DeePKS-kit

Dec 14, 2022

CONTENTS

DeePKS-kit is a program to generate accurate energy functionals for quantum chemistry systems (in connection with PySCF) and periodic systems (in connection with ABACUS), for both perturbative scheme (DeePHF) and self-consistent scheme (DeePKS).

This documentation will currently focus on running DeePKS-kit for periodic systems, i.e., in connection with ABACUS. For molecular systems, we refer the users to DeePKS-kit documentation on GitHub.

Important: The project DeePKS-kit is licensed under GNU LGPLv3.0. If you use this code in any future publications, please cite *Yixiao Chen, Linfeng Zhang, Han Wang, and Weinan E. "DeePKS-kit: a package for developing machine learning-based chemically accurate energy and density functional models." arXiv:2012.14615v2.*

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 DeePKS-kit

DeePKS-kit is a pure python library so it can be installed following the standard *git clone* then *pip install* procedure. Note that the two main requirements *pytorch* and *ABACUS* will not be installed automatically so you will need to install them manually in advance. Below is a more detailed instruction that includes installing the required libraries in the environment.

We use *conda* here as an example. So first you may need to install Anaconda or Miniconda.

To reduce the possibility of library conflicts, we suggest create a new environment (named *deepks*) with basic dependencies installed (optional):

```
conda create -n deepks numpy scipy h5py ruamel.yaml paramiko
conda activate deepks
```

Now you are in the new environment called *deepks*. Next, install PyTorch

```
# assuming a GPU with cudatoolkit 10.2 support
conda install pytorch cudatoolkit=10.2 -c pytorch
```

Once the environment has been setup properly, using *pip* to install DeePKS-kit:

```
$ pip install git+https://github.com/deepmodeling/deepks-kit@abacus
```

1.1.2 ABACUS with DeePKS enabled

To run DeePKS-kit in connection with ABACUS, users first need to install ABACUS with DeePKS enabled. Details of such installation guide can be found at installation with DeePKS.

1.1.3 DPDispatcher (optional)

While DeePKS-kit has its built-in job dispacther, users are welcome to use DPDispatcher for automatic job submission. The usage of these two types of dispatchers is given in xxx. DPDispacther can simply be installed via

```
$ pip install dpdispatcher
```

More details about DPDispacther can be found via DPDispatcher's documentation.

1.2 Getting Started

To give it a shot on a DeePKS-ABACUS sample run, users may try the single water example provided here.

In this example, 1000 structures of the single water molecules with corresponding PBE property labels (including energy and force) have been prepared in advance. Four subfolders, i.e., `group.00-03` can be found under the folder `systems`. `group.00-group.02` contain 300 frames each and can be applied as training sets, while `group.03` contains 100 frames and can be applied as testing set. More details about the file structures and preparation are introduced at *Label preperation*.

This sample job can either be run on a local machine or on Bohrium. Users may modify the input files to make it run on various environment following the instruction in *Input files preperation*. To run this job on a local machine, simply issue:

```
cd deepks-kit/examples/water_single_lda2pbe_abacus/iter
bash run.sh
```

To run this job on Bohrium (which uses DPDispacther for job submission and data gathering), simply issue:

```
cd deepks-kit/examples/water_single_lda2pbe_abacus/iter
bash run_dpdispatcher.sh
```

Outputs generated during the process are introduced in *Important outputs explanation*.

1.3 Label preperation

1.3.1 System structure file

To train a DeePKS model, users must provide the structures of the interested system(s). Structures can be obtained either from a short AIMD run or by adding structural perturbations on top of an optimized geometry. The structures of the system can be provided via three formats as follows

- **(recommended) grouped into** `atom.npy`

  The shape of `atom.npy` file is [nframes, natoms, 4]:

  - **nframes** refers to the number of frames (structures) of the interested system;
  - **natoms** refers to the number of atoms of the interested system, e.g., for single water system, **natoms = 3**;
  - the last dimension **4** corresponds to the nuclear charge of the given atom and its *xyz* coordinates (either in Cartesian form or in fractional form, which needs to be specified with keyword `coord_type` in *scf_abacus.yaml*).

Note: If coordinates saved in `atom.npy` are in unit of *Bohr*, then `lattice_constant` should be set to 1 in *scf_abacus.yaml*. If coordinates saved in `atom.npy` are in unit of *Angstrom*, then `lattice_constant` should be set to 1.8897259886 in *scf_abacus.yaml*. If fractional coordinates are used, `lattice_constant` also needs to be set accordingly. See ABACUS documentation for more details.

