DeePMD-kit

Deep Potential

Jun 21, 2021

CONTENTS:

CHAPTER

ONE

DOWNLOAD AND INSTALL

Please follow our github webpage to download the latest released version and development version.

1.1 Easy installation methods

There various easy methods to install DeePMD-kit. Choose one that you prefer. If you want to build by yourself, jump to the next two sections.

After your easy installation, DeePMD-kit (`dp`) and LAMMPS (`lmp`) will be available to execute. You can try `dp -h` and `lmp -h` to see the help. `mpirun` is also available considering you may want to run LAMMPS in parallel.

1.1.1 Offline packages

Both CPU and GPU version offline packages are avaiable in the Releases page.

1.1.2 With conda

DeePMD-kit is avaiable with conda. Install Anaconda or Miniconda first.

To install the CPU version:

```
conda install deepmd-kit=*=*cpu lammps-dp=*=*cpu -c deepmodeling
```

To install the GPU version containing CUDA 10.1:

```
conda install deepmd-kit=*=*gpu lammps-dp=*=*gpu -c deepmodeling
```

1.1.3 With Docker

A docker for installing the DeePMD-kit is available here.

To pull the CPU version:

```
docker pull ghcr.io/deepmodeling/deepmd-kit:1.2.2_cpu
```

To pull the GPU version:

```
docker pull ghcr.io/deepmodeling/deepmd-kit:1.2.2_cuda10.1_gpu
```

1.2 Install the python interface

1.2.1 Install the Tensorflow's python interface

First, check the python version on your machine

```
python --version
```

We follow the virtual environment approach to install the tensorflow's Python interface. The full instruction can be found on the tensorflow's official website. Now we assume that the Python interface will be installed to virtual environment directory `$tensorflow_venv`

```
virtualenv -p python3 $tensorflow_venv
source $tensorflow_venv/bin/activate
pip install --upgrade pip
pip install --upgrade tensorflow==2.3.0
```

It is highly recommanded to keep the consistency of the TensorFlow version for the python and C++ interfaces. Everytime a new shell is started and one wants to use `DeePMD-kit`, the virtual environment should be activated by

```
source $tensorflow_venv/bin/activate
```

if one wants to skip out of the virtual environment, he/she can do

```
deactivate
```

If one has multiple python interpreters named like python3.x, it can be specified by, for example

```
virtualenv -p python3.7 $tensorflow_venv
```

If one does not need the GPU support of deepmd-kit and is concerned about package size, the CPU-only version of tensorflow should be installed by

```
pip install --upgrade tensorflow-cpu==2.3.0
```

To verify the installation, run

```
python -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

One should remember to activate the virtual environment every time he/she uses deepmd-kit.

1.2.2 Install the DeePMD-kit's python interface

Execute

```
pip install deepmd-kit
```

To test the installation, one may execute

```
dp -h
```

It will print the help information like

```
usage: dp [-h] {train,freeze,test} ...

DeePMD-kit: A deep learning package for many-body potential energy
representation and molecular dynamics

optional arguments:
  -h, --help            show this help message and exit

Valid subcommands:
  {train,freeze,test}
    train               train a model
    freeze              freeze the model
    test                test the model
```

1.3 Install the C++ interface

If one does not need to use DeePMD-kit with Lammps or I-Pi, then the python interface installed in the previous section does everything and he/she can safely skip this section.

1.3.1 Install the Tensorflow's C++ interface

Check the compiler version on your machine

```
gcc --version
```

The C++ interface of DeePMD-kit was tested with compiler gcc >= 4.8. It is noticed that the I-Pi support is only compiled with gcc >= 4.9.

First the C++ interface of Tensorflow should be installed. It is noted that the version of Tensorflow should be in consistent with the python interface. We assume that you have followed our instruction and installed tensorflow python interface 1.14.0 with, then you may follow the instruction for CPU to install the corresponding C++ interface (CPU only). If one wants GPU supports, he/she should follow the instruction for GPU to install the C++ interface.

1.3.2 Install the DeePMD-kit's C++ interface

Clone the DeePMD-kit source code

```
cd /some/workspace
git clone --recursive https://github.com/deepmodeling/deepmd-kit.git deepmd-kit
```

For convenience, you may want to record the location of source to a variable, saying `deepmd_source_dir` by

```
cd deepmd-kit
deepmd_source_dir=`pwd`
```

Now goto the source code directory of DeePMD-kit and make a build place.

```
cd $deepmd_source_dir/source
mkdir build
cd build
```

I assume you want to install DeePMD-kit into path `$deepmd_root`, then execute cmake

```
cmake -DTENSORFLOW_ROOT=$tensorflow_root -DCMAKE_INSTALL_PREFIX=$deepmd_root ..
```

where the variable `tensorflow_root` stores the location where the tensorflow's C++ interface is installed. The DeePMD-kit will automatically detect if a CUDA tool-kit is available on your machine and build the GPU support accordingly. If you want to force the cmake to find CUDA tool-kit, you can speicify the key `USE_CUDA_TOOLKIT`,

```
cmake -DUSE_CUDA_TOOLKIT=true -DTENSORFLOW_ROOT=$tensorflow_root -DCMAKE_INSTALL_PREFIX=
↪$deepmd_root ..
```

and you may further asked to provide `CUDA_TOOLKIT_ROOT_DIR`. If the cmake has executed successfully, then

```
make
make install
```

If everything works fine, you will have the following executable and libraries installed in `$deepmd_root/bin` and `$deepmd_root/lib`

```
$ ls $deepmd_root/bin
dp_ipi
$ ls $deepmd_root/lib
libdeepmd_ipi.so  libdeepmd_op.so  libdeepmd.so
```

1.3.3 Install LAMMPS's DeePMD-kit module

DeePMD-kit provide module for running MD simulation with LAMMPS. Now make the DeePMD-kit module for LAMMPS.

```
cd $deepmd_source_dir/source/build
make lammps
```

DeePMD-kit will generate a module called `USER-DEEPMD` in the `build` directory. Now download your favorite LAMMPS code, and uncompress it (I assume that you have downloaded the tar `lammps-stable.tar.gz`)

```
cd /some/workspace
tar xf lammps-stable.tar.gz
```

The source code of LAMMPS is stored in directory, for example `lammps-31Mar17`. Now go into the LAMMPS code and copy the DeePMD-kit module like this

```
cd lammps-31Mar17/src/
cp -r $deepmd_source_dir/source/build/USER-DEEPMD .
```

Now build LAMMPS

```
make yes-user-deepmd
make mpi -j4
```

The option `-j4` means using 4 processes in parallel. You may want to use a different number according to your hardware.

If everything works fine, you will end up with an executable `lmp_mpi`.

The DeePMD-kit module can be removed from LAMMPS source code by

```
make no-user-deepmd
```

- *Use DeePMD-kit*
  - *Prepare data*
  - *Train a model*
    - *The DeePMD model*
    - *The DeepPot-SE model*
  - *Freeze a model*
  - *Test a model*
  - *Model inference*
  - *Run MD with Lammps*
    - *Include deepmd in the pair style*
    - *Long-range interaction*
  - *Run path-integral MD with i-PI*
  - *Use deep potential with ASE*

CHAPTER

TWO

USE DEEPMD-KIT

In this text, we will call the deep neural network that is used to represent the interatomic interactions (Deep Potential) the **model**. The typical procedure of using DeePMD-kit is

1. Prepare data
2. Train a model
3. Freeze the model
4. Test the model
5. Inference with the model

2.1 Prepare data

One needs to provide the following information to train a model: the atom type, the simulation box, the atom coordinate, the atom force, system energy and virial. A snapshot of a system that contains these information is called a **frame**. We use the following convention of units:

Property| Unit — | :—: Time | ps Length | Å Energy | eV Force | eV/Å Pressure| Bar

The frames of the system are stored in two formats. A raw file is a plain text file with each information item written in one file and one frame written on one line. The default files that provide box, coordinate, force, energy and virial are `box.raw`, `coord.raw`, `force.raw`, `energy.raw` and `virial.raw`, respectively. *We recommend you use these file names*. Here is an example of force.raw:

