NequIP

MIR

Mar 26, 2023

CONTENTS:

NequIP is an open-source package for creating, training, and using E(3)-equivariant machine learning interatomic potentials.

CHAPTER

ONE

OVERVIEW

TODO

CHAPTER

TWO

CITING NEQUIP

CHAPTER

THREE

INSTALLATION

NequIP requires:

- Python >= 3.6
- PyTorch >= 1.8, <=1.11.*. PyTorch can be installed following the instructions from their documentation. Note that neither `torchvision` nor `torchaudio`, included in the default install command, are needed for NequIP.

To install:

- We use Weights&Biases to keep track of experiments. This is not a strict requirement — you can use our package without it — but it may make your life easier. If you want to use it, create an account here and install the Python package:

```
pip install wandb
```

- Install the latest stable NequIP:

```
pip install https://github.com/mir-group/nequip/archive/main.zip
```

To install previous versions of NequIP, please clone the repository from GitHub and check out the appropriate tag (for example `v0.3.3` for version 0.3.3).

To install the current **unstable** development version of NequIP, please clone our repository and check out the `develop` branch.

3.1 Installation Issues

The easiest way to check if your installation is working is to train a _toy_ model:

```
nequip-train configs/minimal.yaml
```

If you suspect something is wrong, encounter errors, or just want to confirm that everything is in working order, you can also run the unit tests:

```
pip install pytest
pytest tests/unit/
```

To run the full tests, including a set of longer/more intensive integration tests, run:

```
pytest tests/
```

If a GPU is present, the unit tests will use it.

CHAPTER

FOUR

YAML INPUT

TODO

CHAPTER

FIVE

HOW-TO TUTORIALS

5.1 How to prepare training dataset

5.1.1 What does NequIP behind the scene

NequIP uses AtomicDataset class to store the atomic configurations. During the initialization of an AtomicDataset object, NequIP reads the atomic structures from the dataset, computes the neighbor list and other data structures needed for the GNN by converting raw data to a list of `AtomicData` objects.

The computed results are then cached on harddisk `root/processed_hashkey` folder. The hashing is based on all the metadata provided for the dataset, which includes the file name, the cutoff radius, float number precision and etc. In the case where multiple training/evaluation runs use the same dataset, the neighbor list will only be computed in the first NequIP run. The later runs will directly load the AtomicDataset object from the cache file to save computation time.

Note: be careful to the cached file. If you update your raw data file but keep using the same filename, NequIP will not automatically update the cached data.

5.1.2 Key concepts

fixed_fields

Fixed fields are the quantities that are shared among all the configurations in the dataset. For example, if the dataset is a trajectory of an NVT MD simulation, the super cell size and the atomic species are indeed a constant matrix/vector through out the whole dataset. In this case, in stead of repeating the same values for many times, we specify the cell and species as fixed fields and only provide them once.

yaml interface

`nequip-train` and `nequip-evaluate` automatically construct the AtomicDataset based on the yaml arguments. Later sections offer a couple different examples.

If the training and validation datasets are from different raw files, the arguments for each set can be defined with `dataset` prefix and `validation_dataset` prefix, respectively.

For example, `dataset_file_name` is used for training data and `validation_dataset_file_name` is for validation data.

Python interface

See `nequip.data.dataset.AtomicInMemoryDataset`.

5.1.3 Prepare dataset and specify in yaml config

ASE format

NequIP accept all format that can be parsed by *ase.io.read* function. We recommend *extxyz*.