- **grouped into `coord.npy` and `type.raw`**

  `coord.npy` is very similar to `atom.npy` with the shape **[nframes, natoms, 3]**. The only difference is that the nuclear charge is not included in the last dimension, which is included in `type.raw` instead. **Note that this format has not been fully tested for periodic systems.**

- **single xyz**

  Save the xyz coordinate of each frame as single xyz file, e.g., `0000.xyz`, `0001.xyz`,... **Note that this format has not been fully tested for periodic systems.**

It should be noted that if the lattice vectors of each frame are *not* the same, users should specify the lattice vector for each frame via `box.npy`, of which the shape is **[nframe, 9]**. If the prepared structures share the same lattice vector, then users may specify it as a keyword in input files. See *scf_abacus.yaml* for details.

1.3.2 Property labels

To train a DeePKS model, the target energy of the interested system is required, and its format should follow the format of the structure file. Additional properties can also be trained, including *force*, *stress*, and *bandgap*. The formats of structure files (taking *atom.npy* as an example) and the corresponding formats of various property labels are summarized as follows:

Filename	Description	Shape	Unit
atom.npy	structural file, **required**	[nframes, natoms, 4]	Bohr or Angstrom or fractional
box.npy	lattice vector file, optional	[nframes, 9]	Bohr or Angstrom
energy.npy	energy label, **required**	[nframes,1]	Hartree
force.npy	force label, optional	[nframes, natoms, 3]	Hartree/Bohr
stress.npy	virial vector file, optional	[nframes, 9]	Hartree
orbital.npy	bandgap label, optional	[nframes,1]	Hartree

1.4 Input files preperation

To run DeePKS-kit in connection with ABACUS, a bunch of input files are required so as to iteratively perform the SCF jobs on ABACUS and the training jobs on DeePKS-kit. Here we will use **single water molecule** as an example to show the required input files for the training of an **LDA**-based DeePKS model that provides **PBE** target energies and forces.

As can be seen in this example, 1000 structures of the single water molecules with corresponding PBE property labels (including energy and force) have been prepared in advance. Four subfolders, i.e., `group.00-03` can be found under the folder `systems`. `group.00-group.02` contain 300 frames each and can be applied as training sets, while `group.03` contains 100 frames and can be applied as testing set. The prepared file structure of a ready-to-run DeePKS iterative traning process should basically look like

```
└── water_single_lda2pbe_abacus
    ├── iter
    │   ├── H_gga_6au_60Ry_2s1p.orb
    │   ├── H_ONCV_PBE-1.0.upf
    │   ├── jle.orb
    │   ├── machines_dpdispatcher.yaml
    │   ├── machines.yaml
    │   ├── O_gga_6au_60Ry_2s2p1d.orb
    │   ├── O_ONCV_PBE-1.0.upf
    │   ├── params.yaml
    │   ├── run_dpdispatcher.yaml
    │   ├── run.sh
    │   ├── scf_abacus.yaml
    │   └── systems.yaml
    ├── README.md
    └── systems
        ├── group.00
        │   ├── atom.npy
        │   ├── energy.npy
        │   └── force.npy
        ├── group.01
        │   ├── atom.npy
        │   ├── energy.npy
        │   └── force.npy
        ├── group.02
        │   ├── atom.npy
        │   ├── energy.npy
        │   └── force.npy
        └── group.03
            ├── atom.npy
            ├── energy.npy
            └── force.npy
```

1.4.1 scf_abacus.yaml

This file controls the SCF jobs performed in ABACUS. The `scf_abacus` block controls the SCF jobs after the init iteration, i.e., with DeePKS model loaded, while the `init_scf_abacus` controls the initial SCF jobs, i.e., bare LDA or PBE SCF calculaiton. The reason to divide this file into two blocks is that after the init iteration, the SCF calculaitons with DeePKS model loaded are sometimes found hard to converge to a tight threshold, e.g., `scf_thr = 1e-7`. Therefore we might want to slightly loose that threshold after the init iteration. Also, even users need to train the model with force label, there is no need to calculate force during the init SCF cycle, since the init training will include the energy label only.

