SIMPLE-NN Documentation

Release 2.0.0

Kyuhyun Lee
Dongsun Yoo
Wonseok Jeong
Jisu Jung
Seungwu Hwang
Sangmin Oh and Seungwu Han

Oct 11, 2023

CONTENTS

SIMPLE-NN (SNU Interatomic Machine-learning PotentiaL packagE – version Neural Network) SIMPLE-NN is a code to construct the neural network interatomic potential (NNP) from ab initio results. SIMPLE-NN also provides an interfacing module to LAMMPS for executing MD.

Note: Please be understood that we are currently rewriting the documentation for better readibility. We will finish it as soon as possible.

CHAPTER ONE

INSTALLATION

1.1 Requirements

- Python 3.6-3.9
- PyTorch 1.5.0-1.10.1 (package for machine learning)
- LAMMPS 23Jun2022 or newer (simulator for molecular dynamics)

Optional:

- mpi4py (library for parallel CPU computation in preprocessing)

1.2 Procedures

1.2.1 1. Pytorch

Install PyTorch: https://pytorch.org/get-started/locally

Choose the PyTorch of stable release for Python. If you have CUDA-capable system, please download PyTorch with CUDA that makes training much faster.

To check if your GPU driver and CUDA are enabled by PyTorch, run the following commands in python to return whether or not the CUDA driver is enabled:

```
import torch.cuda
torch.cuda.is_available()
```

1.2.2 2. SIMPLE-NN

2-1. Download via git clone

You can download a SIMPLE-NN source code through cloning from repository like this:

```
git clone https://github.com/MDIL-SNU/SIMPLE-NN_v2.git SIMPLE-NN
```

2-2. Download as a zip file

Alternatively, you can download a current SIMPLE-NN source code as zip file from link below.

Download SIMPLE-NN: https://github.com/MDIL-SNU/SIMPLE-NN_v2

Note: We recommend using `venv` or `conda` for convenient module managenement.

After downloading the SIMPLE-NN, install SIMPLE-NN with following the command.

```
cd SIMPLE-NN
python setup.py install
```

If you run into permission issues, add a `--user` tag after the last command.

1.2.3 3. LAMMPS

Currently, we support the module for symmetry_function - Neural_network model.

Download LAMMPS: https://github.com/lammps/lammps

Only LAMMPS whose version is `23Jun2022` or newer is supported.

Copy the source code to LAMMPS src directory.

```
cp SIMPLE-NN_v2/simple-nn/features/symmetry_function/pair_nn* /path/to/lammps/src/
cp SIMPLE-NN_v2/simple-nn/features/symmetry_function/symmetry_function.h /path/to/lammps/
↪src/
```

pair_nn* in the first command includes the `pair_nn.cpp`, `pair_nn.h`, `pair_nn_replica.cpp`, and `pair_nn_replica.h`.

Compile LAMMPS code.

```
cd /path/to/lammps/src/
make mpi
```

After this step, you can *test your installation*.

1.2.4 4. mpi4py (optional)

SIMPLE-NN supports the parallel CPU computation in dataset generation and preprocessing for an additional speed gain.

Install mpi4py:

```
pip install mpi4py
```

1.2.5 5. Intel SIMD acceleration (optional)

The filename extension simd refers to Intel-accelerated version of simulating molecular dynamics in SIMPLE-NN. By utilizing vector-matrix multiplication routines in Intel MKL and vectorizing descriptor computation by SIMD, overall speed up would be x3 to x3.5 faster than the regular version.

5.1 Requirements

- Intel CPU supporting AVX
- Compiler supporting AVX instruction set
- IntelMKL `2018.5.274` and `2022.1.0` tested
- Lammps `23Jun2022-Update1(stable)` tested

In our experience, the best performance is achieved when source compiled with intel compiler(icpc) and intel mpi (mpiicpc). LAMMPS provides default makefile for intel compiler, intel mpi and mkl library path setting. Therefore, we recommend to compile lammps source with intel compiler.

The code uses AVX-related functions from intel intrinsic, BLAS routines of MKL, and vector math. So if older versions of MKL and intel compilers support these features, there is no problem for compiling.

5.2 Installation

```
cp {simple_nn_path}/simple_nn/features/symmetry_function/SIMD/{pair_nn_simd.cpp, pair_nn_
↪simd.h, pair_nn_simd_function.h} {lammps_source}/src/
cd {lammps_source}/src
make intel_cpu_intelmpi
```

Note: 'make intel_cpu_intelmpi' is an example of using the intel compiler for lammps. Before using a makefile, you may need to explicitly set some library path and optimization flags (such as -xAVX) in the makefile if necessary.

5.3 Requirements for potential file

- Symmetry function group refers to a group of vector components which have the same target atom specie(s).
- Vector components of the same symmetry function group must have the same cutoff radius.
- Vector components of the same symmetry function group must be contiguous in potential file.
- The zeta value must be an integer in the angular symmetry functions.

Since some assumptions have been made about the potential files for acceleration, the potential file must follow the rules above.

5.4 Usage

In youer LAMMPS script file, regular version uses `pair_style nn`. For the accelerated version, `pair_style nn/intel` should be invoked.

5.5 Further Acceleration

Two additional accelerations are possible if the AVX2 or AVX512 instruction set is available. To enable these features, add "-xCORE-AVX2" or "-xCORE-AVX512" compile flag to your makefile, depending on your CPU. Since AVX512 is released after AVX2, turning on AVX512 automatically turns on AVX2 as well.

Further acceleration by AVX2 is possible by computing unique values of symmetry function parameters to reduce computation. So it puts some requirements on potential file. - The potential file must contain at least one G4 or G5 angular symmetry function. - The number of unique 'eta' value in same angular symmetry function group must be less than 4(AVX2) or 8(AVX512). - The zeta value must be less than 8. This acceleration is about 25~35% faster than the primitive AVX version.

In addition, AVX512 doubles the maximum size of simd calculation, whose speed up is around 10%.

You can check the log file of LAMMPS to see if the installation was successful and if the potential file conditions were met. After LAMMPS reads the potential file, you can see somthing like this :

```
AVX2 for angular descriptor G4 calc : on/off
AVX2 for angular descriptor G5 calc : on/off
AVX512 for descriptor calc : on/off
```

1.3 Test your installation

To check whether SIMPLE-NN and LAMMPS are ready to run or not, we provide the shell script in `test_installation` directory.