```
$ cat force.raw
-0.724  2.039 -0.951  0.841 -0.464  0.363
 6.737  1.554 -5.587 -2.803  0.062  2.222
-1.968 -0.163  1.020 -0.225 -0.789  0.343
```

This `force.raw` contains 3 frames with each frame having the forces of 2 atoms, thus it has 3 lines and 6 columns. Each line provides all the 3 force components of 2 atoms in 1 frame. The first three numbers are the 3 force components of the first atom, while the second three numbers are the 3 force components of the second atom. The coordinate file `coord.raw` is organized similarly. In `box.raw`, the 9 components of the box vectors should be provided on each line. In `virial.raw`, the 9 components of the virial tensor should be provided on each line. The number of lines of all raw files should be identical.

We assume that the atom types do not change in all frames. It is provided by `type.raw`, which has one line with the types of atoms written one by one. The atom types should be integers. For example the `type.raw` of a system that has 2 atoms with 0 and 1:

```
$ cat type.raw
0 1
```

The second format is the data sets of `numpy` binary data that are directly used by the training program. User can use the script `$deepmd_source_dir/data/raw/raw_to_set.sh` to convert the prepared raw files to data sets. For example, if we have a raw file that contains 6000 frames,

```
$ ls
box.raw  coord.raw  energy.raw  force.raw  type.raw  virial.raw
$ $deepmd_source_dir/data/raw/raw_to_set.sh 2000
nframe is 6000
nline per set is 2000
will make 3 sets
making set 0 ...
making set 1 ...
making set 2 ...
$ ls
box.raw  coord.raw  energy.raw  force.raw  set.000  set.001  set.002  type.raw  virial.
↪raw
```

It generates three sets `set.000`, `set.001` and `set.002`, with each set contains 2000 frames. The last set (`set.002`) is used as testing set, while the rest sets (`set.000` and `set.001`) are used as training sets. One do not need to take care of the binary data files in each of the `set.*` directories. The path containing `set.*` and `type.raw` is called a *system*.

2.2 Train a model

2.2.1 Write the input script

The method of training is explained in our [DeePMD][2] and [DeepPot-SE][3] papers. With the source code we provide a small training dataset taken from 400 frames generated by NVT ab-initio water MD trajectory with 300 frames for training and 100 for testing. An example training parameter file is provided. One can try with the training by

```
$ cd $deepmd_source_dir/examples/water/train/
$ dp train water_se_a.json
```

where `water_se_a.json` is the `json` format parameter file that controls the training. It is also possible to use `yaml` format file with the same keys as json (see `water_se_a.yaml` example). You can use script `json2yaml.py` in `data/json/` dir to convert your json files to yaml. The components of the `water.json` contains four parts, `model`, `learning_rate`, `loss` and `training`.

The `model` section specify how the deep potential model is built. An example of the smooth-edition is provided as follows

```
    "model": {
        "type_map":        ["O", "H"],
        "descriptor" :{
            "type":             "se_a",
            "rcut_smth":       5.80,
            "rcut":             6.00,
            "sel":             [46, 92],
            "neuron":             [25, 50, 100],
            "axis_neuron":      16,
```

(continues on next page)

(continued from previous page)

```
            "resnet_dt":        false,
            "seed":                1,
            "_comment":                " that's all"
        },
        "fitting_net" : {
            "neuron":               [240, 240, 240],
            "resnet_dt":        true,
            "seed":                1,
            "_comment":                " that's all"
        },
        "_comment":         " that's all"
    }
```

The **type_map** is optional, which provide the element names (but not restricted to) for corresponding atom types.

The construction of the descriptor is given by option **descriptor**. The **type** of the descriptor is set to `"se_a"`, which means smooth-edition, angular infomation. The **rcut** is the cut-off radius for neighbor searching, and the **rcut_smth** gives where the smoothing starts. **sel** gives the maximum possible number of neighbors in the cut-off radius. It is a list, the length of which is the same as the number of atom types in the system, and `sel[i]` denote the maximum possible number of neighbors with type `i`. The **neuron** specifies the size of the embedding net. From left to right the members denote the sizes of each hidden layers from input end to the output end, respectively. The **axis_neuron** specifies the size of submatrix of the embedding matrix, the axis matrix as explained in the [DeepPot-SE paper][3]. If the outer layer is of twice size as the inner layer, then the inner layer is copied and concatenated, then a ResNet architecture is build between them. If the option **resnet_dt** is set `true`, then a timestep is used in the ResNet. **seed** gives the random seed that is used to generate random numbers when initializing the model parameters.

The construction of the fitting net is give by **fitting_net**. The key **neuron** specifies the size of the fitting net. If two neighboring layers are of the same size, then a ResNet architecture is build between them. If the option **resnet_dt** is set `true`, then a timestep is used in the ResNet. **seed** gives the random seed that is used to generate random numbers when initializing the model parameters.

An example of the `learning_rate` is given as follows

```
    "learning_rate" :{
        "type":                "exp",
        "start_lr":         0.005,
        "decay_steps":         5000,
        "decay_rate":         0.95,
        "_comment":         "that's all"
    }
```

The option **start_lr**, **decay_rate** and **decay_steps** specify how the learning rate changes. For example, the `t`th batch will be trained with learning rate:

```
lr(t) = start_lr * decay_rate ^ ( t / decay_steps )
```

An example of the `loss` is

```
    "loss" : {
        "start_pref_e":         0.02,
        "limit_pref_e":         1,
        "start_pref_f":         1000,
        "limit_pref_f":         1,
        "start_pref_v":         0,
```

(continues on next page)

(continued from previous page)

```
        "limit_pref_v":         0,
        "_comment":         " that's all"
    }
```

The options **start_pref_e**, **limit_pref_e**, **start_pref_f**, **limit_pref_f**, **start_pref_v** and **limit_pref_v** determine how the prefactors of energy error, force error and virial error changes in the loss function (see the appendix of the [DeePMD paper][2] for details). Taking the prefactor of force error for example, the prefactor at batch `t` is

```
w_f(t) = start_pref_f * ( lr(t) / start_lr ) + limit_pref_f * ( 1 - lr(t) / start_lr )
```

Since we do not have virial data, the virial prefactors `start_pref_v` and `limit_pref_v` are set to 0.

An example of `training` is

```
    "training" : {
        "systems":         ["../data1/", "../data2/"],
        "set_prefix":         "set",
        "stop_batch":         1000000,
        "_comment": " batch_size can be supplied with, e.g. 1, or auto (string) or [10,␣
↪20]",
        "batch_size":         1,

        "seed":                 1,

        "_comment": " display and restart",
        "_comment": " frequencies counted in batch",
        "disp_file":         "lcurve.out",
        "disp_freq":         100,
        "_comment": " numb_test can be supplied with, e.g. 1, or XX% (string) or [10, 20]
↪",
        "numb_test":         10,
        "save_freq":         1000,
        "save_ckpt":         "model.ckpt",
        "load_ckpt":         "model.ckpt",
        "disp_training":true,
        "time_training":true,
        "profiling":         false,
        "profiling_file":"timeline.json",
        "_comment":         "that's all"
    }
```

The option **systems** provide location of the systems (path to `set.*` and `type.raw`). It is a vector, thus DeePMD-kit allows you to provide multiple systems. DeePMD-kit will train the model with the systems in the vector one by one in a cyclic manner. **It is warned that the example water data (in folder `examples/data/water`) is of very limited amount, is provided only for testing purpose, and should not be used to train a productive model.**

The option **batch_size** specifies the number of frames in each batch. It can be set to `"auto"` to enable a automatic batch size or it can be input as a list setting batch size individually for each system. The option **stop_batch** specifies the total number of batches will be used in the training.

The option **numb_test** specifies the number of tests that will be used for each system. If it is an integer each system will be tested with the same number of tests. It can be set to percentage **"XX%"** to use XX% of frames of each system for its testing or it can be input as a list setting numer of tests individually for each system (the order should correspond to ordering of the systems key in json).

2.2.2 Training

The training can be invoked by