Below is a sample `scf_abacus.yaml` file for single water molecule, with the explanation of each keyword. Please refer to ABACUS input file documentation for a more detailed explanation of the input parameters in ABACUS.

```
scf_abacus:
  # INPUT args; keywords that related to INPUT file in ABACUS
  ntype: 2                    # int; number of different atom species in this␣
↪calculations, e.g., 2 for H2O
  nbands: 8                   # int; number of bands to be calculated; optional
  ecutwfc: 50                 # real; energy cutoff, unit: Ry
  scf_thr: 1e-7               # real; SCF convergence threshold for density error; 5e-7␣
```

(continues on next page)

(continued from previous page)

```
↪and below is acceptable
  scf_nmax: 50                # int; maximum SCF iteration steps
  dft_functional: "lda"       # string; name of the baseline density functional
  gamma_only: 1               # bool; 1 for gamma-only calculation
  cal_force: 1                # bool; 1 for force calculation
  cal_stress: 0               # bool; 1 for stress calculation

  # STRU args; keywords that related to INPUT file in ABACUS
  # below are default STRU args, users can also set them for each group in
  # ../systems/group.xx/stru_abacus.yaml
  orb_files: ["O_gga_6au_60Ry_2s2p1d.orb", "H_gga_6au_60Ry_2s1p.orb"] # atomic orbital␣
↪file list for each element;
                                                                      # order should be␣
↪consistent with that in atom.npy
  pp_files: ["O_ONCV_PBE-1.0.upf", "H_ONCV_PBE-1.0.upf"]              # pseudopotential␣
↪file list for each element;
                                                                      # order should be␣
↪consistent with that in atom.npy
  proj_file: ["jle.orb"]                                              # projector file;␣
↪generated in ABACUS; see file desriptions for more details
  lattice_constant: 1                                                 # real; lattice␣
↪constant
  lattice_vector: [[28, 0, 0], [0, 28, 0], [0, 0, 28]]                # [3, 3] matrix;␣
↪lattice vectors
  coord_type: "Cartesian"                                             # "Cartesian" or
↪"Direct"; the latter is for fractional coordinates

  # cmd args; keywords that related to running ABACUS
  run_cmd : "mpirun"                                                  # run command
  abacus_path: "/usr/local/bin/abacus"                                # ABACUS␣
↪executable path

# below is the init_scf_abacus block, which is basically same as above
# just note that the recommended value for scf_thr is 1e-7,
# and force calculation can be omitted since the init training includes energy label␣
↪only.
init_scf_abacus:
  orb_files: ["O_gga_6au_60Ry_2s2p1d.orb", "H_gga_6au_60Ry_2s1p.orb"]
  pp_files: ["O_ONCV_PBE-1.0.upf", "H_ONCV_PBE-1.0.upf"]
  proj_file: ["jle.orb"]
  ntype: 2
  nbands: 8
  ecutwfc: 50
  scf_thr: 1e-7
  scf_nmax: 50
  dft_functional: "lda"
  gamma_only: 1
  cal_force: 0
  lattice_constant: 1
  lattice_vector: [[28, 0, 0], [0, 28, 0], [0, 0, 28]]
  coord_type: "Cartesian"
  #cmd args
```

(continues on next page)

(continued from previous page)

```
  run_cmd : "mpirun"
  abacus_path: "/usr/local/bin/abacus"
```

For multi k-points systems, the number of k-points can either be set explicitly as:

```
scf_abacus:
  <...other keywords>
  k_points: [4,4,4,0,0,0]
init_scf_abacus:
  <...other keywords>
  k_points: [4,4,4,0,0,0]
```

or via `kspacing` as:

```
scf_abacus:
  <...other keywords>
  kspacing: 0.1
init_scf_abacus:
  <...other keywords>
  kspacing: 0.1
```

1.4.2 machine.yaml

Note: This file is *not* required when running jobs on Bohrium via DPDispachter. In such case, users need to prepare *machine_dpdispatcher.yaml* instead.

