Computer Physics Communications 228 (2018) 178–184

Contents lists available at ScienceDirect

Computer Physics Communications

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

DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics☆

Han Wang [a,b,*], Linfeng Zhang [c,**], Jiequn Han [c], Weinan E [c,d,e]

[a] Institute of Applied Physics and Computational Mathematics, Fenghao East Road 2, Beijing 100094, PR China
[b] CAEP Software Center for High Performance Numerical Simulation, Huayuan Road 6, Beijing 100088, PR China
[c] Program in Applied and Computational Mathematics, Princeton University, Princeton, NJ 08544, USA
[d] Department of Mathematics, Princeton University, Princeton, NJ 08544, USA
[e] Beijing Institute of Big Data Research, Beijing, 100871, PR China

ARTICLE INFO

Article history:
Received 11 December 2017
Received in revised form 25 February 2018
Accepted 5 March 2018
Available online 21 March 2018

Keywords:
Many-body potential energy
Molecular dynamics
Deep neural networks

ABSTRACT

Recent developments in many-body potential energy representation via deep learning have brought new hopes to addressing the accuracy-versus-efficiency dilemma in molecular simulations. Here we describe DeePMD-kit, a package written in Python/C++ that has been designed to minimize the effort required to build deep learning based representation of potential energy and force field and to perform molecular dynamics. Potential applications of DeePMD-kit span from finite molecules to extended systems and from metallic systems to chemically bonded systems. DeePMD-kit is interfaced with TensorFlow, one of the most popular deep learning frameworks, making the training process highly automatic and efficient. On the other end, DeePMD-kit is interfaced with high-performance classical molecular dynamics and quantum (path-integral) molecular dynamics packages, i.e., LAMMPS and the i-PI, respectively. Thus, upon training, the potential energy and force field models can be used to perform efficient molecular simulations for different purposes. As an example of the many potential applications of the package, we use DeePMD-kit to learn the interatomic potential energy and forces of a water model using data obtained from density functional theory. We demonstrate that the resulted molecular dynamics model reproduces accurately the structural information contained in the original model.

Program summary
Program Title: DeePMD-kit
Program Files doi: http://dx.doi.org/10.17632/hvfh9yvncf.1
Licensing provisions: LGPL
Programming language: Python/C++
Nature of problem: Modeling the many-body atomic interactions by deep neural network models. Running molecular dynamics simulations with the models.
Solution method: The Deep Potential for Molecular Dynamics (DeePMD) method is implemented based on the deep learning framework TensorFlow. Supports for using a DeePMD model in LAMMPS and i-PI, for classical and quantum (path integral) molecular dynamics are provided.
Additional comments including Restrictions and Unusual features: The code defines a data protocol such that the energy, force, and virial calculated by different third-party molecular simulation packages can be easily processed and used as model training data.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

The dilemma of accuracy versus efficiency in modeling the potential energy surface (PES) and interatomic forces has confronted the molecular simulation communities for a long time. On one hand, *ab initio* molecular dynamics (AIMD) has the accuracy of the density functional theory (DFT) [1–3], but the computational cost of DFT in evaluating the PES and forces restricts its typical applications to system size of hundreds to thousands of atoms and time

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author at: Institute of Applied Physics and Computational Mathematics, Fenghao East Road 2, Beijing 100094, PR China.
** Corresponding author.
E-mail addresses: wang_han@iapcm.ac.cn (H. Wang), linfengz@princeton.edu (L. Zhang), jiequnh@princeton.edu (J. Han), weinan@math.princeton.edu (W. E).

https://doi.org/10.1016/j.cpc.2018.03.016
0010-4655/© 2018 Elsevier B.V. All rights reserved.

scale of ∼100 ps. One the other hand, a great deal of effort has been made in developing empirical force fields (FFs) [4–6], which allows for much larger and longer simulations. However, the accuracy and transferability of FFs is often in question. Moreover, fitting the parameters of an FF is usually a tedious and ad hoc process.

In the last few years, machine learning methods have been suggested as a tool to model PES of molecular systems with DFT data, and have achieved some remarkable success [7–16]. Some examples (not a comprehensive list) include the Behler–Parrinello neural network (BPNN) [9], the Gaussian approximation potentials (GAP) [11], the Gradient-domain machine learning (GDML) [14], and the Deep potential for molecular dynamics (DeePMD) [17,18]. In particular, it has been demonstrated for a wide variety of systems that the "deep potential" and DeePMD allow us to perform molecular dynamics simulation with accuracy comparable to that of DFT (or other fitted data) and the efficiency competitive with empirical potential-based molecular dynamics [17,18].

Machine learning, particularly deep learning has been shown to be a powerful tool in a variety of fields [19,20] and even has outperformed human experts in some applications like the AlphaGo in the board game Go [21]. A number of open source deep learning platforms, e.g. TensorFlow [22], Caffe [23], Torch [24], and MXNet [25] are available. These open source platforms have significantly lowered the technical barrier for the application of deep learning. Considering the potential impact that deep learning-based methods will have on molecular simulation, it is of considerable interest to develop open source platforms that serve as the interface between deep neural network models and molecular simulation tools such as LAMMPS [26], Gromacs [27] and NAMD [28], and path-integral MD packages like i-PI [29].

The contribution of this work is to provide an implementation of the DeePMD method, namely DeePMD-kit,[1] which interfaces with TensorFlow for fast training, testing, and evaluation of the PES and forces, and with LAMMPS and i-PI for classical and path-integral molecular dynamics simulations, respectively. In DeePMD-kit, we implement the atomic environment descriptors and chain rules for force/virial computations in C++ and provide an interface to incorporate them as new operators in standard TensorFlow. This allows the model training and MD simulations to benefit from TensorFlow's highly optimized tensor operations. The support of DeePMD for LAMMPS is implemented as a new "pair style", the standard command in LAMMPS. Therefore, only a slight modification in the standard LAMMPS input script is required for energy, force, and virial evaluation through DeePMD-kit. The support for i-PI is implemented as a new force client communicating through sockets with the standard i-PI server, which handles the bead integrations. Given these features provided by DeePMD-kit, training deep neural network model for potential energy and running MD simulations with the model is made much easier than implementing everything from scratch.