```
$ dp train water_se_a.json
```

During the training, the error of the model is tested every **disp_freq** batches with **numb_test** frames from the last set in the **systems** directory on the fly, and the results are output to **disp_file**. A typical `disp_file` looks like

```
# batch      l2_tst    l2_trn    l2_e_tst  l2_e_trn    l2_f_tst  l2_f_trn        lr
      0    2.67e+01  2.57e+01    2.21e-01  2.22e-01    8.44e-01  8.12e-01    1.0e-03
    100    6.14e+00  5.40e+00    3.01e-01  2.99e-01    1.93e-01  1.70e-01    1.0e-03
    200    5.02e+00  4.49e+00    1.53e-01  1.53e-01    1.58e-01  1.42e-01    1.0e-03
    300    4.36e+00  3.71e+00    7.32e-02  7.27e-02    1.38e-01  1.17e-01    1.0e-03
    400    4.04e+00  3.29e+00    3.16e-02  3.22e-02    1.28e-01  1.04e-01    1.0e-03
```

The first column displays the number of batches. The second and third columns display the loss function evaluated by `numb_test` frames randomly chosen from the test set and that evaluated by the current training batch, respectively. The fourth and fifth columns display the RMS energy error (normalized by number of atoms) evaluated by `numb_test` frames randomly chosen from the test set and that evaluated by the current training batch, respectively. The sixth and seventh columns display the RMS force error (component-wise) evaluated by `numb_test` frames randomly chosen from the test set and that evaluated by the current training batch, respectively. The last column displays the current learning rate.

Checkpoints will be written to files with prefix **save_ckpt** every **save_freq** batches. If **restart** is set to `true`, then the training will start from the checkpoint named **load_ckpt**, rather than from scratch.

Several command line options can be passed to `dp train`, which can be checked with

```
$ dp train --help
```

An explanation will be provided

```
positional arguments:
  INPUT                 the input json database

optional arguments:
  -h, --help            show this help message and exit
  --init-model INIT_MODEL
                        Initialize a model by the provided checkpoint
  --restart RESTART     Restart the training from the provided checkpoint
```

The keys `intra_op_parallelism_threads` and `inter_op_parallelism_threads` are Tensorflow configurations for multithreading, which are explained here. Skipping `-t` and `OMP_NUM_THREADS` leads to the default setting of these keys in the Tensorflow.

`--init-model model.ckpt`, for example, initializes the model training with an existing model that is stored in the checkpoint `model.ckpt`, the network architectures should match.

`--restart model.ckpt`, continues the training from the checkpoint `model.ckpt`.

On some resources limited machines, one may want to control the number of threads used by DeePMD-kit. This is achieved by three environmental variables: `OMP_NUM_THREADS`, `TF_INTRA_OP_PARALLELISM_THREADS` and `TF_INTER_OP_PARALLELISM_THREADS`. `OMP_NUM_THREADS` controls the multithreading of DeePMD-kit implemented operations. `TF_INTRA_OP_PARALLELISM_THREADS` and `TF_INTER_OP_PARALLELISM_THREADS` controls `intra_op_parallelism_threads` and `inter_op_parallelism_threads`, which are Tensorflow configurations for multithreading. An explanation is found here.

For example if you wish to use 3 cores of 2 CPUs on one node, you may set the environmental variables and run DeePMD-kit as follows:

```
export OMP_NUM_THREADS=6
export TF_INTRA_OP_PARALLELISM_THREADS=3
export TF_INTER_OP_PARALLELISM_THREADS=2
dp train input.json
```

2.3 Freeze a model

The trained neural network is extracted from a checkpoint and dumped into a database. This process is called "freezing" a model. The idea and part of our code are from Morgan. To freeze a model, typically one does

```
$ dp freeze -o graph.pb
```

in the folder where the model is trained. The output database is called `graph.pb`.

2.4 Test a model

The frozen model can be used in many ways. The most straightforward test can be performed using `dp test`. A typical usage of `dp test` is

```
dp test -m graph.pb -s /path/to/system -n 30
```

where `-m` gives the tested model, `-s` the path to the tested system and `-n` the number of tested frames. Several other command line options can be passed to `dp test`, which can be checked with

```
$ dp test --help
```

An explanation will be provided

```
usage: dp test [-h] [-m MODEL] [-s SYSTEM] [-S SET_PREFIX] [-n NUMB_TEST]
               [-r RAND_SEED] [--shuffle-test] [-d DETAIL_FILE]

optional arguments:
  -h, --help            show this help message and exit
  -m MODEL, --model MODEL
                        Frozen model file to import
  -s SYSTEM, --system SYSTEM
                        The system dir
  -S SET_PREFIX, --set-prefix SET_PREFIX
                        The set prefix
  -n NUMB_TEST, --numb-test NUMB_TEST
                        The number of data for test
  -r RAND_SEED, --rand-seed RAND_SEED
                        The random seed
  --shuffle-test        Shuffle test data
  -d DETAIL_FILE, --detail-file DETAIL_FILE
                        The file containing details of energy force and virial
                        accuracy
```

2.5 Model inference

One may use the python interface of DeePMD-kit for model inference, an example is given as follows

```
import deepmd.DeepPot as DP
import numpy as np
dp = DP('graph.pb')
coord = np.array([[1,0,0], [0,0,1.5], [1,0,3]]).reshape([1, -1])
cell = np.diag(10 * np.ones(3)).reshape([1, -1])
atype = [1,0,1]
e, f, v = dp.eval(coord, cell, atype)
```

where `e`, `f` and `v` are predicted energy, force and virial of the system, respectively.

2.6 Run MD with LAMMPS

2.6.1 Include deepmd in the pair style

Running an MD simulation with LAMMPS is simpler. In the LAMMPS input file, one needs to specify the pair style as follows

```
pair_style     deepmd graph.pb
pair_coeff
```

where `graph.pb` is the file name of the frozen model. The `pair_coeff` should be left blank. It should be noted that LAMMPS counts atom types starting from 1, therefore, all LAMMPS atom type will be firstly subtracted by 1, and then passed into the DeePMD-kit engine to compute the interactions. A detailed documentation of this pair style is available..

2.6.2 Long-range interaction

The reciprocal space part of the long-range interaction can be calculated by LAMMPS command `kspace_style`. To use it with DeePMD-kit, one writes

```
pair_style        deepmd graph.pb
pair_coeff
kspace_style        pppm 1.0e-5
kspace_modify        gewald 0.45
```

Please notice that the DeePMD does nothing to the direct space part of the electrostatic interaction, because this part is assumed to be fitted in the DeePMD model (the direct space cut-off is thus the cut-off of the DeePMD model). The splitting parameter `gewald` is modified by the `kspace_modify` command.

2.7 Run path-integral MD with i-PI

The i-PI works in a client-server model. The i-PI provides the server for integrating the replica positions of atoms, while the DeePMD-kit provides a client named `dp_ipi` that computes the interactions (including energy, force and virial). The server and client communicates via the Unix domain socket or the Internet socket. The client can be started by

```
$ dp_ipi water.json
```

It is noted that multiple instances of the client is allow for computing, in parallel, the interactions of multiple replica of the path-integral MD.

`water.json` is the parameter file for the client `dp_ipi`, and an example is provided:

```
{
    "verbose":                false,
    "use_unix":                true,
    "port":                  31415,
    "host":                  "localhost",
    "graph_file":           "graph.pb",
    "coord_file":           "conf.xyz",
    "atom_type" : {
        "OW":                   0,
        "HW1":                  1,
        "HW2":                  1
    }
}
```

The option **`use_unix`** is set to `true` to activate the Unix domain socket, otherwise, the Internet socket is used.

The option **`graph_file`** provides the file name of the frozen model.

The `dp_ipi` gets the atom names from an XYZ file provided by **`coord_file`** (meanwhile ignores all coordinates in it), and translates the names to atom types by rules provided by **`atom_type`**.

2.8 Use deep potential with ASE

Deep potential can be set up as a calculator with ASE to obtain potential energies and forces.

