Field-induced Recursively Embedded Atom Neural Network Package

-B. Jiang Group (2023)

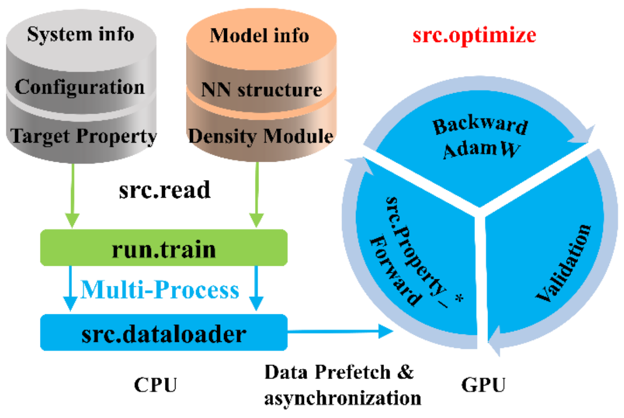
Introduction

Field-induced Recursively embedded atom neural network (FIREANN) is a PyTorch-based end-to-end multi-functional Deep Neural Network Package for Molecular, Reactive and Periodic Systems under the presence of the external field with rigorous rotational equivariance. As a results, FIREANN framework intrinsically describes the response of the potential energy to an external field up to an arbitrary order (dipole moments, polarizabilities …) by taking the analytical gradients of the potential energy with respect to the field vector. FIREANN is developed based on the REANN package and inherits all features of the REANN package including Distributed DataParallel parallelized training on both GPU and CPU and a efficient interface with LAMMPS for GPU and CPU.

Training Workflow

The training process can be divided into four parts: information loading, initialization, dataloader and optimization. First, the "src.read" will load the information about the systems and NN structures from the dataset and input files ("input_nn" and "input_density") respectivrly. Second, the "run.train" module utilizes the loaded information to initialize various classes, including property calculator, dataloader, and optimizer. For each process, an additional thread will be activated in the "src.dataloader" module to prefetch data from CPU to GPU in an asynchronous manner.

Meanwhile, the optimization will be activated in the "src.optimize" module once the first set of data is transferred to the GPU. During optimization, a learning rate scheduler, namely "ReduceLROnPlateau" provided by PyTorch, is used to decay the learning rate. Training is stopped when the learning rate drops below “end_lr” and the model that performs best on the validation set is saved for further investigation.



How to Use FIREANN Package

Users can employ geometries, external fields energies, atomic force vectors (or some other physical properties which are invariant under rigid translation, rotation, and permutation of identical atoms and their corresponding gradients) and arbitrary response of potential energy wrt external fields (dipole moments, polarizabilities, etc.) to construct a model. There are three routines to use this package:

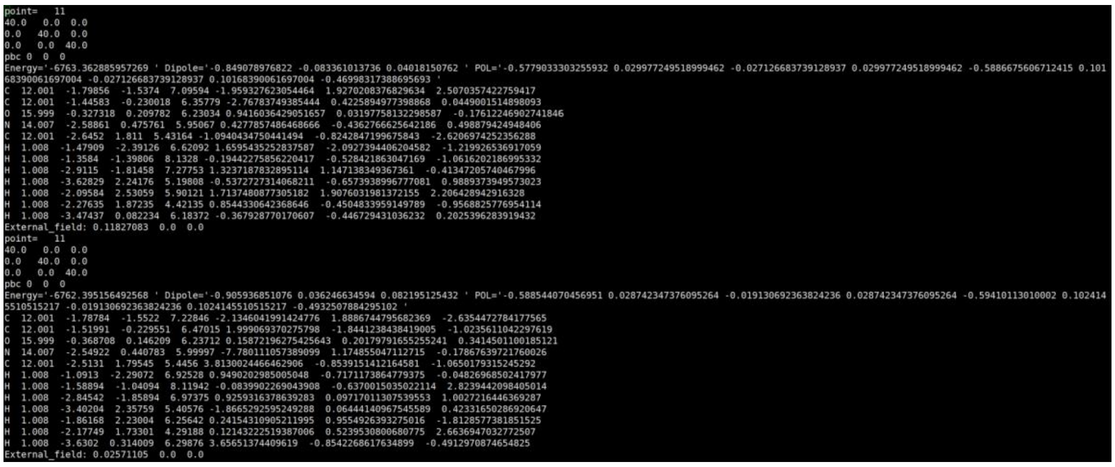
1. Prepare the environment
2. Prepare data
3. Set up parameters

