mace

Release 0.1.0

Ilyes Batatia, David Kovacs, Gregor Simm, and contributors

Apr 09, 2024

USER GUIDE

CHAPTER

ONE

OVERVIEW

MACE is a machine learning software for predicting many-body atomic interactions and generating force fields. It utilizes higher order equivariant message passing for fast and accurate predictions.

CHAPTER

TWO

FEATURES

- Predicts many-body atomic interactions with high accuracy
- Generates force fields for use in molecular dynamics simulations
- Utilizes higher order equivariant message passing
- Fast and efficient predictions

CHAPTER

THREE

GETTING STARTED

CHAPTER

FOUR

DOCUMENTATION

For detailed information on how to use `MACE`, please refer to the following documentation:

4.1 Quick Start

See Installation section under User Guide.

4.2 User Guide

4.2.1 Introduction

What is MACE ?

MACE provides fast and accurate machine learning interatomic potentials with higher order equivariant message passing.

Also available: * [MACE in JAX](https://github.com/ACEsuit/mace-jax), currently about 2x times faster at evaluation, but training is recommended in Pytorch for optimal performances. * [MACE layers](https://github.com/ACEsuit/mace-layer) for constructing higher order equivariant graph neural networks for arbitrary 3D point clouds.

4.2.2 Installation

Requirements: * Python >= 3.7 * [PyTorch](https://pytorch.org/) >= 1.12

(for openMM, use Python = 3.9)

pip installation

To install via `pip`, follow the steps below:

```
pip install --upgrade pip
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/
↪cu118
pip install mace-torch
```

For CPU or MPS (Apple Silicon) installation, use `pip install torch torchvision torchaudio` instead.

conda installation

If you do not have CUDA pre-installed, it is **recommended** to follow the conda installation process:

```
# Create a virtual environment and activate it
conda create mace_env
conda activate mace_env

# Install PyTorch
conda install pytorch torchvision torchaudio pytorch-cuda=11.6 -c pytorch -c nvidia

# (optional) Install MACE's dependencies from Conda as well
conda install numpy scipy matplotlib ase opt_einsum prettytable pandas e3nn

# Clone and install MACE (and all required packages)
git clone https://github.com/ACEsuit/mace.git
pip install ./mace
```

pip installation from source

To install via `pip`, follow the steps below:

```
# Create a virtual environment and activate it
python -m venv mace-venv
source mace-venv/bin/activate

# Install PyTorch (for example, for CUDA 11.6 [cu116])
pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/
↪whl/cu116

# Clone and install MACE (and all required packages)
git clone https://github.com/ACEsuit/mace.git
pip install ./mace
```

Note: The homonymous package on PyPI has nothing to do with this one.

4.2.3 Training

Script

To train a MACE model, you can use the `run_train.py` script (note that if you used `pip install` to install mace, you can also use the executable `mace_run_train` entry point which should be in your path).

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="MACE_model" \
    --train_file="train.xyz" \
    --valid_fraction=0.05 \
    --test_file="test.xyz" \
    --config_type_weights='{"Default":1.0}' \
    --E0s='{1:-13.663181292231226, 6:-1029.2809654211628, 7:-1484.1187695035828, 8:-2042.
↪0330099956639}' \
```

(continues on next page)

(continued from previous page)