The manuscript is organized as follows. In Section 2, the theoretical framework of the DeePMD method is provided. We show in detail how the system energy is constructed and how to take derivatives with respect to the atomic position and box tensor to compute the force and virial. In Section 3, we provide a brief introduction on how to use DeePMD-kit to train a model and run MD simulations with the model. In Section 4, we demonstrate the performance of DeePMD-kit by training a DeePMD model from AIMD data. Results from the MD simulation using the trained DeePMD model are compared to the original AIMD data to validate the modeling. The paper concludes with a discussion about the future work planed for DeePMD-kit.

2. Theory

We consider a system consisting of N atoms and denote the coordinates of the atoms by $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_N\}$. The potential energy E of the system is a function with $3N$ variables, i.e., $E = E(\boldsymbol{R}_1, \ldots, \boldsymbol{R}_N)$, with each $\boldsymbol{R}_i \in \mathbb{R}^3$. In the DeepMD method, E is decomposed into a sum of atomic energy contributions,

$$E = \sum_i E_i, \tag{1}$$

with i being the indexes of the atoms. Each atomic energy is fully determined by the position of the ith atom and its near neighbors,

$$E_i = E_{s(i)}(\boldsymbol{R}_i, \{\boldsymbol{R}_j \,|\, j \in N_{R_c}(i)\}), \tag{2}$$

where $N_{R_c}(i)$ denotes the index set of the neighbors of atom i within the cut-off radius R_c, i.e. $R_{ij} = |\boldsymbol{R}_{ij}| = |\boldsymbol{R}_i - \boldsymbol{R}_j| \leq R_c$. $s(i)$ is the chemical species of atom i. The most straightforward idea to model the atomic energy $E_{s(i)}$ through DNN is to train a neural network with the input simply being the positions of the ith atom $\boldsymbol{R}_i$ and its neighbors $\{\boldsymbol{R}_j \,|\, j \in N_{R_c}(i)\}$. This approach is less than optimal as it does not guarantee the translational, rotational, and permutational symmetries lying in the PES. Thus, a proper preprocessing of the atomic positions, which maps the positions to "descriptors" of atomic chemical environment [30] is needed.

In the DeePMD method, to construct the descriptor for atom i, the positions of its neighbors are firstly shifted by the position of atom i, viz. $\boldsymbol{R}_{ij} = \boldsymbol{R}_i - \boldsymbol{R}_j$. The coordinate of the relative position $\boldsymbol{R}_{ij}$ under lab frame $\{\boldsymbol{e}_x^0, \boldsymbol{e}_y^0, \boldsymbol{e}_z^0\}$ is denoted by $(x_{ij}^0, y_{ij}^0, z_{ij}^0)$, i.e.,

$$\boldsymbol{R}_{ij} = x_{ij}^0 \boldsymbol{e}_x^0 + y_{ij}^0 \boldsymbol{e}_y^0 + z_{ij}^0 \boldsymbol{e}_z^0. \tag{3}$$

Both $\boldsymbol{R}_{ij}$ and the coordinate $(x_{ij}^0, y_{ij}^0, z_{ij}^0)$ preserve the translational symmetry. The rotational symmetry is preserved by constructing a local frame and recording the local coordinate for each atom. First, two atoms, indexed $a(i)$ and $b(i)$, are picked from the neighbors $N_{R_c}(i)$ by certain user-specified rules. The local frame $\{\boldsymbol{e}_{i1}, \boldsymbol{e}_{i2}, \boldsymbol{e}_{i3}\}$ of atom i is then constructed by

$$\boldsymbol{e}_{i1} = \boldsymbol{e}(\boldsymbol{R}_{ia(i)}), \tag{4}$$

$$\boldsymbol{e}_{i2} = \boldsymbol{e}\big(\boldsymbol{R}_{ib(i)} - (\boldsymbol{R}_{ib(i)} \cdot \boldsymbol{e}_{i1})\boldsymbol{e}_{i1}\big), \tag{5}$$

$$\boldsymbol{e}_{i3} = \boldsymbol{e}_{i1} \times \boldsymbol{e}_{i2}, \tag{6}$$

where $\boldsymbol{e}(\boldsymbol{R})$ denotes the normalized vector of $\boldsymbol{R}$, i.e., $\boldsymbol{e}(\boldsymbol{R}) = \boldsymbol{R}/|\boldsymbol{R}|$. Then the local coordinate (x_{ij}, y_{ij}, z_{ij}) (under the local frame) is transformed from the global coordinate $(x_{ij}^0, y_{ij}^0, z_{ij}^0)$ through

$$(x_{ij}, y_{ij}, z_{ij}) = (x_{ij}^0, y_{ij}^0, z_{ij}^0) \cdot \mathcal{R}(\boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)}), \tag{7}$$

where

$$\mathcal{R}(\boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)}) = [\boldsymbol{e}_{i1}, \boldsymbol{e}_{i2}, \boldsymbol{e}_{i3}] \tag{8}$$

is the rotation matrix with the columns being the local frame vectors. The descriptive information of atom i given by neighbor j is constructed by using either full information (both radial and angular) or radial-only information:

$$\{D_{ij}^{\alpha}\} = \begin{cases} \left\{\dfrac{1}{R_{ij}}, \dfrac{x_{ij}}{R_{ij}}, \dfrac{y_{ij}}{R_{ij}}, \dfrac{z_{ij}}{R_{ij}}\right\}, & \text{full information;} \\ \left\{\dfrac{1}{R_{ij}}\right\}, & \text{radial-only information.} \end{cases} \tag{9}$$

When $\alpha = 0, 1, 2, 3$, full (radial plus angular) information is provided. When $\alpha = 0$, only radial information is used. Physical intuition suggests that covalent bonding interactions, such as bond stretching and bending, and dihedral angle forces, are described by the full coordinate information of the first two neighboring shells in the input data. Longer range interactions like the Van der Waals effects are sufficiently accurately captured by the radial information of more distant neighbors. Therefore, for the sake of efficiency, it is usually enough to take into account the full

[1] https://github.com/deepmodeling/deepmd-kit.

information for the closest neighbors up to a certain neighbor shell but radial-only information for the rest of the neighbors within the cut-off radius. It is noted that the order of the neighbor indexes j's in $\{D_{ij}^{\alpha}\}$ is fixed by sorting them firstly according to their chemical species and then, within each chemical species, according to their inversed distances to atom i, i.e., $1/R_{ij}$. The permutational symmetry is naturally preserved in this way. Following the aforementioned procedures, we have constructed the mapping from atomic positions to descriptors, which is denoted by

$$\boldsymbol{D}_i = \boldsymbol{D}_i(\boldsymbol{R}_i, \{\boldsymbol{R}_j \,|\, j \in N_{R_c}(i)\}). \tag{10}$$

The components $D_{ij}^{\alpha} = D_{ij}^{\alpha}(\boldsymbol{R}_{ij}, \boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)})$ are given by Eqs. (3)–(9). The descriptors $\boldsymbol{D}_i$ preserve the translational, rotational, and permutational symmetries and are passed to a DNN to evaluate the atomic energy. We refer to the Supplementary Materials of Ref. [18] for further details in selection of axis atoms and standardization of input data.