1. Prepare the environment

The FIREANN Package is built based on PyTorch and uses the "opt_einsum" package for optimizing einsum-like expressions frequently used in the calculation of the embedded density. In order to run the REANN package, users need to install PyTorch (version: 2.0.0) based on the instructions on the PyTorch official website (https://pytorch.org/get-started/locally/) and the package named opt_einsum (https://optimized-einsum.readthedocs.io/en/stable/).

2. Prepare data

There are two directories that users need to prepare, namely, “train” and “val”, each of which includes a file “configuration” used to preserve the required information including lattice parameters, periodic boundary conditions, configurations, external field, energy and atomic forces (if needed), dipole moments, polarizabilities, etc. For example, users want to represent the NMA system that has available atomic forces. The file "configuration" should be written in the following format.



```
point=   11
40.0   0.0  0.0
0.0   40.0  0.0
0.0   0.0  40.0
pbc 0  0  0
Energy='-6763.362885957269 ' Dipole='-0.849078976822 -0.083361013736 0.04018150762 ' POL='-0.5779033303255932 0.029977249518999462 -0.027126683739128937 0.029977249518999462 -0.5886675606712415 0.101
68390061697004 -0.027126683739128937 0.10168390061697004 -0.46998317388695693 '
C  12.001  -1.79856  -1.5374  7.09594 -1.959327623054464  1.9270208376829634  2.5070357422759417
C  12.001  -1.44583  -0.230018  6.35779 -2.76783749385444  0.4225894977398868  0.0449001514898093
O  15.999  -0.327318  0.209782  6.23034 0.9416036429051657  0.03197758132298587  -0.17612246902741846
N  14.007  -2.58861  0.475761  5.95067 0.4277857486468666  -0.4362766625642186  0.498879424948406
C  12.001  -2.6452  1.811  5.43164 -1.0940434750441494  -0.8242847199675843  -2.6206974252356288
H  1.008  -1.47909  -2.39126  6.62092 1.6595435252837587  -2.0927394406204582  -1.219926536917059
H  1.008  -1.3584  -1.39806  8.1328 -0.19442275856220417  -0.528421863047169  -1.0616202186995332
H  1.008  -2.9115  -1.81458  7.27753 1.3237187832895114  1.147138349367361  -0.41347205740467996
H  1.008  -3.62829  2.24176  5.19808 -0.5372727314068211  -0.6573938996777081  0.9889373949573023
H  1.008  -2.09584  2.53059  5.90121 1.7137480877305182  1.9076031981372155  2.206428942916328
H  1.008  -2.27635  1.87235  4.42135 0.8544330642368646  -0.4504833959149789  -0.9568825776954114
H  1.008  -3.47437  0.082234  6.18372 -0.367928770170607  -0.446729431036232  0.2025396283919432
External_field: 0.11827083  0.0  0.0
point=   11
40.0   0.0  0.0
0.0   40.0  0.0
0.0   0.0  40.0
pbc 0  0  0
Energy='-6762.395156492568 ' Dipole='-0.905936851076 0.036246634594 0.082195125432 ' POL='-0.588544070456951 0.028742347376095264 -0.019130692363824236 0.028742347376095264 -0.59410113010002 0.102414
5510515217 -0.019130692363824236 0.1024145510515217 -0.4932507884295102 '
C  12.001  -1.78784  -1.5522  7.22846 -2.1346041991424776  1.8886744795682369  -2.6354472784177565
C  12.001  -1.51991  -0.229551  6.47015 1.9990693702757980  -1.8441238438419005  -1.0235611042297619
O  15.999  -0.368708  0.146209  6.23712 0.15872196275425643  0.20179791655255241  0.3414501100185121
N  14.007  -2.54922  0.440783  5.99997 -7.780111057389099  1.174855047112715  -0.17867639721760026
C  12.001  -2.5131  1.79545  5.4456 3.8130024466462906  -0.8539151412164581  -1.0650179315245292
H  1.008  -1.0913  -2.29072  6.92528 0.9490202985005048  -0.7171173864779375  -0.04826968502417977
H  1.008  -1.58894  -1.04094  8.11942 -0.0839902269043908  -0.6370015035022114  2.8239442098405014
H  1.008  -2.84542  -1.85894  6.97375 0.9259316378639283  0.09717011307539553  1.0027216446369287
H  1.008  -3.40204  2.35759  5.40576 -1.8665292595249288  0.06444140967545589  0.42331650286920647
H  1.008  -1.86168  2.23004  6.25642 0.24154310905211995  0.9554926393275016  -1.8128577381851525
H  1.008  -2.17749  1.73301  4.29188 0.12143222519387006  0.5239530800680775  2.6636947032772507
H  1.008  -3.6302  0.314009  6.29876 3.65651374409619  -0.8542268617634899  -0.4912970874654825
External_field: 0.02571105  0.0  0.0
```

