TensorAlloy: a highly efficient atomistic neural network program for alloys

Xin Chen, De-Ye Lin, Hai-Feng Song*

Institute of Applied Physics and Computational Math, Beijing 100088, China and

CAEP Software Center for High Performance Numerical Simulation, Beijing 100088, China

Abstract

Atomistic modeling is important for studying physical and chemical properties of materials. Recently, machine learning interaction potentials have gained much more attentions as they can give density functional theory level predictions within second. Currently, symmetry function based atomistic neural network is the most widely used model for alloys. To precisely describe atomistic interactions, integrating advanced metrics, such as force or virial stress, into training can be of great help. However, the traditional way to construct a machine learning model computation graph starts from pre-computed descriptors, but not positions, due to technial challenges. Thus, deriving analytical atomic force or virial stress of a machine learning model becomes quite difficult as the partial derivations of descriptors with respect to positions need separate implementations, which requires plenty of effort and its efficiency can not be guaranteed. In this paper, we designed a new method, named the virtual atom approach, for implementing the symmetry function descriptor and the corresponding atomistic neural networks. With the virtual atom approach, we are able to build computation graph from atomic positions —but not descriptors —to total energy directly, thus asking modern machine learning platforms to compute atomic force and virial stress becoming possible. We also derived a simple and machine learning friendly equation for calculating virial stress. All these new algorithms have been implemented in our Python package, TensorAlloy. TensorAlloy supports both *molecules* and *solids* and one hour is sufficient for TensorAlloy to obtain models with state-of-art level accuracy.

I. INTRODUCTION

Atomistic simulations (*ab initio* calculation, classic molecular dynamics simulation (MD), etc) are powerful tools for studying physical and chemical properties of materials. To obtain reliable simulation results, atomistic interactions must be properly described. The very accurate *ab initio* and quantum chemical methods have gain significant improvements over the past decades but their applicabilities are restricted by computation expenses. Physical model based empirical potentials, such as the embedded-atom method[1–5], are much more popular for long-time simulation because of their reasonable accuraty and acceptable costs. However, designing empirical potentials and optimizing parameters for alloys are still challenging tasks.

In the past few years, machine learning (ML) has become one of the most influential topic in almost all fields of science. A lot of effort has been made by various physicists to explore the possiblity of using ML models —instead of traditional empirical potentials —to describe atomistic interactions during MD simulations. ML potentials are much easier to design and optimize. With sufficient training data, ML models can achieve similar or even better performances compared with density functional theory. Until now, various ML models (kCON[6], SchNet[7], DTNN[8], SNAP[9, 10], etc) have been published by researchers. Among them, the symmetry function based atomistic neural network (ANN) model, first proposed by Parinello and Behler in 2007[11–15], is still the most widely used method for *solids* and its effectness and accuracy has been proven by several works[16–19]. The Amp[20] package, developed by Peterson's group, is the first open-source framework that implements the symmetry function descriptor. Amp supports both *molecules* and *solids* and it has successfully powered quite a few researches since 2016[21, 22]. ANI[23] and TensorMol[24] are also symmetry function based packages but they mainly focus on *molecules*.

In order to precisely model atomistic interactions, both total energy and atomic forces are necessary. For *solids*, the virial stress tensor is also an essential metric. However, integrating force and stress into the loss function is not an easy task. The machine learning approaches favor vectorizable operations but the calculations of atomic descriptors are typically complicated so that many traditional ML packages (Amp, kCON, etc) choose to pre-compute descriptors and only build computation graph from descriptors to total energy. This approach can significantly reduce technical difficulty and works very well if only total energy

is considered in the loss function. But to fit atomic forces, numerous codes must be written manually, such as the derivation of descriptors with respect to positions. Although the derivation is theoretically clear, the implementation will be a challenge and the efficiency can not be guaranteed.