The DNN that maps the descriptors $\boldsymbol{D}_i$ to atomic energy is denoted by

$$E_{s(i)} = \mathcal{N}_{s(i)}(\boldsymbol{D}_i). \tag{11}$$

It is a feedforward network in which data flows from the input layer as $\boldsymbol{D}_i$, through multiple fully connected hidden layers, to the output layer as the atomic energy $E_{s(i)}$. Mathematically, DNN with N_h hidden layers is a mapping

$$\mathcal{N}_{s(i)}(\boldsymbol{D}_i) = \mathcal{L}_{s(i)}^{\text{out}} \circ \mathcal{L}_{s(i)}^{N_h} \circ \mathcal{L}_{s(i)}^{N_h-1} \circ \cdots \circ \mathcal{L}_{s(i)}^{1}(\boldsymbol{D}_i), \tag{12}$$

where the symbol "$\circ$" denotes function composition. Here $\mathcal{L}_{s(i)}^{p}$ is the mapping from layer $p-1$ to p, which is a composition of a linear transformation and a non-linear transformation, the so-called activation function:

$$\boldsymbol{d}_i^p = \mathcal{L}_{s(i)}^{p}(\boldsymbol{d}_i^{p-1}) = \varphi\big(\boldsymbol{W}_{s(i)}^{p}\boldsymbol{d}_i^{p-1} + \boldsymbol{b}_{s(i)}^{p}\big), \tag{13}$$

where $\boldsymbol{d}_i^p \in \mathbb{R}^{M_p}$ denotes the value of neurons in layer p and M_p the number of neurons. The weight matrix $\boldsymbol{W}_{s(i)}^{p} \in \mathbb{R}^{M_p \times M_{p-1}}$ and bias vector $\boldsymbol{b}_{s(i)}^{p} \in \mathbb{R}^{M_p}$ are free parameters of the linear transformation that are to be optimized. The non-linear activation function φ is in general a component-wise function, and here it is taken to be the hyperbolic tangent, i.e.,

$$\varphi(d_1, d_2, \ldots, d_M) = (\tanh(d_1), \tanh(d_2), \ldots, \tanh(d_M)). \tag{14}$$

The output mapping $\mathcal{L}_{s(i)}^{\text{out}}$ is a linear transformation

$$E_{s(i)} = \mathcal{L}_{s(i)}^{\text{out}}(\boldsymbol{d}_i^{N_h}) = \boldsymbol{W}_{s(i)}^{\text{out}}\boldsymbol{d}^{N_h} + b_{s(i)}^{\text{out}}, \tag{15}$$

where weight vector $\boldsymbol{W}_{s(i)}^{\text{out}} \in \mathbb{R}^{1\times M_{N_h}}$ and bias $b_{s(i)}^{\text{out}} \in \mathbb{R}$ are free parameters to be optimized as well.

The force on the ith atom is computed by taking the negative gradient of the system energy with respect to its position, which is given by

$$\begin{aligned}\boldsymbol{F}_i = &-\sum_{j\in N(i),\alpha} \frac{\partial \mathcal{N}_{s(i)}}{\partial D_{ij}^{\alpha}} \frac{\partial D_{ij}^{\alpha}}{\partial \boldsymbol{R}_i} - \sum_{j\neq i} \sum_{k\in N(j),\alpha} \delta_{i,a(j)} \frac{\partial \mathcal{N}_{s(j)}}{\partial D_{jk}^{\alpha}} \frac{\partial D_{jk}^{\alpha}}{\partial \boldsymbol{R}_i} \\ &- \sum_{j\neq i} \sum_{k\in N(j),\alpha} \delta_{i,b(j)} \frac{\partial \mathcal{N}_{s(j)}}{\partial D_{jk}^{\alpha}} \frac{\partial D_{jk}^{\alpha}}{\partial \boldsymbol{R}_i} - \sum_{j\neq i} \sum_{k\in \tilde{N}(j),\alpha} \delta_{i,k} \frac{\partial \mathcal{N}_{s(j)}}{\partial D_{jk}^{\alpha}} \frac{\partial D_{jk}^{\alpha}}{\partial \boldsymbol{R}_i},\end{aligned} \tag{16}$$

where $\tilde{N}(j) = N(j) - \{a(j), b(j)\}$. The virial of the system is given by

$$\begin{aligned}\Xi = &-\sum_{i\neq j} \boldsymbol{R}_{ij} \sum_{\alpha} \frac{\partial \mathcal{N}_{s(i)}}{\partial D_{ij}^{\alpha}} \frac{\partial D_{ij}^{\alpha}}{\partial \boldsymbol{R}_{ij}} - \sum_{i\neq j} \delta_{j,a(i)} \boldsymbol{R}_{ij} \sum_{q,\alpha} \frac{\partial \mathcal{N}_{s(i)}}{\partial D_{iq}^{\alpha}} \frac{\partial D_{iq}^{\alpha}}{\partial \boldsymbol{R}_{ij}} \\ &- \sum_{i\neq j} \delta_{j,b(i)} \boldsymbol{R}_{ij} \sum_{q,\alpha} \frac{\partial \mathcal{N}_{s(i)}}{\partial D_{iq}^{\alpha}} \frac{\partial D_{iq}^{\alpha}}{\partial \boldsymbol{R}_{ij}}.\end{aligned} \tag{17}$$

