatic batch size reduction will happen and you could start from initial large *batch_size* value.

7.2 My fit on GPU crushes with OOM error, what to do ?

This means that you are trying to fit at once too much data into the GPU memory. The amount of data processed by GPU at once is controlled by batch_size parameter, try to reduce it or set `batch_size_reduction=True`. An optimal value for this parameter is totally empirical as it depends on data, potential configuration and GPU itself.

7.3 Can I toggle between CPU and GPU when starting pacemaker?

If you have GPU configured on your machine it will be used by default. You can have additional control over GPU configuration via `input.yaml::backend::gpu_config`:

```
backend:
  gpu_config: {gpu_ind: <int>, mem_limit: <int>}
```

- `gpu_ind`: index of the GPU you want to use for fitting. This need to be specified in case your machine has multiple GPUs (multi GPU fitting is not supported at the moment). Set this parameter to -1 to disable GPU utilization. Default is 0.
- `mem_limit`: maximum amount of GPU memory in MB that is allowed to be used by fitting process. Default is 0 which allows to consume the whole available memory.
  NOTE: memory reserved by the fitting process is not available to anything else. Therefore, it's recommended to set this restriction if you also use the same machine for processes requiring GUI.

7.4 I dont have a GPU. Should I use backend, evaluator= `tensorpotential` or `pyace`?

It is recommended to use `tensorpotential` evaluator for fitting anyways. Even without GPU acceleration non-linear optimization greatly benefits from autogradients provided by TensorFlow.

7.5 How to continue fitting ?

Fitting an ACE potential can be continued or restarted from any `.yaml` potential file produced previously.
If you want to continue fit without changing the basis size, you can do the following: - Provide the path to the starting potential in corresponding field in the `input.yaml` file `yaml      potential: /path/to/your/potential.yaml` - or provide this path through the command line interface `text      pacemaker input.yaml -p /path/to/your/potential.yaml` doing this will override specifications in the `input.yaml`.

If you want to extend the basis (aka do the ladder scheme fitting): - Specify your potential as initial potential `yaml      potential:              ...              initial_potential: /path/to/your/potential.yaml ...` - or use the CLI: `text      pacemaker input.yaml -ip /path/to/your/potential.yaml`

7.6 I want to preserve the "shape" of potential, but refit it from scratch

```yaml
#input.yaml

potential:
  filename: /path/to/your/potential.yaml
  reset: true
```

It will reset potential from potential.yaml, i.e. set radial coefficients to delta_nk and B-basis function coefficients to zero.

7.7 My potential behaves unphysical at short distances, how to fix it?

If training data lacks data at shorter distances, expected repulsive behaviour is not always reproduced. In order to avoid it, you should use core-repulsion potential when you define the potential in `input.yaml` which replaces ACE potential with an exponential repulsion:

```yaml
## input.yaml

potential:
  embeddings:
    ALL: {
      ...
      # core repulsion parameters
      rho_core_cut: 5,
      drho_core_cut: 5
      ...
    }

  bonds:
    ALL: {
      ## inner cutoff, applied in a range [r_in - delta_in, r_in]
      r_in: 2.3, # distance, below which the core repulsion start
      delta_in: 0.1,

      ## core-repulsion parameters `prefactor` and `lambda` in
      ## prefactor*exp(-lambda*r^2)/r, >0 only r<r_in+delta_in
      core-repulsion: [1e3, 1.0],
    }
```

If you did not specify it before the fit, you still could add it after with Python API:

```python
from pyace import *

bbasisconf = BBasisConfiguration("original_potential.yaml")
```

```
for block in bbasisconf.funcspecs_blocks:
    block.r_in = 2.3 # minimal interatomic distance in dataset
    block.delta_in = 0.1
    block.core_rep_parameters=[1e3, 1.0]
    block.rho_cut = block.drho_cut = 5
bbasisconf.save("tuned_potential.yaml")
```

or by manually changing corresponding parameters in `original_potential.yaml` file.