Recently, machine learning platforms (TensorFlow[25], PyTorch[26], MXNet[27], etc) have gain significant developments. These modern machine learning frameworks can fully take advantage of GPUs for acceleration. The automatic differentiation capability brings a new route for devloping ML models capable of predicting energy, force and stress as we may 'let' ML platforms to handle the calculations of force and stress because these metrics can be derived from positions and total energy directly. To achieve this, a direct computation graph from positions to total energy must be built. Thus, the problem becomes: how to design such route?

In this work, we propose a new algorithm, named the virtual atom approach, to implement the symmetry function descriptor and the ANN model based on TensorFlow. This algorithm is very suitable for parallel execution. With this approach, stochastic gradient descent (SGD) based mini-batch training can be adopted. We also derive a general and vectorizable expression for computing virial stress for arbitrary ANN model. All these algorithms have been implemented in our Python program: **TensorAlloy**.

This paper is organized as follows. Section II introduces the theory of the symmetry function descriptor and the ANN framework. Section III describes the virtual atom approach and a modified ANN model used by **TensorAlloy**. Section IV displays performances of **TensorAlloy** on two public datasets. The appendix V gives derivation and implementation details of key algorithms. The compiled version of **TensorAlloy** program and all the associated model files are available on GitHub.

II. THEORY

In the atomistic neural network (**ANN**) framework, the total energy, E^{total}, of a structure with N atoms, is the sum of all atomic energies:

$$E^{total} = \sum_{i}^{N} E_i \tag{1}$$

where E_i, the energy of atom i, is the output of the neural network $\mathbf{NN}_{el}$ for element el:

$$E_i = \mathbf{NN}_{el}\left(\mathbf{G}_i\right) \tag{2}$$

Here $\mathbf{G}_i$ is a vector representing atomic descriptors of atom i. $\mathbf{G}_i$ typically only depends on local environments —neighbors —of atom i:

$$\mathbf{G}_i = F_G(\{r_{ij\mathbf{n}}, |r_{ij\mathbf{n}}| < r_c\}) \tag{3}$$

where F_G can be arbitrary function, r_c is the cutoff radius and $r_{ij\mathbf{n}}$ is the interatomic distance considering the periodic boundary conditions:

$$r_{ij\mathbf{n}} = \|\mathbf{r}_i^{(0)} - \mathbf{r}_j^{(0)} + \mathbf{n}^T\mathbf{h}\| \tag{4}$$

where $\mathbf{r}_i^{(0)}$ and $\mathbf{r}_j^{(0)}$ denote positions of i and j in the primitive cell, $\mathbf{n}$ is a column vector specifying the cell shifts along (X, Y, Z) directions

$$\mathbf{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \tag{5}$$

and $\mathbf{h}$ is the **row-major** 3×3 lattice tensor:

$$\mathbf{h} = \begin{pmatrix} h_{xx} & h_{xy} & h_{xz} \\ h_{yx} & h_{yy} & h_{yz} \\ h_{zx} & h_{zy} & h_{zz} \end{pmatrix} \tag{6}$$

There are many choices of F_G. The symmetry function descriptor, proposed by Parinello and Behler in 2007[11], is currently the most widely used atomic descriptor, especially for alloys. The symmetry function descriptor consists of two sets of invariant (tranlational, rotational and permutational) functions: the radial symmetry function $\mathbf{G}^{(2)}$ and the angular symmetry function $\mathbf{G}^{(4)}$:

$$\mathbf{G}_i^{(2)}(\eta, R_s) = \sum_{j\neq i}\sum_{\mathbf{n}} g_2(\eta, i, j, R_s) \tag{7}$$

$$\mathbf{G}_i^{(4)}(\beta, \gamma, \zeta) = 2^{1-\zeta}\sum_{j,k\neq j,k\neq i}\sum_{\mathbf{n}_1}\sum_{\mathbf{n}_2}\sum_{\mathbf{n}_3} g_4(\beta, \gamma, \zeta, i, j, k, \mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3) \tag{8}$$

