MACE Tutorial

Antonio Rivas Blanco

June 6, 2025

Contents

1 MACE installation

MACE requires `Python >= 3.7`, but it is recommended to use `Python >= 3.9 & Python < 3.13` and `PyTorch >= 2.1`, since versions `3.7 and 3.8` will soon be unsupported (`https://devguide.python.org/versions/`). On the other hand, as of June 6, 2025, `Python >= 3.13` has issues with the `cuequivariance=0.4.0` library, which is required for GPU support.

To avoid problems with library dependencies, it is recommended to create a `conda` or `mamba` environment.

Listing 1: Creating the environment.

```
conda create -n mace_env python=3.9 -y
conda activate mace_env
```

1.1 Installation from PyPI (recommended)

The easiest and recommended way to install MACE is via `pip` from the PyPI repository:

Listing 2: Installation from PyPI.

```
pip install --upgrade pip           # To upgrade pip
pip install mace-torch              # To install MACE
```

CAUTION! There is another package called `mace` on PyPI that is unrelated.

1.2 Installation from source code

It is also possible to install MACE from the source code hosted on GitHub:

Listing 3: Installation from source code.

```
git clone https://github.com/ACEsuit/mace.git
pip install ./mace
```

2 Input Generation

To train a model with MACE, the data must be in `.extxyz` format and include specific labels for total energy, forces, and *stresses*, which are the properties we will train. You can also train on virials and charges. Since we want to train or fine-tune pretrained models, we will use DFT data, which usually comes from VASP.

The VASP outputs we will use are `vasprun.xml` or `vasprun.h5`, which we will convert into our *dataset*. For this, we use the Python script `generate_train_files.py`, which can be found at `https://github.com/RivasAntonio/MACE_functions`. This script contains an adaptation of `atoms_utils.py` from `https://github.com/microsoft/mattersim`.

This function "needs" a directory that contains, either in itself or in its subdirectories, the `vasprun.xml` or `vasprun.h5` files. The arguments are as follows:

- `--data_path, -dv`: Path to the directory with the files.
- `--save_path, -sv`: Path where the outputs will be saved.
- `--train_ratio, -tr` $\in [0, 1]$. Percentage of data for the training set.

- `--validation_ratio, -vr` $\in [0, 1]$. Percentage of data for the validation set.
- `--shuffle` (optional) to shuffle the configurations.
- `--seed` (optional) for reproducibility.

The outputs of the function are the files `train.extxyz`, `validation.extxyz`, `test.extxyz`, and `summary.txt`. The last file shows the total number of configurations and how many have been assigned to each set. The other files have a general format like:

Listing 4: Example configuration with two atoms

```
2
Lattice="a b c d e f g h i" Properties=species:S:1:pos:R:3:REF_forces:R:3
    REF_energy=E_REF REF_stress="Sxx Sxy Sxz Syx Syy Syz Szx Szy Szz" pbc="T T T"
X    x1    y1    z1    fx1    fy1    fz1
X    x2    y2    z2    fx2    fy2    fz2
```

- `2`: Total number of atoms in the configuration.
- `Lattice`: Simulation cell matrix in row-major order: $\vec{a}_1$, $\vec{a}_2$, $\vec{a}_3$.
- `Properties`: Defines the per-atom properties available in each line of the atom block. In this case: atomic species, position (x, y, z), and reference forces (f_x, f_y, f_z).
- `REF_energy`[1]: Reference total energy of the configuration (e.g. obtained with DFT).
- `REF_stress`: Reference stress tensor (symmetric 3×3 matrix flattened by rows).
- `pbc`: Periodic boundary conditions in the x, y, and z directions.
- Each subsequent line contains information for one atom:
  - `X`: Chemical symbol of the atom.
  - `x, y, z`: Cartesian coordinates of the atom.
  - `fx, fy, fz`: Components of the reference force on that atom.

Once we have our dataset ready, it is a good idea to check it to see if all the structures are correct. In the repository `https://github.com/RivasAntonio/MACE_functions` you can find different functions for handling the dataset. Sometimes the input and/or output of the DFT calculation can be wrong and give an unrealistic or unstable configuration, which should not be used for training the model.