```
from ase import Atoms
from deepmd.calculator import DP

water = Atoms('H2O',
              positions=[(0.7601, 1.9270, 1),
                         (1.9575, 1, 1),
                         (1., 1., 1.)],
              cell=[100, 100, 100],
              calculator=DP(model="frozen_model.pb"))
print(water.get_potential_energy())
print(water.get_forces())
```

Optimization is also available:

```
from ase.optimize import BFGS
dyn = BFGS(water)
dyn.run(fmax=1e-6)
print(water.get_positions())
```

CHAPTER

THREE

TRAINING PARAMETERS

model:

type: `dict`
argument path: `model`

type_map:

type: `list`, optional
argument path: `model/type_map`

A list of strings. Give the name to each type of atoms.

data_stat_nbatch:

type: `int`, optional, default: `10`
argument path: `model/data_stat_nbatch`

The model determines the normalization from the statistics of the data. This key specifies the number of *frames* in each *system* used for statistics.

use_srtab:

type: `str`, optional
argument path: `model/use_srtab`

The table for the short-range pairwise interaction added on top of DP. The table is a text data file with (N_t + 1) * N_t / 2 + 1 columes. The first colume is the distance between atoms. The second to the last columes are energies for pairs of certain types. For example we have two atom types, 0 and 1. The columes from 2nd to 4th are for 0-0, 0-1 and 1-1 correspondingly.

smin_alpha:

type: `float`, optional
argument path: `model/smin_alpha`

The short-range tabulated interaction will be swithed according to the distance of the nearest neighbor. This distance is calculated by softmin. This parameter is the decaying parameter in the softmin. It is only required when *use_srtab* is provided.

sw_rmin:

type: `float`, optional
argument path: `model/sw_rmin`

The lower boundary of the interpolation between short-range tabulated interaction and DP. It is only required when *use_srtab* is provided.

sw_rmax:

type: `float`, optional
argument path: `model/sw_rmax`

The upper boundary of the interpolation between short-range tabulated interaction and DP. It is only required when *use_srtab* is provided.

descriptor:

type: `dict`
argument path: `model/descriptor`

The descriptor of atomic environment.

Depending on the value of *type*, different sub args are accepted.

type:

type: `str` (flag key)
argument path: `model/descriptor/type`

The type of the descritpor. Valid types are *loc_frame*, *se_a*, *se_r* and *se_ar*.

- *loc_frame*: Defines a local frame at each atom, and the compute the descriptor as local coordinates under this frame.
- *se_a*: Used by the smooth edition of Deep Potential. The full relative coordinates are used to construct the descriptor.
- *se_r*: Used by the smooth edition of Deep Potential. Only the distance between atoms is used to construct the descriptor.
- *se_ar*: A hybrid of *se_a* and *se_r*. Typically *se_a* has a smaller cut-off while the *se_r* has a larger cut-off.

When *type* is set to `loc_frame`:

sel_a:

type: `list`
argument path: `model/descriptor[loc_frame]/sel_a`

A list of integers. The length of the list should be the same as the number of atom types in the system. *sel_a[i]* gives the selected number of type-i neighbors. The full relative coordinates of the neighbors are used by the descriptor.

sel_r:

type: `list`
argument path: `model/descriptor[loc_frame]/sel_r`

A list of integers. The length of the list should be the same as the number of atom types in the system. *sel_r[i]* gives the selected number of type-i neighbors. Only relative distance of the neighbors are used by the descriptor. sel_a[i] + sel_r[i] is recommended to be larger than the maximally possible number of type-i neighbors in the cut-off radius.

rcut:

type: `float`, optional, default: `6.0`
argument path: `model/descriptor[loc_frame]/rcut`

The cut-off radius. The default value is 6.0

axis_rule:

type: `list`

argument path: model/descriptor[loc_frame]/axis_rule

A list of integers. The length should be 6 times of the number of types.

- axis_rule[i*6+0]: class of the atom defining the first axis of type-i atom. 0 for neighbors with full coordinates and 1 for neighbors only with relative distance.
- axis_rule[i*6+1]: type of the atom defining the first axis of type-i atom.
- axis_rule[i*6+2]: index of the axis atom defining the first axis. Note that the neighbors with the same class and type are sorted according to their relative distance.
- axis_rule[i*6+3]: class of the atom defining the first axis of type-i atom. 0 for neighbors with full coordinates and 1 for neighbors only with relative distance.
- axis_rule[i*6+4]: type of the atom defining the second axis of type-i atom.
- axis_rule[i*6+5]: class of the atom defining the second axis of type-i atom. 0 for neighbors with full coordinates and 1 for neighbors only with relative distance.

When *type* is set to se_a:

sel:

type: list
argument path: model/descriptor[se_a]/sel

A list of integers. The length of the list should be the same as the number of atom types in the system. *sel[i]* gives the selected number of type-i neighbors. *sel[i]* is recommended to be larger than the maximally possible number of type-i neighbors in the cut-off radius.

rcut:

type: float, optional, default: 6.0
argument path: model/descriptor[se_a]/rcut

The cut-off radius.

rcut_smth:

type: float, optional, default: 0.5
argument path: model/descriptor[se_a]/rcut_smth

Where to start smoothing. For example the 1/r term is smoothed from *rcut* to *rcut_smth*

neuron:

type: list, optional, default: [10, 20, 40]
argument path: model/descriptor[se_a]/neuron

Number of neurons in each hidden layers of the embedding net. When two layers are of the same size or one layer is twice as large as the previous layer, a skip connection is built.

axis_neuron:

type: int, optional, default: 4
argument path: model/descriptor[se_a]/axis_neuron

Size of the submatrix of G (embedding matrix).

activation_function:

type: str, optional, default: tanh
argument path: model/descriptor[se_a]/activation_function

The activation function in the embedding net. Supported activation functions are “relu”, “relu6”, “softplus”, “sigmoid”, “tanh”, “gelu”.

resnet_dt:

type: `bool`, optional, default: `False`
argument path: `model/descriptor[se_a]/resnet_dt`

Whether to use a “Timestep” in the skip connection

type_one_side:

type: `bool`, optional, default: `False`
argument path: `model/descriptor[se_a]/type_one_side`

Try to build N_types embedding nets. Otherwise, building N_types^2 embedding nets

precision:

type: `str`, optional, default: `float64`
argument path: `model/descriptor[se_a]/precision`

The precision of the embedding net parameters, supported options are “float64”, “float32”, “float16”.

trainable:

type: `bool`, optional, default: `True`
argument path: `model/descriptor[se_a]/trainable`

If the parameters in the embedding net is trainable

seed:

type: `int | NoneType`, optional
argument path: `model/descriptor[se_a]/seed`

Random seed for parameter initialization

exclude_types:

type: `list`, optional, default: `[]`
argument path: `model/descriptor[se_a]/exclude_types`

The Excluded types

set_davg_zero:

type: `bool`, optional, default: `False`
argument path: `model/descriptor[se_a]/set_davg_zero`

Set the normalization average to zero. This option should be set when *atom_ener* in the energy fitting is used

When *type* is set to `se_r`:

sel:

type: `list`
argument path: `model/descriptor[se_r]/sel`

A list of integers. The length of the list should be the same as the number of atom types in the system. *sel[i]* gives the selected number of type-i neighbors. *sel[i]* is recommended to be larger than the maximally possible number of type-i neighbors in the cut-off radius.

rcut:

type: float, optional, default: 6.0
argument path: model/descriptor[se_r]/rcut

The cut-off radius.

rcut_smth:

type: float, optional, default: 0.5
argument path: model/descriptor[se_r]/rcut_smth

Where to start smoothing. For example the 1/r term is smoothed from *rcut* to *rcut_smth*

neuron:

type: list, optional, default: [10, 20, 40]
argument path: model/descriptor[se_r]/neuron

Number of neurons in each hidden layers of the embedding net. When two layers are of the same size or one layer is twice as large as the previous layer, a skip connection is built.

activation_function:

type: str, optional, default: tanh