The first line can be an arbitrary character other than a blank line. The next three lines are the lattice vectors defining the unit cell of the system. The fifth line is used to enable(1)/disable(0) the periodic boundary conditions in each direction. In this example, NMA is not a periodic system, the fifth line should be “pbc 0 0 0”. For some gas-surface systems, only the x-y plane is periodic and the corresponding fifth line is “pbc 1 1 0”. The sixth line is the targeted properties used to the training the model, recognizable name including “Energy”, “Dipole” and “POL”. Following N lines (N is the number of atoms in the system, here is 12): the columns from the left to right represent the atomic name, relative atomic mass, coordinates(x, y, z) of the geometry, atomic force vectors (if the force vector is not incorporated in the training, these three columns can be omitted). Next line: Start with " External_field:" and then follow by the external field.

3. Set up parameters

In this section we will introduce hyperparameters about the embedded density, training algorithms and architectures of neural networks that are essential for obtaining an exact representation. Next, we will give an example to explain these hyperparameters in detail. There are two input files named "input_nn" and "input_density" saved in the "para" directory in the work directory.

Parameters in “input_nn” and “input_density” file

Although there are dozens of parameters in "input_density" and "input_nn", users can obtain a pretty good accuracy with the default set (the value shown in this manual) except for some parameters marked in red and “required” about the systems for example the "atomtype" that tells the elements involved in the system. The order of parameters is arbitrary and users can give parameters in any order they prefer. If users do not set the parameters "input_density" or "input_nn", code will automatically accept default values.

File “input_nn”

1. Prop_list_init={"Energy":0.1,"Dipole":50,"Force":10,"POL":10} # type: dict
2. Prop_list_final={"Energy":0.1,"Dipole":0.5,"Force":0.5,"POL":0.5} # type: dict

   (The targeted properties used to training the model and corresponding initial/final weights)

3. table_coor = 0 # required parameters type: integer 0/1;

(The form of the coordinates in “1” file, “0”: Cartesian coordinates. “1”: Fraction coordinates.)

4. table_init = 0 # type: integer 0/1;

(Initialize parameters for a model, "0": general method; "1": restart from a previous training)

5. nblock = 1 # type: integer

(Number of residual NN blocks)

6. nl=[64,64] # type: list [n_1, n_2, …, n_p] n_p: integer

(A list holds the number of neurons in each hidden layer of residual blocks. Each residual NN block will have the $n_1 \times n_2 \times \ldots \times n_p \times n_1$. The last n_1 is inserted for identity mapping. The number of neuron should be a power of 2 for a better efficiency.)