An example of these configurations are those with forces much higher than the average of the dataset. To remove these configurations, we have the function `filter_forces.py`, which removes configurations whose maximum force is greater than a threshold we set. To see if there are any erroneous configurations, we can use the ASE graphical interface with `ASE gui <*.extxyz>`, which allows us to quickly plot the maximum force for each configuration and see if there are any to discard. **Note:** Keep in mind that the internal units of MACE are $eV, \mathring{A}$.

We can change the units of the stresses with the function `convert_stress_units.py`. On the other hand, if you already have a complete dataset, you can split it into the three subsets with the function `split_xyz_files.py`. As we will see later, it is useful to know the atomic numbers present in your dataset; for this, you can use `get_elements_from_xyz.py`.

[1] **Note:** It is recommended to add the suffix REF_ to the labels of the properties to be trained to avoid issues with the current version of `ASE`.

3 Data Preprocessing

Before training a model with MACE, it is advisable to preprocess the *dataset*. This is done using the command `python <mace_repo_dir>/mace/cli/preprocess_data.py` or `mace_prepare_data`.

An example of its usage:

Listing 5: Standard preprocessing

```
mace_prepare_data \
 --train_file train.extxyz \
 --valid_file validation.extxyz \
 --test_file test.extxyz \
 --energy_key REF_energy \
 --forces_key REF_forces \
 --forces_key REF_stress \
 --atomic_numbers "[1,6,7,35,53,55,82]" \
 --E0s "{1:-3.667168021358939,6:-8.405573550273285,7:-8.405573550273285,35:␣
    -2.5184940099633986,53:-1.6355986842433357,55:-2.765284507132287,82:␣
    -3.730042357127322}" \
 --r_max=4.5 \
 --h5_prefix="processed_data/" \
 --compute_statistics \
 --seed=123 \
```

With:

- `--atomic_numbers="[1, 6, 7, 35, 53, 55, 82]"`
  List of atomic numbers present in the dataset.
- `--E0s="{1 : -3.6671..., 6 : -8.4055..., ...}"`
  Isolated Atom Energy (in eV) for each element. You can use averaged values, but it is highly recommended to perform *spin-polarized* DFT calculations with the same functional as your configurations to get this energy.
- `--r_max=4.5`
  Cutoff radius in Ångströms. The larger it is, the better your model will be, but it will also be slower.
- `--h5_prefix="processed_data/"`
  Output path where the preprocessed `.h5` files will be saved.
- `--compute_statistics`
  Indicates that global statistics of the dataset should be calculated (average number of neighbors, mean and standard deviation of energies). These are saved in the `statistics.json` file.
- `--seed=123`
  Sets a random seed to ensure that the data split (if performed) is reproducible.

After running the preprocessing command, a directory (`processed_data`) is created with the following structure:

Listing 6: Typical structure of the preprocessed data directory

```
processed_data/
|- statistics.json
|- train/
|  |- train_0.h5
|  |- train_1.h5
|  |- ...
|- val/
|  |- val_0.h5
|  |- val_1.h5
|  |- ...
|- test/
|  |- Default_Default_1.h5
|  |- Default_Default_2.h5
|  \- ...
```

To make handling the arguments easier, you can create a `.yaml` file with the arguments organized as follows:

```
train_file : "train.xyz"
valid_file : "validation.xyz"
test_file : "test.xyz"
h5_prefix : "processed_data_no_changed/"
compute_statistics : True
seed : 123
atomic_numbers : "[1, 6, 7, 35, 53, 55, 82]"
E0s : "{ 1 : -3.667168021358939, 6 : -8.405573550273285, 7 : -8.405573550273285, 35
    : -2.5184940099633986, 53 : -1.6355986842433357, 55 : -2.765284507132287, 82 :
    -3.730042357127322}"
r_max : 8
energy_key : "REF_energy"
forces_key : "REF_forces"
stress_key : "REF_stress"
```

For more options and details about preprocessing, it is recommended to run:

```
python <mace_repo_dir>/mace/cli/preprocess_data.py --help
```

or visit `https://github.com/ACEsuit/mace/blob/main/mace/cli/preprocess_data.py`.

