Tutorial de MACE: Entrenamiento de Potenciales Atómicos

Nombre del autor

June 6, 2025

Contents

1 Instalación de MACE

MACE requiere `Python >= 3.7`, pero se recomienda usar `Python >= 3.9 & Python < 3.13` y `PyTorch >= 2.1`, ya que las versiones `3.7 y 3.8` dejarán de tener soporte pronto (`https://devguide.python.org/versions/`), por otro lado, las versiones de `Python >= 3.13` a fecha de 6 de Junio de 2025 tienen problemas con la biblioteca `cuequivariance=0.4.0`, necesaria para correr en GPU.

Para evitar problemas con las dependencias entre bibliotecas es recomendable crear un entorno de `conda` o `mamba`.

Listing 1: Creación de enviroment.

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

1.1 Instalación desde PyPI (recomendada)

La forma más sencilla y recomendada de instalar MACE es mediante `pip` desde el repositorio de PyPI:

Listing 2: Instalación desde PyPI.

```
pip install --upgrade pip           # Para actualizar pip
pip install mace-torch              # Para instalar MACE
```

CUIDADO! Existe otro paquete llamado `mace` en PyPI que no tiene nada que ver.

1.2 Instalación desde el código fuente

También es posible instalar MACE desde el código fuente alojado en GitHub:

Listing 3: Instalación desde código fuente.

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

2 Generación de Inputs

Para entrenar un modelo con MACE, los datos deben estar en formato `.extxyz` e incluir etiquetas específicas para la energía total, las fuerzas y los *stresses*, que son las propiedades que nosotros entrenaremos, también se pueden entrenar viriales y cargas. Puesto que queremos entrenar o hacer un *finetuning* de modelos preentrenados, utilizaremos datos DFT, que generalmente vendrán de VASP.

Los *outputs* de VASP que utilizaremos son los `vasprun.xml` o `vasprun.h5`, que convertiremos en nuestro *dataset*. Para ello, usaremos la función de *Python* `generate_train_files.py` que se puede encontrar en `https://github.com/RivasAntonio/MACE_functions`. Esta función contiene una adaptación de la de `atoms_utils.py` de `https://github.com/microsoft/mattersim`.

Esta función "necesita" un directorio que contenga en este mismo o en sus subdirectorios los archivos `vasprun.xml` o `vasprun.h5`. Los argumentos son los siguientes:

- `--data_path, -dv`: *Path* al directorio con los archivos.
- `--save_path, -sv`: *Path* donde se guardaran los outputs.

- `--train_ratio, -tr` $\in [0, 1]$. Porcentaje de datos del *set* de entrenamiento.
- `--validation_ratio, -vr` $\in [0, 1]$. Porcentaje de datos del *set* de validación.
- `--shuffle` (opcional) para mezclar las configuraciones.
- `--seed` (opcional) para la reproducibilidad.

El/Los outputs de la función son los archivos `train.extxyz validation.extxyz test.extxyz` y `summary.txt`. El último archivo nos muestra el número total de configuraciones y cuántas han sido asignadas a cada *set*. Por otro lado, los otros archivos tiene un formato general de la forma:

Listing 4: Ejemplo de configuración con dos átomos

```
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`: Número total de átomos en la configuración.
- `Lattice`: Matriz de la celda de simulación en orden fila-mayor: $\vec{a}_1$, $\vec{a}_2$, $\vec{a}_3$.
- `Properties`: Define las propiedades por átomo disponibles en cada línea del bloque de átomos. En este caso: especie atómica, posición (x, y, z) y fuerzas de referencia (f_x, f_y, f_z).
- `REF_energy`[1]: Energía total de referencia de la configuración (por ejemplo, obtenida con DFT).
- `REF_stress`: Tensor de tensiones de referencia (orden simétrico 3×3 aplanado por filas).
- `pbc`: Condiciones periódicas en las direcciones x, y y z.
- Cada línea posterior contiene información para un átomo:
  - `X`: Símbolo químico del átomo.
  - `x, y, z`: Coordenadas cartesianas del átomo.
  - `fx, fy, fz`: Componentes de la fuerza de referencia sobre ese átomo.

Una vez tenemos nuestro *dataset* preparado, conviene revisarlo para ver si todas las estructuras son correctas. En el repositorio `https://github.com/RivasAntonio/MACE_functions` encontramos diferentes funciones para el manejo del conjunto de datos. En ocasiones el input y/o el output del cálculo DFT puede ser erróneo y dar una configuración irreal o inestable, con las cuales no conviene entrenar el modelo.

Un ejemplo de estas configuraciones son las que tienen fuerzas excesivamente mayores a la media del conjunto de datos. Para eliminar estas configuraciones tenemos la función `filter_forces.py`, que elimina las configuraciones cuya fuerza máxima sea mayor que un umbral que establecemos. Para ver si hay alguna configuración errónea podemos ayudarnos de la interfaz gráfica de `ASE` con `ASE gui <*.extxyz>`. Esta nos permite graficar rápidamente la fuerza máxima para cada configuración y ver si hay configuraciones que descartar. **Nota:** Debemos tener en cuenta que las unidades internas de MACE son $eV, \mathring{A}$

[1] **Nota:** a las etiquetas de las propiedades a entrenar se les recomienda poner el sufijo REF_ para evitar problemas con la versión actual de `ASE`.

Podemos cambiar las unidades de los *stresses* con la función `convert_stress_units.py`. Por otro lado, en el caso de tener ya un conjunto de datos completo, lo podemos dividir en los tres subconjuntos con la función `split_xyz_files.py`. Como veremos posteriormente, es interesante saber los números atómicos de nuestro conjunto de datos, para ello podemos utilizar `get_elements_from_xyz.py`.

3 Preprocesamiento de Datos

Antes de entrenar un modelo con MACE, es conveniente hacer un preprocesamiento del *dataset*. Esto se realiza mediante el comando `python <mace_repo_dir>/mace/cli/preprocess_data.py` o `mace_prepare_data`.

Un ejemplo de su uso:

Listing 5: Preprocesamiento estándar

```
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 \
```

Con:

- `--atomic_numbers="[1, 6, 7, 35, 53, 55, 82]"`
  Lista de números atómicos presentes en el conjunto de datos.
- `--E0s="{1 : -3.6671..., 6 : -8.4055..., ...}"`
  Isolated Atom Energy (en eV) para cada elemento. Se pueden promediar, pero es muy recomendable hacer cálculos *spin-polarized* DFT con el mismo funcional que las configuraciones para calcular esta energía.
- `--r_max=4.5`
  Radio de corte en Ångströms. Cuanto mayor sea, mejor será nuestro modelo, pero será más lento.
- `--h5_prefix="processed_data/"`
  Ruta de salida en la que se guardarán los archivos `.h5` preprocesados.
- `--compute_statistics`
  Indica que deben calcularse estadísticas globales del conjunto (número medio de vecinos, media y desviación estándar de energías). Estas se guardan en el archivo `statistics.json`.
- `--seed=123`
  Establece una semilla aleatoria para asegurar que el particionado de los datos (en caso de que se realice) sea reproducible.

Tras ejecutar el comando de preprocesamiento, se crea un directorio (`processed_data`) con la siguiente estructura:

Listing 6: Estructura típica del directorio de datos preprocesados

```
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
|  \- ...
```

Para un manejo más fácil de los argumentos, podemos crear un archivo `.yaml` con los argumentos organizados de la siguiente forma:

```
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"
```

Para más opciones y detalles sobre el preprocesamiento, se recomienda ejecutar:

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

o visitar `https://github.com/ACEsuit/mace/blob/main/mace/cli/preprocess_data.py`.

4 Entrenamiento del Modelo

Una vez hecho el preprocesamiento de los datos, podemos pasar al entrenamiento del modelo. MACE permite entrenar modelos tanto desde cero como hacer *fine-tuning*de modelos entrenados. Para ambos casos podemos utilizar `python <mace_repo_dir>/mace/cli/run_train.py` o `mace_run_train`.

4.1 Entrenamiento desde cero

Para entrenar un modelo MACE desde cero, ejecutamos:

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

Un ejemplo típico de este archivo `config.yaml` para entrenar desde 0 puede ser el que encontramos en 7. Veamos para qué sirve cada argumento:

- `name`: Nombre del modelo. Se utiliza para nombrar carpetas de salida, checkpoints, logs, etc.

- **Datos de entrada**
  - `train_file`, `valid_file`: Rutas a los conjuntos de entrenamiento y validación en formato `.h5`, generados tras el preprocesado.
  - `statistics_file`: Archivo JSON con estadísticas del *dataset*.

- **Acceso a propiedades**
  - `energy_key`, `forces_key`, `stress_key`: Etiquetas del diccionario que identifican la energía, las fuerzas y las tensiones en los archivos de entrada.

- **Directorios de salida**
  - `checkpoints_dir`: Ruta donde se guardan losmodelos periódicamente durante el entrenamiento.
  - `results_dir`: Ruta donde se almacenan resultados de validación y métricas.
  - `log_dir`: Ruta para los archivos de *logging* del proceso de entrenamiento.
  - `model_dir`: Ruta donde se guarda el modelo final entrenado.

- **Hiperparámetros del modelo**
  - `r_max`: Radio de corte en Å.
  - `batch_size`: Tamaño del lote para el entrenamiento. Generalmente suele ser 32, pero depende de la memoria de la GPU que utilicemos y de cuántos átomos tengan las configuraciones. MACE sugiere que se cumpla la siguiente relación heurística $200000 = \text{max\_num\_epochs}\frac{\text{num\_configs}}{\text{batch\_size}}$ para un buen entrenamiento.
  - `ema_decay`: *Exponential Moving Average decay*. Básicamente, cuanto más cerca de 1 esté más lento será el entrenamiento pero mejor será
  - `max_L`: Grado máximo de los armónicos esféricos.
  - `num_channels`: Número de canales ocultos en cada capa del modelo.
  - `num_radial_basis`: Número de funciones radiales utilizadas para representar la distancia.

  - `num_interactions`: Número de bloques de interacción (profundidad del modelo).
  - `lr`: Tasa de aprendizaje.
  - `energy_weight`, `forces_weight`, `stress_weight`: Pesos relativos de cada término en la función de pérdida.
  - `compute_stress`, `compute_forces`: Determinan si el modelo debe predecir tensiones y fuerzas además de la energía.
  - `scaling`: Método para escalar las propiedades físicas; `rms_forces_scaling` es el más común.

- **Control del entrenamiento**
  - `max_num_epochs`: Número máximo de *epochs* del entrenamiento.
  - `eval_interval`: Frecuencia (en *epochs*) con la que se evalúa el modelo en el conjunto de validación.
  - `plot_frequency`: Frecuencia para guardar gráficas de diagnóstico.
  - `restart_latest`: Si está activado, permite reanudar el entrenamiento desde el último checkpoint.
  - `seed`: Semilla aleatoria para asegurar reproducibilidad.
  - `device`: Dispositivo de entrenamiento, normalmente `cuda` (GPU) o `cpu`.
  - `enable_cueq`: Permite acelerar algunas operaciones gracias al paquete `cuequivariance` de NVIDIA.

- **Optimización para hardware**
  - `distributed`: Permite entrenamiento distribuido si se dispone de múltiples GPUs.
  - `num_workers`: Número de procesos paralelos para el `DataLoader`.
  - `pin_memory`: Mejora la eficiencia al transferir datos a la GPU.

- **Evaluación**
  - `error_table`: Define cómo se calculan las métricas de error. Por ejemplo, `PerAtomRMSEstressvirials` incluye RMSE por átomo en energía, fuerzas y tensiones.
  - `loss`: Tipo de función de pérdida utilizada; `universal` es una opción general válida para múltiples tareas.

Listing 7: Archivo de configuración YAML para entrenamiento desde cero

```
# ====================== NOMBRE DEL EXPERIMENTO ======================
name : 'mace-entrenamiento-desde-cero'