To run ABACUS-DeePKS training process on a local machine or on a cluster via slurm or PBS, it is recommended to use the DeePKS built-in dispatcher and prepare `machine.yaml` file as follows.

```
# this is only part of input settings.
# should be used together with systems.yaml and params.yaml
scf_machine:
  group_size: 125         # number of SCF jobs that are grouped and submitted together;␣
↪these jobs will be run sequentially
  resources:
    task_per_node: 1      # number of CPUs for one SCF job

  sub_size: 1             # keyword for PySCF; set to 1 for ABACUS SCF jobs
  dispatcher:
    context: local        # "local" to run on local machine, or "ssh" to run on a remote␣
↪machine
    batch: shell          # set to shell to run on local machine, you can also use `slurm`␣
↪or `pbs`

train_machine:
  dispatcher:
    context: local        # "local" to run on local machine, or "ssh" to run on a remote␣
↪machine
    batch: shell          # set to shell to run on local machine, you can also use `slurm`␣
↪or `pbs`
```

(continues on next page)

(continued from previous page)

```
    remote_profile: null # use lazy local
  # resources are no longer needed, and the task will use gpu automatically if there is␣
↪one.
  python: "python"       # use python in path


# other settings (these are default; can be omitted)
cleanup: false           # whether to delete slurm and err files
strict: true             # do not allow undefined machine parameters

#paras for abacus
use_abacus: true         # use abacus in scf calculation
```

To run ABACUS-DeePKS via PBS or slurm, the following parameters can be specified under `resources` block in both `scf_machine` and `train_machine`:

```
# this is only part of input settings.
# should be used together with systems.yaml and params.yaml
scf_machine:
  <...other kerwords>
  resources:
    numb_node:          # int; number of nodes; default value is 1
    task_per_node:      # int; ppn required; default value is 1;
    numb_gpu:           # int; number of GPUs; default value is 1
    time_limit:         # time limit; default value is 1:0:0
    mem_limit:          # int; memeory limit in GB
    partition:          # string; queue name
    account:            # string; account info
    qos:                # string;
    module_list:        # e.g., [abacus]
    source_list:        # e.g., [/opt/intel/oneapi/setvars.sh; conda activate deepks]
    <... other keywords>
 train_machine:
   <...other kerwords>
   resources:
     <... same as above>
```

1.4.3 machine_dpdispatcher.yaml

Note: This file is *not* required when running jobs on a local machine or on a cluster via slurm or PBS *with the built-in dispatcher*. In such case, users may prepare *machine.yaml* instead. That being said, users may also modify keywords in this file to submit jobs to a cluster via slurm or PBS. Please refer to DPDispatcher documentation for more details on slurm/PBS job submission.

To run ABACUS-DeePKS on Bohrium or via slurm, users need to use DPDispatcher and prepare `machine_dpdispatcher.yaml` file as follows. Most of the keyword in this file share the same meaning as those in `machine.yaml`. The unique part here is to specify keywords in `dpdispatcher_resources:` block. Below is an example for running jobs in Bohrium:

```
# this is only part of input settings.
# should be used together with systems.yaml and params.yaml
scf_machine:
  resources:
    task_per_node: 4
  dispatcher: dpdispatcher
  dpdispatcher_resources:
    number_node: 1
    cpu_per_node: 8
    group_size: 125
    source_list: [/opt/intel/oneapi/setvars.sh]
  sub_size: 1
  dpdispatcher_machine:
    context_type: lebesguecontext
    batch_type: lebesgue
    local_root: ./
    remote_profile:
      email: (your-account-email)          # email address registered on Bohrium
      password: (your-password)            # password on Bohrium
      program_id: (your-program-id)        # program ID on Bohrium
      input_data:
        log_file: log.scf
        err_file: err.scf
        job_type: indicate
        grouped: true
        job_name: deepks-scf
        disk_size: 100
        scass_type: c8_m8_cpu              # machine type
        platform: ali
        image_name: abacus-workshop        # image name
        on_demand: 0
train_machine:
  dispatcher: dpdispatcher
  dpdispatcher_machine:
    context_type: lebesguecontext
    batch_type: lebesgue
    local_root: ./
    remote_profile:
      email: (your-account-email)
      password: (your-password)
      program_id: (your-program-id)
      input_data:
        log_file: log.train
        err_file: err.train
        job_type: indicate
        grouped: true
        job_name: deepks-train
        disk_size: 100
        scass_type: c8_m8_cpu
        platform: ali
        image_name: abacus-workshop
        on_demand: 0
  dpdispatcher_resources:
```

(continues on next page)

(continued from previous page)

```
    number_node: 1
    cpu_per_node: 8
    group_size: 1
    source_list: [~/.bashrc]
  python: "/usr/bin/python3" # use python in path
  # resources are no longer needed, and the task will use gpu automatically if there is␣
↪one

# other settings (these are default; can be omitted)
cleanup: false # whether to delete slurm and err files
strict: true # do not allow undefined machine parameters

#paras for abacus
use_abacus: true # use abacus in scf calculation
```