7. dropout_p=[0.0, 0.0] # type: list[p1,p2,…] p_i: real number between 0~1

(A list holds the Bernoulli distribution probability of disabling the neuron in each hidden layer used in the dropout for regularization. The length of the list must be larger than or equal to the number of hidden layers.)

8. table_norm=True # type: bool True/False

(A switch enables (True)/disables (False) the layer normalization in hidden layers, which can help with regularization and acceleration of convergence.)

9. re_coeff=0.0 # type: real number between 0~1

(L2 regularization coefficient. Default is 0; usually, a very small value or 0 are preferred)

Note: The dropout is a frequently-used technology in machine learning (ML) and has made a huge success. However, in many traditional ML models, only the outputs serve as the targets, which differ in the construction of potential energy surfaces. The minus gradients of the energy (atomic forces) are always used as an important term to improve the accuracy and reduce the required number of configurations. Meanwhile, the introduction of atomic forces can act as regularization and reduce overfitting. So if users have a sufficient number of datasets including atomic forces and an appropriate size of NN, not that many or strong regularization needs to be included in models (including the L2 regularization coefficients).

10. Epoch=10000   # type: integer

(Maximum epochs in one fitting.)

11. patience_epoch=50   # type: integer

12. decay_factor=0.5   # type: real number between 0~1

(The learning rate will be decayed by a factor of “decay_factor” if the loss is not improved after “patience_epoch” epochs. new_lr = lr * decay_factor.)

13. print_epoch=1   # type: integer

(Calculate and print the training and validation rmse every “print_epoch” epochs.)

14. ratio = 0.9   # type: real between 0~1

(If an empty "validation" folder is provided, then the train will be divided into two parts (ratio:1-ratio) used for training and validation respectively.)

15. start_lr=0.01   #   type: real number

(Initial learning rate. A comparatively high start_lr and small patience_epoch are preferred for better efficiency and accuracy.)

16. end_lr=0.00001   # type: real number

(Final learning rate)

(The learning rate will decrease from the “start_lr” to the “end_lr” by a factor of “decay_factor” according to the learning rate scheduler “ReduceLROnPlateau” provided by PyTorch.)

17. batchsize_train=64 # required parameters type: integer

18. **batchsize_val=128** # required parameters type: integer

(Number of configurations used in each batch for train (batchsize_train) and validation (batchsize_val). Note, this "batchsize_train" is a key parameter concerned with efficiency. Normally, a large enough value is given to achieve high usage of the GPU and lead to higher efficiency in training if you have sufficient data. However, for small training data, a large "batchsize" can lead to a decrease in accuracy, probably owing to the decrease in the number of gradient descents during each epoch. The decrease in accuracy may be compensated by more epochs (increase the "patience_epoch") or a larger learning rate. Some detailed testing is required here to achieve a balance of accuracy and efficiency in training. The value of "batch_val" has no effect on accuracy, and thus a larger value is preferred.)

19. activate = 'Relu_like' # type: string options Relu_like/Tanh_like

(activation functions)

20. queue_size=10 # type: integer

(The number of batch data prefetched to the GPU to ensure good accelerated performance.)

21. DDP_backend=”nccl” # type: string options nccl/gloo

(The built-in backends included in PyTorch distributed. The nccl is preferred for distributed GPU training.)

22. find_unused=False # type: bool True/False

(False is the default value. If there is an error reporting unused parameters, change find_unused to True.)

23. dtype='float32 # type: string “float32” / “float64”

24. folder = ”./”

(The path save the directories “train” and “val”)

The following parameters have a prefix "oc_" and play the same role as defined in the former that determines the NN structure. The difference is that the former represents the mapping from density to atomic energy, and the latter represents the mapping from density to orbital coefficients.

25. oc_loop = 1 # type: integer

(Number of iterations used to represent the orbital efficients.)

26. oc_nl = [128,128] # type: list [n_1, n_2, …, n_p] n_p: integer

(Same as nl.)