$$g_2 = \exp\left\{-\frac{\eta(r_{ij\mathbf{n}} - R_s)^2}{r_c^2}\right\} \cdot f_c(r_{ij\mathbf{n}}) \tag{9}$$

$$g_4 = (1 + \gamma\cos\theta)^\zeta \exp\left\{-\frac{\beta(r_{ij\mathbf{n_1}}^2 + r_{ik\mathbf{n_2}}^2 + r_{jk\mathbf{n_3}}^2)}{r_c^2}\right\} f_c(r_{ij\mathbf{n_1}}) f_c(r_{ik\mathbf{n_2}}) f_c(r_{jk\mathbf{n_3}}) \tag{10}$$

where η, R_s, β, γ and ζ are empirically chosen parameters, $f_c(r)$ is a cutoff (damping) function and its original form is:

$$f_c(r) = \begin{cases} 0 & r > r_c \\ \frac{1}{2} + \frac{1}{2}\cos\left(r/r_c \cdot \pi\right) & r \leq r_c \end{cases} \tag{11}$$

and r_c is the cutoff radius. In our implementation, R_s is always fixed to 0 so it becomes neglectable[20]. Equation (11) can be further transformed to a vectorized expression:

$$f_c(r) = \frac{1}{2}\left(1 + \cos\left[\min(\frac{r}{r_c}, 1)\pi\right]\right) \tag{12}$$

and $\cos\theta$ also has a vectorized form:

$$\cos\theta = \frac{r^2_{ij\mathbf{n_1}} + r^2_{ik\mathbf{n_2}} - r^2_{jk\mathbf{n_3}}}{2r_{ij\mathbf{n_1}} \cdot r_{ik\mathbf{n_2}}} \tag{13}$$

The summations in $\mathbf{G}^{(2)}$ and $\mathbf{G}^{(4)}$ should go over all neighbors within r_c.

Analytical atomic forces can be obtained directly by computing the first-order derivative of E^{total} with respect to $\mathbf{r}_a^{(0)}$ where a is an index:

$$f_a = -\frac{\partial E^{total}}{\partial \mathbf{r}_a^{(0)}} \tag{14}$$

According to the chain rule, we can expand Equation 14:

$$\begin{aligned} \frac{\partial E^{total}}{\partial \mathbf{r}_a^{(0)}} &= \sum_i^N \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial \mathbf{r}_a^{(0)}} \\ &= \sum_i^N \sum_l^{N_G} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial G_{il}}{\partial \mathbf{r}_a^{(0)}} \end{aligned} \tag{15}$$

and

$$\frac{\partial G^{(2)}_{il}}{\partial \mathbf{r}_a^{(0)}} = \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\partial r_{ij\mathbf{n}}}{\partial r_k^0} \tag{16}$$

$$\frac{\partial G^{(4)}_{il}}{\partial \mathbf{r}_a^{(0)}} = 2^{1-\zeta} \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n}_1} \sum_{\mathbf{n}_2} \sum_{\mathbf{n}_3} \left(\frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \frac{\partial r_{ij\mathbf{n_1}}}{\partial \mathbf{r}_a^{(0)}} + \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \frac{\partial r_{ik\mathbf{n_2}}}{\partial \mathbf{r}_a^{(0)}} + \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \frac{\partial r_{jk\mathbf{n_3}}}{\partial \mathbf{r}_a^{(0)}} \right) \tag{17}$$

The derivations are theoretically straightforward but coding these equations manually will be a challenge[20]. Thanks to the virtual atom approach, **TensorAlloy** can now calculate f_a using Equation 14 directly.