Note: If you use the `venv` or `conda` for SIMPLE-NN, activate the virtual environment before check.

Run `run.sh` with the path of lammps binary.

```
./run.sh /path/to/lammps/src/lmp_mpi
```

While `run.sh` tests SIMPLE-NN, LAMMPS with neural network potential, and LAMMPS with replica ensemble, pass or fail messages will be printed like:

```
Test is going on...
SIMPLE-NN test is passed (or failed).
LAMMPS with neural network test is passed (or failed).
LAMMPS with replica ensemble test is passed (or failed).
```

If you have a problem in installation, post a issues in here.

CHAPTER

TWO

INPUTS

In this section, you can find the information of input files that are used in SIMPLE-NN.

2.1 input.yaml

In this section, you can find the features in `input.yaml` that are used in SIMPLE-NN.

2.1.1 Generate_features

- `True` (default) / `False`

---

generate_features determines whether SIMPLE-NN converts the *ab initio* calculation result into `.pt` format used as input dataset or not. Detailed setting for `generate_features` can be found in *Data*

2.1.2 Preprocess

- `True` (default) / `False`

---

preprocess determines whether SIMPLE-NN splits the whole dataset into train/validation dataset and calculates the scaling, PCA matrix, and atomic weights for input features or not. Detailed setting for `preprocess` can be found in *Preprocessing*.

2.1.3 Train_model

- `True` (default) / `False`

---

train_model determines whether SIMPLE-NN optimizes(or evaluate) the neural network based on the `train_list` and `valid_list`. Detailed setting for `train_model` can be found in *Neural network*.

2.1.4 Random_seed

- `null` (default) / non-negative integer

---

random_seed is used to set the seed of random number generator in SIMPLE-NN. SIMPLE-NN has randomness in train/valid separation, data loading, and weight initialization. When `random_seed` is set to `null`, SIMPLE-NN generates the random number based on your system time. Users can reproduce the same training result with the random seed value written at the top of the `LOG` file.

2.1.5 Params

- Type: `dict`

params contains the path of parameter files for each atom. For example, when the system consists of Si and O atoms, params should be written down like this:

```
params:
    Si: params_Si
    O: params_O
```

The detailed description of `params_XX` can be found in *params_XX*. The order of species determines the index in `params_XX`.

2.1.6 Data

In this section, you can find the parameters related to reference data that are used in SIMPLE-NN.

Path and format

type

- `symmetry_function` (default)

---

type chooses the kind of input feature descriptor. Currently, SIMPLE-NN supports only the Belher-Parrinello-type atom-centered symmetry function.[1] Parameters for generating symmetry functions are provided in *params_XX*.

struct_list

- `structure_list` (default)

---

struct_list stands for the path of the file that contains the reference dataset. Detailed format of `structure_list` is found in *structure_list*.

[1] J. Behler, J. Chem. Phys. 134 (2011) 074106

refdata_format

- `vasp-out` (default) / `espresso-out` / etc…

---

refdata_format describes the file format of reference dataset. As SIMPLE-NN reads the reference dataset via Atomic Simulation Environment (ASE) module, only data format listed in here can be used as dataset.

compress_outcar

- `True` (default) / `False`

---

compress_outcar decides whether to compress OUTCAR or not before reading by Atomic Simulation Environment (ASE) module. Compressing OUTCAR enhances the reading speed.

Note: **compress_outcar** only works to output of VASP called as OUTCAR

save_directory

- `data` (default)

---

save_directory defines the path, where `data*.pt` files are located.

save_list

- `total_list` (default)

---

save_list contains the whole `data*.pt` generated. It will be splited into train/validation set.

absolute_path

- `True` (default) / `False`

---

absolute_path determines whether all data paths are written as an absolute or relative path. Users can choose a useful format.

Data extraction

read_force

- `True` (default) / `False`

---

read_force should be `True` if you want to extract force information from *ab initio* calculation result.

read_stress

- `True` (default) / `False`

---

read_stress should be `True` if you want to extract stress information from *ab initio* calculation result.

dx_save_sparse

- `True` (default) / `False`

---

dx_save_sparse determines whether the derivative of input feature matrix, which is used to calculate force from atomic energy in training process, is saved as sparse or dense tensor. Generally, sparse tensor has smaller capacity but provides slower training speed. We recommend testing on your system before setting. It only works when *read_force* is `True`.

2.1.7 Preprocessing

In this section, you can find the information of preprocessing in SIMPLE-NN.

Train/validation

data_list

- `total_list` (default)

---

data_list contains all the paths of reference data. Users can change the name of `total_list` as their favor.

train_list

- `train_list` (default)

---

train_list contains all the paths of training data which is separated as the rate of *valid_rate*.

valid_list

- `valid_list` (default)

---

valid_list contains all the paths of validation data which is separated as the rate of *valid_rate*.

valid_rate

- `0.1` (default) / `0.0` ~ `1.0`

---

valid_rate separates the rate of validation data and training data as a specified value(`0.1`). For example, if the **valid_rate** is set as `0.1` 10 % of total data are classified as validation data, and the remaining 90 % data are classified as training data.

shuffle

- `True` (default) / `False`

---

shuffle determines whether training data and validation data are randomly shuffled(`True`) or in order(`False`) based on the *valid_rate*.

Scaling parameters

calc_scale

- `True` (default) / `False`

---

calc_scale determines whether SIMPLE-NN calculates scaling parameters(`True`) or not(`False`). Feature scaling is a method used to normalize the range of independent variables or features of data. It is required because as the range of raw data varies widely, the range of all features should be normalized in order to match the contribution of each feature proportionately to the final width. SIMPLE-NN supports several scaling method as described in *scale_type*.

scale_type

- `minmax` (default) / `meanstd`, `uniform gas`

SIMPLE-NN supports `minmax`, `meanstd`, `uniform gas` for scaling calculation. The usage of **scal_type** is as below.

```yaml
# input.yaml
preprocessing:
    scale_type: minmax
```

Note: If the **scale_type** tag is set as `uniform gas`, there is an additional tag named *scale_rho*.

---

scale_rho

- atom_type: `atomic density`

The usage of **scale_rho** is as below. Users can give **scale_rho** value as atomic density(# of atoms / volume) for each atom. The unit of **scale_rho** is $^{-3}$

```
#input.yaml
preprocessing:
    scale_type: uniform gas
    scale_rho:
        Si: 0.01
        O : 0.02
```

scale_width

- `1.0` (default)

---

scale_width determines the width of the distribution of scaled data.