```
    --model="MACE" \
    --hidden_irreps='128x0e + 128x1o' \
    --r_max=5.0 \
    --batch_size=10 \
    --max_num_epochs=1500 \
    --swa \
    --start_swa=1200 \
    --ema \
    --ema_decay=0.99 \
    --amsgrad \
    --restart_latest \
    --device=cuda \
```

Training files

To train a MACE model, you will use the `run_train.py`command which takes the following arguments:

First specify the name of your model and final log file using the `--name` flag.

You can specify the training file with the `--train_file` flag. The validation set can either be specified as a separate file using the `--valid_file` keyword, or it can be specified as a fraction of the training set using the `--valid_fraction` keyword. The validation set is not used for optimizing the model but to estimate the model accuracy during training.

It is also possible to provide a test set using the `--test_file` keyword. This set is entirely independent and only gets evaluated at the end of the training process.

Model

Model options

The `--model` flag specifies the type of model to be trained. The vanilla MACE model is specified by `--model="MACE"`. The `--model="ScaleShiftMACE"` model includes a residual connection at first, which will usually improve the model's accuracy but will make the model output incorrect isolated atoms energies. Use this model if you are not interested in bond-breaking energies. To train a model on dipole moments, use `--model="AtomicDipolesMACE"`. If you want to also train simultaneously on energies, use `--model="EnergyDipolesMACE"`.

The Messages

Change `-hidden_irreps` to control the model size. For most applications, the recommended default model size is `--hidden_irreps='256x0e'` (meaning 256 invariant messages) or `--hidden_irreps='128x0e + 128x1o'` (meaning 128 equivariant messages). If the model is not accurate enough, you can include higher order features, e.g., `128x0e + 128x1o + 128x2e`, or increase the number of channels to `256`. The number of message passing layers can be controlled via the `--num_ineractions` parameter. **Increasing the model size and the number of layers will lead to more accurate but slower models.**

Correlation order

MACE uses a body order expansion on the site energy:

$E_i = E_i^{(0)} + \sum_j E_{ij}^{(1)} + \sum_{jk} E_{ijk}^{(2)} + ...$

The correlation order corresponds to the order that MACE induces at each layer. Choosing `--correlation=3` will create basis function of up to 4-body (ijke) indices, for each layer. Because of the multiple layers of MACE, the total correlation order is much higher. A two layers mace, with `--correlation=3` has a total body order of 13.

Angular resolution

The angular resolution describes how precise the model can identify angles. This is controled by `l_max`. The higher this integer, more precise is the angular resolution. Larger value will result in more accurate but slower models. The default is `l_max=3`.

Cutoff radius

The cutoff radius controls the locality of the model. A `--r_max=3.0` means that the model assumes atoms seperated by a distance of more than 3.0 A do not directly `communicate`. Because the model has two layers, atoms further than 3.0 A can still `communicate` by proxy. The actual receptive field of the model is the number of layers times the cutoff distance.

Reference energies

It is usually preferred to add the isolated atoms to the training set, rather than reading in their energies through the command line like in the example above. To label them in the training set, set `config_type=IsolatedAtom` in their info fields. If you prefer not to use or do not know the energies of the isolated atoms, you can use the option `--E0s="average"` which estimates the atomic energies using least squares regression.

SWA and EMA

If the keyword `--swa` is enabled, the energy weight of the loss is increased for the last ~20% of the training epochs (from `--start_swa` epochs). This setting usually helps lower the energy errors.

Data keys

When parsing the data files, the energies are read using the keyword `energy` and the forces using the keyword `forces`. To change that, specify the `--energy_key` and `--forces_key`. You can also specify `--stress_key` to read the stress tensor, `--virials_key` to read the virial tensor, and `--dipole_key` to read the dipole moments.

Float precision

The precision can be changed using the keyword `--default_dtype`, the default is `float64` but `float32` gives a significant speed-up (usually a factor of x2 in training).

Set batch size

The keywords `--batch_size` and `--max_num_epochs` should be adapted based on the size of the training set. The batch size should be increased when the number of training data increases, and the number of epochs should be decreased. An heuristic for initial settings, is to consider the number of gradient update constant to 200 000, which can be computed as $text{max-num-epochs}*frac{text{num-configs-training}}{text{batch-size}}$.

Validation parameters

The validation set controls the stopping of the training. At each `--eval_interval` the model is tested on the validation set. We also evaluate the set by batch size, controlled by `--valid_batch_size`. If the accuracy of the model stops improving on the validation set for `--patience` number of epochs. This is called **early stopping**.

Heterogeneous labels

The code can handle training set with heterogeneous labels, for example containing both bulk structures with stress and isolated molecules. In this example, to make the code ignore stress on molecules, append to your molecules configuration a `config_stress_weight = 0.0`.

Devices

To use GPUs, specify `--device=cuda`. To use CPUs, specify `--device=cpu`. To use Apple Silicon GPU acceleration make sure to install the latest PyTorch version and specify `--device=mps`.

Checkpoints

For trainings that require restarting, you can continue the fitting from the last checkpoint by using the flag `--restart_latest`. The checkpoint saves the best model that currently has been trained. All checkpoints are saved in ./checkpoints folder. We can also continue from a restart when extending the dataset or changing any hyperparameters that do not affect the model size.

4.2.4 Evaluation

To evaluate your MACE model on an XYZ file, run the `eval_configs.py`:

```
python3 <mace_repo_dir>/mace/cli/eval_configs.py \
    --configs="your_configs.xyz" \
    --model="your_model.model" \
    --output="./your_output.xyz"
```

4.2.5 ASE calculator

MACE models can run molecular dynamics or geometry optimisation through the ASE calculator. The ASE calculator is a Python module that can be used to run MD simulations or geometry optimisations.

The native ASE neighbour list being slow, it is recommended to use the matscipy branch. **Note** that the matscipy branch is not compatible with non-periodic systems.

Running MD simulations

The ASE calculator can be used to run MD simulations with MACE. The following example shows how to run a MD simulation of a 3BPA molecule in vacuum.