argument path: model/descriptor[se_r]/activation_function

The activation function in the embedding net. Supported activation functions are “relu”, “relu6”, “softplus”, “sigmoid”, “tanh”, “gelu”.

resnet_dt:

type: bool, optional, default: False
argument path: model/descriptor[se_r]/resnet_dt

Whether to use a “Timestep” in the skip connection

type_one_side:

type: bool, optional, default: False
argument path: model/descriptor[se_r]/type_one_side

Try to build N_types embedding nets. Otherwise, building N_types^2 embedding nets

precision:

type: str, optional, default: float64
argument path: model/descriptor[se_r]/precision

The precision of the embedding net parameters, supported options are “float64”, “float32”, “float16”.

trainable:

type: bool, optional, default: True
argument path: model/descriptor[se_r]/trainable

If the parameters in the embedding net is trainable

seed:

type: int | NoneType, optional
argument path: model/descriptor[se_r]/seed

Random seed for parameter initialization

exclude_types:

type: list, optional, default: []

argument path: model/descriptor[se_r]/exclude_types

The Excluded types

set_davg_zero:

type: bool, optional, default: False
argument path: model/descriptor[se_r]/set_davg_zero

Set the normalization average to zero. This option should be set when *atom_ener* in the energy fitting is used

When *type* is set to se_ar:

a:

type: dict
argument path: model/descriptor[se_ar]/a

The parameters of descriptor *se_a*

r:

type: dict
argument path: model/descriptor[se_ar]/r

The parameters of descriptor *se_r*

fitting_net:

type: dict
argument path: model/fitting_net

The fitting of physical properties.

Depending on the value of *type*, different sub args are accepted.

type:

type: str (flag key), default: ener
argument path: model/fitting_net/type

The type of the fitting. Valid types are *ener*, *dipole*, *polar* and *global_polar*.

- *ener*: Fit an energy model (potential energy surface).
- *dipole*: Fit an atomic dipole model. Atomic dipole labels for all the selected atoms (see *sel_type*) should be provided by *dipole.npy* in each data system. The file has number of frames lines and 3 times of number of selected atoms columns.
- *polar*: Fit an atomic polarizability model. Atomic polarizability labels for all the selected atoms (see *sel_type*) should be provided by *polarizability.npy* in each data system. The file has number of frames lines and 9 times of number of selected atoms columns.
- *global_polar*: Fit a polarizability model. Polarizability labels should be provided by *polarizability.npy* in each data system. The file has number of frames lines and 9 columns.

When *type* is set to ener:

numb_fparam:

type: int, optional, default: 0
argument path: model/fitting_net[ener]/numb_fparam

The dimension of the frame parameter. If set to >0, file *fparam.npy* should be included to provided the input fparams.

numb_aparam:

type: `int`, optional, default: `0`

argument path: `model/fitting_net[ener]/numb_aparam`

The dimension of the atomic parameter. If set to >0, file *aparam.npy* should be included to provided the input aparams.

neuron:

type: `list`, optional, default: `[120, 120, 120]`

argument path: `model/fitting_net[ener]/neuron`

The number of neurons in each hidden layers of the fitting net. When two hidden layers are of the same size, a skip connection is built.

activation_function:

type: `str`, optional, default: `tanh`

argument path: `model/fitting_net[ener]/activation_function`

The activation function in the fitting net. Supported activation functions are “relu”, “relu6”, “softplus”, “sigmoid”, “tanh”, “gelu”.

precision:

type: `str`, optional, default: `float64`

argument path: `model/fitting_net[ener]/precision`

The precision of the fitting net parameters, supported options are “float64”, “float32”, “float16”.

resnet_dt:

type: `bool`, optional, default: `True`

argument path: `model/fitting_net[ener]/resnet_dt`

Whether to use a “Timestep” in the skip connection

trainable:

type: `bool` | `list`, optional, default: `True`

argument path: `model/fitting_net[ener]/trainable`

Whether the parameters in the fitting net are trainable. This option can be

- bool: True if all parameters of the fitting net are trainable, False otherwise.
- list of bool: Specifies if each layer is trainable. Since the fitting net is composed by hidden layers followed by a output layer, the length of tihs list should be equal to len(*neuron*)+1.

rcond:

type: `float`, optional, default: `0.001`

argument path: `model/fitting_net[ener]/rcond`

The condition number used to determine the inital energy shift for each type of atoms.

seed:

type: `int` | `NoneType`, optional

argument path: `model/fitting_net[ener]/seed`

Random seed for parameter initialization of the fitting net

atom_ener:

type: list, optional, default: []
argument path: model/fitting_net[ener]/atom_ener

Specify the atomic energy in vacuum for each type

When *type* is set to dipole:

neuron:

type: list, optional, default: [120, 120, 120]
argument path: model/fitting_net[dipole]/neuron

The number of neurons in each hidden layers of the fitting net. When two hidden layers are of the same size, a skip connection is built.

activation_function:

type: str, optional, default: tanh
argument path: model/fitting_net[dipole]/activation_function

The activation function in the fitting net. Supported activation functions are "relu", "relu6", "softplus", "sigmoid", "tanh", "gelu".

resnet_dt:

type: bool, optional, default: True
argument path: model/fitting_net[dipole]/resnet_dt

Whether to use a "Timestep" in the skip connection

precision:

type: str, optional, default: float64
argument path: model/fitting_net[dipole]/precision

The precision of the fitting net parameters, supported options are "float64", "float32", "float16".

sel_type:

type: int | NoneType | list, optional
argument path: model/fitting_net[dipole]/sel_type

The atom types for which the atomic dipole will be provided. If not set, all types will be selected.

seed:

type: int | NoneType, optional
argument path: model/fitting_net[dipole]/seed

Random seed for parameter initialization of the fitting net

When *type* is set to polar:

neuron:

type: list, optional, default: [120, 120, 120]
argument path: model/fitting_net[polar]/neuron

The number of neurons in each hidden layers of the fitting net. When two hidden layers are of the same size, a skip connection is built.

activation_function:

type: str, optional, default: tanh
argument path: model/fitting_net[polar]/activation_function

The activation function in the fitting net. Supported activation functions are “relu”, “relu6”, “softplus”, “sigmoid”, “tanh”, “gelu”.

resnet_dt:

type: `bool`, optional, default: `True`
argument path: `model/fitting_net[polar]/resnet_dt`

Whether to use a “Timestep” in the skip connection

precision:

type: `str`, optional, default: `float64`
argument path: `model/fitting_net[polar]/precision`

The precision of the fitting net parameters, supported options are “float64”, “float32”, “float16”.

fit_diag:

type: `bool`, optional, default: `True`
argument path: `model/fitting_net[polar]/fit_diag`

Fit the diagonal part of the rotational invariant polarizability matrix, which will be converted to normal polarizability matrix by contracting with the rotation matrix.

scale:

type: `float` | `list`, optional, default: `1.0`
argument path: `model/fitting_net[polar]/scale`

The output of the fitting net (polarizability matrix) will be scaled by `scale`

diag_shift:

type: `float` | `list`, optional, default: `0.0`
argument path: `model/fitting_net[polar]/diag_shift`

The diagonal part of the polarizability matrix will be shifted by `diag_shift`. The shift operation is carried out after `scale`.

sel_type:

type: `int` | `NoneType` | `list`, optional
argument path: `model/fitting_net[polar]/sel_type`

The atom types for which the atomic polarizability will be provided. If not set, all types will be selected.

seed:

type: `int` | `NoneType`, optional
argument path: `model/fitting_net[polar]/seed`

Random seed for parameter initialization of the fitting net

When *type* is set to `global_polar`:

neuron:

type: `list`, optional, default: `[120, 120, 120]`
argument path: `model/fitting_net[global_polar]/neuron`

The number of neurons in each hidden layers of the fitting net. When two hidden layers are of the same size, a skip connection is built.