27. oc_nblock = 1 # type: integer

(Same as nblock)

28. oc_dropout_p=[0.0, 0.0] # type: list[p1,p2,…] p_i: real number between 0~1

(Same as dropout_p)

29. oc_activate = 'Relu_like' # type: string options “Relu_like” / “Tanh_like”

(Same as activate)

30. oc_table_norm=False # # type: bool True/False

(Same as table_norm)

File “input_density”

1. cutoff = 4.5 # type: real number

(Cutoff distances)

2. neigh_atoms = 150 # type: integer

(Maximum number of neighbor atoms within the cutoff)

3. atomtype= ['O', 'H',] # required parameters type: list [elmenet1, element2,…]

(The involved elements in the systems)

4. nipsin= 2 # type: integer

(Maximal angular momenta determine the orbital type (s, p, d ..))

5. nwave=8 # type: integer

(Number of radial Gaussian functions. This number should be a power of 2 for better efficiency.)

With all needed files prepared, users can execute the training in a single node with multiple GPUs with the directive:

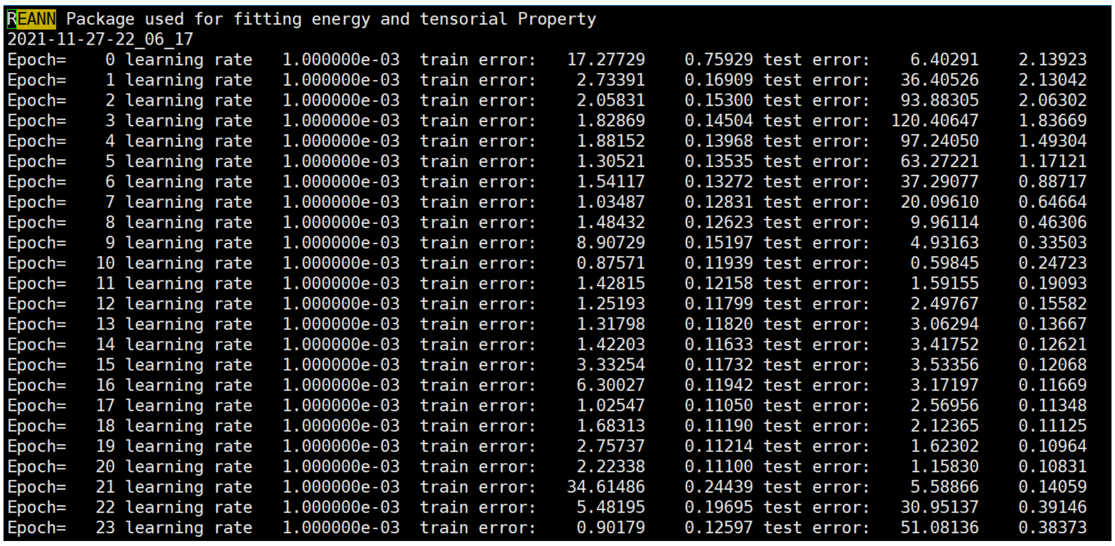
"python3 -m torch.distributed.run --nproc_per_node=$NPROC_PER_NODE --nnodes=1 --standalone $path ", where "path" is the path that code locate in and the NPROC_PER_NODE is the number of GPUs in each node.

The multi-nodes multi-GPUs can be launched with a similar directive which need to specify the node and the port of the master node. Details can be found in https://pytorch.org/docs/stable/elastic/run.html?highlight=distributed%20run#module-torch.distributed.run.

In the training process, some iterative information will output to the file “nn.err”.

- nn.err

First, some information about the model will be printed out.

