Computer Physics Communications 292 (2023) 108883

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computer Programs in Physics

Atom-centered machine-learning force field package ☆,☆☆

Lei Li [a,*,1], Ryan A. Ciufo [b,1], Jiyoung Lee [b,1], Chuan Zhou [a], Bo Lin [a], Jaeyoung Cho [b], Naman Katyal [b], Graeme Henkelman [b,*]

[a] Department of Materials Science and Engineering, Guangdong Provincial Key Laboratory of Functional Oxide Materials and Devices, Southern University of Science and Technology, Shenzhen 518055, Guangdong, China
[b] Department of Chemistry and the Oden Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712-0231, USA

ARTICLE INFO

Article history:
Received 13 June 2022
Received in revised form 2 June 2023
Accepted 31 July 2023
Available online 18 August 2023

Keywords:
Machine learning
Adaptive kinetic Monte Carlo
Density functional theory
Atomic force field

ABSTRACT

In recent years, machine learning algorithms have been widely used for constructing force fields with an accuracy of ab initio methods and the efficiency of classical force fields. Here, we developed a python-based atom-centered machine-learning force field (PyAMFF) package to provide a simple and efficient platform for fitting and using machine learning force fields by implementing an atom-centered neural-network algorithm with Behler-Parrinello symmetry functions as structural fingerprints. The following three features are included in PyAMFF: (1) integrated Fortran modules for fast fingerprint calculations and Python modules for user-friendly integration through scripts and facile extension of future algorithms; (2) a pure Fortran backend to interface with the software, including the long-timescale dynamic simulation package EON, enabling both molecular dynamic simulations and adaptive kinetic Monte Carlo simulations with machine-learning force fields; and (3) integration with the Atomic Simulation Environment package for active learning and ML-based algorithm development. Here, we demonstrate an efficient parallelization of PyAMFF in terms of CPU and memory usage and show that the Fortran-based PyAMFF calculator exhibits a linear scaling relationship with the number of symmetry functions and the system size.

Program summary

Program title: python-based atom-centered machine-learning force field (PyAMFF)
CPC Library link to program files: https://doi.org/10.17632/fsn6dkcvrv.1
Developer's repository link: https://gitlab.com/pyamff/pyamff
Licensing provisions: Apache License, 2.0
Nature of problem: Determine an approximate (surrogate) model based upon atomic forces and energies from density functional theory (DFT). With a surrogate model that is less computationally expensive to evaluate than DFT, there can be a rapid exploration of the potential energy surface, accelerated optimization to minima and saddle points, and ultimately, accelerated design of active materials where the kinetics are key to the material function.
Solution method: The atomic environments of training data are calculated in terms of Behler-Parrinello fingerprints. These fingerprints are passed to a neural network which is trained to reproduce the energy and force of the training data. A parallel implementation and Fortran backend allow for efficient training and calculation of the resulting surrogate model. Examples of long-time simulations of materials on the surrogate model surfaces are provided.

1. Introduction

The use of atomistic calculations to study condensed matter systems has become standard for determining structures, reaction pathways, transition states, and barriers of complex systems [1,2]. These calculations are typically performed with *ab initio* methods, which return high accuracy with a tradeoff of high computational cost. Researchers have amassed large quantities of data from

☆ The review of this paper was arranged by Prof. Weigel Martin.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding authors.
E-mail addresses: lil33@sustech.edu.cn (L. Li), henkelman@utexas.edu (G. Henkelman).
[1] Contribute equally to this work.

https://doi.org/10.1016/j.cpc.2023.108883
0010-4655/

these calculations representing the potential energy surfaces (PESs) in terms of atomic positions. Increasingly, researchers have taken advantage of these large datasets by applying machine learning (ML) methods to approximate the PES for various atomic systems. Some examples of the software for fitting ML PESs include but are not limited to AMP [3], DeePMD-kit [4], RuNNer [1,5–8], N2P2 [9,10], and FitSNAP [11]. DeePMD-kit, RuNNer, and N2P2 construct a machine learning interatomic potential based on a particular descriptor and provide an interface with molecular dynamics (MD) software, LAMMPS [12]. AMP rather supports a wide range of descriptors and regression models so users can have more flexibility during building models and performing large-scale and long-time MD simulations. The remarkable success of these high-quality machine learning packages is demonstrated by numerous studies.

Of the above mentioned ML methods, one that stands out for its versatility is the Behler-Parrinello neural network (BPNN) [1,5–8]. The BPNN approach represents the PES from many-body individual atomic contributions. A sub-neural net for each atom type is generated, which takes atomic descriptors as input and provides atomic energies as output, which are then summed to give the total energy. The purpose of the input descriptors is to describe the relevant local environment of each atom. The atomic-centered NN structure allows for a high degree of transferability of the network while also being applicable to systems of different sizes.

The contribution of this current work is to provide a framework named the Python Atom-Centered Machine Learning Force Field (PyAMFF), which is based on the BPNN concept. PyAMFF supports rapid calculation of atomic descriptors, optimization of NN potentials, and calculation of energies and forces from NN in a transferable manner. PyAMFF provides an interface with the Atomic Simulation Environment (ASE) [13] as a calculator so that the optimized force fields can be integrated with ASE calculations or with the EON code, allowing long-timescale dynamics simulations at a reasonable cost with high accuracy. PyAMFF allows easy construction of the ML force field and a platform to perform fundamental NN studies.

The manuscript is organized as follows: Section 2 describes the atom-centered machine learning algorithm used in PyAMFF. Section 3 provides the PyAMFF code framework and describes how force fields are trained and utilized. Section 4 describes the performance of PyAMFF. Finally, Section 5 reports calculations from several systems and applies PyAMFF for long-timescale simulations.