activation_function:

type: str, optional, default: tanh
argument path: model/fitting_net[global_polar]/activation_function

The activation function in the fitting net. Supported activation functions are "relu", "relu6", "softplus", "sigmoid", "tanh", "gelu".

resnet_dt:

type: bool, optional, default: True
argument path: model/fitting_net[global_polar]/resnet_dt

Whether to use a "Timestep" in the skip connection

precision:

type: str, optional, default: float64
argument path: model/fitting_net[global_polar]/precision

The precision of the fitting net parameters, supported options are "float64", "float32", "float16".

fit_diag:

type: bool, optional, default: True
argument path: model/fitting_net[global_polar]/fit_diag

Fit the diagonal part of the rotational invariant polarizability matrix, which will be converted to normal polarizability matrix by contracting with the rotation matrix.

scale:

type: float | list, optional, default: 1.0
argument path: model/fitting_net[global_polar]/scale

The output of the fitting net (polarizability matrix) will be scaled by scale

diag_shift:

type: float | list, optional, default: 0.0
argument path: model/fitting_net[global_polar]/diag_shift

The diagonal part of the polarizability matrix will be shifted by diag_shift. The shift operation is carried out after scale.

sel_type:

type: int | NoneType | list, optional
argument path: model/fitting_net[global_polar]/sel_type

The atom types for which the atomic polarizability will be provided. If not set, all types will be selected.

seed:

type: int | NoneType, optional
argument path: model/fitting_net[global_polar]/seed

Random seed for parameter initialization of the fitting net

loss:

type: dict
argument path: loss

The definition of loss function. The type of the loss depends on the type of the fitting. For fitting type *ener*, the prefactors before energy, force, virial and atomic energy losses may be provided. For fitting type *dipole*, *polar* and *global_polar*, the loss may be an empty *dict* or unset.

Depending on the value of *type*, different sub args are accepted.

type:

type: `str` (flag key), default: `ener`
argument path: `loss/type`

The type of the loss. For fitting type *ener*, the loss type should be set to *ener* or left unset. For tensorial fitting types *dipole*, *polar* and *global_polar*, the type should be left unset. .

When *type* is set to `ener`:

start_pref_e:

type: `float | int`, optional, default: `0.02`
argument path: `loss[ener]/start_pref_e`

The prefactor of energy loss at the start of the training. Should be larger than or equal to 0. If set to none-zero value, the energy label should be provided by file energy.npy in each data system. If both start_pref_energy and limit_pref_energy are set to 0, then the energy will be ignored.

limit_pref_e:

type: `float | int`, optional, default: `1.0`
argument path: `loss[ener]/limit_pref_e`

The prefactor of energy loss at the limit of the training, Should be larger than or equal to 0. i.e. the training step goes to infinity.

start_pref_f:

type: `float | int`, optional, default: `1000`
argument path: `loss[ener]/start_pref_f`

The prefactor of force loss at the start of the training. Should be larger than or equal to 0. If set to none-zero value, the force label should be provided by file force.npy in each data system. If both start_pref_force and limit_pref_force are set to 0, then the force will be ignored.

limit_pref_f:

type: `float | int`, optional, default: `1.0`
argument path: `loss[ener]/limit_pref_f`

The prefactor of force loss at the limit of the training, Should be larger than or equal to 0. i.e. the training step goes to infinity.

start_pref_v:

type: `float | int`, optional, default: `0.0`
argument path: `loss[ener]/start_pref_v`

The prefactor of virial loss at the start of the training. Should be larger than or equal to 0. If set to none-zero value, the virial label should be provided by file virial.npy in each data system. If both start_pref_virial and limit_pref_virial are set to 0, then the virial will be ignored.

limit_pref_v:

type: `float | int`, optional, default: `0.0`
argument path: `loss[ener]/limit_pref_v`

The prefactor of virial loss at the limit of the training, Should be larger than or equal to 0. i.e. the training step goes to infinity.

start_pref_ae:

type: float | int, optional, default: 0.0
argument path: loss[ener]/start_pref_ae

The prefactor of virial loss at the start of the training. Should be larger than or equal to 0. If set to none-zero value, the virial label should be provided by file virial.npy in each data system. If both start_pref_virial and limit_pref_virial are set to 0, then the virial will be ignored.

limit_pref_ae:

type: float | int, optional, default: 0.0
argument path: loss[ener]/limit_pref_ae

The prefactor of virial loss at the limit of the training, Should be larger than or equal to 0. i.e. the training step goes to infinity.

relative_f:

type: float | NoneType, optional
argument path: loss[ener]/relative_f

If provided, relative force error will be used in the loss. The difference of force will be normalized by the magnitude of the force in the label with a shift given by *relative_f*, i.e. DF_i / (|| F || + relative_f) with DF denoting the difference between prediction and label and || F || denoting the L2 norm of the label.

learning_rate:

type: dict
argument path: learning_rate

The learning rate options

start_lr:

type: float, optional, default: 0.001
argument path: learning_rate/start_lr

The learning rate the start of the training.

stop_lr:

type: float, optional, default: 1e-08
argument path: learning_rate/stop_lr

The desired learning rate at the end of the training.

decay_steps:

type: int, optional, default: 5000
argument path: learning_rate/decay_steps

The learning rate is decaying every this number of training steps.

training:

type: dict
argument path: training

The training options

systems:

type: list | str
argument path: training/systems

The data systems. This key can be provided with a listthat specifies the systems, or be provided with a string by which the prefix of all systems are given and the list of the systems is automatically generated.

set_prefix:

type: `str`, optional, default: `set`
argument path: `training/set_prefix`

The prefix of the sets in the systems.

stop_batch:

type: `int`
argument path: `training/stop_batch`

Number of training batch. Each training uses one batch of data.

batch_size:

type: `int` | `list` | `str`, optional, default: `auto`
argument path: `training/batch_size`

This key can be

- list: the length of which is the same as the *systems*. The batch size of each system is given by the elements of the list.
- int: all *systems* uses the same batch size.
- string "auto": automatically determines the batch size os that the batch_size times the number of atoms in the system is no less than 32.
- string "auto:N": automatically determines the batch size os that the batch_size times the number of atoms in the system is no less than N.

seed:

type: `int` | `NoneType`, optional
argument path: `training/seed`

The random seed for training.

disp_file:

type: `str`, optional, default: `lcueve.out`
argument path: `training/disp_file`

The file for printing learning curve.

disp_freq:

type: `int`, optional, default: `1000`
argument path: `training/disp_freq`

The frequency of printing learning curve.

numb_test:

type: `int` | `list` | `str`, optional, default: `1`
argument path: `training/numb_test`

Number of frames used for the test during training.

save_freq:

type: `int`, optional, default: `1000`

argument path: `training/save_freq`

The frequency of saving check point.

save_ckpt:

type: `str`, optional, default: `model.ckpt`
argument path: `training/save_ckpt`

The file name of saving check point.

disp_training:

type: `bool`, optional, default: `True`
argument path: `training/disp_training`

Displaying verbose information during training.

time_training:

type: `bool`, optional, default: `True`
argument path: `training/time_training`

Timing durining training.

profiling:

type: `bool`, optional, default: `False`
argument path: `training/profiling`

Profiling during training.

profiling_file:

type: `str`, optional, default: `timeline.json`
argument path: `training/profiling_file`

Output file for profiling.

CHAPTER

FOUR

PAIR_STYLE DEEPMD COMMAND

4.1 Syntax