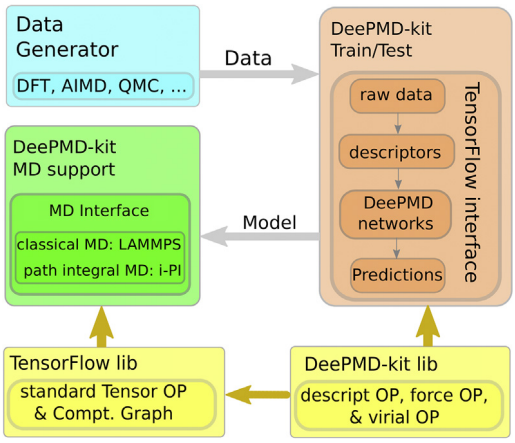

Fig. 1. Schematic plot of the DeePMD-kit architecture and the workflow. The gray arrows present the workflow. The data, including energy, force, virial, box, and type, are passed from the Data Generator to the DeePMD-kit Train/Test module to perform training. After training, the DeePMD model is passed to the DeePMD-kit MD support module to perform MD. The TensorFlow and DeePMD-kit libraries are used for supporting different calculations. See text for detailed descriptions.

The derivation of the force and virial formula Eqs. (16)–(17) is given in Appendix.

The unknown parameters $\{\boldsymbol{W}_s^p, \boldsymbol{b}_s^p\}$ in the linear transformations of the DNN are determined by a training process that minimizes the *loss function* L, i.e.,

$$\min_{\{\boldsymbol{W}_s^p, \boldsymbol{b}_s^p\}} L(p_\epsilon, p_f, p_\xi). \tag{18}$$

The L is defined as a sum of different mean square errors of the DNN predictions

$$L(p_\epsilon, p_f, p_\xi) = \frac{p_\epsilon}{N}\Delta E^2 + \frac{p_f}{3N}\sum_i |\Delta \boldsymbol{F}_i|^2 + \frac{p_\xi}{9N}\|\Delta \Xi\|^2, \tag{19}$$

where ΔE, $\Delta \boldsymbol{F}_i$ and $\Delta \Xi$ denote root mean square (RMS) error in energy, force, and virial, respectively. The prefactors p_ϵ, p_f, and p_ξ are free to change even during the optimization process. In this work, the prefactors are given by

$$p(t) = p^{\text{limit}}\left[1 - \frac{r_l(t)}{r_l^0}\right] + p^{\text{start}}\left[\frac{r_l(t)}{r_l^0}\right], \tag{20}$$

where $r_l(t)$ and r_l^0 are the learning rate at training step t and the learning rate at the beginning, respectively. The prefactor varies from p^{start} at the beginning and goes to p^{limit} as the learning ends. We adopt an exponentially decaying learning rate

$$r_l(t) = r_l^0 \times d_r^{t/d_s}, \tag{21}$$

where d_r and d_s are the decay rate and decay steps, respectively. The decay rate d_r is required to be less than 1.

3. Software

The DeePMD-kit is composed of three parts: (1) a library that implements the computation of descriptors, forces, and virial in C++, including interfaces to TensorFlow and third-party MD packages; (2) training and testing programs built on TensorFlow's Python API; (3) supports for LAMMPS and i-PI. This section illustrates the usage of DeePMD-kit along a typical workflow: preparing data, training the model, testing the model, and running classical/path-integral MD simulations with the model. A schematic plot of the DeePMD-kit architecture and the workflow is shown in Fig. 1.

3.1. Data preparation

The data for training/testing a DeePMD model is composed of a list of *systems*. Each system contains a number of *frames*. Some of the frames are used as training data, while the others are used as testing data. Each frame records the shape of simulation region (box tensor) and the positions of all atoms in the system. The order of the frames in a system is not relevant, but the number of atoms and the atom types should be the same for all frames in the same system. Each frame is labeled with the energy, the forces, and the virial. Any one or two of the labels can be absent. When a label is absent, its corresponding prefactor in the loss function Eq. (19) is set to zero. The labels can be computed by any molecular simulation package that takes in the atomic positions and the box tensor and returns the energy, the forces, and/or the virial. The DeePMD-kit defines a data protocol called RAW format. The labels computed by different packages should be converted to RAW format to serve as training/testing data. The box tensor, atomic coordinates (under lab frame), and the labels are stored in separate text files, with names `box.raw`, `coord.raw`, `energy.raw`, `force.raw`, and `virial.raw`, respectively. Each line of a RAW file corresponds to one frame of the data, with the properties of each atom presented in succession. For example, consider a `coord.raw` file that has two frames of a two-atom system. It has the following content

```
7.726  1.886  4.640  8.998  5.513  11.071
8.229  1.621  4.164  7.110  5.970  10.351
```

The first line stores the atomic coordinates of the first frame, while the second line stores those of the second frame. The coordinate of the first atom in the first frame is (7.726, 1.886, 4.640), while that of the second atom in the first frame is (8.998, 5.513, 11.071). Similarly, the coordinate of the first atom in the second frame is (8.229, 1.621, 4.164), while that of the second atom in the second frame is (7.110 5.970 10.351). The order of the frames appearing in a RAW files and the order of atoms in each frame should be consistent across all the RAW files. The units of length, energy, and force in the RAW files are Å, eV, and eV/Å, respectively. The data is organized in this way because the frames can be combined or split in a convenient way using standard text processing tools such as `cat`, `sed`, and `awk` provided by Unix-like operating systems, and the files can also be manipulated and analyzed as array text data by the NumPy module of Python. The atom types are recorded in the file `type.raw`, which has only one line with atom types as integers presented in succession. Again, it is addressed that the atom types should be consistent in all frames of the same system.

The data is composed of several systems. The RAW files of the different systems should be placed in different folders, and the number of atoms and the atom types are NOT required to be the same for different systems. Frequent loading of the RAW text files from hard disk may become the bottleneck of efficiency. Therefore, the RAW files except the `type.raw` are firstly converted to NumPy binary files and then used by the training and testing programs in DeePMD-kit. DeePMD-kit provides a Python script for this conversion.