4 Model Training

Once the data preprocessing is done, we can move on to training the model. MACE allows you to train models either from scratch or by fine-tuning pre-trained models. For both cases, you can use `python <mace_repo_dir>/mace/cli/run_train.py` or `mace_run_train`.

4.1 Training from scratch

To train a MACE model from scratch, run:

```
mace_run_train --config config.yaml
```

A typical example of this `config.yaml` file for training from scratch can be found in 7. Let's see what each argument is for:

  - `name`: Model name. Used to name output folders, checkpoints, logs, etc.

- **Input data**
  - `train_file`, `valid_file`: Paths to the training and validation sets in `.h5` format, generated after preprocessing.
  - `statistics_file`: JSON file with dataset statistics.
- **Property access**
  - `energy_key`, `forces_key`, `stress_key`: Dictionary labels that identify energy, forces, and stresses in the input files.
- **Output directories**
  - `checkpoints_dir`: Path where models are periodically saved during training.
  - `results_dir`: Path where validation results and metrics are stored.
  - `log_dir`: Path for training process log files.
  - `model_dir`: Path where the final trained model is saved.
- **Model hyperparameters**
  - `r_max`: Cutoff radius in Å.
  - `batch_size`: Batch size for training. Usually 32, but it depends on your GPU memory and how many atoms are in your configurations. MACE suggests following this heuristic relation
    $200000 = \text{max\_num\_epochs} \frac{\text{num\_configs}}{\text{batch\_size}}$ for good training.
  - `ema_decay`: Exponential Moving Average decay. Basically, the closer to 1, the slower the training but the better the results.
  - `max_L`: Maximum degree of spherical harmonics.
  - `num_channels`: Number of hidden channels in each model layer.
  - `num_radial_basis`: Number of radial functions used to represent distance.
  - `num_interactions`: Number of interaction blocks (model depth).
  - `lr`: Learning rate.

  - `energy_weight`, `forces_weight`, `stress_weight`: Relative weights for each term in the loss function.
  - `compute_stress`, `compute_forces`: Whether the model should predict stresses and forces in addition to energy.
  - `scaling`: Method to scale physical properties; `rms_forces_scaling` is the most common.

- **Training control**
  - `max_num_epochs`: Maximum number of training epochs.
  - `eval_interval`: Frequency (in epochs) to evaluate the model on the validation set.
  - `plot_frequency`: Frequency to save diagnostic plots.
  - `restart_latest`: If enabled, allows resuming training from the last checkpoint.
  - `seed`: Random seed for reproducibility.
  - `device`: Training device, usually `cuda` (GPU) or `cpu`.
  - `enable_cueq`: Enables acceleration for some operations using NVIDIA's `cuequivariance` package.

- **Hardware optimization**
  - `distributed`: Enables distributed training if you have multiple GPUs.
  - `num_workers`: Number of parallel processes for the `DataLoader`.
  - `pin_memory`: Improves efficiency when transferring data to the GPU.

- **Evaluation**
  - `error_table`: Defines how error metrics are calculated. For example, `PerAtomRMSEstressvirials` includes per-atom RMSE for energy, forces, and stresses.
  - `loss`: Type of loss function used; `universal` is a general option valid for multiple tasks.

Listing 7: YAML configuration file for training from scratch

```
# ====================== EXPERIMENT NAME ======================
name : 'mace-training-from-scratch'

# ====================== INPUT DATA ======================
train_file: './inputs/processed_data/train'
valid_file: './inputs/processed_data/val'
statistics_file : "./inputs/processed_data/statistics.json"

# ====================== PROPERTY ACCESS ======================
energy_key: 'REF_energy'
forces_key : 'REF_forces'
stress_key : 'REF_stress'

# ====================== OUTPUT DIRECTORIES ======================
checkpoints_dir : './outputs/checkpoints'
results_dir : './outputs/results'
log_dir : './outputs/logs'
model_dir : './outputs/models'

# ====================== MODEL HYPERPARAMETERS ======================
r_max : 7.0
batch_size : 32
ema_decay : 0.999

max_L : 3
max_ell : 3
num_channels : 128
num_radial_basis : 10
num_interactions : 3

lr : 0.001
energy_weight : 1.0
forces_weight : 100.0
stress_weight : 0.1

compute_stress : True
compute_forces : True
scaling : 'rms_forces_scaling'

# ====================== TRAINING CONTROL ======================
max_num_epochs : 15
eval_interval : 1
plot_frequency : 5
restart_latest : True
seed : 123
device : 'cuda'
enable_cueq : True

# ====================== MULTI-GPU AND CPU OPTIMIZATION ======================
distributed : True
num_workers : 16
pin_memory : True

# ====================== EVALUATION OPTIONS ======================
error_table : "PerAtomRMSEstressvirials"
loss : "universal"
```