```
pair_style deepmd models ... keyword value ...
```

- deepmd = style of this pair_style
- models = frozen model(s) to compute the interaction. If multiple models are provided, then the model deviation will be computed
- keyword = *out_file* or *out_freq* or *fparam* or *atomic* or *relative*

4.2 Examples

```
pair_style deepmd graph.pb
pair_style deepmd graph.pb fparam 1.2
pair_style deepmd graph_0.pb graph_1.pb graph_2.pb out_file md.out out_freq 10 atomic␣
↪relative 1.0
```

4.3 Description

Evaluate the interaction of the system by using Deep Potential or Deep Potential Smooth Edition. It is noticed that deep potential is not a "pairwise" interaction, but a multi-body interaction.

This pair style takes the deep potential defined in a model file that usually has the .pb extension. The model can be trained and frozen by package DeePMD-kit.

The model deviation evalulate the consistency of the force predictions from multiple models. By default, only the maximal, minimal and averge model deviations are output. If the key `atomic` is set, then the model deviation of force prediction of each atom will be output.

By default, the model deviation is output in absolute value. If the keyword `relative` is set, then the relative model deviation will be output. The relative model deviation of the force on atom `i` is defined by

```
           |Df_i|
Ef_i = -------------
       |f_i| + level
```

where `Df_i` is the absolute model deviation of the force on atom `i`, `|f_i|` is the norm of the the force and `level` is provided as the parameter of the keyword `relative`.

4.4 Restrictions

- The `deepmd` pair style is provided in the USER-DEEPMD package, which is compiled from the DeePMD-kit, visit the DeePMD-kit website for more information.
- The `atom_style` of the system should be `atomic`.
- When using the `atomic` key word of `deepmd` is set, one should not use this pair style with MPI parallelization.

CHAPTER

FIVE

DEEPMD-KIT API

class deepmd.Data.**DataSets**(*sys_path*, *set_prefix*, *seed=None*, *shuffle_test=True*)

check_batch_size(*batch_size*)

check_test_size(*test_size*)

get_batch(*batch_size*)
returned property prefector [4] in order: energy, force, virial, atom_ener

get_ener()

get_natoms()

get_natoms_2(*ntypes*)

get_natoms_vec(*ntypes*)

get_numb_set()

get_set(*data*, *idx=None*)

get_sys_numb_batch(*batch_size*)

get_test()
returned property prefector [4] in order: energy, force, virial, atom_ener

get_type_map()

load_batch_set(*set_name*)

load_data(*set_name*, *data_name*, *shape*, *is_necessary=True*)

load_energy(*set_name*, *nframes*, *nvalues*, *energy_file*, *atom_energy_file*)
return : coeff_ener, ener, coeff_atom_ener, atom_ener

load_set(*set_name*, *shuffle=True*)

load_test_set(*set_name*, *shuffle_test*)

load_type(*sys_path*)

load_type_map(*sys_path*)

numb_aparam()

numb_fparam()

reset_iter()

set_numb_batch(*batch_size*)

stats_energy()

class deepmd.Data.**DeepmdData**(*sys_path*, *set_prefix='set'*, *shuffle_test=True*, *type_map=None*, *modifier=None*)

add(*key*, *ndof*, *atomic=False*, *must=False*, *high_prec=False*, *type_sel=None*, *repeat=1*)

avg(*key*)

check_batch_size(*batch_size*)

check_test_size(*test_size*)

get_atom_type()

get_batch(*batch_size*)

get_data_dict()

get_natoms()

get_natoms_vec(*ntypes*)

get_ntypes()

get_numb_batch(*batch_size*, *set_idx*)

get_numb_set()

get_sys_numb_batch(*batch_size*)

get_test(*ntests=- 1*)

get_type_map()

reduce(*key_out*, *key_in*)

reset_get_batch()

class deepmd.DataModifier.**DipoleChargeModifier**(*model_name*, *model_charge_map*, *sys_charge_map*, *ewald_h=1*, *ewald_beta=1*)

build_fv_graph()

eval(*coord*, *box*, *atype*, *eval_fv=True*)

eval_fv(*coords*, *cells*, *atom_types*, *ext_f*)

modify_data(*data*)

class deepmd.DataSystem.**DataSystem**(*systems*, *set_prefix*, *batch_size*, *test_size*, *rcut*, *run_opt=None*)

check_type_map_consistency(*type_map_list*)

compute_energy_shift()

format_name_length(*name*, *width*)

get_batch(*sys_idx=None*, *sys_weights=None*, *style='prob_sys_size'*)

get_batch_size()

get_nbatches()

get_nsystems()

get_ntypes()

get_sys(*sys_idx*)

get_test(*sys_idx=None*)

get_type_map()

numb_fparam()

print_summary(*run_opt*)

process_sys_weights(*sys_weights*)

class deepmd.DataSystem.**DeepmdDataSystem**(*systems*, *batch_size*, *test_size*, *rcut*, *set_prefix='set'*, *shuffle_test=True*, *type_map=None*, *modifier=None*)

add(*key*, *ndof*, *atomic=False*, *must=False*, *high_prec=False*, *type_sel=None*, *repeat=1*)

add_dict(*adict*)

compute_energy_shift(*rcond=0.001*, *key='energy'*)

get_batch(*sys_idx=None*, *sys_probs=None*, *auto_prob_style='prob_sys_size'*)
Get a batch of data from the data system

sys_idx: int The index of system from which the batch is get. If sys_idx is not None, *sys_probs* and *auto_prob_style* are ignored If sys_idx is None, automatically determine the system according to *sys_probs* or *auto_prob_style*, see the following.

sys_probs: list of float The probabilitis of systems to get the batch. Summation of positive elements of this list should be no greater than 1. Element of this list can be negative, the probability of the corresponding system is determined automatically by the number of batches in the system.

auto_prob_style: float Determine the probability of systems automatically. The method is assigned by this key and can be - "prob_uniform" : the probability all the systems are equal, namely 1.0/self.get_nsystems() - "prob_sys_size" : the probability of a system is proportional to the number of batches in the system - "prob_sys_size;stt_idx:end_idx:weight;stt_idx:end_idx:weight;..." :

the list of systems is devided into blocks. A block is specified by *stt_idx:end_idx:weight*, where *stt_idx* is the starting index of the system, *end_idx* is then ending (not including) index of the system, the probabilities of the systems in this block sums up to *weight*, and the relatively probabilities within this block is proportional to the number of batches in the system.

get_batch_size()

get_data_dict()

get_nbatches()

get_nsystems()

get_ntypes()

get_sys(*idx*)

get_sys_ntest(*sys_idx=None*)
Get number of tests for the currently selected system, or one defined by sys_idx.

get_test(*sys_idx=None*, *n_test=- 1*)

get_type_map()

print_summary(*run_opt*, *sys_probs=None*, *auto_prob_style='prob_sys_size'*)