3.2. Model training

The computation of atomic energy $E_{s(i)}$ (see Eq. (2)) consists of two successive mappings: first, from the positions of the atom i and its neighbors to its descriptors, i.e., Eq. (10); second, from the descriptors to the atomic energy through DNN, i.e., Eq. (11). The DNN part is implemented by standard tensor operations provided by the TensorFlow deep learning framework. However, the descriptor part is not a standard operation in TensorFlow, thus it is implemented with C++ and is interfaced to TensorFlow as a new "operator". The force and virial computation requires derivatives of system energy with respect to atomic position and box tensor, respectively. This is done with the chain rule in Eqs. (16) and (17), respectively. The gradient of the DNN, i.e., $\partial E_{s(i)}/\partial D^{\alpha}_{jk}$, is implemented by the `tf.gradients` operator provided by TensorFlow. The derivatives $\partial D^{\alpha}_{jk}/\partial \boldsymbol{R}_l$ and the chain rules defined in Eqs. (16) and (17) are implemented in C++ and then interfaced with TensorFlow. By using the TensorFlow with the user implemented operators, we are now able to compute the system energy, the atomic forces, and the virial, thus we are able to evaluate the loss function (forward propagation). The derivatives of the loss function with respect to the parameters $\{\boldsymbol{W}^p_s, \boldsymbol{b}^p_s\}$ (backward propagation) are automatically computed by TensorFlow.

The optimization problem (18) is currently solved by the TensorFlow's implementation of the Adam stochastic gradient descent method [31]. At each step of optimization (equivalent to training step), the value and gradients of the loss function is computed against only a subset of the training data, which is called a *batch*. The number of frames in a batch is called the *batch size*. Taking the RMS energy error ΔE for instance, it is evaluated by

$$\Delta E^2 = \frac{1}{S_b}\sum_{k=1}^{S_b} |E^k - E(\boldsymbol{R}^k_1, \ldots, \boldsymbol{R}^k_N)|^2 \tag{22}$$

where $\{\boldsymbol{R}^k\}$, E^k, and S_b denote the atomic positions, system energy of the kth frame in the batch, and the batch size, respectively. The errors $|\Delta \boldsymbol{F}_i|^2$ and $\|\Delta \Xi\|^2$ are evaluated analogously. It is noted that the evaluation of the loss function for different frames in the batch is embarrassingly parallel. Therefore, ideally, the batch size S_b should be divisible by the number of CPU cores in the computation.

We denote the systems in the training data by $\{\Omega_1, \ldots, \Omega_{S_s}\}$ with S_s being the total number of systems and denote the number of frames in Ω_i by $|\Omega_i|$. The systems $\{\Omega_1, \ldots, \Omega_{S_s}\}$ are used in the training in a cyclic way. First, the model is trained for $|\Omega_1|/S_b$ steps by using $|\Omega_1|/S_b$ batches randomly taken from Ω_1 without replacement. Next, the model is trained for $|\Omega_2|/S_b$ steps by using $|\Omega_2|/S_b$ batches randomly taken from Ω_2 without replacement. In such a way, the systems in the set $\{\Omega_1, \ldots, \Omega_{S_s}\}$ are used in training successively.

The training program in the DeePMD-kit is called `dp_train`. It reads a parameter file in JSON format that specifies the training process. Some important settings in the parameter file are

```
{
"n_neuron":     [240, 120, 60, 30, 10],
"systems":      ["/path/to/water", "/path/to/ice"],
"stop_batch":   1000000,
"batch_size":   4,
"start_lr":     0.001,
"decay_steps":  5000,
"decay_rate":   0.95,
}
```

In this file, the item `n_neuron` sets the number of hidden layers to 5, and the number of neurons in each layer is set to $(M_1, M_2, M_3, M_4, M_5) = (240, 120, 60, 30, 10)$, from the innermost to the outermost layer. The training has two `systems`, with Ω_1 stored in the folder `/path/to/water` and Ω_2 in the folder `/path/to/ice`. The batch size is set to 4. In total the model is optimized for 10^6 steps (set by `stop_batch`), i.e., 10^6 batches are used in the training. The starting learning rate, decay steps, and decay rate (see Eq. (21)) are set to 0.001, 5000, and 0.95, respectively.

The parameters of the DeePMD model are saved to TensorFlow checkpoints during the training process, thus one can break the training at any time and restart it from any of the checkpoints.

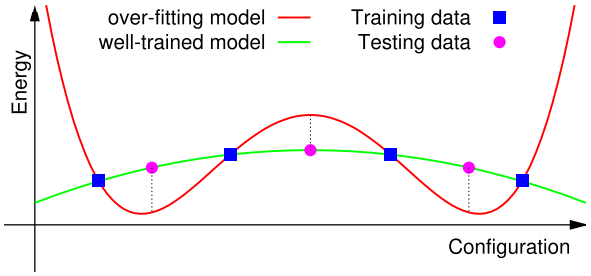

Fig. 2. Schematic illustration of over-fitting. The blue squares denote the training data, while the pink filled circles denote the testing data. Only the training data is used in training models. Both the over-fitting model and the well-trained model have small training error, however, the over-fitting model presents a significantly larger testing error.

Once the training finishes, the model parameters and the network topology are *frozen* from the checkpoint file by the tool `dp_frz`. The frozen model can be used in model testing and MD simulations.

3.3. Model testing

DeePMD-kit provides two modes of model testing. (1) During the training, the RMS energy, force and virial errors and the loss function are evaluated by both the training batch data and the testing data and displayed on the fly. Sometimes, for the sake of efficiency, only a subset of the testing data is used to test the model on the fly. (2) After the model is frozen, it can be tested by the tool `dp_test`. Ideally the training error and the testing error should be roughly the same. A signal of overfitting is indicated by a much lower training error compared to the testing error, see Fig. 2 for an illustration. In this case, it is suggested to either reduce the number of layers and/or the number of neurons of each layer, or increase the size of the training data.

3.4. Molecular dynamics

Once the model parameters are frozen, MD simulations can be carried out. We provide an interface that inputs the atom types and positions and returns energy, forces, and virial computed by the DeePMD model. Therefore, in principle, it can be called in any MD package during MD simulations. In the current release of DeePMD-kit, we provide supports for the LAMMPS and i-PI packages.

The evaluation of interactions is implemented by using the TensorFlow's C++ API. First, the model parameters are loaded, then the network operations defined in the frozen model are executed in exactly the same way as the evaluation in the model training stage, see Section 3.2. The DeePMD-kit's implementation of descriptors, derivatives of descriptors, chain rules for force and virial computations are called as non-standard operators by the TensorFlow.

LAMMPS support.