The 3×3 virial stress tensor ϵ is an important metric for solids. For arbitrary many-body interaction potentials, ϵ can be calculated with the following equation[28]:

$$\epsilon = \left(-\sum_{i=1}^{N} \mathbf{r}_i^{(0)} \otimes f_i - \sum_{\mathbf{n}} \mathbf{h}^T \mathbf{n} \otimes \sum_{i=1}^{N} F'_{i\mathbf{n}} \right)^T \tag{18}$$

where $\otimes$ indicates tensor product, f_i is the total force acting on atom i and $F'_{i\mathbf{n}}$ is the partial force. Yntil now, most machine learning force-field models focus on fitting total energy and atomic forces, very few of them[29–31] had attempted to include stress in their loss functions. The current expression of Equation 18 is not very compatible with NN or other ML potentials since the partial forces are difficult to compute with vectorized operations. To resolve this issue, we derived an equivalent form of Equation 18:

$$\epsilon = -F^T R + \left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} \tag{19}$$

where R is the $N \times 3$ positions matrix and F is the corresponding total forces matrix. The new equation can be easily implemented with simple matrix operations. The detailed derivation is given in the appendix.

III. METHOD

In this section we will describe details of **TensorAlloy**: the virtual atom approach, the atomic residual potential and the loss function. The Ni-Mo alloy system, with four η, two β, two γ and one ζ, will be used as the example to show this algorithm. For simplicity, only radial symmetry function related algorithms are visualized in the figures.

A. The virtual atom approach

Fig 1 shows the initialization procedure of the virtual atom approach. The cutoff radius r_{cut} must be fixed. The first step is to go through the entire dataset and determining four prerequisite constants: $N_{ij}^{\max}$, $N_{ijk}^{\max}$, $N_{\mathrm{Ni}}^{\max}$ and $N_{\mathrm{Mo}}^{\max}$. Here $N_{ij}^{\max}$ represents the maximum number of neighbor pairs of arbitrary structure and $N_{ijk}^{\max}$ represents the maximum number of triples. $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ only depends on r_{cut}. The associated codes are given in the appendix. $N_{\mathrm{Ni}}^{\max}$ and $N_{\mathrm{Mo}}^{\max}$ indicate the maximum appearances of Ni and Mo in any structure, respectively. With $N_{\mathrm{Ni}}^{\max}$ and $N_{\mathrm{Mo}}^{\max}$, the global symbol list (GSL) can be created. GSL defines

the global reference of this dataset and must be maintained during entire training. Then we can construct *offsets*. This vector marks the starting indices of the **ordered** interaction pairs (e.g. Ni-Mo) and triples (e.g. Ni-Ni-Mo). The order of the interactions must be maintained as well.