NOTE However, it is strongly recommended to add more data, that describe the behaviour ar short distances rather than relying on the core repulsion completely.

7.8 How to split train/test data for the fitting?

Just use `test_size` keyword in `input.yaml::data`:

```
data:
  test_size: 0.1 # for 10% of data used for testing
```

Alternatively, you can provide train and test datasets separately:

```
data:
  filename: /path/to/train_data.pckl.gzip
  test_filename: /path/to/test_data.pckl.gzip
```

7.9 I want to change the cutoff, what should I do ?

If you decrease the cutoff, i.e. from `rcut: 7` to `rcut: 6.5`, then no neighbours will be lost, and you could continue to use the dataset, but it would be less computational efficient.

If you increase the cutoff, then it is necessary to rebuild the neighbour lists by adding `--rebuild` option to pacemaker, i.e.

pacemaker . . . –rebuild

7.10 How better to organize my dataset files ?

It is recommended to store all dataset files (i.e. `df*.pckl.gzip`) in one folder and specify the environment variable `$PACEMAKERDATAPATH` (exectue it in terminal or add to for example `.bashrc`)

export PACEMAKERDATAPATH=/path/to/my/datases/files

7.11 What are good values for regularization parameters ?

Ideally, one would prefer avoid using regularizations and would use additional data instead. When this is not possible, it is recommended that relative contribution of the regularization terms into the total loss do not exceed a few percents. So, regularization parameters of order **1e-5** ∼ **1e-8** are good initial values, but check their relative contribution in detailed statistics, printed every `input.yaml::backed::display_step` step.

7.12 How to fit only certain part of the potential, i.e. binary interaction only ?

If you have already fitted potential `Al.yaml` and `Ni.yaml` and would like to create a binary potential by fitting to binary data, i.e. AlNi structures, then in `input.yaml::potential` you could provide only binary interaction parts:

```
potential:
  deltaSplineBins: 0.001,
  elements: [Al, Ni],
  bonds: {
        BINARY: {
            rcut': 6.2,
            dcut': 0.01,
            core-repulsion': [0.0, 5.0],
            radbase': ChebExpCos,
            radparameters': [5.25]
    }
  }
  functions: {
      BINARY: {
        nradmax_by_orders:  [5, 2, 2, 1],
        lmax_by_orders: [0, 2, 2, 1],
        }
  }

  ## provide list of initial potentials
  initial_potential: [Al.yaml, Ni.yaml]
```

and in `input.yaml::fit` you add

```
fit:
  ...
  trainable_parameters: BINARY
  ...
```

7.13 I see different metrics text files during the fit, what is it ?

All metrics files contain values of loss function (*loss*), its energy/forces contributions (*e_loss_contrib*, *f_loss_contrib*), regularization contributions (*reg_loss*) and also root mean squared error (RMSE)/ mean absolute error (MAE) (*rmse_**, *mae_**) of energies (*rmse_epa*,*mae_epa*) and forces (norm of error vector *rmse_f* and per-component *mae_f_comp*) for whole dataset as well as for structures within 1eV/atom above minumum (*low_**). `metrics.txt` and `test_metrics.txt` are update every train/test step, whereas `ladder_metrics.txt/test_ladder_metrics.txt` are updated after each ladder step and `cycle_metrics.txt/test_cycle_metrics.txt` are updated after each cycle on ladder step.

7.14 Optimization stops too early due to too small updates, but I want to run it longer...

You need to decrease certain tolerance parameters for corresponding minimization algorithm. For example, for BFGS, there is `gtol: 1e-5` default parameter, that you could decrease in `input.yaml`

```
fit:
  options: {gtol: 5e-7}
```

7.15 How to create a custom weights dataframe for ExternalWeightingPolicy?

7.16 How to add more weights to certain structures ?

7.17 How to run on-the-fly validation of the potential ?