The LAMMPS support for DeePMD is shipped as a third-party package with the DeePMD-kit source code. The installation of package is similar to other third-party packages for LAMMPS and is explained in detail in the DeePMD-kit manual. In the current release, only serial MD simulations with DeePMD model are supported. To enable the DeePMD model, only two lines are added in the LAMMPS input file.

```
pair_style      deepmd graph.pb
pair_coeff
```

The command `deepmd` in `pair_style` means to use the DeePMD model to compute the atomic interactions in the MD simulations. The parameter `graph.pb` is the file containing the frozen model. The `pair_coeff` should be left blank.

i-PI support.

The i-PI is implemented based on a client–server model. The i-PI works as a server that integrates the trajectories of the nucleus. The DeePMD-kit provides a client called `dp_ipi` that gets coordinates of atoms from the i-PI server and returns the energy, forces, and virial computed by the DeePMD model to the i-PI server. The communication between the server and client is implemented through either the UNIX domain sockets or the Internet sockets. It is noted that multiple instances of the client are allowed, thus the computation of the interactions in multiple path-integral replicas is embarrassingly parallelized. The parameters of running the client are provided by a JSON file. An example for a water system is

```
{
    "verbose":          false,
    "use_unix":         true,
    "port":             31415,
    "host":             "localhost",
    "graph_file":       "graph.pb",
    "coord_file":       "conf.xyz",
    "atom_type" :       {"OW": 0, "HW1": 1, "HW2": 1}
}
```

In this example, the client communicates with the server through the UNIX domain sockets at port 31415. The forces are computed according to the frozen model stored in `graph.pb`. The `conf.xyz` file provides the atomic names and coordinates of the system. The `dp_ipi` ignores the coordinates in `conf.xyz` and translates the atom names to types according to the rule provided by `atom_type`.

4. Example

The performance of the DeePMD-kit package is demonstrated by a bulk liquid water system of 64 molecules subject to periodic boundary conditions.[2] The dataset is generated by a 20 ps, 330 K NVT AIMD simulation with PBE0+TS exchange–correlation functional. The frames are recorded from the trajectory in each time step, i.e., 0.0005 ps. Thus in total we have 40 000 frames. The order of the frames is randomly shuffled. 38 000 of them are used as training data, while the remaining 2000 are used as testing data.

The cut-off radius of neighbor atoms is 6.0 Å. The network input (descriptors) contains both radial and angular information of 16 closest neighboring oxygen atoms and 32 closest neighboring hydrogen atoms, while contains only the radial information of the rest of neighbors.

The first and second neighboring shells of a water molecule have on average 4 and 12 molecules, respectively, and thus our setting roughly considers the angular information of neighbors up to the second neighboring shell. The DNN contains 5 hidden layers. The size of each layer is $(M_1, M_2, M_3, M_4, M_5) = (240, 120, 60, 30, 10)$, from the innermost to the outermost layer. The model is trained by the Adam stochastic gradient descent method, with the learning rate decreasing exponentially. The decay rate and decay step are set to 0.95 and 5000, respectively. The prefactors in the loss function are taken as $p_e^{\text{start}} = 0.02$, $p_e^{\text{limit}} = 8$, $p_f^{\text{start}} = 1000$, and $p_f^{\text{limit}} = 1$. No virial is available in the data, so the virial prefactors are set to 0, i.e., $p_v^{\text{start}} = p_v^{\text{limit}} = 0$.

The model is trained on a desktop machine with an Intel Core i7-3770 CPU and 32 GB memory using 4 OpenMP threads. The total wall time of the training is 16 h. The learning curves of the RMS energy and force errors as functions of training step are plotted in Fig. 3. The errors are tested on the fly by 100 frames randomly

[2] For this example, the raw data and the JSON parameter files for training and MD simulation are provided in the online package. More details on how to use them are explained in the manual.

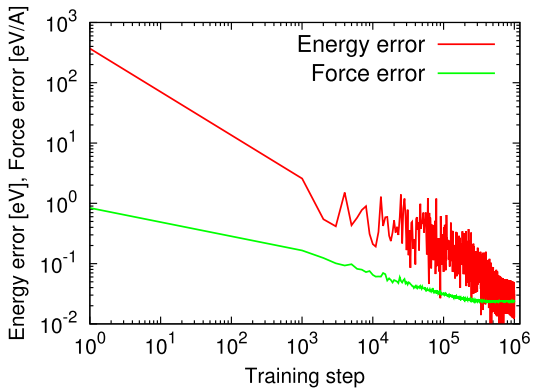

Fig. 3. The learning curves of the liquid water system. The root mean square energy and force testing errors are presented against the training step. The energy error is given in the unit of eV, while the force error is given in unit of eV/Å. The axes of the plot are logscaled.

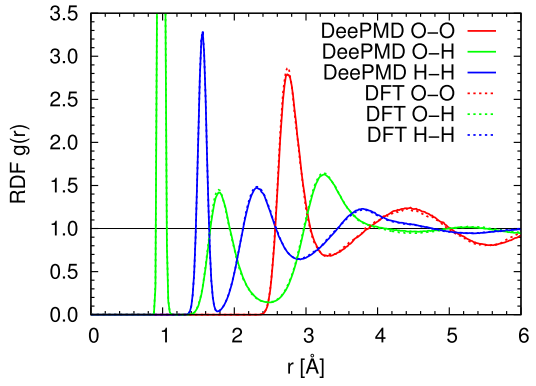

Fig. 4. The radial distribution functions of the DeePMD compared with the PBE0+TS DFT water model.

picked from the testing set. At the beginning, the model parameters $\{\boldsymbol{W}_s^p, \boldsymbol{b}_s^p\}$ are randomly initialized, and the RMS energy and force errors are 3.7×10^2 eV and 8.4×10^{-1} eV/Å, respectively. At the end of training, the RMS energy and force errors over the whole testing set are 2.8×10^{-2} eV and 2.4×10^{-2} eV/Å, respectively. The standard deviation of the energy and the forces in the data are 6.5×10^{-1} eV and 8.1×10^{-1} eV/Å, respectively. Therefore, the relative errors of energy and force with respect to the data standard deviation are 4.3% and 2.9%, respectively.