2. Theory

The algorithms employed by PyAMFF have been described in detail elsewhere [1,3,5–8]. In brief, PyAMFF uses modified Behler-Parrinello (BP) symmetry functions as inputs to the NN model [5,14]. BP symmetry functions provide a rotationally and translationally invariant description of local atomic environments that can be mapped to energies and forces using the NN model. Both radial (G^I) and angular (G^{II}) terms are constructed for pairs with a value of the width parameter η and triplets with different values of parameters, η, λ, and ζ, respectively.

$$G_i^I = \sum_{j\neq i}^{all} e^{-\eta \frac{\left(R_{ij}-R_s\right)^2}{R_c^2}} f_c\left(R_{ij}\right) \tag{1}$$

$$G_i^{II} = 2^{1-\zeta} \sum_{\substack{j,k\neq i \\ | j\neq k}}^{all} (1+\lambda cos(\theta_{ijk} - \theta_s))^{\zeta} e^{-\eta \frac{(R_{ij}^2+R_{ik}^2+R_{jk}^2)}{R_c^2}} f_c\left(R_{ij}\right) f_c\left(R_{ik}\right) f_c\left(R_{jk}\right) \tag{2}$$

with a cutoff function

$$f_c\left(R_{ij}\right) = \begin{cases} 0.5 \cdot \left[cos\left(\frac{\pi R_{ij}}{R_c}\right)+1\right] \text{ for } R_{ij} \leq R_c \\ 0 \text{ for } R_{ij} = R_c \end{cases} \tag{3}$$

where R_{ij} is the distance between atom i and j. The center of the Gaussian functions (G^I) and (G^{II}) can be shifted to non-zero distance and angle R_s and θ_s, respectively, which can capture the local atomic environment effectively. The symmetry functions act as inputs into a NN for their respective atom type to calculate the energy for each atom. The atomic energies are summed to produce the system's total energy, and the atomic forces are calculated using the chain rule, where the Jacobian matrix of the fingerprints is computed in advance. This structure allows for a flexible potential that is invariant to the ordering of the atoms as input into the network. After a forward pass through the network, the calculated energies E and forces F are compared to the target energies E^{True} and forces F^{True} of the training set, and a loss value is generated,

$$Loss = \sum_{k=1}^{M} \left\{ \alpha \left(\frac{E_k^{True}}{N_k} - \frac{E_k}{N_k} \right)^2 + \frac{\beta}{3N_k} \sum_{l=1}^{3} \sum_{i=1}^{N_j} \left(F_{il}^{True} - F_{il} \right)^2 \right\} \tag{4}$$

where M is the number of training images, N_j is the number of atoms in image k, l represents the x, y, or z direction in the cartesian coordinate system, and α and β are energy and force coefficients, respectively. Backpropagation through the network calculates gradients of the loss function with respect to the weights and biases, which are used to update the model according to a specified optimization scheme. An overview of the process is presented in Scheme 1.

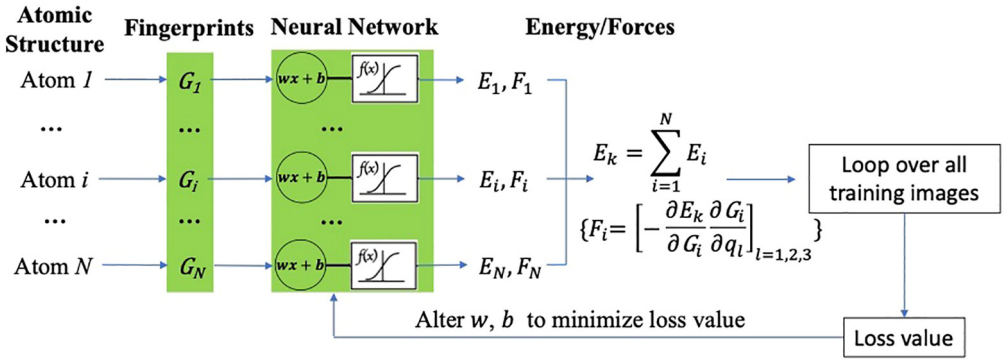
3. The PyAMFF code

The PyAMFF code is written in Python and Fortran. The layout of the code is shown in Scheme 2. PyAMFF has modules for the descriptors, neighbor-list, and fingerprints, and the ML module is supported by PyTorch [15] to fit numerical relationships between structural information and the DFT-calculated energy and force using a back-propagating NN. The backend Fortran code speeds up the calculations and provides an interface with programs, including ASE and EON.

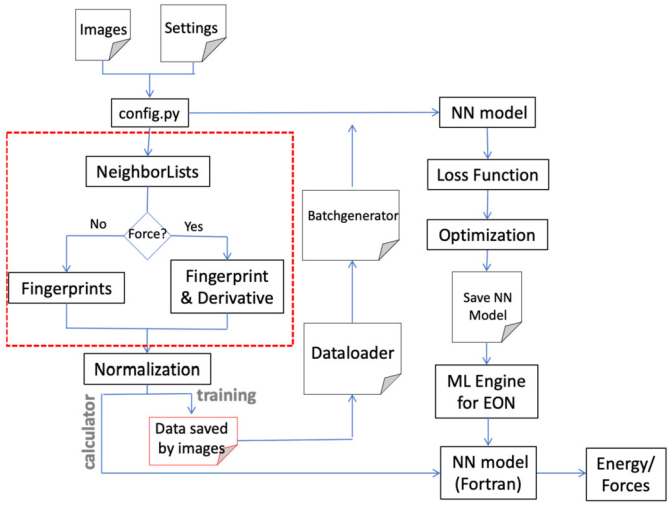
3.1. Code framework

Once a set of training structures with energies and forces is supplied and user-defined settings are configured, PyAMFF calculates neighbor-lists and pair/triplet data for each atom in each training structure. The pair/triplet information is passed from our neighbor-list module to our fingerprint module, where fingerprint and fingerprint derivatives (or fingerprint only if force calculation is not requested) are calculated and stored on disk. For flexibility, PyAMFF supports a Python and Fortran implementation of the neighbor-list and fingerprint modules. Once calculated, symmetry functions go through data-processing, which normalizes the fingerprint functions for input to the BPNN model. Finally, all the data is passed to the ML module for the NN training, testing, and evaluation.