# ====================== DATOS DE ENTRADA ======================
train_file: './inputs/processed_data/train'
valid_file: './inputs/processed_data/val'
statistics_file : "./inputs/processed_data/statistics.json"

# ====================== ACCESO A PROPIEDADES ======================
energy_key: 'REF_energy'
forces_key : 'REF_forces'
stress_key : 'REF_stress'

# ====================== DIRECTORIOS DE SALIDA ======================
checkpoints_dir : './outputs/checkpoints'
results_dir : './outputs/results'
log_dir : './outputs/logs'
model_dir : './outputs/models'

# ====================== HIPERPARAMETROS DEL MODELO ======================
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'

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

# ====================== OPTIMIZACION MULTI-GPU Y CPU ======================
distributed : True
num_workers : 16
pin_memory : True

# ====================== OPCIONES DE EVALUACION ======================
error_table : "PerAtomRMSEstressvirials"
loss : "universal"
```

5 Fine-tuning

Una vez entrenado un modelo en MACE, es posible afinarlo, es decir hacerle *fine-tuning* para mejorar su rendimiento en un conjunto de datos más específico. Este procedimiento resulta útil cuando se dispone de un modelo general entrenado con datos de alta calidad (por ejemplo, de un gran conjunto de estructuras DFT) y se desea especializarlo para una tarea particular

En MACE existen dos enfoques principales para realizar el *fine-tuning*:

- **Fine-tuning** ingenuo *(naive fine-tuning)*: consiste en continuar el entrenamiento del modelo base usando el nuevo conjunto de datos, sin mecanismos adicionales de protección del conocimiento previo. Es sencillo de implementar, pero puede llevar al llamado *catastrophic forgetting*, donde el modelo pierde parte de la información aprendida previamente.
- **Fine-tuning con múltiples cabezas (multi-head fine-tuning)**: en este enfoque, se congela el modelo base y se entrena una o más cabezas (redes adicionales) especializadas en el nuevo conjunto de datos. De este modo, se preserva el conocimiento aprendido previamente mientras se adapta el modelo a nuevas tareas. Esta técnica también permite mantener el rendimiento sobre los datos originales si se usan varias cabezas de forma simultánea.

En las siguientes secciones se describen en detalle ambos enfoques, comenzando por el método más sencillo: el *fine-tuning ingenuo*. Simplemente tenemos que añadir los siguientes argumentos al archivo `config.yaml`

```
foundation_model : <>                # Ruta al modelo al que hacerle fine-tuning
multiheads_finetuning : False        # Para no activar el multiheads
```

Los modelos pueden ser o bien entrenados por el usuario o los proporcionados por ACE-Suit, que se encuentran `https://github.com/ACEsuit/mace-foundations/tree/main`. Este argumento tambień acepta las entradas "small", "medium" y "large", que descarga uno de los 3 primeros modelos entrenados con MACE.

6 Multihead Fine-tuning

Esta una técnica de MACE que permite ajustar un modelo base simultáneamente sobre un conjunto de datos objetivo y un conjunto de datos de *replay* con el que se entrenó el modelo. Esto ayuda a prevenir el *catastrophic forgetting* manteniendo las generalidades del modelo mientras se adapta a un caso de uso específico.

6.1 Opciones de conjuntos de datos de *replay*

Antes de comenzar el proceso de *fine-tuning*, es necesario obtener un conjunto de datos de *replay*. Estos están disponibles en:

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

Existen dos tipos de conjuntos de datos de *replay* disponibles:

- **Modo 1: Datos de entrenamiento originales con etiquetas DFT verdaderas**
  - Contiene cálculos DFT reales utilizados para entrenar el modelo base, por lo que suele ser bueno para cálculos de de energías y fuerzas en el estado fundamental.
- **Modo 2: Configuraciones diversas evaluadas con el modelo base**

- Contiene configuraciones de varios materiales evaluadas utilizando el modelo base.
- Las métricas iniciales sobre fuerzas y energía mostrarán ceros (ya que las predicciones coinciden exactamente con las etiquetas).
- Se centra en preservar el comportamiento existente del modelo en lugar de la precisión obtenida en el anterior método.

6.2 Método 1: Preprocesamiento mediante la línea de comandos

El primer enfoque utiliza la herramienta de línea de comandos `fine_tuning_select.py` para preparar su conjunto de datos de *replay* antes del entrenamiento.

Utilice el siguiente comando para seleccionar configuraciones del conjunto de datos de *replay* basadas en su conjunto de datos objetivo:

Listing 8: Selección de configuraciones para el conjunto de datos de *replay*