5 Fine-tuning

Once a model has been trained in MACE, it is possible to fine-tune it to improve its performance on a more specific dataset. This procedure is useful when you have a general model trained on high-quality data (for example, from a large set of DFT structures) and you want to specialize it for a particular task.

In MACE, there are two main approaches for fine-tuning:

- **Naive fine-tuning**: This consists of simply continuing the training of the base model using the new dataset, without any additional mechanisms to protect the previously learned knowledge. It is easy to implement, but can lead to so-called *catastrophic forgetting*, where the model loses some of the information it had previously learned.
- **Multi-head fine-tuning**: In this approach, the base model is frozen and one or more heads (additional neural networks) are trained, specialized on the new dataset. This way, the previously learned knowledge is preserved while the model adapts to new tasks. This technique also allows you to maintain performance on the original data if you use several heads at the same time.

The following sections describe both approaches in detail, starting with the simplest method: naive fine-tuning. You just need to add the following arguments to your `config.yaml` file:

```
foundation_model : <>                # Path to the model to fine-tune
multiheads_finetuning : False        # Do not activate multiheads
```

The models can be either user-trained or those provided by ACESuit, which are available at `https://github.com/ACEsuit/mace-foundations/tree/main`. This argument also accepts the entries "small", "medium", and "large", which will download one of the first 3 models trained with MACE.

6 Multihead Fine-tuning

This is a MACE technique that allows you to adjust a base model simultaneously on a target dataset and a *replay* dataset used to train the model. This helps prevent *catastrophic forgetting* by maintaining the generality of the model while adapting it to a specific use case.

6.1 Replay dataset options

Before starting the fine-tuning process, you need to get a replay dataset. These are available at:

`https://github.com/ACEsuit/mace-foundations/releases`

There are two types of replay datasets available:

- **Mode 1: Original training data with true DFT labels**
  - Contains real DFT calculations used to train the base model, so it is usually good for ground-state energy and force calculations.
- **Mode 2: Diverse configurations evaluated with the base model**
  - Contains configurations of various materials evaluated using the base model.

- The initial metrics for forces and energy will show zeros (since the predictions match exactly the labels).
- This mode focuses on preserving the existing behavior of the model rather than the accuracy obtained in the previous method.

6.2 Method 1: Preprocessing via Command Line

The first approach uses the command-line tool `fine_tuning_select.py` to prepare your replay dataset before training.

Use the following command to select configurations from the replay dataset based on your target dataset:

Listing 8: Selecting configurations for the replay dataset

```
python -m <mace_repo>/mace.cli.fine_tuning_select \
  --configs_pt path/to/replay_dataset.xyz \
  --configs_ft path/to/your_dataset.xyz \
  --num_samples 10000 \
  --subselect fps \
  --model path/to/foundation_model.model \
  --output selected_configs.xyz \
  --filtering_type combinations \
  --head_pt pt_head \
  --head_ft target_head \
  --weight_pt 1.0 \
  --weight_ft 10.0
```

Key parameters:

- `--configs_pt`: Path to the replay dataset.
- `--configs_ft`: Path to your target dataset.
- `--num_samples`: Number of configurations to select from the replay dataset.
- `--subselect`: Method for subselection (`fps` for Farthest Point Sampling or `random`).
- `--filtering_type`: How to filter configurations based on elements:
  - `combinations`: Keep configurations if they only contain elements with the provided atomic numbers, but not necessarily all of them.
  - `exclusive`: Keep configurations that contain only elements with the provided atomic numbers.
  - `inclusive`: Keep configurations if they contain all the provided atomic numbers (but may have additional elements).
  - `none`: No filtering.
- `--atomic_numbers`: Specify atomic numbers for filtering.