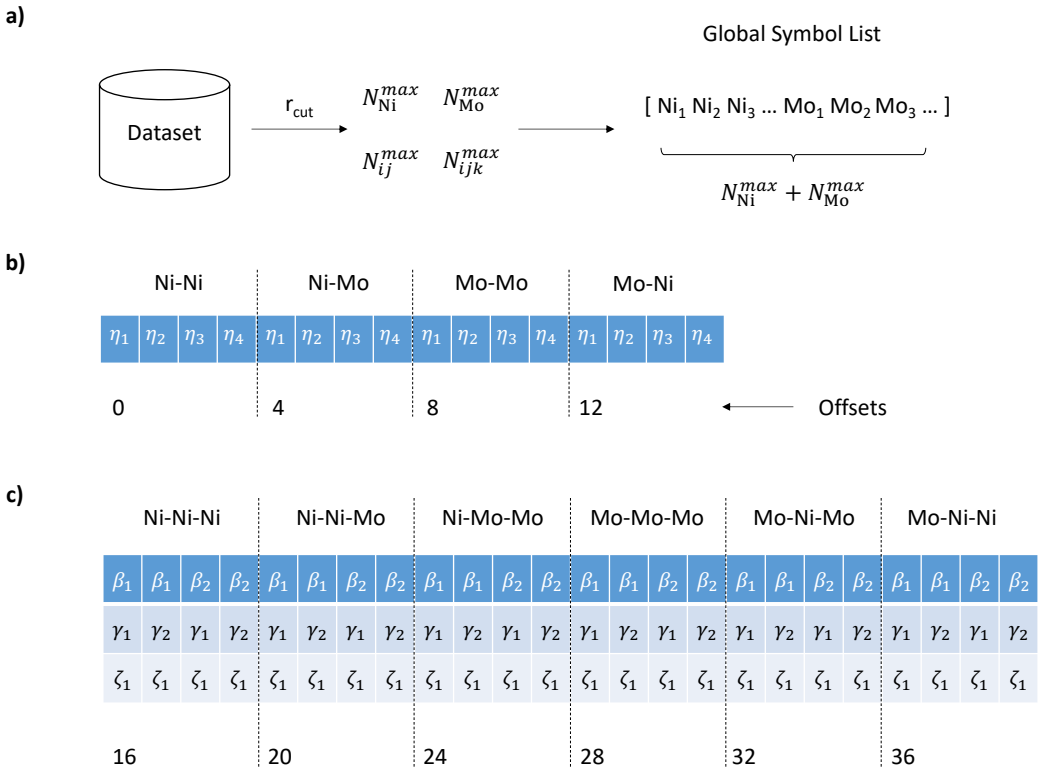

FIG. 1. The initialization. **a)** the detections of $N_{\mathrm{Ni}}^{\mathrm{max}}$, $N_{\mathrm{Ni}}^{\mathrm{max}}$, N_{ij}^{max}, N_{ijk}^{max} and the global symbols list. **b)** and **c)** the *offsets* and their corresponding symmetry function parameters.

Fig 2 shows the algorithm to construct the radial mapping matrix g_{map}^{2} which plays a key role in building the computation graph (Fig 3). This algorithm starts from the $N \times 3$ local positions matrix $\mathbf{R}$. Here 'local' means original and 'global' means GSL. The first step is to find all neighbor pairs: $\mathbf{n}$, $i^{(0)}$ and $j^{(0)}$. This can be done by the neighbor_list routine of ASE[32]. $\mathbf{n}$ is a $N_{ij} \times 3$ matrix representing the periodic boundary shift vectors where N_{ij} is the total number of interaction pairs within r_{cut}. Both $i^{(0)}$ and $j^{(0)}$ are $N_{ij} \times 1$ column vectors representing the indices of i and j of Equation 4. $\mathbf{t}$ is a manually-created $N_{ij} \times 1$ column vector storing the types —represented by integers —of the corresponding interaction pairs.

The second step is to convert the local indices, $i^{(0)}$ and $j^{(0)}$, to global indices based on

GSL and then **increase their values by one**. By padding with zeros, we can finally get the $N_{ij}^{\max} \times 3$ matrix $\mathbf{n_g}$ and $N_{ij}^{\max} \times 1$ column vectors $i_{+,\mathrm{g}}^{(0)}$, $j_{+,\mathrm{g}}^{(0)}$ and $\mathbf{t}_\mathrm{g}$. The increment is extremely important because the indices of all padded 'virtual' or 'dummy' atoms are fixed to 0.

The last step is to construct $g_{\mathrm{map}}^{(2)}$, which should be a $N_{ij}^{\max} \times 2$ matrix. Its first column is just $i_{+,\mathrm{g}}^{(0)}$ and its second column are corresponding offset values of $\mathbf{t}_\mathrm{g}$. $g_{\mathrm{map}}^{(2)}$, $j_{+,\mathrm{g}}^{(0)}$ and $\mathbf{n_g}$ can be pre-computed and cached. The algorithm to build $g_{\mathrm{map}}^{(4)}$ for angular symmetry functions is similar but too complicated to visualize. Thus, we provide a Python implementation in the appendix.