Example: Given an atomic data stored in "H2.extxyz" that looks like below:

```
2
Properties=species:S:1:pos:R:3 energy=-10 user_label=2.0 pbc="F F F"
H       0.00000000       0.00000000       0.00000000
H       0.00000000       0.00000000       1.02000000
```

The yaml input should be

```
dataset: ase
dataset_file_name: H2.extxyz
ase_args:
format: extxyz
include_keys:
  - user_label
key_mapping:
  user_label: label0
chemical_symbol_to_type:
  H: 0
```

For other formats than *extxyz*, be careful to the ase parsers; they may have different behavior from the extxyz parser. For example, the ase vasp parser store potential energy to *free_energy* instead of *energy*. Because we optimize our code to the *extxyz* parser, NequIP will not be able to load any *total_energy* labels. We need some additional keys to help NequIP to understand the situtaion Here's an example for vasp outcar.

```
dataset: ase
dataset_file_name: OUTCAR
ase_args:
  format: vasp-out
key_mapping:
  free_energy: total_energy
chemical_symbol_to_type:
  H: 0
```

The way around is to use key mapping, please see more note below.

NPZ format

If your dataset constitute configurations that always have the same number of atoms, npz data format can be an option.

In the npz file, all the values should have the same row as the number of the configurations. For example, the force array of 36 atomic configurations of an N-atom system should have the shape of (36, N, 3); their total_energy array should have the shape of (36).

Below is an example of the yaml specification.

```
dataset: npz
dataset_file_name: example.npz
include_keys:
  - user_label1
  - user_label2
npz_fixed_field_keys:
  - cell
  - atomic_numbers
key_mapping:
  position: pos
  force: forces
  energy: total_energy
  Z: atomic_numbers
```

Note on key mapping

NequIP has default key names for energy, force, cell (defined at nequip.data._keys) Unlike in the ASE format where these information is automatically parsed, in the npz data format, the correct key names have to be provided. The common key names are: *total_energy*, *forces*, *atomic_numbers*, *pos*, *cell*, *pbc*. the key_mapping can help to convert the user defined name (key) to NequIP default name (value).

5.1.4 Advanced options

skip frames during data processing

The *include_frame* argument can be specified in yaml to skip certain frames in the raw datafile. The item has to be a list or a python iteratable object.

register user-defined graph, node, edge fields

Graph, node, edge fields are quantities that belong to the whole graph, each atom, each edge, respectively. Example graph fields include cell, pbc, and total_energy. Example node fields include pos, forces

To help NequIP to properly assemble the batch data, graph quantity other than cell, pbc, total_energy should be registered.

5.2 How to migrate to newer NequIP versions

(Written for migration from 0.3.3 to 0.4. Nov. 3. 2021)

If the model are mostly the same and there is only some internal variable changes, it is possible to migrate your NequIP model from the older version to the newer version.

5.2.1 Upgrade NequIP

1. Record the old version

Go to the code folder in your virtual environment, find out the last commit that you are using

```
# bash code
NEQUIP_FOLDER=$(python -c "import nequip; print(\"/\".join(nequip.__file__.
↪split(\"/\")[:-1]))")
cd ${NEQUIP_FOLDER}
pwd
git show --oneline -s
OLD_COMMIT=$(git show --oneline -s|awk '{print $1}')
```

2. Update your main nequip repo

```
# bash code
git pull origin main
pip install -e ./
```

5.2.2 Obtain the state_dict from the old ver

For version before 0.3.3, the `last_model.pth` stores the whole pickle model. So you need to save the `state_dict()`; otherwise, skip this section.

1. Back up the old version

Git clone the old commit to a new folder

```
# bash code
git clone git@github.com:mir-group/nequip.git -n old_nequip
cd old_nequip
git checkout ${OLD_COMMIT}
```

2. Save the state_dict from the old verion

Go to the old_nequip folder, make sure that your current nequip is overloaded by local nequip folder. The result of the code below should show the old_nequip folder instead of the one usually used in the virtualenv.

```
#python
import nequip
print(nequip.__file__)
```

Load the old model with the old verion in python.

```
# save_state_dict.py
import torch
import sys
model_folder = sys.argv[1]
old_model=torch.load(
    f"{model_folder}/last_model.pth",
    map_location=torch.device('cpu') # if it operates on CPU
    )
torch.save(old_model.state_dict(), f"{model_folder}/new_last_model.pth")
```

5.2.3 Load the state_dict in the new version

Go to any other directorys that are not in the old version nequip folder.