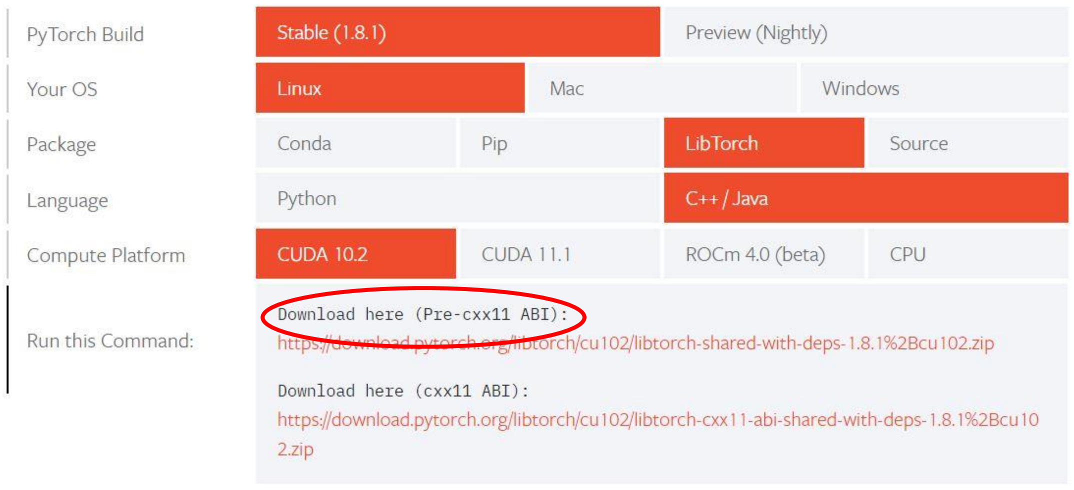


```
REANN Package used for fitting energy and tensorial Property
2021-11-27-22_06_17
Epoch=    0 learning rate   1.000000e-03  train error:   17.27729    0.75929 test error:    6.40291    2.13923
Epoch=    1 learning rate   1.000000e-03  train error:    2.73391    0.16909 test error:   36.40526    2.13042
Epoch=    2 learning rate   1.000000e-03  train error:    2.05831    0.15300 test error:   93.88305    2.06302
Epoch=    3 learning rate   1.000000e-03  train error:    1.82869    0.14504 test error:  120.40647    1.83669
Epoch=    4 learning rate   1.000000e-03  train error:    1.88152    0.13968 test error:   97.24050    1.49304
Epoch=    5 learning rate   1.000000e-03  train error:    1.30521    0.13535 test error:   63.27221    1.17121
Epoch=    6 learning rate   1.000000e-03  train error:    1.54117    0.13272 test error:   37.29077    0.88717
Epoch=    7 learning rate   1.000000e-03  train error:    1.03487    0.12831 test error:   20.09610    0.64664
Epoch=    8 learning rate   1.000000e-03  train error:    1.48432    0.12623 test error:    9.96114    0.46306
Epoch=    9 learning rate   1.000000e-03  train error:    8.90729    0.15197 test error:    4.93163    0.33503
Epoch=   10 learning rate   1.000000e-03  train error:    0.87571    0.11939 test error:    0.59845    0.24723
Epoch=   11 learning rate   1.000000e-03  train error:    1.42815    0.12158 test error:    1.59155    0.19093
Epoch=   12 learning rate   1.000000e-03  train error:    1.25193    0.11799 test error:    2.49767    0.15582
Epoch=   13 learning rate   1.000000e-03  train error:    1.31798    0.11820 test error:    3.06294    0.13667
Epoch=   14 learning rate   1.000000e-03  train error:    1.42203    0.11633 test error:    3.41752    0.12621
Epoch=   15 learning rate   1.000000e-03  train error:    3.33254    0.11732 test error:    3.53356    0.12068
Epoch=   16 learning rate   1.000000e-03  train error:    6.30027    0.11942 test error:    3.17197    0.11669
Epoch=   17 learning rate   1.000000e-03  train error:    1.02547    0.11050 test error:    2.56956    0.11348
Epoch=   18 learning rate   1.000000e-03  train error:    1.68313    0.11190 test error:    2.12365    0.11125
Epoch=   19 learning rate   1.000000e-03  train error:    2.75737    0.11214 test error:    1.62302    0.10964
Epoch=   20 learning rate   1.000000e-03  train error:    2.22338    0.11100 test error:    1.15830    0.10831
Epoch=   21 learning rate   1.000000e-03  train error:   34.61486    0.24439 test error:    5.58866    0.14059
Epoch=   22 learning rate   1.000000e-03  train error:    5.48195    0.19695 test error:   30.95137    0.39146
Epoch=   23 learning rate   1.000000e-03  train error:    0.90179    0.12597 test error:   51.08136    0.38373
```