After creating the combined dataset, to use it for training, run:

Listing 9: Training with the combined dataset

```
python -m mace.cli.run_train \
  --name mymodel_finetuned \
  --pt_train_file selected_configs.xyz \
  --train_file path/to/your_dataset.xyz \
  --valid_fraction 0.05 \
  --foundation_model path/to/foundation_model.model \
  --energy_weight 1.0 \
  --forces_weight 100.0 \
```

6.3 Method 2: MP Shortcut for Compatible Models

Just like with naive finetuning, MACE allows you to download pretrained models using its arguments. These are trained using datasets from the Materials Project (MP) database or other DFT-compatible databases. This method simplifies the process by allowing you to automatically download a suitable replay dataset by using the special value `mp` for the `--pt_train_file` parameter.

Listing 10: Multihead fine-tuning using the MP shortcut

```
python -m mace.cli.run_train \
  --train_file path/to/your_dataset.xyz \
  --foundation_model medium \
  --pt_train_file mp \
  --atomic_numbers "[1,6,7,8]" \
  --multiheads_finetuning True
```

Compatible base models:

- `MACE-MP-0` (`small`, `medium`, `large`)
- `MACE-MP-0b`, `MACE-MP-0b2`, `MACE-MP-0b3`
- `MACE-MPA-0`

6.4 Method 3: Direct Multihead Fine-tuning with `run_train`

The third method integrates the selection of the replay dataset directly into the training process using the `run_train.py` script.

Listing 11: Direct multihead fine-tuning with `run_train`

```
python -m mace.cli.run_train \
  --name mymodel_finetuned \
  --train_file path/to/your_dataset.xyz \
  --foundation_model path/to/foundation_model.model \
  --pt_train_file path/to/replay_dataset.xyz \
  --num_samples_pt 10000 \
  --filter_type_pt combinations \
  --subselect_pt fps \
  --weight_pt 1.0 \
  --atomic_numbers "[1,6,7,8]" \
  --multiheads_finetuning True \
  --force_mh_ft_lr False
```

Key parameters:

- `--train_file`: Path to the dataset for fine-tuning.
- `--foundation_model`: Path to the base model.
- `--pt_train_file`: Path to the replay dataset (you can also use `mp` for Materials Project data).
- `--num_samples_pt`: Number of samples to use from the replay dataset.
- `--filter_type_pt`: Filtering strategy for configurations (`combinations`, `exclusive`, `inclusive`, `none`).
- `--subselect_pt`: Subselection method (`fps` or `random`).
- `--weight_pt`: Loss weight for the pretraining head.
- `--atomic_numbers`: Atomic numbers to keep in the replay dataset.
- `--multiheads_finetuning`: Enable multi-head fine-tuning.
- `--force_mh_ft_lr`: Allows changing the learning parameters of the pretrained model when set to `True`.

This method gives you more flexibility and control over the fine-tuning process, letting you adjust different parameters according to your dataset and model needs.

6.5 Tips for successful multi-head fine-tuning

- Make sure the atomic numbers in your target dataset are present in the replay dataset.
- Adjust the head weights (`--weight_pt` and `--weight_ft`) to properly balance the influence of the replay and target datasets.

Note: For more information about the parameters and how they work, check `https://github.com/ACEsuit/mace` or read Batatia, I., Kovacs, D. P., Simm, G., Ortner, C., & Csányi, G. (2022). MACE: Higher order equivariant message passing neural networks for fast and accurate force fields. *Advances in neural information processing systems,* 35, 11423-11436. Or Batatia, I., Batzner, S., Kovács, D. P., Musaelian, A., Simm, G. N., Drautz, R., ... & Csányi, G. (2022). The design space of e (3)-equivariant atom-centered interatomic potentials. *arXiv preprint* arXiv:2205.06643.