```
from ase import units
from ase.md.langevin import Langevin
from ase.io import read, write
import numpy as np
import time

from mace.calculators import MACECalculator

calculator = MACECalculator(model_path='/content/checkpoints/MACE_model_run-123.model',␣
↪device='cuda')
init_conf = read('BOTNet-datasets/dataset_3BPA/test_300K.xyz', '0')
init_conf.set_calculator(calculator)

dyn = Langevin(init_conf, 0.5*units.fs, temperature_K=310, friction=5e-3)
def write_frame():
        dyn.atoms.write('md_3bpa.xyz', append=True)
dyn.attach(write_frame, interval=50)
dyn.run(100)
print("MD finished!")
```

To get the model and the data associated with this example, please run the colab tutorial here.

4.2.6 MACE descriptors

MACE descriptors are atomic features obtained after each message passing block of a MACE model. As MACE is a very expressive model, these descriptors are rich and can be used for various tasks, such as classification or analysis. To extract these descriptors directly from an ase.Atoms object, you can use the convenient function provided in the MACECalculator class.

Here is a simple example:

```
from ase.io import read
import numpy as np
from mace.calculators import MACECalculator
calculator = MACECalculator(model_paths='/content/checkpoints/MACE_model_run-123.model',␣
↪device='cuda')
init_conf = read('BOTNet-datasets/dataset_3BPA/test_300K.xyz', '0')
descriptors = calculator.get_descriptors(init_conf)
```

Currently only the base `MACE` and `ScaleShiftMACE` models are supported. By default, only the invariant part of the descriptors is returned. To get the full descriptors, you can set the `invariants_only` argument to `False`.

Also, by default, the descriptors for each layer are returned. To get the descriptors of the first `n` layers, you can set the `num_layers` argument to `n`.

Note: The descriptors are returned in the form of numpy arrays, structured as follows:

- **First Dimension**: Corresponds to the number of atoms in the system.
- **Second Dimension**: Relates to the number of descriptors, dependent on the model used.

Depending on the value of the `invariants_only` argument, the number of descriptors, $N_{\text{descriptors}}$, is calculated as follows:

- If `invariants_only=True`: $N_{\text{descriptors}} = \text{nchannels} \times \text{nlayers}$.
- If `invariants_only=False`: $N_{\text{descriptors}} = \text{nchannels} \times (\text{nlayers} - 1) \times (L_{\text{max}} + 1)^2 + \text{nchannels}$.

4.2.7 OpenMM Interface

MACE models can be used to run molecular dynamics through OpenMM. A wide variety of simulations can be run in this way, and it allows for execution of the full simulation on the GPU.

Manual Installation steps

Create the conda environment from the provided environment file. Note we use mamba as a drop in conda replacement.

`conda install mamba -c conda-forge`

If you are installing on a headnode, override the virtual cuda package to match the compute node CUDA version.

```
export CONDA_OVERRIDE_CUDA=11.8
mamba env create -f mace-openmm.yml
```

- Install additional MACE/openMM related packages from GitHub

```
pip install git+https://github.com/choderalab/mpiplus.git
pip install git+https://github.com/ACEsuit/mace.git
pip install git+https://github.com/openmm/openmm-ml.git
pip install git+https://github.com/jharrymoore/openmmtools.git@development
```

- To quickly test the installation, run the following command to run a short MACE MD simulation

```
wget https://github.com/ACEsuit/mace-off/blob/main/mace_off23/MACE-OFF23_small.model
mace-md -f ejm_31.sdf --ml_mol ejm_31.sdf --model_path MACE-OFF23_small.model --output_
↪dir md_test --nl torch_nl --steps 1000 --minimiser ase --dtype float64 --remove_cmm --
↪unwrap
```

Testing your Installation

Run the unit tests for mace-md with the following:

```
pytest -s openmmtools/tests/test_mace-md.py
```

Running MD simulations

The following snippets use files from the `examples/example_data` folder of the openmmtools repository.

Run pure MACE MD simulation with Langevin dynamics:

```
from openmmtools.openmm_torch.hybrid_md import PureSystem
import torch

torch.set_default_dtype(torch.float64)

file = "ejm_31.sdf"
model_path = "MACE_SPICE_larger.model"
temperature = 298

system=PureSystem(
    ml_mol=file
    model_path=model_path
    potential="mace"
    output_dir="output_md"
    temperature=298,
    nl="torch"
)

system.run_mixed_md(
    steps=5000, interval=25, output_file="output.pdb", restart=False,
)
```

Example of a MACE NPT simulation with periodic boundary conditions:

```
from openmmtools.openmm_torch.hybrid_md import PureSystem
import torch

torch.set_default_dtype(torch.float64)

file = "waterbox.xyz"
model_path = "MACE_SPICE_larger.model"
temperature = 298
pressure = 1.0

system=PureSystem(
    file=file,
    model_path=model_path,
    potential="mace",
    temperature=temperature,
    output_dir="output_md",
            pressure = pressure
```

(continues on next page)

(continued from previous page)

```
)

system.run_mixed_md(
    steps=10000, interval=50, output_file="output_md_water.pdb", restart=False
)
```

Example of a hybrid ML-MM simulation where the small molecule is parametrised by MACE, whilst the solvent and ions are modelled with a classical FF

```
from openmmtools.openmm_torch.hybrid_md import HybridSystem
import torch

torch.set_default_dtype(torch.float64)

file = "ejm_31.sdf"
model_path = "MACE_SPICE_larger.model"
temperature = 298

system = MixedSystem(
    file=file,
    ml_mol=file,
    model_path=model_path,
    potential="mace",
    output_dir="output_hybrid",
    temperature=298,
    nl="nnpops",
    nnpify_type="resname",
    resname="UNK",
)

system.run_mixed_md(
    steps=10000, interval=50, output_file="output_md_mlmm.pdb", restart=False
)
```

Alternatively, simulations can also be run through the mace-md interface, which exposes exactly the same functionality.

Pure MD simulations

The simplest use case is where the full system is simulated with the MACE potential. The simulation can be started from a `.xyz` file as follows, which will run the simulation for 1000 steps, reporting structures and run information every 100 steps

```
mace-md -f molecule.xyz --ml_mol molecule.xyz --model_path /path/to/my-mace.model --steps
1000 --timestep 1.0 --integrator langevin --interval 100 --output_dir ./test_output
```

For a full set of command line argument options, run `mace-md -h`

Hybrid ML/MM simulations

It is also possible to run MD simulations where only a subset of the system is treated with a MACE potential, with the rest treated using a classical potential. This is a 'mechanical embedding' regime, in that only the intramolecular components are described by the ML potential, whilst the long-range dispersion and coulomb interactions are still described clasically

To run these simulations, there are more stringent requirements on the filetypes, since a full MM topology must also be built, requiring explicit bonds and atomtypes. This typically means the full system should be provided as a PDB file, whilst the small molecule (or the part to be evaluated with MACE) is provided as an sdf file.

Whilst it is possible to run a plain MD trajectory like this, this setup is particularly useful for computing free energy corrections from the full MM to the ML/MM hamiltonian. By specifying `--run_type repex`, a replica exchange simulation will be performed, in which each intermediate state has a fractional contribution of the MM and ML components for the small molecule. The full command to run a replica exchange job looks like this

```
mace-md -f complex.pdb --ml_mol ligand.sdf --run_type repex --replicas 8 --output_dir ./
↪repex_output --steps 1000 --model_path /path/to/my-mace.model
```

This will run 1 ns (1000 x 1 ps MCMC swap attempts), writing all information required to analyse the simulation and compute free energy corrections to the output dir.

4.2.8 MACE in LAMMPS

Warning: Please be cautious about benchmarking your LAMMPS model against, for example, the equivalent ASE calculator.

Both CPU and GPU evaluation are possible, but GPU evaluation will give MUCH better performance (at least at present).

Preparing your model

Train the model using the `develop` branch. Afterwards, use the `create_lammps_model.py` script to prepare a torchscript-compiled LAMMPS_MACE model:

```
python <mace_repo_dir>/mace/cli/create_lammps_model.py my_mace.model
```

Instructions for GPU

Installation

These instructions are for Cambridge-relevant machines and should be adapted as needed.

CSD3 Ampere Nodes

First steps:

```
git clone --branch=mace --depth=1 https://github.com/ACEsuit/lammps
wget https://download.pytorch.org/libtorch/cu121/libtorch-shared-with-deps-2.2.0%2Bcu121.
↪zip
unzip libtorch-shared-with-deps-2.2.0+cu121.zip
rm libtorch-shared-with-deps-2.2.0+cu121.zip
mv libtorch libtorch-gpu
```

Request an interactive job to obtain a GPU node for the installation:

```
sintr -A YOUR-ACCOUNT-GPU -p ampere -N 1 --gres=gpu:1 -t 1:00:00
```

After logging on to the GPU node interactively, prepare the environment:

```
module purge
module load intel-mkl-2017.4-gcc-5.4.0-2tzpyn7
module load rhel8/slurm rhel8/global gcc/9 openmpi/gcc/9.3/4.0.4 cuda/12.1 cudnn
```

Compile LAMMPS:

```
cd lammps
mkdir build-ampere
cd build-ampere
cmake \
    -D CMAKE_BUILD_TYPE=Release \
    -D CMAKE_INSTALL_PREFIX=$(pwd) \
    -D CMAKE_CXX_STANDARD=17 \
    -D CMAKE_CXX_STANDARD_REQUIRED=ON \
    -D BUILD_MPI=ON \
    -D BUILD_SHARED_LIBS=ON \
    -D PKG_KOKKOS=ON \
    -D Kokkos_ENABLE_CUDA=ON \
    -D CMAKE_CXX_COMPILER=$(pwd)/../lib/kokkos/bin/nvcc_wrapper \
    -D Kokkos_ARCH_AMDAVX=ON \
    -D Kokkos_ARCH_AMPERE100=ON \
    -D CMAKE_PREFIX_PATH=$(pwd)/../../libtorch-gpu \
    -D PKG_ML-MACE=ON \
    ../cmake
make -j 20
make install
```

Using the model in LAMMPS

Warning: At present, only single-GPU evaluation is recommended.

Begin your LAMMPS input with the following commands::

```
units         metal
atom_style    atomic
atom_modify   map yes
newton        on
```

Your pair commands should look something like this::

```
pair_style mace no_domain_decomposition
pair_coeff * * my_mace.model-lammps.pt C H N O
```

With no_domain_decomposition, LAMMPS builds a periodic graph rather than treating ghost atoms as independent nodes.

Finally, you should initiate Kokkos by calling LAMMPS with something like the following:

```
lmp -k on g 1 -sf kk -in in.lammps
```

Instructions for CPU

Installation

These instructions are for Cambridge-relevant machines and should be adapted as needed.

CSD3 Cascade Lake Nodes

This environment is specific to the Cascade Lake compute nodes. If you want to use Ice Lake, you'll need something slightly different - see the CSD3 documentation. Moreover, you may see MPI errors if you try running on a CSD3 head node; just use the compute nodes.:

```
module purge
module load rhel7/default-ccl
module load gcc/9
```

Download libtorch:

```
wget https://download.pytorch.org/libtorch/cpu/libtorch-shared-with-deps-1.13.0%2Bcpu.zip
unzip libtorch-shared-with-deps-1.13.0+cpu.zip
rm libtorch-shared-with-deps-1.13.0+cpu.zip
```

Install Lammps:

```
git clone --branch mace --depth=1 https://github.com/ACEsuit/lammps
cd lammps; mkdir build; cd build
cmake -DCMAKE_INSTALL_PREFIX=$(pwd) \
```

(continues on next page)

(continued from previous page)

```
      -D CMAKE_CXX_STANDARD=17 \
      -D CMAKE_CXX_STANDARD_REQUIRED=ON \
      -D BUILD_MPI=ON \
      -D BUILD_OMP=ON \
      -D PKG_OPENMP=ON \
      -D PKG_ML-MACE=ON \
      -D CMAKE_PREFIX_PATH=$(pwd)/../../libtorch \
      ../cmake
make -j 4
make install
```

Using the model in LAMMPS

Begin your LAMMPS input with the following commands::

```
units         metal
atom_style    atomic
atom_modify   map yes
newton        on
```

Your pair commands should look something like this::

```
pair_style mace
pair_coeff * * my_mace.model-lammps.pt C H N O
```

If you are using a single MPI process with threading (recommended for small systems), use the no_domain_decomposition option for speedups::

```
# add this atom_modify command after your atom_style command
atom_modify map yes

# add the no_domain decomposition option to the pair_style declaration
pair_stye mace no_domain_decomposition
```

With no_domain_decomposition, LAMMPS builds a periodic graph rather than treating ghost atoms as independent nodes.

Here is an example slurm script (for Cascade Lake). For now, it is best to rely on threading for smaller systems. For larger systems, you'll need to experiment - multiple-node jobs will work, but it is likely best to use a small number of MPI processes per node and threading for the rest. You may want the –exclusive option to get access to the full-node memory.:

```
#!/bin/bash

#SBATCH -J lammps-mace
#SBATCH -A MY-ACCOUNT-CPU
#SBATCH -p cclake
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --exclusive
#SBATCH --time=08:00:00
#SBATCH --mail-type=FAIL
```

(continues on next page)

(continued from previous page)