After that, the learning rate, RMSE of energy and atomic force in each direction of each epoch will be printed out. Note: The RMSE of training is estimated from the average of each batch rather than the realistic RMSE. The parameters of the model with the smallest RMSE in energy will be saved to “REANN.pth”. Then they will be converted to serialized models (“PES.pt”) via the torchscript, which can be loaded in python/C++ environments used for high-performance inference.

MD Simulations with Lammps

The program will save “LAMMPS.pt”, which are two serializable models (float/double) designed for the interface as a new “pair_style” named “reann” in the LAMMPS (version: 10 Feb 2021). To run Lammps based on the “reann” pair_style, users need to build the Lammps from source with CMake. First, users need to download the Lammps and copy the file in each floder located in (“lammps-interface/” (for eann when oc_loop=0) or “lammps-REANN-interface/” (for reann when oc_loop >0)) to the corresponding directory in Lammps (create folder if it does not exist). Then users need to download the Torch C++ frontend (Libtorch) from the PyTorch website consistent with the environments in the targeted device (CPU or GPU with different versions of CUDA and cudnn). In addition, the downloaded Libtorch needs to uncompress to “libtorch” folder, which is located in the same directory with the src of Lammps. Some attention should be taken here that the pre-cxx11 ABI Libtorch (circled in the following picture) should be adopted and the version of CUDA should be exactly consistent with the device environments supported in Libtorch, which means only the “CUDA 10.2” and “CUDA 11.1” are available.



PyTorch Build	[✓ Stable (1.8.1)]	Preview (Nightly)		
Your OS	[✓ Linux]	Mac	Windows	
Package	Conda	Pip	[✓ LibTorch]	Source
Language	Python	[✓ C++ / Java]		
Compute Platform	[✓ CUDA 10.2]	CUDA 11.1	ROCm 4.0 (beta)	CPU

Run this Command:

```
Download here (Pre-cxx11 ABI):
https://download.pytorch.org/libtorch/cu102/libtorch-shared-with-deps-1.8.1%2Bcu102.zip

Download here (cxx11 ABI):
https://download.pytorch.org/libtorch/cu102/libtorch-cxx11-abi-shared-with-deps-1.8.1%2Bcu10
2.zip
```

Finally, users can generate the Makefile using the directive “sh bulid.sh” in the “build” folder and then “make” to obtain the executable file “lmp_mpi”, more details about building the lammps with cmake can be found on the website of Lammps. Now, MD simulations in Lammps can obtain a pretty good efficiency. There is an example in the folder “lammps-interface/example”. The (sequence number) of elements should begin with 1 not 0 and the sequence of elements should be the same as the "atomtype" list in "input_density". Note: The multi-process parallelization algorithm for REANN is less straightforward as in classical force fields or conventional locally atomic descriptor-based ML models. Currently, the MD simulation based on the REANN model can only perform on a single process (GPU/CPU). However, if you set oc_loop=0, REANN will degrade to eann. The MD simulations can run on multiple CPUs or GPUs (the number of GPUs should be equal to the number of processes) and obtain a pretty good acceleration.

Inference Based on ASE

In the "ASE" folder, there is a python script "ase_fireann.py" used as an example to calculate the energy and atomic forces by invoking the model save in "PES.pt". Before running the script, you need to copy the "calculator.py and fireann.py" from the calculator folder to the ASE installation path. Note: The "atomtype" script in the inference should be same with that in the "input_density". In this interface, we use a cell-linked algorithm to construct the neighbor list by a high efficient Fortran implementation. Thus, the Fortran code should be compiled by f2py, which generates a dynamic link library by running a "run" script, which can be called by python.