After optimization, PyAMFF calculators can predict the energy and forces of a given new related structure by reading the saved weights, biases and architectural information of the NN. Currently, PyAMFF supports the ASE-compatible [13] and EON-compatible [16] calculators. For details, see Sections 3.3 and 3.4, respectively.



Scheme 1. An overview of the atom-centered NN applied for each element in the training set. G_i represents fingerprints of atom i. q_l is the x, y, or z component of the Cartesian coordinate system. Atoms of one element share a single NN model.



Scheme 2. Schematic representation of the PyAMFF code. Descriptors are calculated for each image based on pair and triplet information. These descriptors are normalized, batched, and saved to disk. Batches are passed to the NN module, and a potential is generated. The NN potential can be utilized using an ASE [13] style calculator or a ML Engine for EON [16]. The modules in the dotted red box are available in both Python and Fortran. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

3.2. Training force fields

A large training set is typically required to obtain an accurate ML potential which increases the computational cost of fitting. For performance, PyAMFF parallelizes the training process via the distributed multiprocessing module of PyTorch. The model is replicated across N processes, while M batches are trained by each process (see Fig. 2a) that can be controlled via user-supplied flags. Batching performance is reported in Section 4.1. In addition to parallel training capabilities, PyAMFF allows for batching fingerprint calculations to account for system memory constraints. Finally, in addition to being trained on CPUs, PyAMFF can be trained using GPUs via the CUDA toolkit supplied by PyTorch for a significant performance increase on modern hardware, which is discussed in Section 4.1.

Once the training data set is distributed, PyAMFF initiates the NN model with an unoptimized set of weights and biases (NN parameters). This NN model is utilized to predict the energy and forces for each image, and the difference between the predicted energy and forces from target/training energy and forces (called loss) is used to optimize the NN parameters. PyAMFF optimizes the NN parameters until either the loss in energy and forces become smaller than the stopping criteria or the optimization is performed for a maximum number of epochs. After training, the NN parameters are saved in two different file formats, "pyamff.pt" and "mlff.pyamff", which can be used by built-in calculators or to restart training. Note that ".pt" is a PyTorch readable format, and ".pyamff" is a user-readable format.

Additionally, PyAMFF allows for NN models to be updated and retrained with the addition of new data for on-the-fly training of systems as they evolve. Retraining becomes beneficial for accelerating methods such as structural optimization or nudged elastic band calculations [17,18]. To enable the on-the-fly training, the user only needs to write three lines of commands, as shown below. First, define a pyamffRunner object by turning on the active learning flag. Second, add new images generated during structural optimization, nudged elastic band calculations, or molecular dynamics simulations to the training set. Finally, call the run function in the pyamffRunner object to conduct the force field training. The obtained ML force field can be reused by calling the PyAMFF ASE calculator (see Section 3.3 for details). Fig. 1 illustrates an example of the speedup of structure optimizations using PyAMFF. While we have observed performance improvements, more work is required to develop a mature algorithm that utilizes machine-learning force fields to speed up structural optimization. PyAMFF code provides an easy-to-use tool for developers to design their algorithm, potentially facilitating the development of machine-learning-assisted structural optimization, NEB, and MD methods.

```
   self.pyamff_train = pyamffRunner(activelearning=
True)
   self.training_set.append(new_images)
   self.pyamff_train.run(self.epoches_max, self.train-
ing_set)
```

3.3. ASE calculator

PyAMFF supports a built-in ASE-compatible calculator that can predict the energy and force of a given structure using a trained ML model [13]. For modularity, we provide both Python and Fortran based ASE-compatible calculators. The Python calculator can be used as pure Python or as a hybrid of Python and Fortran (to speed up the fingerprint calculations) and PyTorch for the NN. The Fortran calculator is implemented entirely in Fortran, with a Python wrapper to interface with ASE. That is, the entire process, from reading the trained network parameters to predicting energy and force, is performed in Fortran; this calculator does not rely on PyTorch for evaluating the NN.

When using the PyAMFF calculator, the trained ML model parameters are read from a file in either ".pt" for the Python calculator or ".pyamff" for the Fortran calculator. The Python and Fortran calculators can be imported and initialized respectively as:

```
#Python:
from pyamff.aseCalc import aseCalc
```