```
. /etc/profile.d/modules.sh
module purge
module load rhel7/default-ccl

export OMP_NUM_THREADS=56
export MKL_NUM_THREADS=56
mpirun -np 1 ../../lammps/build/lmp -in in.lammps
```

4.3 Example scripts for training MACE models

This page collects various training scripts for training MACE models in the paper "Evaluation of the MACE Force Field Architecture: from Medicinal Chemistry to Materials Science"

4.3.1 MD22: large molecules

The MD22 dataset (http://www.sgdml.org) contains configurations with energies and forces computed at the DFT level of QM for 7 different large molecular systems. In the paper we used a very large MACE model to showcase primarily the achievable accuracy with a model that is still usefully fast.

An example script for training the 2-layer model on the carbon nanotube (largest) system is given below

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="nanotube_large_r55" \
    --train_file="nanotube_large.xyz" \
    --valid_fraction=0.05 \
    --test_file="nanotube_test.xyz" \
    --E0s="average" \
    --model="MACE" \
    --num_interactions=2 \
    --num_channels=256 \
    --max_L=2 \
    --correlation=3 \
    --r_max=5.0 \
    --forces_weight=1000 \
    --energy_weight=10 \
    --batch_size=2 \
    --valid_batch_size=2 \
    --max_num_epochs=650 \
    --start_swa=450 \
    --scheduler_patience=5 \
    --patience=15 \
    --eval_interval=3 \
    --ema \
    --swa \
    --swa_forces_weight=10 \
    --error_table='PerAtomMAE' \
    --default_dtype="float64"\
    --device=cuda \
    --seed=123 \
```

(continues on next page)

(continued from previous page)

```
    --restart_latest \
    --save_cpu
```

In comparison, the single layer model uses `max_L=0`, because there is no equivariant message to be passed. this model is considerably (ca 10x) faster, and somewhat less accurate as shown in the paper.

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="nano_large_r6" \
    --train_file="nanotube_large.xyz" \
    --valid_fraction=0.05 \
    --test_file="nanotube_test.xyz" \
    --E0s="average" \
    --model="MACE" \
    --num_interactions=1 \
    --num_channels=256 \
    --max_L=0 \
    --correlation=3 \
    --r_max=6.0 \
    --forces_weight=1000 \
    --energy_weight=10 \
    --batch_size=4 \
    --valid_batch_size=8 \
    --max_num_epochs=1000 \
    --start_swa=600 \
    --scheduler_patience=5 \
    --patience=15 \
    --eval_interval=3 \
    --ema \
    --swa \
    --swa_forces_weight=10 \
    --error_table='PerAtomMAE' \
    --default_dtype="float64"\
    --device=cuda \
    --seed=123 \
    --restart_latest \
    --save_cpu
```

4.3.2 ANI-1x dataset: H, C, N, O transferable FF

We used the subset of the ANI-1x datatset (https://www.nature.com/articles/s41597-020-0473-z) that also has couple cluster reference data to train 3 transferable MACE models of increaing accuracy.

To train MACE on large datasets one can preprocess the data and use on the fly data loading. This option is currently available on the multi-GPU branch of MACE.

```
python <mace_repo_dir>/mace/cli/preprocess_data.py \
    --train_file="ani1x_cc_dft.xyz" \
    --valid_fraction=0.03 \
    --energy_key="DFT_energy" \
    --forces_key="DFT_forces" \
    --r_max=5.0 \
    --h5_prefix="ANI1x_cc_DFT_rc5_" \
```

(continues on next page)

(continued from previous page)

```
    --compute_statistics \
    --E0s="{1: -13.62222753701504, 6: -1029.4130839658328, 7: -1484.8710358098756, 8: -
↪2041.8396277138045}" \
    --seed=12345
```

This produces 3 files: ANI1x_cc_DFT_rc5_train.h5, ANI1x_cc_DFT_rc5_valid.h5, ANI1x_cc_DFT_rc5_statistics.json. The statistics file contains the mean and standard deviation of the energies and forces, which are used to normalize the data as well as other statistics like the cutoff and average number of neighbours used for internal normailsation of the model. For the smallest model we used `r_max=4.5` and for the medium and large models `r_max=5.0`. The training script for the smallest model is given below.

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="ani500k_small" \
    --train_file="ANI1x_cc_DFT_rc5_train.h5" \
    --valid_file="ANI1x_cc_DFT_rc5_valid.h5" \
    --statistics_file="ANI1x_cc_DFT_rc5_statistics.json" \
    --E0s="{1: -13.62222753701504, 6: -1029.4130839658328, 7: -1484.8710358098756, 8: -
↪2041.8396277138045}" \
    --model="MACE" \
    --num_interactions=2 \
    --num_channels=64 \
    --max_L=0 \
    --correlation=3 \
    --r_max=5.0 \
    --forces_weight=1000 \
    --energy_weight=40 \
    --weight_decay=1e-7 \
    --clip_grad=1.0 \
    --batch_size=128 \
    --valid_batch_size=128 \
    --max_num_epochs=500 \
    --scheduler_patience=20 \
    --patience=50 \
    --eval_interval=1 \
    --ema \
    --swa \
    --start_swa=250 \
    --swa_lr=0.00025 \
    --swa_forces_weight=10 \
    --num_workers=32 \
    --error_table='PerAtomMAE' \
    --default_dtype="float64"\
    --device=cuda \
    --seed=123 \
    --restart_latest \
    --save_cpu
```