PCA parameters

calc_pca

- `True` (default) / `False`

The principal component analysis(PCA) is the process of computing the principal components and using them to modify the basis of data. It is useful to reduce the number of dimensions in the vectors in a dataset. **calc_pca** determines whether SIMPLE-NN calculates pca(`True`) or not(`False`).

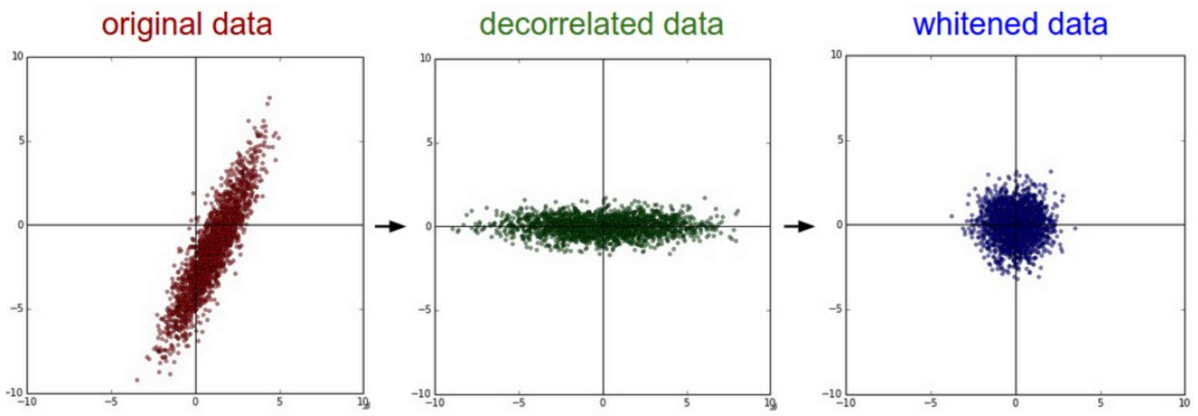
pca_whiten

- `True` (default) / `False`

---

pca_whiten determines whether SIMPLE-NN performs pca whitening(`True`) or not(`False`). The effect of pca whitening is shown as below.[1]

[1] CS231n-Stanford



min_whiten_level

- `1.0e-8` (default)

When *pca_whiten* is set as `True`, the **min_whiten_level** is activated. The minimum width of the distributed data after the PCA process must be bigger than **min_whiten_level** to apply the PCA whitening.

Atomic weights

calc_atomic_weights

- `False` (default) / `gdf`

As mentioned in *Advanced features* section, tuning the weight of atomic force in loss function can be used to reduce the force errors of sparsely sampled atoms. In order to activate atomic weights, the usage of **calc_atomic_weights** is shown as below. SIMPLE-NN supports automatic parameter generation scheme for σ and c. Use the setting `params: Auto` to get a robust σ and c.

```
# input.yaml
preprocessing:
    calc_atomic_weights:
        type: gdf
        params: Auto
```

2.1.8 Neural network

In this section, you can find the information of neural network in SIMPLE-NN.

Running mode

train

- `True` (default) / `False`

---

train determines whether SIMPLE-NN conducts training(`True`) or not(`False`). For the test process and drawing correlation graph, **train** tag must be set as `False`.

train_list

- `train_list` (default)

---

train_list reads the list of training data that is produced from *Preprocessing* step.

valid_list

- `valid_list` (default)

---

valid_list reads the list of validation data that is produced from *Preprocessing* step.

test

- `False` (default) / `True`

If the **test** tag is set as `True`, the predicted energy and forces for the test set are calculated.

test_list

- `test_list` (default)

If the *test* tag is set as `True`, SIMPLENN reads the `test_list` to perform the test step.

add_NNP_ref

- `False` (default) / `True`

In order to apply replica ensemble to Neural Network Potentials, **add_NNP_ref** must be set as `True`. Then SIMPLE-NN reads *ref_list*, producing atomic energies into the data.pt file. The *continue* tag must be set as shown below.

```
#input.yaml
neural_network:
    continue: weights
```

ref_list

- `ref_list` (default)

---

ref_list is required when *add_NNP_ref* is activated. As shown below, users must make **ref_list** with preprocessed training data and validation data.

```
cat train_list valid_list > ref_list
```

train_atomic_E

- `False` (default) / `True`

If the **train_atomic_E** tag is set as `True`, based on the train_list and valid_list which were produced from *add_NNP_ref* step, SIMPLE-NN trains one set of replica ensemble. By varying weight parameters and network size, users can apply replica ensemble to Neural Network Potentials. The *continue* tag must be set as shown below.

```
#input.yaml
neural_network:
    continue: null
```

test_atomic_E

- `False` (default) / `True`

If the **test_atomic_E** tag is set as `True`, the predicted total and atomic energies for the test set are calculated.

use_force

- `True` (default) / `False`

If the **use_force** tag is set as `True`, force is used for training. From our experience, we recommend training with both energy and forces for robust Neural Network Potential, since training with only energy induces overfitting, while training with forces only gives large errors in total energy.

use_stress

- `True` (default) / `False`

If the **use_stress** tag is set as `True`, stress is used for training.

shuffle_dataloader

- `True` (default) / `False`

When SIMPLE-NN divides training data based on the batch size, **shuffle_dataloader** determines whether to select data randomly(`True`) or in order(`False`).