1.4.4 params.yaml

This file controls the init and iterative training processes performed in DeePKS-kit. Default values for hyperparameters set for the training process (as given below) are recommended for users who are not very experienced in machine-learning, while machine-learning gurus are welcome to play with them.

```
# this is only part of input settings.
# should be used together with systems.yaml and machines.yaml

# number of iterations to do, can be set to zero for DeePHF training
n_iter: 1

# directory setting (these are default choices, can be omitted)
workdir: "."
share_folder: "share" # folder that stores all other settings

# scf settings, set to false when n_iter = 0 to skip checking
scf_input: false


# train settings for training after init iteration,
# set to false when n_iter = 0 to skip checking
train_input:
  # model_args is omitted, which will inherit from init_train
  data_args:
    batch_size: 16          # training batch size; 16 is recommended
    group_batch: 1          # number of batches to be grouped; set to 1 for ABACUS-
↪related training
    extra_label: true       # set to true to train the model with force, stress, or␣
↪bandgap labels.
                            # note that these extra labels will only be included after␣
↪the init iteration
                            # only energy label will be included for the init training
    conv_filter: true       # if set to true (recommended), will read the convergence␣
↪data from conv_name
                            # and only use converged datapoints to train; including any␣
```

(continues on next page)

(continued from previous page)

```
↪unconverged
                        # datapoints may screw up the training!
    conv_name: conv     # npy file that records the converged datapoints
  preprocess_args:
    preshift: false     # restarting model already shifted. Will not recompute shift␣
↪value
    prescale: false     # same as above
    prefit_ridge: 1e1   # the ridge factor used in linear regression
    prefit_trainable: false # make the linear regression fixed during the training
  train_args:
    # start learning rate (lr) will decay a factor of `decay_rate` every `decay_steps`␣
↪epoches
    decay_rate: 0.5
    decay_steps: 1000
    display_epoch: 100  # show training results every n epoch
    force_factor: 1     # the prefactor multiplied infront of the force part of the␣
↪loss
    n_epoch: 5000       # total number of epoch needed in training
    start_lr: 0.0001    # the start learning rate, will decay later

# init training settings, these are for DeePHF task
init_model: false       # do not use existing model to restart from

init_scf: True          # whether to perform init SCF;

init_train:             # parameters for init nn training; basically the same as␣
↪those listed in train_input
  model_args:
    hidden_sizes: [100, 100, 100] # neurons in hidden layers
    output_scale: 100           # the output will be divided by 100 before compare␣
↪with label
    use_resnet: true            # skip connection
    actv_fn: mygelu             # same as gelu, support force calculation
  data_args:
    batch_size: 16
    group_batch: 1
  preprocess_args:
    preshift: true              # shift the descriptor by its mean
    prescale: false             # scale the descriptor by its variance (can cause␣
↪convergence problem)
    prefit_ridge: 1e1           # do a ridge regression as prefitting
    prefit_trainable: false
  train_args:
    decay_rate: 0.96
    decay_steps: 500
    display_epoch: 100
    n_epoch: 5000
    start_lr: 0.0003
```

Even though the DeePKS training scheme is relativley robust, there might be a chance that the SCF procedure fails to converge after loading the DeePKS model. Such convergence failure might be caused by insufficient variety of the training data, and/or the discontinuities issue due to the sorting of the eigenvalues in eigenvalue decomposition step when constructing the descriptors. One thing that worths trying is to add more training data with sufficient variety in

structure, and if the convergence failure remains or the training data is indeed sufficient, users may further symmetrize the descriptors by modifing the `init_train` block in `params.yaml` as follows:

```
init_train:                # parameters for init nn training; basically the same as␣
↪those listed in train_input
  proj_basis: [[0,[0,...,0]],
               [1, [0,...,0]],
               [2, [0,...,0]]] # projected basis for thermal embedding, 0, 1, and 2 in␣
↪the first column correspond to s, p, and d orbitals,
                               # and the number of zeros afterwards should equal the␣
↪number of Bessel functions in jle.orb.
  model_args:
    hidden_sizes: [100, 100, 100] # neurons in hidden layers
    output_scale: 100             # the output will be divided by 100 before compare␣
↪with label
    use_resnet: true              # skip connection
    actv_fn: mygelu               # same as gelu, support force calculation
    embedding: {embd_sizes: null, init_beta: 5, type: thermal} # apply thermal averaging␣
↪to further symmetrize the descriptors
  <...other keywords>
```