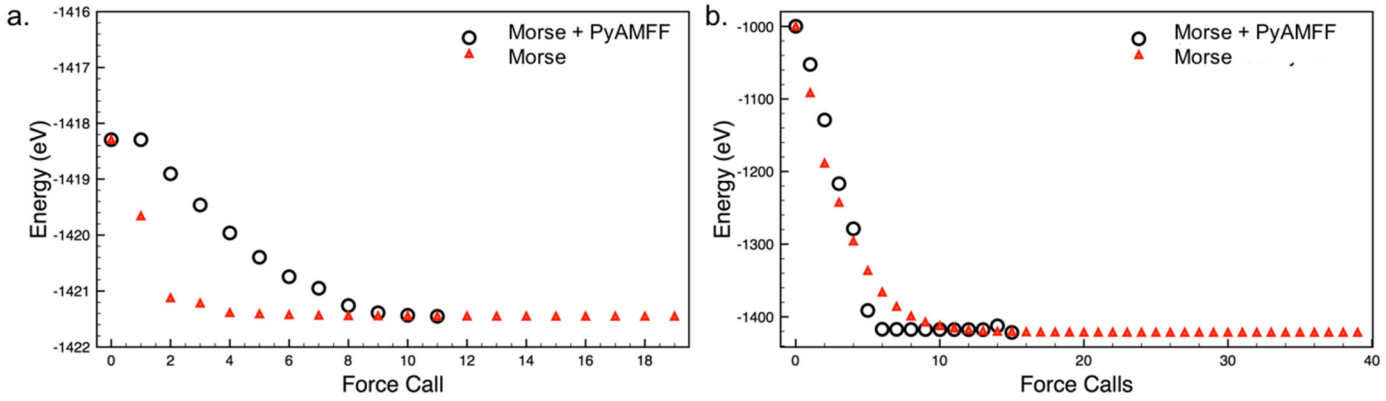


Fig. 1. a.) Mildly and b.) severely perturbed bulk 256 atom Pt structures described by a Morse potential. Using PyAMFF with 2 G^Is and 2 G^{II}s and a (5,5) neural network reduces the total number of force evaluations needed. Structures were obtained from optbench.org, [19] where additional information about this benchmark system can be found for reference.

```
calc = aseCalc('./pyamff.pt')

#Fortran:
from pyamff.aseCalcF import aseCalcF
calc = aseCalcF(mlff_file='mlff.pyamff')
```

Once initialized, both calculators inherit the methods of an atoms object in ASE [13].

3.4. Fortran module for EON

One of the key features of PyAMFF is that the trained ML potential can be utilized for modeling dynamics of atomic-scale systems over long time scales with our EON code [16]. EON provides the adaptive kinetic Monte Carlo (AKMC) method, which can achieve long time scale dynamics within a computationally reasonable time by finding relevant transitions on the fly [20,21]. While the availability of a reliable and inexpensive force field is a significant challenge for the method, ML potentials have become a good candidate due to their DFT accuracy with orders of magnitude lower computational cost. To take full advantage of ML in EON, we built a pure Fortran module for PyAMFF, which follows a very similar workflow to the Fortran ASE calculator described in Section 3.3, that can be executed by EON. The actual application and its performance validations are discussed in the following sections.

4. Performance test

4.1. Parallelization for training

Fig. 2a shows a framework of the parallelization algorithm applied in PyAMFF to speed up the training process. PyAMFF splits the training dataset to $nProc \times nbatch$ total batches, where $nProc$ defines the number of processes and $nbatch$ defines the number of batches per process. Each process then receives a batch corresponding to one column of $nbatch$ in Fig. 2a. Each $nbatch$ is passed through the NN on the corresponding process, and the loss is evaluated. The total loss is calculated by looping over $nbatch$ columns of batches. $nProc$ and $nbatch$ can be used to manage the computational efficiency and memory usage, respectively.

To demonstrate the parallelization efficiency of PyAMFF, we performed a training job with 2000 images of a $Pd_{13}H_2$ nanoparticle for 1000 epochs and NN architecture of one hidden layer with 50 neurons. The optimized fingerprints from Ref. [14] were adopted without modifications. The left panel of Fig. 2b shows the training time and the corresponding speedup factor with $nProc$ ($nbatch$=1). The speedup factor is used to measure the parallelization efficiency relative to the sequential execution and is defined as

$$speedup = \frac{runtime\ of\ serial\ execution}{runtime\ of\ parallel\ execution}.$$

The training time reduces linearly with increasing $nProc$, i.e., the speedup factor is linearly correlated with $nProc$ and maximum computational efficiency at $nProc = 20$. Further increase of the process number beyond 20 led to a loss of computational efficiency, and an inverse linear relationship between the speedup factor and $nProc$ was observed due to more frequent data loading and a communication bottleneck. Note that these values will vary for different machines.

In PyAMFF, $nbatch$ is used to manage memory usage during training. The right panel of Fig. 2b presents the memory usage and the calculated scaledown factor with $nbatch$ ($nProc = 1$). Similar to the definition of speedup, the scaledown factor is used to measure the scaling efficiency of memory usage relative to execution with $nbatch = 1$, which is calculated as

$$scaledown = \frac{memory\ usage\ of\ execution\ with\ nbatch = 1}{memory\ usage\ of\ execution\ with\ nbatch > 1}.$$

The training job with $nbatch = 1$ used 4 GB of memory, which was reduced to 1.64 GB at $nbatch$ 8, corresponding to a scaledown factor of 2.4. A further increase of $nbatch$ provides limited gains in the scaledown efficiency. As expected, the maximum memory usage is inversely proportional to $nbatch$.

PyAMFF package supports GPU training which we have compared to CPU training. While we are still optimizing our GPU performance, the current version of PyAMFF GPU training is up to 1.5 times faster than CPU training. We would like to emphasize that our benchmarking was performed with only one system discussed below, using an A100 GPU machine, and rigorous testing will be performed in the future. Finally, PyAMFF only supports GPU training on one machine $nProc = 1$, and we are building an interface that can support multi-GPU training. The GPU vs. CPU speedup was tested using a lithium DFT dataset from VASP calculations. The dataset consists of bulk structures, surfaces with adatoms, strained bulk structures from 4 different lithium phases (BCC, FCC, 9R, HCP), and lithium clusters from sizes 6-100. The GPU training was performed with $nProc = 1$ and batches per process $nbatch = 1, 2, 4, 8, 16, 32, 64$, and the CPU training was performed such that total batches ($nProc$ * $nbatch$) was equal to 1, 2, 4, 8, 16, 32, 64 for a fair comparison with GPU. The GPU vs. CPU training time comparison with total batches is shown in Fig. 2c. CPU-Z1 means CPU training with Z=2^x $nProc$ and $nbatch = 1$ (multiple processes, one batch per process), CPU-1Z means CPU training with $nProc = 1$, and Z=2^x $nbatch$ (one process, multiple batches per process), and CPU-4,8,16,32,64 is CPU training with 4, 8, 16, 32, 64 total batches with 2^x $nProc$. For the CPU-4 example, the hexagonal point represents training time for $nProc = 2^1$ and $nbatch = 2$ (total batches/nProc, 4/2). The best training time for GPU is $nProc, nbatch = (1, 1)$ 143 seconds which is ~1.5 times faster than the best CPU training time with $nProc, nbatch = (16, 1)$ 217.9 seconds and ~8 times faster than CPU training time with $nProc, nbatch = (1, 1)$ 752 seconds. The GPU vs. CPU training time difference becomes 1.64 with 1500 training images. Hence, with a bigger dataset, the GPU vs. CPU speed up can be higher, which we plan to investigate in future studies.