Double check now the `nequip.__file__` should locate at the `${NEQUIP_FOLDER}`

Then try to load the old `state_dict()` to the new model.

```
# in new nequip
import torch
from nequip.utils import Config
from nequip.model import model_from_config

config = Config.from_file("config_final.yaml")

# only needed for version 0.3.3
config["train_on_keys"]=["forces", "total_energy"]
config["model_builders"] = ["EnergyModel", "PerSpeciesRescale", "ForceOutput",
↪"RescaleEnergyEtc"]

model = model_from_config(config, initialize=False)

d = torch.load("new.pth")
# load the state dict to the new model
model.load_state_dict(d)
```

The code will likely to fail. Render some outputs like below:

```
RuntimeError: Error(s) in loading state_dict for RescaleOutput:
        Missing key(s) in state_dict: "model.func.per_species_rescale.shifts",
↪"model.func.per_species_rescale.scales".
        Unexpected key(s) in state_dict: "model.func.per_species_scale_shift.
```

(continues on next page)

(continued from previous page)

```
↪shifts", "model.func.per_species_scale_shift.scales", "model.func.radial_
↪basis.cutoff.p", "model.func.radial_basis.cutoff.r_max"
```

According to this output and the CHANGELOG.md file, we can revise the dictionary by renaming or removing variables.

```
# rename all parameters listed in the change log as changed.
d["model.func.per_species_rescale.shifts"]=d.pop("model.func.per_species_scale_
↪shift.shifts")
d["model.func.per_species_rescale.scales"]=d.pop("model.func.per_species_scale_
↪shift.scales")
d.pop("model.func.radial_basis.cutoff.p")
d.pop("model.func.radial_basis.cutoff.r_max")

# load the state dict to the new model
model.load_state_dict(d)

# save the new state dict
import nequip
torch.save(model.state_dict(), f"new_last_model_{nequip.__version__}.pth')
```

5.2.4 Validate the result using nequip-evaluate

Old model

```
python nequip/script/evaluate.py
```

New model

```
nequip-evaluate --train-dir new_model/ --dataset-config data.yaml --output new.
↪xyz
```

```
root: ./
r_max: 4
validation_dataset: ase
validation_dataset_file_name: validate.xyz
chemical_symbol_to_type:
  H: 0
  C: 1
  O: 2
```

CHAPTER

SIX

FAQ

6.1 How do I...

... continue to train a model that reached a stopping condition?
There will be an answer here.

1. Reload the model trained with version 0.3.3 to the code in 0.4. check out the migration note at *How to migrate to newer NequIP versions*.
2. Specify my dataset for *nequip-train* and *nequip-eval*, see _dataset_note.

CHAPTER

SEVEN

COMMAND-LINE EXECUTABLES

7.1 nequip-train

```
usage: nequip-train [-h] [--equivariance-test] [--model-debug-mode] [--grad-
↪anomaly-mode] [--log LOG] config
```

Train (or restart training of) a NequIP model.

positional arguments:
config YAML file configuring the model, dataset, and other options

optional arguments:

-h, --help	show this help message and exit
--equivariance-test	test the model's equivariance before training
--model-debug-mode	enable model debug mode, which can sometimes give much more useful error messages at the cost of some speed. Do not use for production training!
--grad-anomaly-mode	enable PyTorch autograd anomaly mode to debug NaN gradients. Do not use for production training!
--log LOG	log file to store all the screen logging

7.2 nequip-evaluate

```
usage: nequip-evaluate [-h] [--train-dir TRAIN_DIR] [--model MODEL] [--dataset-
↪config DATASET_CONFIG]
                       [--metrics-config METRICS_CONFIG] [--test-indexes TEST_
↪INDEXES] [--batch-size BATCH_SIZE]
                       [--device DEVICE] [--output OUTPUT] [--log LOG]
```