7 Model Evaluation

Once the model is trained, we can evaluate it, for example, on the test set using the commands `python3 <mace_repo_dir>/mace/cli/eval_configs.py` or `mace_eval_configs`. Here's an example:

Listing 12: Model evaluation

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

This will generate a file with the following format:

Listing 13: Example configuration with MACE predictions

```
2
Lattice="a b c d e f g h i" Properties=species:S:1:pos:R:3:REF_forces:R:3:
    MACE_forces:R:3 \
REF_energy=E_REF REF_stress="Sxx Sxy Sxz Syx Syy Syz Szx Szy Szz" \
MACE_energy=E_MACE MACE_stress="_JSON [[Sxx', Sxy', Sxz'], [Syx', Syy', Syz'], [Szx
    ', Szy', Szz']]" pbc="T T T"
X   x1   y1   z1   fx1_REF   fy1_REF   fz1_REF   fx1_MACE   fy1_MACE   fz1_MACE
X   x2   y2   z2   fx2_REF   fy2_REF   fz2_REF   fx2_MACE   fy2_MACE   fz2_MACE
```

To get metrics from this evaluation and see how well the model fits, you can use the `plot_pred.py` script from `https://github.com/RivasAntonio/MACE_functions`.

Appendix: Using MACE Models

You can use MACE models with both ASE and LAMMPS. For ASE, MACE provides a calculator compatible with the Atomic Simulation Environment.

Here's an example of how to run an MD simulation using a MACE model:

Listing 14: MD simulation with MACE and ASE

```
from ase import units
from ase.md.langevin import Langevin
from ase.io import read
from mace.calculators import MACECalculator

# Initialize the MACE calculator with the trained model
calculator = MACECalculator(model_path='path/to/model/MACE_model.model', device='
    cuda')

# Read the initial configuration from a .xyz file
init_conf = read('path/to/file/test_300K.xyz', index=0)
init_conf.set_calculator(calculator)

# Set up Langevin dynamics for an NVT ensemble
dyn = Langevin(init_conf, timestep=0.5 * units.fs, temperature_K=310, friction=5e
    -3)

# Function to save simulation frames
def write_frame():
  dyn.atoms.write('md_3bpa.xyz', append=True)

# Attach the writing function and run the simulation
dyn.attach(write_frame, interval=50)
dyn.run(100)

print("Simulation␣finished!")
```

- `MACECalculator`: Initializes the MACE calculator with the trained model specified in `model_path`. The `device` parameter sets the device to use (e.g., `'cuda'` for GPU).
- `read`: Reads the initial atomic configuration from a `.xyz` file.
- `set_calculator`: Assigns the MACE calculator to the atomic configuration.
- `Langevin`: Sets up Langevin dynamics to simulate an NVT ensemble, specifying the timestep, temperature, and friction coefficient.
- `write_frame`: Function that saves the current atomic configuration to a `.xyz` file every certain number of steps.
- `attach`: Attaches the writing function to the dynamics object so it runs at defined intervals.
- `run`: Runs the molecular dynamics simulation for the specified number of steps.

To use a MACE model as a potential in LAMMPS, you first need to convert it to a LAMMPS-compatible format. You can do this with `python <mace_repo_dir>/mace/cli/create_lammps_model.py` or `mace_create_lammps_model` as follows:

```
python <mace_repo_dir>/mace/cli/create_lammps_model.py my_mace.model --format=
    mliap
```

You can skip the format option, but then it will convert to another LAMMPS version with lower performance. Once you have the potential, you need to compile LAMMPS with this version, as explained on the MACE website. If you're not familiar with compiling[2], you can follow this tutorial.

Below is a complete example of how to use a MACE model with ML-IAP in LAMMPS:

Listing 15: LAMMPS script using MACE with ML-IAP

```
# MACE ML-IAP example
units          metal
atom_style     atomic
atom_modify    map yes
newton         on

# Read structure
read_data      structure.data

# Set up MACE potential
pair_style     mliap unified model-mliap_lammps.pt 0
pair_coeff     * * C H O N

# Run settings
timestep       0.0001
thermo         100

# MD run
fix            1 all nvt temp 300 300 100
run            1000
```

You can run this script with the following command:

```
lmp -k on g 1 -sf kk -pk kokkos newton on neigh half -in input.in
```

[2]Be careful with the `cmake.sh` file, since depending on your cluster's GPU you'll need to change the Kokkos_ARCH_ option. You can check your GPU type by running `nvidia-smi` in an interactive job.