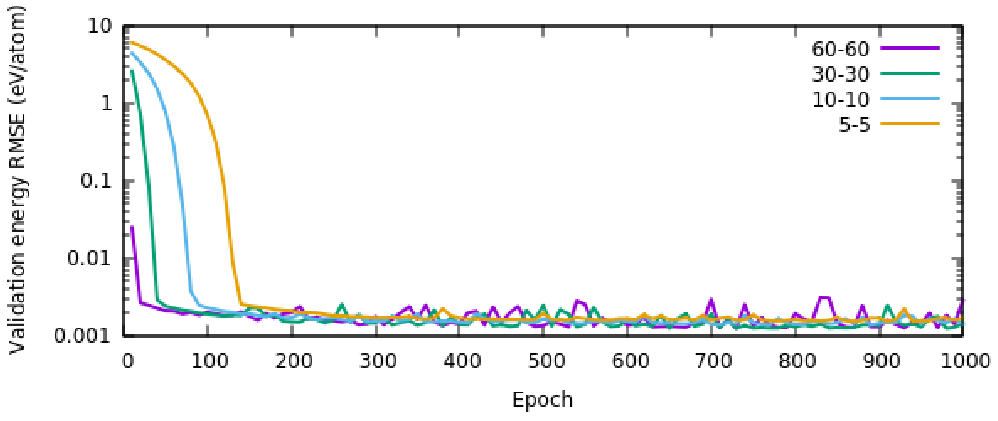
Network

nodes

- `30-30` (default)

---

nodes indicate the network architecture. `30-30` means 2 hidden layers with 30 hidden nodes. As shown below, increasing the number of nodes guarantees low energy RMSE but slow computation, while decreasing the number of nodes is fast but gives high energy RMSE value.



Note: Note that this figure is the result of Si MD, which is too simple to show the effect of nodes.

acti_func

- `sigmoid` (default) / `tanh`, `relu`, `selu`, `swish`

SIMPLE-NN supports several activation functions, such as `sigmoid` function which is the default setting, hyperbolic tangent(`tanh`) function, rectified linear unit(`relu`) function, scaled exponential linear unit(`selu`) function and `swish` function. The usage of **acti_func** is shown as below.

```
# input.yaml
neural_network:
   acti_func: sigmoid
```

double_precision

- `True` (default) / `False`

---

double_precision determines whether the double-precision(`True`) or single-precision(`False`) is used.

weight_initializer

- type: `xavier normal` (default) / `xavier uniform`, `normal`, `constant`, `kaiming normal`, `kaiming uniform`, `he normal`, `he uniform`, `orthogonal`, `sparse`

Weight initialization is used to define the initial values for the parameters in Neural Network models prior to training the models on dataset. SIMPLE-NN supports several **weight_initializer** and the usage of **weight_initializer** is as below.

```
# input.yaml
neural_network:
    weight_initializer:
        type: xavier normal
        params:
```

dropout

- `0` (default) / `0` ~ `1`

The main idea of dropout is to randomly drop units from the neural network during training, resulting in a significant reduction of overfitting. Users must type a value between `0` and `1` to enable **dropout**. For example, if users type `0.25`, 25 % of nodes in neural network hidden layers are dropped.

use_scale

- `True` (default) / `False`

---

use_scale determines whether SIMPLE-NN uses scaling parameters(`True`) that is calculated from *calc_scale* step or not(`False`).

use_pca

- `True` (default) / `False`

---

use_pca determines whether SIMPLE-NN uses pca(`True`) that is calculated from *calc_pca* step or not(`False`).

use_atomic_weights

- `False` (default) / `True`

If **use_atomic_weights** tag is set as `True`, SIMPLE-NN uses the atomic weights that are produced from *calc_atomic_weights* step.

weight_modifier

- type: `null` (default) / `modified sigmoid`

Dictionary for weight modifier. The usage of **weight_modifier** is as below.

```
# input.yaml
neural_network:
    weight_modifier:
        type: modified sigmoid
        params:
```

Optimization

optimizer

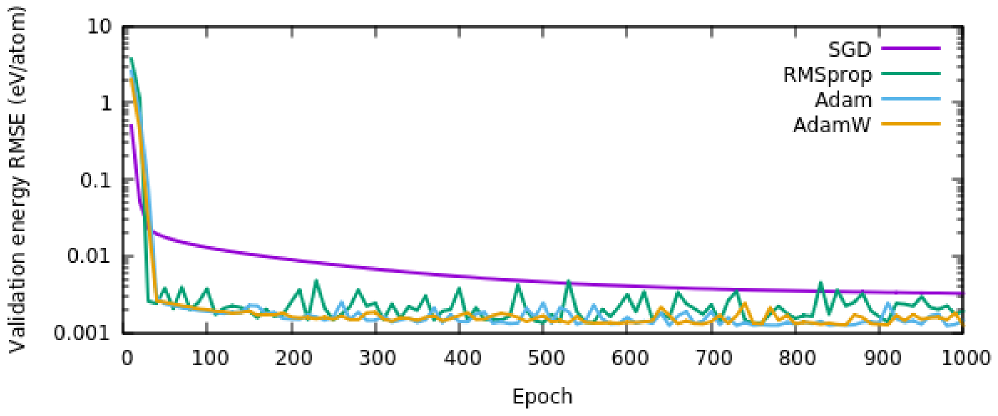
- method: `Adam` (default) / `Adadelta`, `Adagrad`, `AdamW`, `Adamax`, `ASGD`, `SGD`, `RMSprop`, `Rprop`

---

optimizer determines the optimization method. The usage of **optimizer** is as below. SIMPLE-NN supports `Adam`, `Adadelta`, `Adagrad`, `AdamW`, `Adamax`, `ASGD`, `SGD`, `RMSprop` and `Rprop`.

```
# input.yaml
neural_network:
    optimizer:
        method: Adam
        params:
```

As shown below, in general, `Adam` type optimizer shows the best convergence.

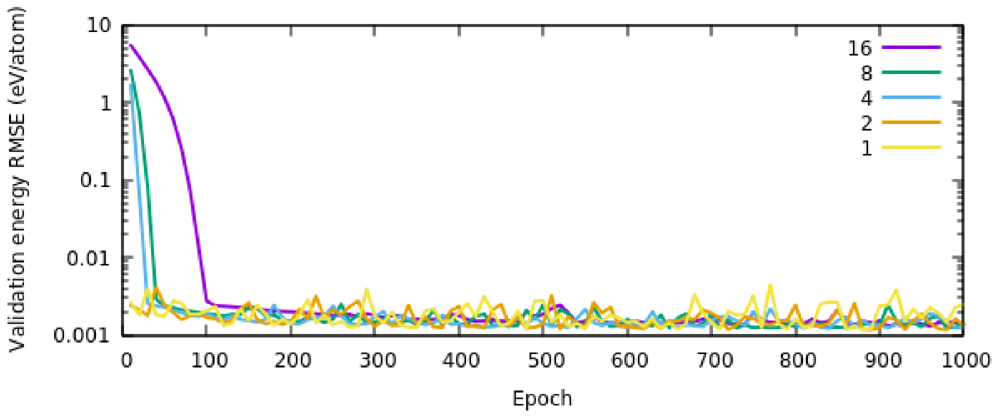


batch_size

- 8 (default)