```
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
```

Parámetros clave:

- `--configs_pt`: Ruta al conjunto de datos de *replay*.
- `--configs_ft`: Ruta a su conjunto de datos objetivo.
- `--num_samples`: Número de configuraciones a seleccionar del conjunto de datos de *replay*.
- `--subselect`: Método para la subselección (`fps` para Farthest Point Sampling o `random`).
- `--filtering_type`: Cómo filtrar configuraciones basadas en elementos:
  - `combinations`: Mantener configuraciones si estas solo contiene elementos con los números atómicos que le proporcionamos, pero no necesariamente todos.
  - `exclusive`: Mantener configuraciones que contienen solo elementos con los números atómicos que le proporcionamos.
  - `inclusive`: Mantener configuraciones si estas contienen todos los elementos de los números atómicos proporcionados (pero puede tener otros adicionales).
  - `none`: Sin filtrado.
- `--atomic_numbers`: Especifica números atómicos para filtrar.

Después de crear el conjunto de datos combinado,Para utilizarlo en el entrenamiento se debe ejecutar:

Listing 9: Entrenamiento con el conjunto de datos combinado

```
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 Método 2: Atajo MP para modelos compatibles

De igual forma que para el *naive finetuning*, MACE permite la opción de descargar modelos preeentrenados mediante sus argumentos. Estos están entrenados utilizando conjuntos de datos en la base de datos Materials Project (MP) o bases de datos con DFT compatibles. Este método simplifica el proceso al permitir la descarga automática de un conjunto de datos de *replay* adecuado mediante el uso del valor especial `mp` para el parámetro `--pt_train_file`.

Listing 10: Fine-tuning multihead utilizando el atajo MP

```
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
```

Modelos base compatibles:

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

6.4 Método 3: Fine-tuning multihead directo con `run_train`

El tercer método integra la selección del conjunto de datos de *replay* directamente en el proceso de entrenamiento utilizando el script `run_train.py`.

Listing 11: Fine-tuning multihead directo con `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
```

Parámetros clave:

- `--train_file`: Ruta al conjunto de datos para hacer el *finetuning*.
- `--foundation_model`: Ruta al modelo base.
- `--pt_train_file`: Ruta al conjunto de datos de *replay* (también se puede usar `mp` para datos del Materials Project).
- `--num_samples_pt`: Número de muestras a utilizar del conjunto de datos de *replay*.
- `--filter_type_pt`: Estrategia de filtrado para las configuraciones (`combinations`, `exclusive`, `inclusive`, `none`).
- `--subselect_pt`: Método de subselección (`fps` o `random`).
- `--weight_pt`: Peso para la pérdida de la cabeza de preentrenamiento.
- `--atomic_numbers`: Números atómicos a mantener en el conjunto de datos de *replay*.
- `--multiheads_finetuning`: Activar el *fine-tuning* con múltiples cabezas.
- `--force_mh_ft_lr`: Permite cambiar los parámetros de aprendizaje del modelo preentrenado cuando es `True`.

Este método proporciona una mayor flexibilidad y control sobre el proceso de *fine-tuning*, permitiendo ajustar diversos parámetros según las necesidades específicas del conjunto de datos y del modelo.

6.5 Consejos para un *fine-tuning* exitoso con múltiples cabezas

- Asegúrese de que los números atómicos en su conjunto de datos objetivo estén presentes en el conjunto de datos de *replay*.
- Ajuste los pesos de las cabezas (`--weight_pt` y `--weight_ft`) para equilibrar adecuadamente la influencia de los conjuntos de datos de *replay* y objetivo.

Nota: para más información sobre los parámetros y su funcionamiento, puede acudir a `https://github.com/ACEsuit/mace` o leer 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. O 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 Evaluación del Modelo

Una vez entrenado el modelo, podemos evaluarlo sobre por ejemplo el conjunto test con los comandos `python3 <mace_repo_dir>/mace/cli/eval_configs.py` o `mace_eval_configs`. Un ejemplo de su funcionamiento

Listing 12: Evaluación del modelo

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

Esto devolverá un archivo cuya forma es:

Listing 13: Ejemplo de configuración con predicciones de MACE