1.4.5 projector file

The descriptors applied in DeePKS model is generated from the projected density matrix, therefore a set of projectors are required in advance. To obtain these projectors for periodic system, users need to run a specific sample job in ABACUS. These projectors are products of spherical Bessel functions (radial part) and spherical harmonic functions (angular part), which are similar to numerical atomic orbitals. The number of Bessel functions are controled by the radial and wavefunction cutoff, for which 5 or 6 Bohr and `ecutwfc` set in *scf_abacus.yaml* are recommeded, respectively.

Note that it is not necessary to change the STRU file of this sample job, since all elements share the same descriptor. Basically, users *only* need to specify `calculation` as `gen_bessel` and then adjust the energy cutoff and the radial cutoff of the wavefunctions. The angular part is controled via the keyword `bessel_lmax` and the value 2 (including *s*, *p*, and *d* orbitals) is strongly recommended. See below for related input parameters:

```
calculation gen_bessel # calculation type should be gen_bessel
bessel_lmax 2   # maximum angular momentum for projectors; 2 is recommended
bessel_rcut 5   # radial cutoff in unit Bohr; 5 or 6 is recommended
ecutwfc   100   # kinetic energy cutoff in unit Ry; should be consistent with that set␣
↪for ABACUS SCF calculation
```

After running this sample job, users will find `jle.orb` in folder `OUT.abacus` and will need to copy this file to the `iter` folder.

Note: Note that the `jle.orb` file provided in the example is with extremely low cutoff for efficient job running and therefore is not indended for any practical production-level projects. Users need to generate a more practical projector file based on the recommended cutoffs provided above.

1.4.6 orbital files and pseudopotential files

The DeePKS-related calculations are implemented with **lcao** basis set in ABACUS, therefore the orbital and pseudopotential files for each elements are required. Since the numerical atomic orbitals in ABACUS are generated based on SG15 optimized Norm-Conserving Vanderbilt (ONCV) pseudopotentials, users are required to use this set of pseudopotentials. Atomic orbitals with 100Ry energy cutoff are recommended, and `ewfcut` **is recommended to set to 100 Ry, i.e., consistent with the one applied in atomic orbital generation.**

Both the pseudopotential and the atomic orbital files can be downloaded from ABACUS official website. The required files are recommended to be placed on `iter` folder, as shown in the *file structure* .

1.5 Important outputs explanation

During the training process, a bunch of outputs will be generated. First, `ABACUS` folder will be generated under each training/testing group (`group.xx` under `systems`), which further includes *N=nframes* subfolders, `0`, `1`, ..., `${nframes}`. For example, for `water_single_lda2pbe_abacus`, `ABACUS` in folder `systems/group.00` contains 300 subfolders, while `ABACUS` in folder `systems/group.03` contains 100 subfolders. Each subfolder contains the input and output file of the ABACUS SCF job of corresponding frame at current iteration, and will be overwritten on the next iteration.

For each iteration, error statistics and training outputs are generated in `iter.xx` folder. For example, the file structure of `iter.init` basically looks like:

If `niter` is larger than 0, then `iter.00`, `iter.01`, ..., will be generated at corresponding iteration. These folders share similar file structures as `iter.init` does. Important output files during the training processes are explained as below.

1.5.1 log.data

path: `iter/iter.xx/00.scf/log.data`

This file contains error statistics as well as SCF convergence ratio of each iteration. For example, for `water_single_lda2pbe_abacus`, `log.data` of the init iteration (located at `iter/iter.init/00.scf`) looks like

```
Training:
  Convergence:
    900 / 900 =          1.00000
  Energy:
    ME:          -0.09730528149450003
    MAE:         0.09730528149450003
    MARE:        0.00030881151639484673
Testing:
  Convergence:
    100 / 100 =          1.00000
  Energy:
    ME:          -0.09730505954754445
    MAE:         0.09730505954754445
    MARE:        0.0003349933606729039
```

where ME = mean error, MAE = mean absolute error, MARE = mean relative absolute error. MARE is calculated via removing any constant energy shift between the target and base energy. Note that only energy error is included here since only energy label is trained in the init iteration.