reduce(*key_out*, *key_in*)

class deepmd.DeepDipole.**DeepDipole**(*model_file*, *load_prefix='load'*, *default_tf_graph=False*)

class deepmd.DeepEval.**DeepEval**(*model_file*, *load_prefix='load'*, *default_tf_graph=False*)
common methods for DeepPot, DeepWFC, DeepPolar, ...

make_natoms_vec(*atom_types*)

reverse_map(*vec*, *imap*)

sort_input(*coord*, *atom_type*, *sel_atoms=None*)

class deepmd.DeepEval.**DeepTensor**(*model_file*, *variable_name*, *variable_dof*, *load_prefix='load'*, *default_tf_graph=False*)
Evaluates a tensor model

eval(*coords*, *cells*, *atom_types*, *atomic=True*)

get_ntypes()

get_rcut()

get_sel_type()

get_type_map()

class deepmd.DeepPolar.**DeepGlobalPolar**(*model_file*, *default_tf_graph=False*)

eval(*coords*, *cells*, *atom_types*)

class deepmd.DeepPolar.**DeepPolar**(*model_file*, *default_tf_graph=False*)

class deepmd.DeepPot.**DeepPot**(*model_file*, *default_tf_graph=False*)

eval(*coords*, *cells*, *atom_types*, *fparam=None*, *aparam=None*, *atomic=False*)

eval_inner(*coords*, *cells*, *atom_types*, *fparam=None*, *aparam=None*, *atomic=False*)

get_dim_aparam()

get_dim_fparam()

get_ntypes()

get_rcut()

get_type_map()

class deepmd.DeepWFC.**DeepWFC**(*model_file*, *default_tf_graph=False*)

class deepmd.DescrptLocFrame.**DescrptLocFrame**(*jdata*)

build(*coord_*, *atype_*, *natoms*, *box_*, *mesh*, *suffix=''*, *reuse=None*)

compute_input_stats(*data_coord*, *data_box*, *data_atype*, *natoms_vec*, *mesh*)

get_dim_out()

get_nlist()

get_ntypes()

get_rcut()

get_rot_mat()

prod_force_virial(*atom_ener*, *natoms*)

class deepmd.DescrptSeA.**DescrptSeA**(*jdata*)

build(*coord_*, *atype_*, *natoms*, *box_*, *mesh*, *suffix=''*, *reuse=None*)

compute_input_stats(*data_coord*, *data_box*, *data_atype*, *natoms_vec*, *mesh*)

get_dim_out()

get_dim_rot_mat_1()

get_nlist()

get_ntypes()

get_rcut()

get_rot_mat()

prod_force_virial(*atom_ener*, *natoms*)

class deepmd.DescrptSeAR.**DescrptSeAR**(*jdata*)

build(*coord_*, *atype_*, *natoms*, *box*, *mesh*, *suffix=''*, *reuse=None*)

compute_input_stats(*data_coord*, *data_box*, *data_atype*, *natoms_vec*, *mesh*)

get_dim_out()

get_nlist_a()

get_nlist_r()

get_ntypes()

get_rcut()

prod_force_virial(*atom_ener*, *natoms*)

class deepmd.DescrptSeR.**DescrptSeR**(*jdata*)

build(*coord_*, *atype_*, *natoms*, *box_*, *mesh*, *suffix=''*, *reuse=None*)

compute_input_stats(*data_coord*, *data_box*, *data_atype*, *natoms_vec*, *mesh*)

get_dim_out()

get_nlist()

get_ntypes()

get_rcut()

prod_force_virial(*atom_ener*, *natoms*)

class deepmd.EwaldRecp.**EwaldRecp**(*hh*, *beta*)

eval(*coord*, *charge*, *box*)

class deepmd.Fitting.**DipoleFittingSeA**(*jdata*, *descrpt*)

build(*input_d*, *rot_mat*, *natoms*, *reuse=None*, *suffix=''*)

get_out_size()

get_sel_type()

class deepmd.Fitting.**EnerFitting**(*jdata*, *descrpt*)

build(*inputs*, *input_dict*, *natoms*, *reuse=None*, *suffix=''*)

compute_input_stats(*all_stat*, *protection*)

compute_output_stats(*all_stat*)

get_numb_aparam()

get_numb_fparam()

class deepmd.Fitting.**GlobalPolarFittingSeA**(*jdata*, *descrpt*)

build(*input_d*, *rot_mat*, *natoms*, *reuse=None*, *suffix=''*)

get_out_size()

get_sel_type()

class deepmd.Fitting.**PolarFittingLocFrame**(*jdata*, *descrpt*)

build(*input_d*, *rot_mat*, *natoms*, *reuse=None*, *suffix=''*)

get_out_size()

get_sel_type()

class deepmd.Fitting.**PolarFittingSeA**(*jdata*, *descrpt*)

build(*input_d*, *rot_mat*, *natoms*, *reuse=None*, *suffix=''*)

compute_input_stats(*all_stat*, *protection=0.01*)

get_out_size()

get_sel_type()

class deepmd.Fitting.**WFCFitting**(*jdata*, *descrpt*)

build(*input_d*, *rot_mat*, *natoms*, *reuse=None*, *suffix=''*)

get_out_size()

get_sel_type()

get_wfc_numb()

class deepmd.LearningRate.**LearningRateExp**(*jdata*)

build(*global_step*, *stop_batch=None*)

start_lr()

value(*batch*)

class deepmd.Loss.**EnerDipoleLoss**(*jdata*, ***kwarg*)

build(*learning_rate*, *natoms*, *model_dict*, *label_dict*, *suffix*)

print_header()

print_on_training(*sess*, *natoms*, *feed_dict_test*, *feed_dict_batch*)

class deepmd.Loss.**EnerStdLoss**(*jdata*, ***kwarg*)

build(*learning_rate*, *natoms*, *model_dict*, *label_dict*, *suffix*)

print_header()

print_on_training(*sess*, *natoms*, *feed_dict_test*, *feed_dict_batch*)

class deepmd.Loss.**TensorLoss**(*jdata*, ***kwarg*)

build(*learning_rate*, *natoms*, *model_dict*, *label_dict*, *suffix*)

print_header()

print_on_training(*sess*, *natoms*, *feed_dict_test*, *feed_dict_batch*)

class deepmd.Model.**DipoleModel**(*jdata*, *descrpt*, *fitting*)

class deepmd.Model.**GlobalPolarModel**(*jdata*, *descrpt*, *fitting*)

class deepmd.Model.**Model**(*jdata*, *descrpt*, *fitting*)

build(*coord_*, *atype_*, *natoms*, *box*, *mesh*, *input_dict*, *suffix=''*, *reuse=None*)

data_stat(*data*)

get_ntypes()

get_rcut()

get_type_map()

model_type = 'ener'

class deepmd.Model.**PolarModel**(*jdata*, *descrpt*, *fitting*)

class deepmd.Model.**TensorModel**(*jdata*, *descrpt*, *fitting*, *var_name*)

build(*coord_*, *atype_*, *natoms*, *box*, *mesh*, *input_dict*, *suffix=''*, *reuse=None*)

data_stat(*data*)

get_ntypes()

get_out_size()

get_rcut()

get_sel_type()

get_type_map()

class deepmd.Model.**WFCModel**(*jdata*, *descrpt*, *fitting*)

deepmd.Model.**make_all_stat**(*data*, *nbatches*, *merge_sys=True*)

pack data for statistics Parameters ———- data:

The data

merge_sys: bool (True) Merge system data

all_stat: A dictionary of list of list storing data for stat. if merge_sys == False data can be accessed by

all_stat[key][sys_idx][batch_idx][frame_idx]

else merge_sys == True can be accessed by all_stat[key][batch_idx][frame_idx]

deepmd.Model.**merge_sys_stat**(*all_stat*)

deepmd.Network.**one_layer**(*inputs*, *outputs_size*, *activation_fn=<function tanh>*, *precision=tf.float64*, *stddev=1.0*, *bavg=0.0*, *name='linear'*, *reuse=None*, *seed=None*, *use_timestep=False*, *trainable=True*, *useBN=False*)

class deepmd.TabInter.**TabInter**(*filename*)

get()

reinit(*filename*)

class deepmd.Trainer.**NNPTrainer**(*jdata*, *run_opt*)

build(*data*, *stop_batch=0*)

get_global_step()

print_head()

test_on_the_fly(*fp*, *data*, *feed_dict_batch*)

train(*data*)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

E

F

G

L

M

N

O

P

R

S

T

V

W