Compute the error of a model on a test set using various metrics. The model, metrics, dataset, etc. can specified in individual YAML config files, or a training session can be indicated with `--train-dir`. In order of priority, the global settings (dtype, TensorFloat32, etc.) are taken from: (1) the model config (for a training session), (2) the dataset config (for a deployed model), or (3) the defaults. Prints only the final result in `name = num` format to stdout; all other information is ``logging.debug``ed to stderr. WARNING: Please note that results of CUDA models are rarely exactly reproducible, and that even CPU models can be nondeterministic.

optional arguments:

-h, --help show this help message and exit

--train-dir TRAIN_DIR Path to a working directory from a training session.

--model MODEL A deployed or pickled NequIP model to load. If omitted, defaults to *best_model.pth* in *train_dir*.

--dataset-config DATASET_CONFIG A YAML config file specifying the dataset to load test data from. If omitted, *config.yaml* in *train_dir* will be used

--metrics-config METRICS_CONFIG A YAML config file specifying the metrics to compute. If omitted, *config.yaml* in *train_dir* will be used. If the config does not specify *metrics_components*, the default is to logging.debug MAEs and RMSEs for all fields given in the loss function. If the literal string *None*, no metrics will be computed.

--test-indexes TEST_INDEXES Path to a file containing the indexes in the dataset that make up the test set. If omitted, all data frames *not* used as training or validation data in the training session *train_dir* will be used.

--batch-size BATCH_SIZE Batch size to use. Larger is usually faster on GPU.

--device DEVICE Device to run the model on. If not provided, defaults to CUDA if available and CPU otherwise.

--output OUTPUT XYZ file to write out the test set and model predicted forces, energies, etc. to.

--log LOG log file to store all the metrics and screen logging.debug

7.3 nequip-deploy

```
usage: nequip-deploy [-h] {info,build} ...
```

Deploy and view information about previously deployed NequIP models.

optional arguments:

-h, --help show this help message and exit

commands:

{info,build}
info Get information from a deployed model file build Build a deployment model

7.3.1 nequip-deploy info

```
usage: nequip-deploy info [-h] model_path
```

positional arguments:
model_path Path to a deployed model file.

optional arguments:

-h, --help	show this help message and exit

7.3.2 nequip-deploy build

```
usage: nequip-deploy build [-h] train_dir out_file
```

positional arguments:
train_dir Path to a working directory from a training session. out_file Output file for deployed model.

optional arguments:

-h, --help	show this help message and exit

7.4 nequip-benchmark

```
usage: nequip-benchmark [-h] [--profile PROFILE] [--device DEVICE] [-n N] [--n-
↪data N_DATA] [--timestep TIMESTEP]
                        config
```

Benchmark the approximate MD performance of a given model configuration / dataset pair.

positional arguments:
config configuration file

optional arguments:

-h, --help	show this help message and exit
--profile PROFILE	Profile instead of timing, creating and outputing a Chrome trace JSON to the given path.
--device DEVICE	Device to run the model on. If not provided, defaults to CUDA if available and CPU otherwise.
-n N	Number of trials.
--n-data N_DATA	Number of frames to use.
--timestep TIMESTEP	MD timestep for ns/day esimation, in fs. Defauts to 1fs.

CHAPTER

EIGHT

INTEGRATION TO LAMMPS, ASE

8.1 LAMMPS

8.2 ASE

CHAPTER

NINE

ALL OPTIONS

9.1 General

9.1.1 Basic

root

Type:
Default: n/a

run_name

Type: path
Default: n/a

`run_name` specifies something about whatever

9.1.2 Advanced

allow_tf32

Type: bool
Default: `False`

If `False`, the use of NVIDIA's TensorFloat32 on Tensor Cores (Ampere architecture and later) will be disabled. If `True`, the PyTorch defaults (use anywhere possible) will remain.