The trained DeePMD model is frozen and passed to LAMMPS to run NVT MD simulation of 64 water molecules. The simulation cell is of size 12.4447 Å × 12.4447 Å × 12.4447 Å under periodic boundary conditions. The simulation lasts for 200 ps. Snapshots in the first 50 ps are discarded, while the rest snapshots in the trajectory are saved in every other 0.01 ps for structural analysis. The oxygen–oxygen, oxygen–hydrogen, and hydrogen–hydrogen radial distribution functions are presented in Fig. 4. The distribution of the tetrahedral packing parameter [32] is presented in Fig. 5. These results show that the DeePMD model is in satisfactory agreement with the DFT model in generating structure properties.

5. Conclusion and future work

We introduced the software DeePMD-kit, which implements DeePMD, a deep neural network representation for atomic interactions, based on the deep learning framework TensorFlow. The descriptors and chain rules for force/virial computation of DeePMD are implemented in C++ and interfaced to TensorFlow as new operators for model training and PES evaluation. Therefore, the

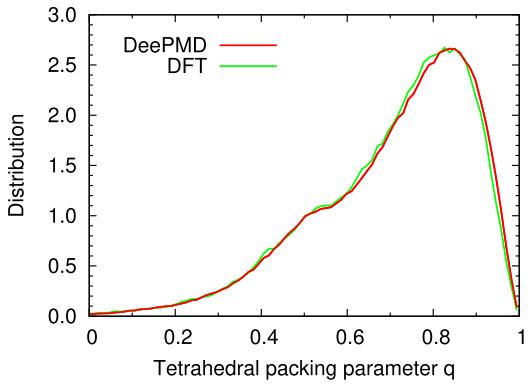

Fig. 5. The distribution of the tetrahedral packing parameter of the DeePMD compared with the PBE0+TS DFT water model.

training, testing, and MD simulations benefit from TensorFlow's state-of-the-art training algorithms and highly optimized tensor operations. Supports for third-party MD packages, LAMMPS and i-PI, are provided such that these softwares can do classical/path-integral MD simulations with the atomic interactions modeled by DeePMD.

In addition, we also provided the analytical details needed to implement the DeePMD method, including the definition of the chemical environment descriptors, the deep neural network architecture, the formula for force and virial calculation, and the definition of the loss function. We explained the RAW data format defined by DeePMD-kit, which provides a protocol for utilizing simulation data generated by other molecular simulation packages and can be easily manipulated by text processing tools in the UNIX-like systems and Python. We provided brief instructions on the model training, testing, and how to set up DeePMD simulation under LAMMPS and i-PI. Finally the accuracy and efficiency of the DeePMD-kit package is illustrated by an example of bulk liquid water system.

The current version of DeePMD-kit only provides CPU implementation of the descriptor computation. In the training stage this computation is embarrassingly parallelized by OpenMP. However, during the evaluation of energy, force, and virial in MD simulations, this computation is not parallelized. In the future we will provide support on the parallel computation of descriptors via CPU multicore and GPU multithreading mechanisms.

Acknowledgments

The work of H. Wang is supported by the National Science Foundation of China under Grants 11501039 and 91530322, the National Key Research and Development Program of China under Grants 2016YFB0201200 and 2016YFB0201203, and the Science Challenge Project No. JCKY2016212A502. The work of L. Zhang, J. Han and W. E is supported in part by ONR grant N00014-13-1-0338, DOE grants DE-SC0008626 and DE-SC0009248, and NSFC grant U1430237. Part of the computational resources is provided by the Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund under Grant No. U1501501.

Appendix. Derivation of force and virial

By using Eqs. (10) and (11), the force of the ith atom is given by

$$\boldsymbol{F}_i = -\frac{\partial}{\partial \boldsymbol{R}_i} \sum_j E_{s(j)}$$

$$= -\sum_{j,k,\alpha} \frac{\partial E_{s(j)}}{\partial D_{jk}^{\alpha}} \frac{\partial D_{jk}^{\alpha}(\boldsymbol{R}_{jk}, \boldsymbol{R}_{ja(j)}, \boldsymbol{R}_{jb(j)})}{\partial \boldsymbol{R}_i}$$

$$
\begin{aligned}
&= -\sum_{k\in N(i),\alpha} \frac{\partial E_{s(i)}}{\partial D^{\alpha}_{ik}} \frac{\partial D^{\alpha}_{ik}(\boldsymbol{R}_{ik}, \boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)})}{\partial \boldsymbol{R}_i} \\
&\quad -\sum_{j\neq i} \sum_{k\in N(j),\alpha} \delta_{i,a(j)} \frac{\partial E_{s(j)}}{\partial D^{\alpha}_{jk}} \frac{\partial D^{\alpha}_{jk}(\boldsymbol{R}_{jk}, \boldsymbol{R}_{ja(j)}, \boldsymbol{R}_{jb(j)})}{\partial \boldsymbol{R}_i} \\
&\quad -\sum_{j\neq i} \sum_{k\in N(j),\alpha} \delta_{i,b(j)} \frac{\partial E_{s(j)}}{\partial D^{\alpha}_{jk}} \frac{\partial D^{\alpha}_{jk}(\boldsymbol{R}_{jk}, \boldsymbol{R}_{ja(j)}, \boldsymbol{R}_{jb(j)})}{\partial \boldsymbol{R}_i} \\
&\quad -\sum_{j\neq i} \sum_{k\in \tilde{N}(j),\alpha} \delta_{i,k} \frac{\partial E_{s(j)}}{\partial D^{\alpha}_{jk}} \frac{\partial D^{\alpha}_{jk}(\boldsymbol{R}_{jk}, \boldsymbol{R}_{ja(j)}, \boldsymbol{R}_{jb(j)})}{\partial \boldsymbol{R}_i}.
\end{aligned}
$$

The virial of the system is given by

$$
\begin{aligned}
\Xi &= \sum_i \boldsymbol{R}_i \boldsymbol{F}_i \\
&= -\sum_i \boldsymbol{R}_i \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_i} - \sum_i \boldsymbol{R}_i \sum_{j\neq i} \frac{\partial E_{s(j)}}{\partial \boldsymbol{R}_i} \\
&= -\sum_i \boldsymbol{R}_i \sum_{j\neq i} \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_{ij}} + \sum_i \boldsymbol{R}_i \sum_{j\neq i} \frac{\partial E_{s(j)}}{\partial \boldsymbol{R}_{ji}} \\
&= -\sum_i \boldsymbol{R}_i \sum_{j\neq i} \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_{ij}} + \sum_j \boldsymbol{R}_j \sum_{i\neq j} \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_{ij}} \\
&= -\sum_{i\neq j} \boldsymbol{R}_{ij} \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_{ij}}.
\end{aligned}
$$