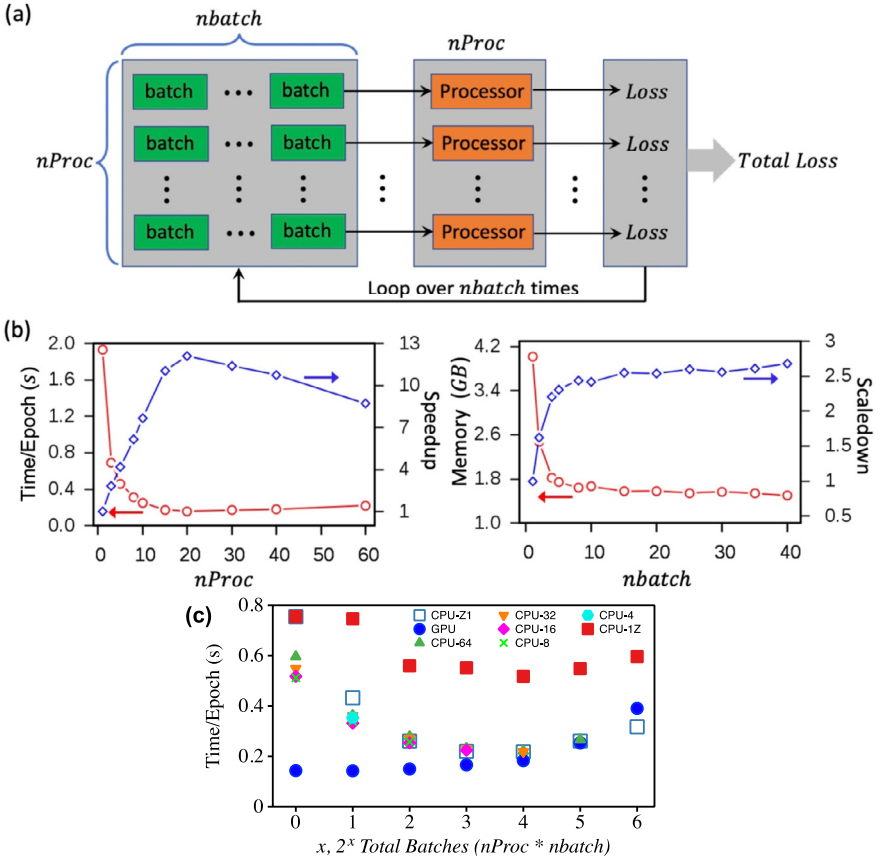


Fig. 2. (a) Schematic illustration of the parallelization algorithm for training in PyAMFF. (b) The execution time and memory used for training as a function of the number of processes (*nProc*) and batches (*nbatch*). (c) GPU vs. CPU training time for lithium dataset with 1000 images. X-axis represents a different number of total batches for GPU, CPU-Z1, and CPU-1Z, where total batches are defined as the number of processes (*nProc*) * batches per processes (*nbatch*) and number of processes for CPU training (*nProc*) for CPU-4, CPU-8, CPU-16, CPU-32, and CPU-64 for 4, 8, 16, 32 and 64 total batches respectively. The training time was calculated for 1000 epochs.

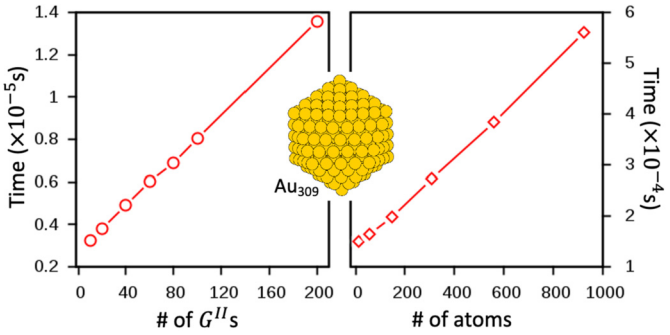


Fig. 3. Computational cost as a function of (left) number of G^{II} functions for a Au_{309} cluster and (right) the number of atoms in a gold nanoparticle.

4.2. Calculator for energy and force evaluation

The scaling relationship between computational cost and system size is an important metric for the efficiency of our computational methods. Here, we tested the dependence of computational cost on the number of fingerprints and the number of atoms by evaluating the energy and force of Au nanoparticles with a pre-trained PyAMFF force field (Fig. 3). The computational cost dependence on the number of fingerprints was tested by varying number of G^{II} functions per atom because the evaluation of the G^{II} fingerprints is the most resource-intensive calculation. As shown in Fig. 3a, the computational cost scales linearly on the number of G^{II} fingerprints. A force evaluation with 200 G^{II} functions took 1.3×10^{-5} s, which is 8 orders of magnitude faster than the comparable DFT calculation. Computational scaling as a function of system size was performed by evaluating the force and energy of icosahedral Au nanoparticles with varying numbers of atoms ranging from 13 to 923 (Fig. 3b); the computational cost increased linearly with the number of atoms. These scaling tests demonstrate that the PyAMFF Fortran calculator has a linear scaling relationship with the number of fingerprints and system size, ensuring the extensibility of our code to large systems.