The model can easily be transfer learned to CC level of theory. For this the preprocesing has to be repated with the CC energies. Than the training can simply be continued. Suince the CC data does not have forces it is crucial to deactivate scaling by the RMS of the forces by setting `scaling="no_scaling"`. For the fine tuning we have also reduced the learning rate.

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="ani500k_small" \
    --train_file="ANI1x_cc_rc5_train.h5" \
    --valid_file="ANI1x_cc_rc5_valid.h5" \
    --statistics_file="ANI1x_cc_rc5_statistics.json" \
    --E0s="{1: -13.62222753701504, 6: -1029.4130839658328, 7: -1484.8710358098756, 8: -
↪2041.8396277138045}" \
    --scaling="no_scaling" \
    --model="MACE" \
    --num_interactions=2 \
    --num_channels=64 \
    --max_L=0 \
    --correlation=3 \
    --r_max=5.0 \
    --forces_weight=0.0 \
    --energy_weight=10000 \
    --lr=0.001 \
    --weight_decay=1e-7 \
    --clip_grad=1.0 \
    --batch_size=128 \
    --valid_batch_size=128 \
    --max_num_epochs=750 \
    --scheduler_patience=15 \
    --patience=30 \
    --eval_interval=1 \
    --ema \
    --num_workers=16 \
    --error_table='PerAtomMAE' \
    --default_dtype="float64"\
    --device=cuda \
    --seed=123 \
    --restart_latest \
```

The medium model had `num_channels=96`, `max_L=1` and `r_max=5.0`. It was trained for 350 epochs, with the second part of the learning rate schedule starting after 175 epochs.

The large model had `num_channels=192`, `max_L=2` and `r_max=5.0`. It was trained for 210 epochs, with the second part of the learning rate schedule starting after 60 epochs. This very long training was necessary, because we were observing some constant shifts in the energies when evaluating the model on much larger systems than in the training set as discussed in the manuscript. The shifts were reduced by training the models longer.

The 6 ANI dataset trained MACE models (3 model size and DFT and CC reference data for each) is available at https://github.com/ACEsuit/mace/blob/docs/docs/examples/ANI_trained_MACE.zip

4.3.3 Liquid water

The liquid water dataset was downloaded from https://github.com/BingqingCheng/ab-initio-thermodynamics-of-water/tree/master/training-set

To train the smaller MACE model used in the simulations of the paper we used the following input line:

```
python <mace_repo_dir>/mace/cli/run_train.py \
    --name="water_1k_small" \
    --train_file="train.xyz" \
    --valid_fraction=0.05 \
    --test_file="test.xyz" \
    --E0s="average" \
    --model="MACE" \
    --num_interactions=2 \
    --num_channels=64 \
    --max_L=0 \
    --correlation=3 \
    --r_max=6.0 \
    --forces_weight=1000 \
    --energy_weight=10 \
    --energy_key="TotEnergy" \
    --forces_key="force" \
    --batch_size=2 \
    --valid_batch_size=4 \
    --max_num_epochs=800 \
    --start_swa=400 \
    --scheduler_patience=15 \
    --patience=30 \
    --eval_interval=4 \
    --ema \
    --swa \
    --error_table='PerAtomMAE' \
    --default_dtype="float64"\
    --device=cuda \
    --seed=123 \
    --restart_latest \
    --save_cpu \
```

4.4 Foundation models

Currently available pretrained MACE models:

1. MACE-MP: pretrained foundation models for materials chemistry, parameterised for 89 chemical elements.
2. MACE-OFF23: transferable organic force fields, parameterised for neutral organic molecules made up of 10 different chemical elements.
3. MACE-ANI-CC: MACE model trained on the coupled cluster accurate ANI training set of organic molecules, parameterised for H, C, N, O elements.

4.4.1 Pretrained MACE-MP-0 models

We have collaborated with the Materials Project (MP) to train a universal MACE checkpoints covering 89 elements on 1.6 M bulk crystals in the MPTrj dataset. The model are releaed on GitHub at https://github.com/ACEsuit/mace-mp.

To access the pretrained checkpoints as an ASE calculator, you can use the following code snippets:

```
from mace.calculators import mace_mp
from ase import build

macemp = mace_mp() # return ASE calculator
atoms = build.molecule('H2O')
descriptors_mp = macemp.get_descriptors(atoms)
```

```
from mace.calculators import mace_mp
from ase import build
from ase.md import Langevin
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase import units

macemp = mace_mp() # return the default medium ASE calculator equivalent to mace_
↪mp(model="medium")
#macemp = mace_mp(model="large") # return a larger model
#macemp = mace_mp(model="https://tinyurl.com/y7uhwpje") # downlaod the model at the␣
↪given url
#macemp = mace_mp(dispersion=True) # return a model with D3 dispersion correction
atoms = build.molecule('H2O')
atoms.calc = macemp

# Initialize velocities.
T_init = 300  # Initial temperature in K
MaxwellBoltzmannDistribution(atoms, T_init * units.kB)

# Set up the Langevin dynamics engine for NVT ensemble.
dyn = Langevin(atoms, 0.5 * units.fs, T_init * units.kB, 0.001)
n_steps = 200 # Number of steps to run
dyn.run(n_steps)
```

A full benchmark of the MACE-MP-0 models across more than 30 applications can be found in the paper.

Please cite,

```
@misc{batatia2023foundation,
    title={A foundation model for atomistic materials chemistry},
    author={Ilyes Batatia and Philipp Benner and Yuan Chiang and Alin M. Elena and Dávid␣
↪P. Kovács and Janosh Riebesell and Xavier R. Advincula and Mark Asta and William J.␣
↪Baldwin and Noam Bernstein and Arghya Bhowmik and Samuel M. Blau and Vlad Cărare and␣
↪James P. Darby and Sandip De and Flaviano Della Pia and Volker L. Deringer and Rokas␣
↪Elijošius and Zakariya El-Machachi and Edvin Fako and Andrea C. Ferrari and Annalena␣
↪Genreith-Schriever and Janine George and Rhys E. A. Goodall and Clare P. Grey and␣
↪Shuang Han and Will Handley and Hendrik H. Heenen and Kersti Hermansson and Christian␣
↪Holm and Jad Jaafar and Stephan Hofmann and Konstantin S. Jakob and Hyunwook Jung and␣
↪Venkat Kapil and Aaron D. Kaplan and Nima Karimitari and Namu Kroupa and Jolla␣
↪Kullgren and Matthew C. Kuner and Domantas Kuryla and Guoda Liepuoniute and Johannes T.
```

(continues on next page)

(continued from previous page)

```
↪ Margraf and Ioan-Bogdan Magdău and Angelos Michaelides and J. Harry Moore and Aakash␣
↪A. Naik and Samuel P. Niblett and Sam Walton Norwood and Niamh O'Neill and Christoph␣
↪Ortner and Kristin A. Persson and Karsten Reuter and Andrew S. Rosen and Lars L.␣
↪Schaaf and Christoph Schran and Eric Sivonxay and Tamás K. Stenczel and Viktor Svahn␣
↪and Christopher Sutton and Cas van der Oord and Eszter Varga-Umbrich and Tejs Vegge␣
↪and Martin Vondrák and Yangshuai Wang and William C. Witt and Fabian Zills and Gábor␣
↪Csányi},
    year={2023},
    eprint={2401.00096},
    archivePrefix={arXiv},
    primaryClass={physics.chem-ph}
}
```

and the relevent papers if you use these checkpoints (see mace_mp docstrings for a list).

4.4.2 MACE-OFF23: Transferable Organic Force Fields

MACE-OFF23 are a series of three transferable organic force fields for organic chemistry. They were parameterised for 10 chemical elements: H, C, N, O, P, S, F, Cl, Br, I. It can be used to study systems of neutral molecules in gas phase liquid phase, or for organic crystals. If you use the model please cite the preprint.

The models are published under the Academic Software License (ASL) and can be downloaded from here.

The models can also be used simply as an ASE calculator:

```
from mace.calculators import mace_off
from ase import build

atoms = build.molecule('H2O')
calc = mace_off(model="medium", device='cuda')
atoms.set_calculator(calc)
print(atoms.get_potential_energy())
```

4.4.3 MACE-ANI-CC: Coupled cluster Accurate Pretrained Model for H, C, N, O elements

If you use the model please cite the paper.

The model can also be used simply as an ASE calculator:

```
from mace.calculators import mace_anicc
from ase import build

atoms = build.molecule('H2O')
calc = mace_anicc()
atoms.set_calculator(calc)
print(atoms.get_potential_energy())
```

CHAPTER

FIVE

SUPPORT

If you need help using `MACE`, please use GitHub Discussions or send an email to ilyes.batatia@ens-paris-saclay.fr