9.2 Dataset

9.2.1 Basic

r_max

See *r_max*.

type_names

Type: NoneType
Default: None

chemical_symbols

Type: NoneType
Default: None

chemical_symbol_to_type

Type: NoneType
Default: None

avg_num_neighbors

Type: NoneType
Default: None

key_mapping

Type: dict
Default: {'positions': 'pos', 'energy': 'total_energy', 'force': 'forces', 'forces': 'forces', 'Z': 'atomic_numbers', 'atomic_number': 'atomic_numbers'}

include_keys

Type: list
Default: []

npz_fixed_field_keys

Type: list
Default: []

file_name

Type: NoneType
Default: None

url

Type: NoneType
Default: `None`

force_fixed_keys

Type: list
Default: `[]`

extra_fixed_fields

Type: dict
Default: `{}`

include_frames

Type: NoneType
Default: `None`

ase_args

Type: dict
Default: `{}`

9.2.2 Advanced

See tutorial on ../guide/_dataset_note.

9.3 Model

9.3.1 Edge Basis

Basic

r_max

Type: float
Default: n/a

The cutoff radius within which an atom is considered a neighbor.

irreps_edge_sh

Type: Irreps or int
Default: n/a

The irreps to use for the spherical harmonic projection of the edges. If an integer, specifies all spherical harmonics up to and including that integer as ℓ_{max}. If provided as explicit irreps, all multiplicities should be 1.

num_basis

Type: int
Default: `8`

The number of radial basis functions to use.

chemical_embedding_irreps_out

Type: Irreps
Default: n/a

The size of the linear embedding of the chemistry of an atom.

Advanced

BesselBasis_trainable

Type: bool
Default: `True`

Whether the Bessel radial basis should be trainable.

basis

Type: type
Default: `<class 'nequip.nn.radial_basis.BesselBasis'>`

The radial basis to use.

9.3.2 Convolution

Basic

num_layers

Type: int
Default: `3`

The number of convolution layers.

feature_irreps_hidden

Type: Irreps
Default: n/a

Specifies the irreps and multiplicities of the hidden features. Typically, include irreps with all ℓ values up to ℓ_{max} (see *irreps_edge_sh*), each with both even and odd parity. For example, for `irreps_edge_sh: 1`, one might provide: `feature_irreps_hidden: 16x0e + 16x0o + 16x1e + 16x1o`.

Advanced

invariant_layers

Type: int
Default: `1`

The number of hidden layers in the radial neural network.

invariant_neurons

Type: int
Default: `8`

The width of the hidden layers of the radial neural network.

resnet

Type: bool
Default: `False`

nonlinearity_type

Type: str
Default: `gate`

nonlinearity_scalars

Type: dict
Default: `{'e': 'ssp', 'o': 'tanh'}`

nonlinearity_gates

Type: dict
Default: `{'e': 'ssp', 'o': 'abs'}`

use_sc

Type: bool
Default: `True`

9.3.3 Output block

Basic

conv_to_output_hidden_irreps_out

Type: Irreps
Default: n/a

The middle (hidden) irreps of the output block. Should only contain irreps that are contained in the output of the network (`0e` for potentials).

Advanced

9.4 Training

9.4.1 Basic

9.4.2 Advanced

9.5 Logging

9.5.1 Basic

9.5.2 Advanced

CHAPTER

TEN

PYTHON API

10.1 nequip.data

10.2 nequip.trainer

CHAPTER

ELEVEN

ERRORS

11.1 Common errors

Various shape errors
Check the sanity of the shapes in your dataset.

Out-of-memory errors with *nequip-evaluate*
Choose a lower `--batch-size`; while the highest value that fits in your GPU memory is good for performance, lowering this does *not* affect the final results (beyond numerics).

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search