5. Computational results

Here we show performance tests of force-field training for a periodic (Ge) and a nanoparticle ($Pd_{13}H_2$) system with the PyAMFF code. We also present an AKMC example of a $Pd_{13}H_2$ nanoparticle with a pre-trained force field.

5.1. Ge

A PyAMFF model was trained for a periodic Ge system. The database of atomic structures was constructed by expanding the Si atomic structures reported by Bartók et al. [22]. In total, 2364 atomic structures were collected for training. Another 7092 atomic structures were created by further expanding or compressing the unit cell, with 73% of the data in the training set and 27% reserved for model validation. The energy and forces were evaluated with DFT as implemented in VASP. To describe the chemical environment of each Ge atom, 24 G^{I}s and 16 G^{II}s were selected, with a cutoff of 5.7 Å (refer to supporting information for details of the fingerprints parameters). A (15, 15) NN model with the sigmoid activation function was adopted to fit the PES of the Ge system. Fig. 4 shows the predicted energy (left) and forces (right) of the training (in red) and validation (in green) data from the PyAMFF model as compared to the reference DFT values. The energy and forces for training and validation data are distributed tightly along the $y = x$ line. The calculated RMSEs per atom of the energy and forces for the training data are 0.013 eV/atom

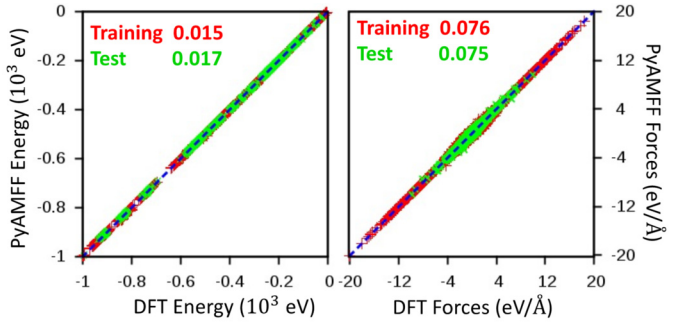


Fig. 4. Comparison of the energy (left) and forces (right) for a periodic Ge system with reference DFT values. The red and green symbols represent the values of the training and test data, respectively. The training and test values reported are root mean square errors of energy (eV/atom) and force (eV/Å) for the training (in red) and test data (in green). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

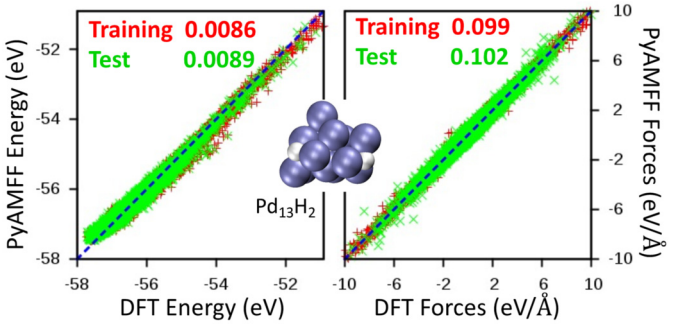


Fig. 5. Comparison of the energy (left) and forces (right) evaluated by the PyAMFF model with the reference DFT values. The red and green symbols represent the values of the training and test data, respectively. Light blue and white spheres represent Pd and H atoms, respectively. The values reported are root mean square errors of energy (in eV/atom) and force (eV/Å) for the training (in red) and test data (in green). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

and 0.076 eV/Å, respectively. Similar RMSEs for the validation data are observed, 0.017 eV/image for energy and 0.075 eV/Å for forces.

5.2. $Pd_{13}H_2$ particles

In our previous study [14], we optimized fingerprints and trained a BPNN force field for the $Pd_{13}H_2$ nanoparticle with the AMP [3] package. Here, we adopted the same fingerprints (36 G^Is and 12 G^{II}s) and training and test data to evaluate the performance of our PyAMFF code for force field training (refer to Ref. [14] for more details). Fig. 5 shows the predicted energy (left) and forces (right) of the training and test data from the PyAMFF model against the reference DFT values. The calculated RMSEs per atom of the energy and forces for the training data are 0.0086 eV/atom and 0.099 eV/Å, respectively and RMSEs for test data are 0.0089 eV/atom and 0.102 eV/Å, respectively, slightly larger than the RMSEs of the training data. Overall, the trained PyAMFF model exhibits a very similar performance as the BPNN model obtained from the AMP package.

5.3. AKMC simulations

Finally, as shown in Fig. 6, we performed AKMC simulations to compute the dynamical evolution of a $Pd_{13}H_2$ nanoparticle at 300 K, using the PyAMFF force field described in Section 5.2. More computational details can be found in Ref. [14]. When compared to results reported previously using AMP in Ref. [14], a similar trend is seen. Due to the improved computational efficiency of fingerprints and their derivatives, the time scale of the dynamic simulation reaches 17 ns, much longer than that reported (100 ps) in Ref. [14]. As observed in Ref. [14], the nanoparticle undergoes a structural transition accompanied by diffusion of the hydrogen atoms on the surface. As shown in Fig. 6a, at ∼0.6 ns, the rotation of a five-Pd motif (highlighted in gold) leads to a transition from an icosahedral structure to