---

batch_size determines the number of samples in the batch training set. As shown below, increasing batch size shows a slow drop of energy RMSE and gives small fluctuations while decreasing batch size shows a fast drop of energy RMSE and gives large fluctuations.



full_batch

- `False` (default) / `True`

If the **full_batch** tag is set as `True`, full batch mode is enabled.

total_epoch

- `1000` (default)

---

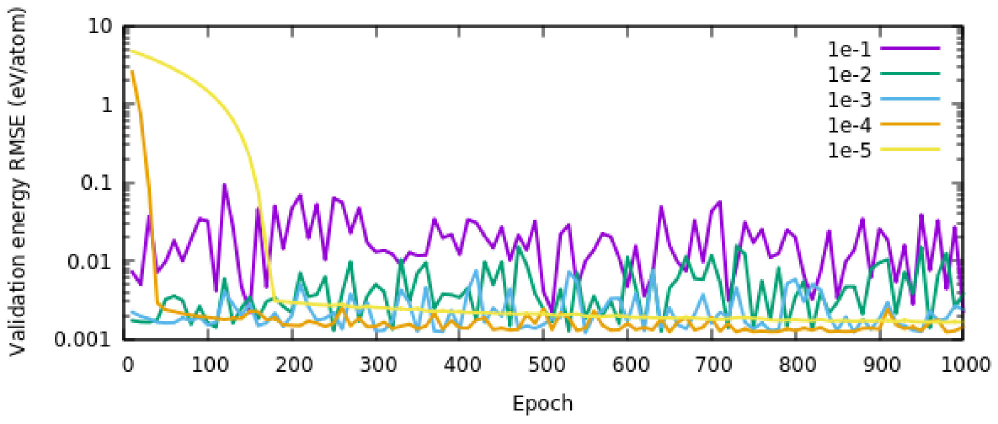
total_epoch indicates the number of total training epoch.

learning_rate

- `0.0001` (default)

---

learning_rate indicates the learning rate for the gradient descendent-based optimization algorithm. As shown below, increasing learning rate gives a fast energy RMSE drop but large fluctuations, while decreasing learning rate gives small fluctuations but slow energy RMSE drop.



decay_rate

- `null` (default)

The *learning_rate* is a parameter that determines how much the update step affects the current value of the weights while the **decay_rate** is an additional term in a weight update rule that exponentially reduces a weight to zero.

l2_regularization

- `1.0e-6` (default)

---

l2_regularization indicates the value of weight decay. Weight decay is a regularization technique by adding a small penalty to the loss function, which can prevent overfitting. SIMPLE-NN supports L2 regularization.

Loss function

loss_scale

- `1.` (default)

---

loss_scale indicates the scaling coefficient for the entire loss function.

E_loss_type

- `1` (default)

E_loss_type determines whether SIMPLE-NN uses normalized(divided by # of atoms) energy loss function(`1`) or not(`2`).

F_loss_type

- `1` (default)

F_loss_type determines whether SIMPLE-NN uses normalized(divided by # of atoms) force loss function(`1`) or not(`2`).

energy_coeff

- `1.` (default)

---

energy_coff indicates the scaling coefficient for energy loss.

force_coeff

- `0.1` (default)

---

force_coff indicates the scaling coefficient for force loss.

stress_coeff

- `1.0e-06` (default)

---

stress_coff indicates the scaling coefficient for stress loss.

Logging & saving

show_interval

- `10` (default)

---

show_interval indicates the interval for printing RMSE in the LOG file.

save_interval

- `0` (default)

---

save_interval indicates the interval for saving the neural network potential file.

energy_criteria

- `null` (default)

---

energy_criteria $(\mathrm{eV/atom})$ determines the stopping criteria for energy RMSE. In our experience, less than 10 $\mathrm{meV/atom}$ gives a reasonable result.

force_criteria

- `null` (default)

---

force_criteria determines the stopping criteria for force RMSE. In our experience, less than 0.3 $\mathrm{eV/}$ gives reasonable result.

stress_criteria

- `null` (default)

---

stress_criteria (kB) determines the stopping criteria for stress RMSE. In our experience, less than 10 kB gives reasonable result.

print_structure_rmse

- `False` (default) / `True`

If the **print_structure_rmse** tag is set as `True`, RMSE's for each structure type is also printed in the LOG file.

Continue

continue

- `null` (default) / `weights`, `checkpoint_bestmodel.pth.tar`

If the **continue** tag is set to `weights`, the training process restarts from the LAMMPS potential file(`potential_saved`). If the tag is set to `checkpoint_bestmodel.pth.tar`, the training process restarts from the checkpoint file. The usage of **continue** is as below.

```
#input.yaml
neural_network:
    continue: weights / checkpoint_bestmodel.pth.tar
```

Note: Users need to copy `pca` and `scale_factor` and potential files if you use LAMMPS potential(change the name of potential file into `potential_saved`).

Users need to copy `checkpoint_bestmodel.pth.tar` into your running directory if you use checkpoint file.

clear_prev_status

- `False` (default) / `True`

---

clear_prev_status determines whether SIMPLE-NN continues from the :doc:*//inputs.input.yaml/neural_network/start_epoch* with the corresponding network inside the `checkpoint_bestmodel.pth.tar` file(`False`)or not. The usage of **clear_prev_status** is shown as below.

```
#input.yaml
neural_network:
    continue: checkpoint_bestmodel.pth.tar
clear_prev_status: True
start_epoch: 5
```

Note: More details of **clear_prev_optimizer** is under construction.

clear_prev_optimizer

- `False` (default) / `True`

---

clear_prev_optimizer determines whether SIMPLE-NN continues with the optimizer inside the `checkpoint_bestmodel.pth.tar` file(`False`)or not.

```yaml
#input.yaml
neural_network:
    continue: checkpoint_bestmodel.pth.tar
clear_prev_optimizer: False
```

Note: More details of **clear_prev_optimizer** is under construction.

---

start_epoch

- `1` (default) / Non-negative integer

---

start_epoch determines at which epoch SIMPLE-NN will start.

Parallelism

use_gpu

- `True` (default) / `False`

---

use_gpu indicates whether SIMPLE-NN operates with GPU (`True`) or CPU (`False`).