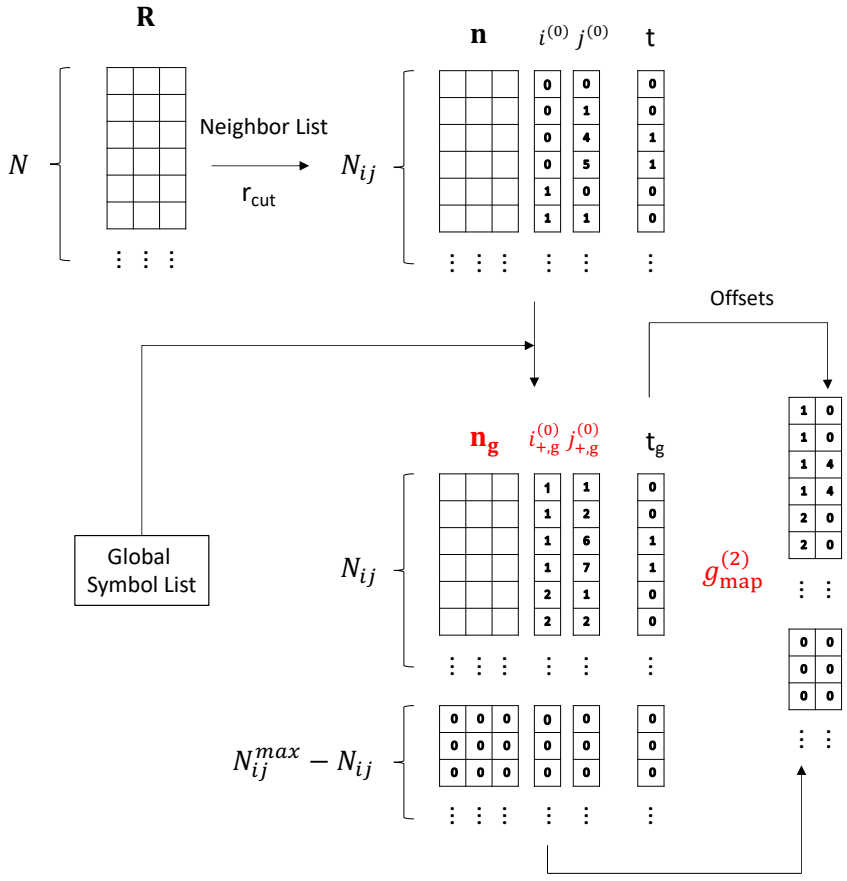

FIG. 2. The second stage of the virtual atom approach: the construction of $g_{\mathrm{map}}^{(2)}$. Tensors marked red can be pre-computed and cached.

Fig 3 shows the final stage of the virtual atom approach: the computation graph for calculating radial symmetry function descriptors. This stage also starts from the $N \times 3$ local positions matrix $\mathbf{R}$ and involves two steps: the calculation of interatomic distances r_{ij} (Fig

3a) and the construction of $\mathbf{G}^{(2)}$ (Fig 3b).

The initial step is to map local positions $\mathbf{R}$ to $(N_{\mathrm{Ni}}^{\max} + N_{\mathrm{Mo}}^{\max} + 1) \times 3$ global positions $\mathbf{R'}$ based on GSL. The first row (marked blue) of $\mathbf{R}'$ are all zeros representing 'positions' of the virtual atom. Then with tf.gather, the global indices $i_{+,\mathrm{g}}^{(0)}$ and $j_{+,\mathrm{g}}^{(0)}$ of Fig 2 can be transformed to global positions: $\mathbf{R}'_{\mathbf{i}}$ and $\mathbf{R}'_{\mathbf{j}}$. Thus, the interatomic distances r_{ij} can be simply calculated with Equation 4.