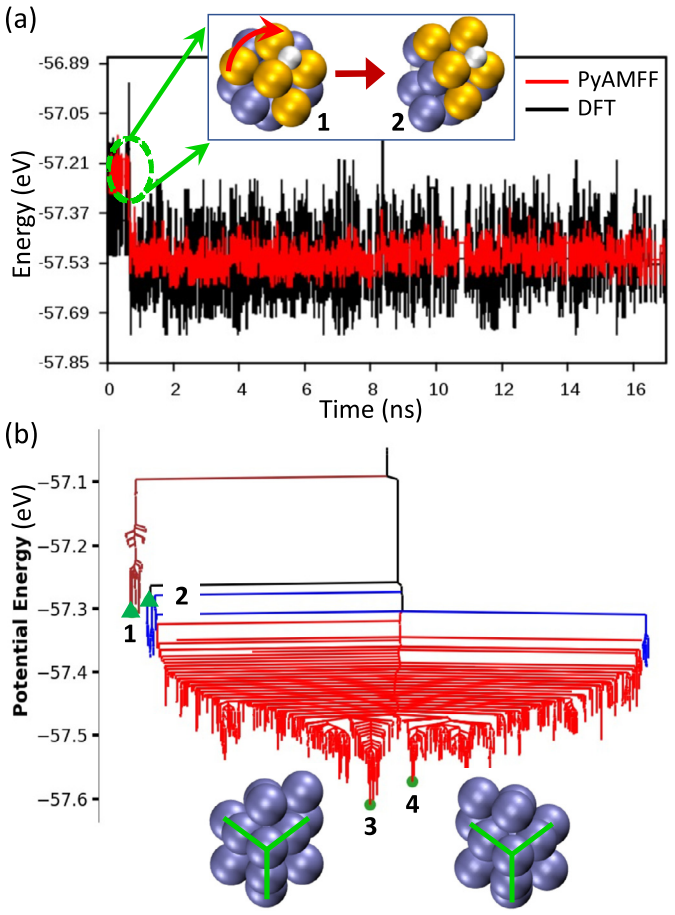


Fig. 6. (a) Time evolution of the total energy and (b) disconnectivity graph from an AKMC simulation of the $Pd_{13}H_2$ nanoparticle at 300 K using PyAMFF-based EON [16]. Inset images in (a) indicate the structure transition from an icosahedral cluster to a deformed cluster, denoted as Structure 1 and 2 in (b). Both the light blue and gold spheres represent Pd atoms, and the white spheres represent H atoms. In (b), brown and red lines represent isomers with an icosahedral Pd structure. Red lines represent isomers with a chiral Pd skeletal structure, as shown in the inset images. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

a deformed structure. Upon that, the system undergoes several metastable states (blue lines in Fig. 6b) and transitions to a more stable state (red lines in Fig. 6b). In such a state, the nanoparticle entails a chiral Pd structure, as shown in the inset images of Fig. 6b. The chiral structures (3 and 4) correspond to the Type 4 and 5 isomers reported in Ref. [14] (corresponding side images are shown in Figure S1).

5.4. Performance comparison with other BPNN packages

We conducted benchmark calculations to evaluate the computational efficiency of the PyAMFF code. Firstly, we used a Ge system with 24 G^Is and 16 G^{II}s to compare the execution time of PyAMFF, N2P2, and AMP for single-point energy and force evaluation. Fig. 7a shows the variation of execution time for single-point energy and force evaluation with the number of atoms for each package. The PyAMFF code was approximately 100 times faster than AMP but about 10 times slower than N2P2. We also compared the training efficiency of PyAMFF and N2P2 and found that PyAMFF outperforms N2P2 in the training process (Fig. 7b). In just 1300 s, PyAMFF achieved an energy RMSE value of 5 meV/atom and a force RMSE value of 0.07 eV/Å, whereas N2P2 maintains larger RMSE values even at 2000 s. Additionally, we evaluated the performance of PyAMFF by running the benchmark systems reported by Zuo and Ong et al. [23]. Our results show that PyAMFF achieves smaller RMSE values compared to the NNP results reported in the same publication (Table 1). Although PyAMFF has a moderate computational efficiency in energy and force evaluation, it exhibits better training performance with higher accuracy and lower computa-

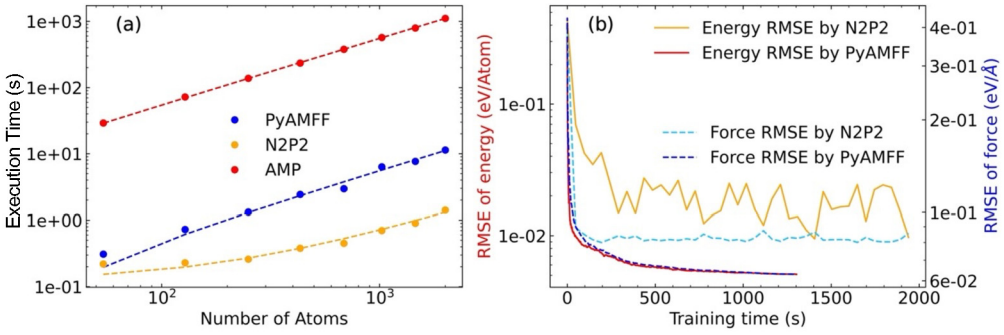


Fig. 7. (a) Variation of the execution time of single-point-energy calculation with the number of atoms using PyAMFF, N2P2, and AMP. (b) Variation of energy and force RMSEs with training time obtained with PyAMFF and N2P2. Training is performed on 40 CPU cores with 100 Ge structures, including 24 G^I and 16 G^{II} functions. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 1
Energy and force RMSEs obtained from the PyAMFF package for the Ni, Cu, Li, Mo, Si, and Ge systems reported in Ref. [23]. The values in column 'NNP' are data from Ref. [23]. In each cell, the values before and after the slash symbol are RMSE values for the training and test sets, respectively.