GPU_number

- `null` (default) / Non-negative integer

GPU_number determines which GPU SIMPLE-NN will use.

inter_op_threads

- `0` (default)

---

inter_op_threads indicates the number of threads for CPU.

intra_op_threads

- `0` (default)

Under construction

subprocesses

- `0` (default)

---

subprocesses indicates how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process.

Note: More details are explained at PyTorch website.[1]

2.2 params_XX

params_XX contains the coefficients for symmetry functions (SFs). XX is the element name in the target system. 'param_XX' is written in the following style:

```
2 1 0 6.0 0.003214 0.0 0.0
2 1 0 6.0 0.035711 0.0 0.0
4 1 1 6.0 0.000357 1.0 -1.0
4 1 1 6.0 0.028569 1.0 -1.0
4 1 1 6.0 0.089277 1.0 -1.0
```

Each line means:

```
[Type of SF (1)] [Atom-type index (2)] [Cutoff radius (1)] [Coefficients for SF (3)]
```

The number inside $(\cdot)$ is the dimension of parameters.

[Type of SF] Currently, G2, G4, and G5 are supported, selected by 2, 4, and 5, respectively.

[Atom-type index] Type indices of neighbor atoms which starts from 1. The order of type index follows that of the `params` tag written in `input.yaml`) The radial part (G2) requires only one neighbor type so the second parameter is neglected. For the angular parts (G4 and G5), two neighboring types are needed. The order of the two parameters does not matter.

[Cutoff radius] The cutoff radius for cutoff functions in the given SF.

[Coefficients for SF] The parameters defining each symmetry function. For G2, the first two values indicate η and $\mathrm{R_s}$ and the third one is neglected. For G4 and G5, η, ζ, and λ are listed in this order.

[1] TORCH.UTILS.DATA

2.3 structure_list

str_list contains the location of reference calculation data. The format is described below:

```
[ structure_type_1 ]
/location/of/calculation/data/oneshot_output_file :
/location/of/calculation/data/MDtrajectory_output_file 100:2000:20

[ structure_type_2 : 3.0 ]
/location/of/calculation/data/same_folder_format{1..10}/oneshot_output_file :
```

You can use the format of braceexpand to set a path to reference file (like last line). The part which is written after the path indicates the index of snapshots. (format is 'start:end:interval'. ':' means all snapshots.) You can group structures like above for convenience ([structure_group_name] above the pathes of reference file). If `print_structure_rmse` is true, RMSEs for each structure type are also prited in LOG file. In addition, you can set the weights for each structure type ([structure_group_name : weights], default: 1.0).

2.4 run.py

run.py is the simple script for running SIMPLE-NN, passing the `input.yaml`.

```
from simple-nn import run

run('input.yaml')
```

CHAPTER

THREE

QUICK TUTORIAL

3.1 Introduction

This section demonstrate SIMPLE-NN with tutorials. Example files are in `SIMPLE-NN/tutorials/`. In this example, snapshots from 500K MD trajectory of amorphous SiO_2 (72 atoms) are used as training set.

To run SIMPLE-NN, type the following command:

```
python run.py
```

If you have installed `mpi4py`, MPI parallelization provides an additional speed gain in *preprocess* (`generate_features` and `preprocess` in `input.yaml`).

```
mpirun -np $numproc python run.py
```

where `numproc` stands for the number of CPU processors.

Note: In this example, all paths in `*_list` such as `train_list` and `valid_list` are written as relative path. Therefore, you should copy `data` directory to each example or change the paths properly after the first example *Preprocess*.

3.2 Preprocess

To preprocess the *ab initio* calculation result for training dataset of NNP, you need three types of input file (`input.yaml`, `structure_list`, and `params_XX`). The example files except params_Si and params_O are introduced below. Detail of params_Si and params_O can be found in *params_XX* section. In this example, 70 symmetry functions consist of 8 radial symmetry functions per 2-body combination and 18 angular symmetry functions per 3-body combination. Input files introduced in this section can be found in `SIMPLE-NN/tutorials/Preprocess`.

```
# input.yaml
generate_features: True
preprocess: True
train_model: False

params:
    Si: params_Si
    O: params_O

preprocessing:
```

(continues on next page)

(continued from previous page)

```
    valid_rate: 0.1
    calc_scale: True
    calc_pca: True
```

```
# str_list
../ab_initio_output/OUTCAR_comp ::10
```

With this input file, SIMPLE-NN calculates feature vectors and its derivatives (`generate_features`) and generates training/validation dataset (`preprocess`). Sample VASP OUTCAR file (the file is compressed to reduce the file size) is in `SIMPLE-NN/tutorials/ab_initio_output`.

In MD trajectory, snapshots are sampled only in the interval of 10 MD steps (20 fs).

Output files are provided in `SIMPLE-NN/tutorials/Preprocess_answer` except for `data` directory due to the large capacity. `data` directory contains the preprocessed *ab initio* calculation results as binary format named `data1.pt`, `data2.pt`, and so on.

If you want to see which data are saved in `.pt` file, use the following command.

```
import torch
result = torch.load('data1.pt')
```

`result` provides the information of input features as dictionary format.

Warning: We strongly recommend turning on the `calc_pca` and `calc_scale` options in the `preprocess`. They significantly reduce the root-mean-square-error (RMSE) in the `training`.

3.3 Training

To train the NNP with the preprocessed dataset, you need to prepare the `input.yaml`, `train_list`, `valid_list`, `scale_factor`, and `pca`. The last two files highly improves the loss convergence and training quality.

```
# input.yaml
generate_features: False
preprocess: False
train_model: True

params:
    Si: params_Si
    O:  params_O

neural_network:
    nodes: 30-30
    batch_size: 8
    optimizer:
        method: Adam
    total_epoch: 100
    learning_rate: 0.001
    use_scale: True
    use_pca: True
```

With this input file, SIMPLE-NN optimizes the neural network (`train_model`). The paths of training/validation dataset should be written in `train_list` and `valid_list`, respectively. The 70-30-30-1 network is optimized by Adam optimizer with the 0.001 of learning rate and batch size of 8 during 1000 epochs. The input feature vectors whose size is 70 are converted by `scale_factor`, following PCA matrix transformation by `pca` The execution log and energy, force, and stress root-mean-squared-error (RMSE) are stored in `LOG`. Input files introduced in this section can be found in `SIMPLE-NN/tutorials/Training`.