Now it's time to get radial symmetry function descriptors matrix $\mathbf{G}^{(2)}$. As shown in Fig 3b, this step can be executed in parallel: for each η_i, we first calcualte $g_2(r_{ij}, \eta_i)$ with Equation 9 and then map the $N_{ij}^{\max}$-length vector $g_2(r_{ij}, \eta_i)$ to $(N_{\mathrm{Ni}}^{\max} + N_{\mathrm{Mo}}^{\max} + 1) \times 16$ matrix $\mathbf{G}_{\eta_i}^{(2)}$ with $g_{\mathrm{map}}^{\eta_i}$ and tf.scatter_nd. Here 16 is the multiplacation of 4 (η) and 4 (radial interactions). $g_{\mathrm{map}}^{\eta_i}$ is calculated by adding a constant $i-1$ to all values of the second column of $g_{\mathrm{map}}^{(2)}$. Only shaded columns in Fig 3b have non-zero values. In fact, the first column of $g_{\mathrm{map}}^{\eta_i}$ means the indices of the central atoms in Equation 7 (or rows of $\mathbf{G}^{(2)}$) and the second column marks the corresponding η_i in Fig 1 (or columns of $\mathbf{G}^{(2)}$). The first row of each $\mathbf{G}_{\eta_i}^{(2)}$ reprsents the descriptors of the virtual atom. Removing these rows with tf.split, we get $\hat{\mathbf{G}}_{\eta_i}^{(2)}$. The final radial symmetry function descriptors, $\mathbf{G}^{(2)}$ is just the sum of all $\hat{\mathbf{G}}_{\eta_i}^{(2)}$.

B. The data flow

The following equation briefly demonstrates the core data flow of **TensorAlloy** during a mini-batch training:

$$\mathbf{R}'_{\mathbf{b}} \rightarrow \mathbf{G}_{\mathbf{b}} \rightarrow \mathbf{E}_{\mathbf{b}} \rightarrow \mathbf{F}_{\mathbf{b}} \rightarrow \epsilon_{\mathbf{b}} \rightarrow \mathbf{Loss} \tag{20}$$

where the $N_b \times (N_{\mathrm{Ni}}^{\max} + N_{\mathrm{Mo}}^{\max} + 1) \times 3$ matrix $\mathbf{R}'_{\mathbf{b}}$ is a batch of positions matrices ($N_b \geq 1$ is the batch size), $\mathbf{G}_{\mathbf{b}}$ is the corresponding symmetry function descriptors matrix of shape $N_b \times (N_{\mathrm{Ni}}^{\max} + N_{\mathrm{Mo}}^{\max}) \times 16$ and $\mathbf{E}_{\mathbf{b}}$ represents total energies of the structures. Section III A describes the implementation of $\mathbf{R}'_{\mathbf{b}} \rightarrow \mathbf{G}_{\mathbf{b}}$. To implement $\mathbf{G}_{\mathbf{b}} \rightarrow \mathbf{E}_{\mathbf{b}}$, **TensorAlloy** uses 1D-convolutional neural networks with 1×1 kernels[6]. The total forces $\mathbf{F}_{\mathbf{b}}$ and the virial stress $\epsilon_{\mathbf{b}}$ are calculated with Equation 14 and Equation 19 by with just few lines of codes.