	Energy_RMSE (meV/atom)		Force_RMSE (eV/Å)	
	PyAMFF	NNP (ref. [23])	PyAMFF	NNP (ref. [23])
Ni	0.28/0.72	2.38/2.25	0.03/0.04	0.06/0.07
Cu	0.24/0.49	2.13/1.68	0.02/0.02	0.05/0.06
Li	0.26/0.79	1.24/0.98	0.02/0.02	0.06/0.06
Mo	1.50/3.73	6.06/5.67	0.16/0.20	0.20/0.20
Si	1.64/4.05	10.84/9.95	0.10/0.13	0.17/0.17
Ge	1.40/3.96	11.27/10.95	0.08/0.10	0.12/0.12

tional cost. Overall, our evaluation indicates that PyAMFF is a promising machine-learning force field development package.

6. Conclusion

The robustness of machine learning and the BPNN method has motivated us to develop the open-source Pythonic Atom-Centered Machine Learning Force Field package. The goals of the project are to provide a quick, easy, and efficient platform for fitting and using machine learning force fields while also providing a platform that can supplement calculations and provide a basis for studying and understanding machine learning for chemistry and materials. PyAMFF allows for the efficient training of machine learning force fields and includes integration with ASE [13] and EON [16] for the utilization of trained force fields. We have shown that PyAMFF is efficient and scalable to large systems while accurately (re)producing results for various periodic and non-periodic systems. Additionally, PyAMFF allows for active training, which can be used to fit potentials on-the-fly to speed up DFT calculations.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Graeme Henkelman reports financial support was provided by Welch Foundation (F-1841), National Science Foundation (CHE-2102317). Lei Li reports financial support was provided by the National Key R&D Program of China (2022YFA1503102), the National Natural Science Foundation of China (22179058), the Department of Science and Technology of Guangdong Province (2021B1212040001).

Data availability

Data will be made available on request.

Acknowledgements

This work was supported by the National Key R&D Program of China (2022YFA1503102), the National Natural Science Foundation of China (22179058), the Guangdong Provincial Key Laboratory Program (2021B1212040001) from the Department of Science and Technology of Guangdong Province, and the Center for Computational Science and Engineering at the Southern University of Science and Technology (SUSTech). Work in Austin was supported by the National Science Foundation (CHE-2102317), the Welch Foundation (F-1841), and the Texas Advanced Computing Center.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2023.108883.

References

[1] J. Behler, Angew. Chem., Int. Ed. 56 (2017) 12828–12840.
[2] O.T. Unke, S. Chmiela, H.E. Sauceda, M. Gastegger, I. Poltavsky, K.T. Schütt, A. Tkatchenko, K.-R. Müller, Chem. Rev. 121 (2021) 10142–10186.
[3] A. Khorshidi, A.A. Peterson, Comput. Phys. Commun. 207 (2016) 310–324.
[4] H. Wang, L. Zhang, J. Han, W. E, Comput. Phys. Commun. 228 (2018) 178–184.
[5] J. Behler, M. Parrinello, Phys. Rev. Lett. 98 (2007) 146401.
[6] J. Behler, Int. J. Quant. Chem. 115 (2015) 1032–1050.
[7] J. Behler, J. Chem. Phys. 134 (2011) 074106.
[8] J. Behler, J. Phys. Condens. Matter 26 (2014) 183001.
[9] A. Singraber, J. Behler, C. Dellago, J. Chem. Theory Comput. 15 (2019) 1827–1840.
[10] A. Singraber, T. Morawietz, J. Behler, C. Dellago, J. Chem. Theory Comput. 15 (2019) 3075–3092.
[11] A. Rohskopf, C. Sievers, N. Lubbers, M.A. Cusentino, J. Goff, J. Janssen, M. McCarthy, D.M. De Oca Zapiain, S. Nikolov, K. Sargsyan, D. Sema, E. Sikorski, L. Williams, A.P. Thompson, M.A. Wood, J. Soc. Struct. 8 (2023) 5118.
[12] A.P. Thompson, H.M. Aktulga, R. Berger, D.S. Bolintineanu, W.M. Brown, P.S. Crozier, P.J. in 't Veld, A. Kohlmeyer, S.G. Moore, T.D. Nguyen, R. Shan, M.J. Stevens, J. Tranchida, C. Trott, S.J. Plimpton, Comput. Phys. Commun. 271 (2022) 108171.
[13] A. Hjorth Larsen, J. Jørgen Mortensen, J. Blomqvist, I.E. Castelli, R. Christensen, M. Dułak, J. Friis, M.N. Groves, B. Hammer, C. Hargus, E.D. Hermes, P.C. Jennings, P. Bjerre Jensen, J. Kermode, J.R. Kitchin, E. Leonhard Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K.S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, K.W. Jacobsen, J. Phys. Condens. Matter 29 (2017) 273002.
[14] L. Li, H. Li, I.D. Seymour, L. Koziol, G. Henkelman, J. Chem. Phys. 152 (2020) 224102.
[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, vol. 32, Curran Associates, Inc., 2019, pp. 8024–8035.
[16] S.T. Chill, M. Welborn, R. Terrell, L. Zhang, J.-C. Berthet, A. Pedersen, H. Jónsson, G. Henkelman, Model. Simul. Mater. Sci. Eng. 22 (2014) 055002.
[17] G. Henkelman, H. Jónsson, J. Chem. Phys. 113 (2000) 9978–9985.
[18] A.A. Peterson, J. Chem. Phys. 145 (2016) 074106.
[19] S. Chill, G. Henkelman, (n.d.).
[20] G. Henkelman, H. Jónsson, J. Chem. Phys. 115 (2001) 9657–9666.
[21] L. Xu, G. Henkelman, J. Chem. Phys. 129 (2008) 114104.
[22] A.P. Bartók, J. Kermode, N. Bernstein, G. Csányi, Phys. Rev. X 8 (2018) 041048.
[23] Y. Zuo, C. Chen, X. Li, Z. Deng, Y. Chen, J. Behler, G. Csányi, A.V. Shapeev, A.P. Thompson, M.A. Wood, S.P. Ong, J. Phys. Chem. A 124 (2020) 731–745.