In this example, force label is triggered on after the init iteration by setting `extra_label` to be `true` and `force_factor` to be 1 in *params.yaml*. And `log.data` in `iter.00/00.scf` therefore has the force error statistics:

```
Training:
  Convergence:
    899 / 900 =          0.99889
  Energy:
    ME:          1.707869318132222e-05
    MAE:         3.188871711078968e-05
    MARE:        3.054509587845316e-05
  Force:
    MAE:         0.00030976685248761896
Testing:
  Convergence:
    100 / 100 =          1.00000
  Energy:
    ME:          1.8457155353139854e-05
    MAE:         3.5420404788446546e-05
    MARE:        3.3798956665677724e-05
  Force:
    MAE:         0.00032716565708601 49
```

To judge whether the DeePKS model has converged, users may compare error statistics in `log.data` between current and former iterations, if the errors almost remain the same, the model can be considered as converged.

1.5.2 log.train

path: `iter/iter.xx/01.train/log.train`

This file records the learning curve of the training process at each iteration. It should be noted that for iterations *after* the initial one, *train error (trn err)* recorded in this file corresponds to the **total error** of the training set, i.e., energy error plus the error from extra labels, while *test error (tst err)* corresponds to only the **energy error** of the testing set. For init training, both the train error and the test error correspond to the energy error since no extra label is included.

For a successful training process, users would expect a remarkable decrease in both the train and the test error, especially during the first one or two iterations. As the iterative training goes on, the decrease in errors will gradually become subtle.

1.5.3 RECORD

path: `iter/RECORD`

This file records every step taken in the iterative training process and is **crucial** when resubmitting the job. Each row of this `RECORD` file corresponds to a unique step, and details are given as follows:

- `(X 0 0)`: at iteration number `X` (`X=0` corresponds to `iter.init`; `X=1` corresponds to `iter.00`; `X=2` corresponds to `iter.01`; etc), pre process of SCF, generate ABACUS work directory and input files in each group of systems
- `(X 0 1)`: run SCF calculations in ABACUS
- `(X 0 2)`: concatenate and check the SCF result and print convergence and accuracy in *log.data* in `iter.xx/00.scf`.
- `(X 0)`: current SCF job done; prepare for training
- `(X 1 0)`: train a new model using the old one (if any) as starting point

- `(X 1 1)`: current training done; learning curve is recorded in *log.train* in `iter.xx/01.train`
- `(X 1)`: test the model on all data to see the pure fitting error in `log.test` in iter.xx/01.train
- `(X)`: current iteration done

For example, if we want to restart the training process for iter.00, then the corresponding `RECORD` file should look like

```
0 0 0
0 0 1
0 0 2
0 0
0 1 0
0 1 1
0 1
0
1 0 0
1 0 1
1 0 2
1 0
```

Note: To re-run the whole procedure, make sure that all `iter.xx` folder, `share` folder and `RECORD` file are deleted! In addition, if previous jobs were submiited via DPDsipatcher and resubmission is desired for some reason, maks sure the .json file located at `~/.dpdispatcher/dp_cloud_server/` is removed.

1.5.4 Model file

path: `iter/iter.xx/01.train/model.pth`; this is the model file generated directly by the neural network in DeePKS-kit

path: `iter/iter.{xx+1}/00.scf/model.ptg`; this is the adjusted format of `model.pth` which will be loaed in ABACUS

To manually convert `model.pth` to `model.ptg`, one needs to run the following script:

```
import torch
import torch.nn as nn
from torch.nn import functional as F
from deepks.model import CorrNet
mp = CorrNet.load("model.pth")
mp.compile_save("model.ptg")
```

1.6 Running ABACUS with DeePKS model

Once the DeePKS training process is converged, users may perform ABACUS SCF calculation with the DeePKS model loaded. Compared to a normal ABACUS SCF job with lcao basis, one needs to add the following keywords to `INPUT` file:

```
<...other keywords>
deepks_scf: 1                # run SCF job with DeePKS model
deepks_model: model.ptg      # provide the model file; should be correctly located
```

Note that the path of `model.ptg` should be provided along with the file itself. The above input works only if `model.ptg` and `INPUT` are placed under the same directory.

Users also need to provide the projector file along with the path in `STRU`:

```
<...other keywords>
NUMERICAL_DESCRIPTOR
jle.orb
```

An example of running ABACUS SCF with trained DeePKS model has been provided here.