3.4 Evaluation

To evaluate the training quality of neural network, `test_list` and result of training (`checkpoint.pth.tar` or `potential_saved`) should be prepared. `test_list` contains the path of testset preprocessed as `.pt` format. `.pt` format data can be generated as described in *preprocess*. Note that you should set `train_list` to `test_list` with `valid_rate` of 0.0. Then, SIMPLE-NN will write all paths of preprocessed data in `test_list`.

```yaml
# input.yaml
generate_features: True
preprocess: True
train_model: False

params:
    Si: params_Si
    O: params_O

preprocessing:
    train_list: 'test_list'
    valid_rate: 0.0
    calc_scale: False
    calc_pca: False
    calc_atomic_weights: False
```

In this example, `test_list` is made by concatenating `train_list` and `valid_list` in *training* for simplicity. Put the name of result of training such as `checkpoint_*.tar` for PyTorch checkpoint file or `weights` for LAMMPS potential in `continue` in `input.yaml`.

```yaml
# input.yaml
generate_features: False
preprocess: False
train_model: True

params:
    Si: params_Si
    O:  params_O

neural_network:
    train: False
    test: True
    continue: checkpoint_bestmodel.pth.tar
```

Input files introduced in this section can be found in `SIMPLE-NN/tutorials/Evaluation`.

Note: If you use LAMMPS potential (`potential_saved`), you need to copy `pca` and `scale_factor` file and change the name of potential as `potential_saved`.

After running SIMPLE-NN with the setting above, output file named `test_result` is generated. The file is pickle format and you can open this file with python code of below

```
import torch
result = torch.load('test_result')
```

In the file, DFT energies/forces, NNP energies/forces are included. We also provide the python code (`correlation.py`) that makes parity plots from `test_result`.

3.5 Molecular dynamics

Note: You have to compile your LAMMPS with `pair_nn.cpp`, `pair_nn.h`, and `symmetry_function.h` to run molecular dynamics simulation.

To run MD simulation with LAMMPS, add the lines into the LAMMPS script file.

```
# lammps.in

units metal

pair_style nn
pair_coeff * * /path/to/potential_saved_bestmodel Si O
```

Warning: This pair_style requires the `newton` setting to be `on(default)` for pair interactions.

Input script for example of NVT MD simulation at 300 K are provided in `SIMPLE-NN/tutorials/Molecular dynamics`. Run LAMMPS via the following command.

```
/path/to/lammps/src/lmp_mpi < lammps.in
```

You also can run LAMMPS with `mpirun` command if multi-core CPU is supported.

```
mpirun -np $numproc /path/to/lammps/src/lmp_mpi < lammps.in
```

Output files can be found in `SIMPLE-NN/tutorials/Molecular_dynamics_answer`.

CHAPTER

FOUR

ADVANCED FEATURES

4.1 Introduction

This section demonstrate SIMPLE-NN with tutorials. Example files are in `SIMPLE-NN/tutorials/`. In this example, snapshots from 500K MD trajectory of amorphous SiO_2 (72 atoms) are used as training set.

4.2 GDF weighting

Tuning the weight of atomic force in loss function can be used to reduce the force errors of the sprasely sampled atoms. Gaussian densigy function (GDF) weighting[1] is one of the methods, which suggests the gaussian type of weighting scheme. To use GDF, you need to calculate the $\rho(\mathbf{G})$ by adding the following lines to the `symmetry_function` section in `input.yaml`. SIMPLE-NN supports automatic parameter generation scheme for σ and c. Use the setting `sigma: Auto` to get a robust σ and c (values are stored in LOG file). Input files introduced in this section can be found in `SIMPLE-NN/tutorials/GDF_weighting`.

```
# input.yaml:

preprocessing:
    valid_rate: 0.1
    calc_scale: True
    calc_pca: True
    calc_atomic_weights:
        type: gdf
        params: Auto
```

$\rho(\mathbf{G})$ indicates the density of each training point. After calculating $\rho(\mathbf{G})$, histograms of $\rho(\mathbf{G})^{-1}$ are also saved as in the file of `GDFinv_hist_XX.pdf`.

Note: If there is a peak in high $\rho(\mathbf{G})^{-1}$ region in the histogram, increasing the Gaussian weight(σ) is recommended until the peak is removed. On the contrary, if multiple peaks are shown in low $\rho(\mathbf{G})^{-1}$ region in the histogram, reduce σ is recommended until the peaks are combined.

In the default setting, the group of $\rho(\mathbf{G})^{-1}$ is scaled to have average value of 1. The interval-averaged force error with respect to the $\rho(\mathbf{G})^{-1}$ can be visualized with the following script.

```
from simple_nn.utils import graph as grp
grp.plot_error_vs_gdfinv(['Si','O'], 'test_result')
```

[1] W. Jeong, K. Lee, D. Yoo, D. Lee and S. Han, J. Phys. Chem. C 122 (2018) 22790

The graph of interval-averaged force errors with respect to the $\rho(\mathbf{G})^{-1}$ is generated as `ferror_vs_GDFinv_XX.pdf`

If default GDF is not sufficient to reduce the force error of sparsely sampled training points, One can use scale function to increase the effect of GDF. In scale function, b controls the decaying rate for low $\rho(\mathbf{G})^{-1}$ and c separates highly concentrated and sparsely sampled training points. To use the scale function, add following lines to the `neural_network` section in `input.yaml`.

```
# input.yaml:

neural_network:
    weight_modifier:
        type: modified sigmoid
        params:
            Si:
                b: 1
                c: 35.
            O:
                b: 1
                c: 74.
```

For our experience, $b = 1.0$ and automatically selected c shows reasonable results. To check the effect of scale function, use the following script for visualizing the force error distribution according to $\rho(\mathbf{G})^{-1}$.

In the script below, `test_result_woscale` is the test result file from the training without scale function and `test_result_wscale` is the test result file from the training with scale function. These `test_result` are made as described in *evaluation*. We do not provide `test_result_wscale`.

```
from simple_nn.utils import graph as grp
grp.plot_error_vs_gdfinv(['Si','O'], 'test_result_woscale', 'test_result_wscale')
```

4.3 Uncertainty estimation

The local configuration shown in the simulation driven by NNP should be included the training set because NNP only guarantees the reliability within the trained domain. Therefore, we suggest to check whether the local environment is trained or not through the standard deviation of atomic energies from replica ensemble[2]. To estimate the uncertainty of atomic configuration, following three steps are needed.