```
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
```

Para obtener métricas de esta evaluación y ver qué tan bien ajusta el modelo, se puede usar la función `plot_pred.py` de `https://github.com/RivasAntonio/MACE_functions`.

Anexo: Uso de modelos de MACE

Podremos usar los modelos con ASE y con LAMMPS. Para usarlo con el primero, nos aprovechamos de que MACE proporciona una calculadora compatible con ASE (*Atomic Simulation Environment*).

Un ejemplo de cómo ejecutar una simulación de MD utilizando un modelo MACE.

Listing 14: Simulación de MD con MACE y ASE

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

#Inicializar la calculadora de MACE con el modelo entrenado
calculator = MACECalculator(model_path='ruta/al/modelo/MACE_modelo.model', device='
    cuda')

# Leer la configuracion inicial desde un archivo .xyz
init_conf = read('ruta/al/archivo/test_300K.xyz', index=0)
init_conf.set_calculator(calculator)

# Configurar la dinamica de Langevin para un ensemble NVT
dyn = Langevin(init_conf, timestep=0.5 * units.fs, temperature_K=310, friction=5e
    -3)

# Funcion para guardar los frames de la simulacion
def write_frame():
    dyn.atoms.write('md_3bpa.xyz', append=True)

# Adjuntar la funcion de escritura y ejecutar la simulacion
dyn.attach(write_frame, interval=50)
dyn.run(100)

print("Simulacion-finalizada!)
```

- `MACECalculator`: Inicializa el calculador de MACE con el modelo entrenado especificado en `model_path`. El parámetro `device` indica el dispositivo a utilizar (por ejemplo, `'cuda'` para GPU).
- `read`: Lee la configuración inicial de los átomos desde un archivo `.xyz`.
- `set_calculator`: Asigna el calculador MACE a la configuración atómica.
- `Langevin`: Configura la dinámica de Langevin para simular un ensamble NVT, especificando el paso de tiempo, la temperatura y el coeficiente de fricción.
- `write_frame`: Función que guarda la configuración actual de los átomos en un archivo `.xyz` cada cierto número de pasos.
- `attach`: Adjunta la función de escritura al objeto de dinámica para que se ejecute en intervalos definidos.
- `run`: Ejecuta la simulación de dinámica molecular durante el número de pasos especificado.

Para usar un modelo de MACE como un potencial en LAMMPS, lo primero que tenemos que hacer es convertirlo a uno compatible con LAMMPS, para ello usaremos `python <mace_repo_dir>/mace/cli/creat` o `mace_create_lammps_model` como sigue:

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

Podemos no añadir el formato, lo que haría que lo convirtiera a otra versión de LAMMPS con menor rendimiento que esta. Una vez tengamos el potencial, debemos compilar LAMMPS con esta versión de LAMMPS, como aparece en la web de MACE. Si no estás familiarizado con compilaciones[2]., puedes seguir este tutorial:

A continuación, se muestra un ejemplo completo de cómo usar un modelo MACE con ML-IAP en LAMMPS:

[2]Hay que tener cuidado con el archivo `cmake.sh` ya que dependiendo de la GPU del cluster hay que variar la opción Kokkos_ARCH_. Puedes ver qué tipo de GPU tienes ejecutando desde un trabajo interactivo `nvidiasmi`

Listing 15: Script de LAMMPS usando MACE con 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
```

Este script puede ejecutarse con el siguiente comando:

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