By using Eq. (10) and Eq. (11), it reads

$$
\begin{aligned}
\Xi &= -\sum_{i\neq j} \boldsymbol{R}_{ij} \frac{\partial E_{s(i)}}{\partial \boldsymbol{R}_{ij}} \\
&= -\sum_{i\neq j} \boldsymbol{R}_{ij} \sum_{q,\alpha} \frac{\partial E_{s(i)}}{\partial D^{\alpha}_{iq}} \frac{\partial D^{\alpha}_{iq}(\boldsymbol{R}_{iq}, \boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)})}{\partial \boldsymbol{R}_{ij}} \\
&= -\sum_{i\neq j} \boldsymbol{R}_{ij} \sum_{\alpha} \frac{\partial E_{s(i)}}{\partial D^{\alpha}_{ij}} \frac{\partial D^{\alpha}_{ij}(\boldsymbol{R}_{ij}, \boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)})}{\partial \boldsymbol{R}_{ij}} \\
&\quad -\sum_{i\neq j} \boldsymbol{R}_{ij} \delta_{j,a(i)} \sum_{q,\alpha} \frac{\partial E_{s(i)}}{\partial D^{\alpha}_{iq}} \frac{\partial D^{\alpha}_{iq}(\boldsymbol{R}_{iq}, \boldsymbol{R}_{ia(i)}, \boldsymbol{R}_{ib(i)})}{\partial \boldsymbol{R}_{ij}} \\
&\quad -\sum_{i\neq j} \boldsymbol{R}_{ij} \delta_{j,b(i)} \sum_{q,\alpha} \frac{\partial E_{s(i)}}{\partial D^{\alpha}_{iq}} \frac{\partial D^{\alpha}_{iq}(\boldsymbol{R}_{iq}, \boldsymbol{R}_{ib(i)}, \boldsymbol{R}_{ib(i)})}{\partial \boldsymbol{R}_{ij}}.
\end{aligned}
$$

References

[1] W. Kohn, L.J. Sham, Phys. Rev. 140 (4A) (1965) A1133.
[2] R. Car, M. Parrinello, Phys. Rev. Lett. 55 (22) (1985) 2471.
[3] D. Marx, J. Hutter, Ab Initio Molecular Dynamics: Basic Theory and Advanced Methods, Cambridge University Press, 2009.
[4] K. Vanommeslaeghe, E. Hatcher, C. Acharya, S. Kundu, S. Zhong, J. Shim, E. Darian, O. Guvench, P. Lopes, I. Vorobyov, A. Mackerell Jr., J. Comput. Chem. 31 (4) (2010) 671–690.
[5] W. Jorgensen, D. Maxwell, J. Tirado-Rives, J. Am. Chem. Soc. 118 (45) (1996) 11225–11236.
[6] J. Wang, R.M. Wolf, J.W. Caldwell, P.A. Kollman, D.A. Case, J. Comput. Chem. 25 (9) (2004) 1157–1174.
[7] A.P. Thompson, L.P. Swiler, C.R. Trott, S.M. Foiles, G.J. Tucker, J. Comput. Phys. 285 (2015) 316–330.
[8] T.D. Huan, R. Batra, J. Chapman, S. Krishnan, L. Chen, R. Ramprasad, NPJ Comput. Mater. 3 (2017) 1.
[9] J. Behler, M. Parrinello, Phys. Rev. Lett. 98 (14) (2007) 146401.
[10] T. Morawietz, A. Singraber, C. Dellago, J. Behler, Proc. Natl. Acad. Sci. (2016) 201602375.
[11] A.P. Bartók, M.C. Payne, R. Kondor, G. Csányi, Phys. Rev. Lett. 104 (13) (2010) 136403.
[12] M. Rupp, A. Tkatchenko, K.-R. Müller, O.A. VonLilienfeld, Phys. Rev. Lett. 108 (5) (2012) 058301.
[13] K.T. Schütt, F. Arbabzadah, S. Chmiela, K.R. Müller, A. Tkatchenko, Nature Commun. 8 (2017) 13890.
[14] S. Chmiela, A. Tkatchenko, H.E. Sauceda, I. Poltavsky, K.T. Schütt, K.-R. Müller, Sci. Adv. 3 (5) (2017) e1603015.
[15] J.S. Smith, O. Isayev, A.E. Roitberg, Chem. Sci. 8 (4) (2017) 3192–3203.
[16] K. Yao, J.E. Herr, D.W. Toth, R. Mcintyre, J. Parkhill, The tensormol-0.1 model chemistry: a neural network augmented with long-range physics, 2017, arXiv preprint arXiv:1711.06385.
[17] J. Han, L. Zhang, R. Car, W. E, Commun. Comput. Phys. 23 (3) (2018) 629–639. http://dx.doi.org/10.4208/cicp.OA-2017-0213.
[18] L. Zhang, J. Han, H. Wang, R. Car, W. E, Phys. Rev. Lett. (2017) in press, arXiv preprint arXiv:1707.09571.
[19] Y. LeCun, Y. Bengio, G. Hinton, Nature 521 (7553) (2015) 436–444.
[20] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT press, 2016.
[21] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Nature 529 (7587) (2016) 484–489.
[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., OSDI, vol. 16, 2016, pp. 265–283.
[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, ACM, 2014, pp. 675–678.
[24] R. Collobert, K. Kavukcuoglu, C. Farabet, BigLearn, NIPS Workshop, no. EPFL-CONF-192376, 2011.
[25] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015, arXiv preprint arXiv:1512.01274.
[26] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19.
[27] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, J. Chem. Theory Comput. 4 (3) (2008) 435–447.
[28] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, K. Schulten, J. Comput. Chem. 26 (16) (2005) 1781–1802.
[29] M. Ceriotti, J. More, D.E. Manolopoulos, Comput. Phys. Comm. 185 (3) (2014) 1019–1026.
[30] A.P. Bartók, R. Kondor, G. Csányi, Phys. Rev. B 87 (18) (2013) 184115.
[31] D. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.
[32] J.R. Errington, P.G. Debenedetti, Nature 409 (6818) (2001) 318.