4.3.1 1. Atomic energy extraction

To estimatet the uncertainty of atomic configuration, the atomic energies extracted from reference NNP should be added into reference dataset (`.pt`).

```
# input.yaml

generate_features: False
preprocess: False
train_model: True

params:
    Si: params_Si
```

(continues on next page)

[2] W. Jeong, D. Yoo, K. Lee, J. Jung and S. Han, J. Phys. Chem. Lett. 2020, 11, 6090-6096

(continued from previous page)

```
    O:  params_O

neural_network:
    train: False
    test: False
    add_NNP_ref: True
    ref_list: 'ref_list'
    train_atomic_E: False
    use_scale: True
    use_pca: True
    continue: checkpoint_bestmodel.pth.tar
```

`ref_list` contains the dataset list to be evaluated to atomic energy. Reference NNP is written in `continue`. After that, the reference dataset (`.pt`) are overwritten with atomic energies.

4.3.2 2. Training with atomic energy

Next, train the replica NNP only with atomic energy. To prevent the convergence among replicas, diversity the network structure by increasing the standard deviation of initial weight distribution (`gain` (default: 1.0)) and change the number of hidden nodes larger than reference NNP.

```
# input.yaml

generate_features: False
preprocess: False
train_model: True
random_seed: 123

params:
    Si: params_Si
    O:  params_O

neural_network:
    train: False
    test: False
    add_NNP_ref: False
    train_atomic_E: True
    nodes: 30-30
    weight_initializer:
        params:
            gain: 2.0
    optimizer:
        method: Adam
    total_epoch: 100
    learning_rate: 0.001
    scale: True
    pca: True
    continue: null
```

Because the atomic energies are needed in training, `data` directory made from *atomic_energy_extraction* is needed.

4.3.3 3. Uncertainty estimation in molecular dynamics

Note: You have to compile your LAMMPS with `pair_nn_replica.cpp`, `pair_nn_replica.h`, and `symmetry_function.h` to evaluate the uncertainty in molecular dynamics simulation.

LAMMPS can calculate the atomic uncertainty through standard deviation of atomic energies. Because atomic uncertainty will be written as atomic charge, prepare LAMMPS data file as charge format and modify your LAMMPS input as below example.

```
# lammps.in

units       metal
atom_style  charge

pair_style  nn/r 3
pair_coeff  * * potential_saved Si O &
            potential_saved_30 &
            potential_saved_60 &
            potential_saved_90

compute     std all property/atom q

dump        mydump all custom 1 dump.lammps id type x y z c_std
dump_modify sort id

run 1
```

We provide the LAMMPS potentials whose network size are 60-60 and 90-90, respectively. Atomic uncertainties are written in a dump file for each atoms. Outputs files are found in `SIMPLE-NN/tutorials/Uncertainty_estimation_answer/3.Uncertainty_estimation_in_molecular_dynamics`.

CHAPTER

FIVE

RELEASE NOTE

5.1 v2.1.0 (29 Sep 2022)

Breaking changes:

- **Accelerated version of simulating molecular dynamics using SIMD and MKL.**
    - Main code developer (Yutack Park).
    - Total 3x ~ 3.5x speed-up : `pair_nn_simd.cpp`, `pair_nn_simd.h` and `pair_nn_simd_function.h`

5.2 v2.0.0 (3 Dec 2021)

Breaking changes:

- **Refactoring SIMPLE-NN from Tensorflow to PyTorch!**
    - Main code developer (Seungwoo Hwang).
    - Main code developer (Sangmin Oh).
    - Project advisor and code developer (Jisu Jung).
    - Project organizer and original code developer (Kyuhyun Lee).

5.3 v1.1.1 (23 Sep 2021)

General changes:

- Independent tags of `generate` and `preprocess` in `input.yaml` for consistency (Jisu Jung).
- Extended buffer in LAMMPS potential (`pair_nn.cpp`) for multinary (> 4) system (Jisu Jung).

Bug fixes:

- Fixed the inconsistency between the direct and cartesian positions from `ASE` (Jisu Jung).
- Fixed the memory leak in LAMMPS potential (`pair_nn.cpp`) (Jisu Jung).

5.4 v1.1.0 (13 Oct 2020)

Development:

- Replica ensemble for quantifying the uncertainty (Wonseok Jeong and Jisu Jung).

Bug fixes:

- Fixed type mismatch in LAMMPS potential (`pair_nn.cpp`) (Jisu Jung).

5.5 v1.0.0 (21 Feb 2020)

Development:

- Stress training (Jisu Jung).

General changes:

- Optimized LAMMPS potential (`pair_nn.cpp`) (Dongsun Yoo).
- Changed the unit in LOG from epoch to iteration (Dongsun Yoo).
- PCA whitening (Dongsun Yoo).

5.6 v0.8.0 (6 Apr 2019)

Development:

- Gaussian density function (GDF) calculation (Kyuhyun Lee, Dongsun Yoo, Wonseok Jeong).

General changes:

- Added information to LOG (Kyuhyun Lee, Dongsun Yoo).

Bug fixes:

- Fixed MPI issues (Kyuhyun Lee, Dongsun Yoo).

5.7 v0.6.0 (20 Nov 2018)

General changes:

- Added brace expansion in `str_list` (Kyuhyun Lee).
- Added early stopping feature (Kyuhyun Lee).

5.8 v0.5.0 (11 Oct 2018)

General changes:

- `stddev` for weight initialization (Dongsun Yoo).

5.9 v0.4.6 (18 Sep 2018)

General changes:

- Warning on undefined tag in `input.yaml` (Dongsun Yoo).

Bug fixes:

- Fixed regularization. (Dongsun Yoo).

5.10 v0.4.5 (4 Sep 2018)

General changes:

- User-defined optimizer (Kyuhyun Lee).

5.11 v0.4.3 (24 Aug 2018)

General changes:

- Changed saving mechanism (Kyuhyun Lee).

CHAPTER

SIX

FAQ

- How to mitigate the overfittng?
  - Try dropout of `True` , larger l2_regularization, less the number of node.
- How to restart from previous training?
  - Write the file name of `checkpoint` of `potential_saved` in `neural_network` of `input.yaml`. Do not forget to copy `scale_factor` and `pca` when using `potential_saved`.