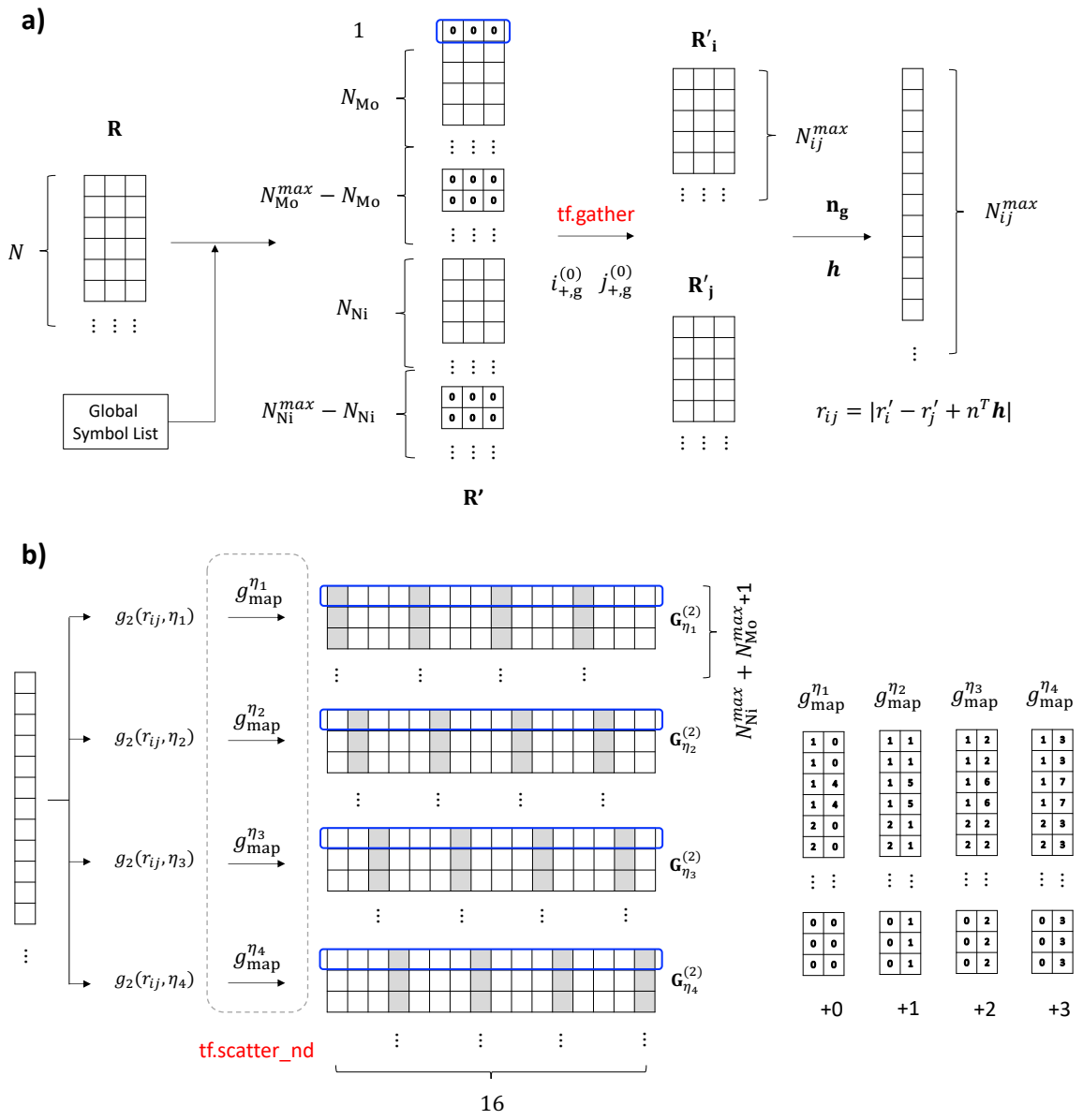

FIG. 3. **a)** The calculation of the interatomic distances r_{ij}. **b)** The construction of the radial symmetry function descriptors $\mathbf{G}_{\eta_i}^{(2)}$. Rows that are marked blue correspond to the virtual atom and will be discared before feeding $\mathbf{G}$ into atomistic neural networks.

C. The residual model

To fit the total energy, we slightly modified the total energy expression of Equation 1:

$$\begin{aligned} E^{total} &= E^{residual} + E^{static} \\ &= \sum_{i=1}^{N} \mathbf{NN}_{el}\left(\{G_i\}\right) + \sum_{el} n_{el} E_{el} \end{aligned} \tag{21}$$

where $\{E_{el}\}$ is a set of *trainable* scalar variables representing the *static* energy of each type of element[6]. The NN in Equation 21 just describes the atomistic interactions (the *residual* energy). The introduction of E^{static} can significantly limit value range of NN outputs, leading to much faster convergence speed and higher training stability (Fig 4c). The initial $\{E_{el}\}$ are calculated by solving the linear system $Ax = b$ where A is a $N_{data} \times N_{el}$ matrix and b is a $N_{data} \times 1$ vector. $A(i,j)$ is the number of j^{th} element in structure i and $b(i)$ is the total energy of structure i.

To train the model, we use the following total loss function:

$$\begin{aligned} \mathbf{Loss} = & \sqrt{\frac{1}{N_b}\sum_{i=1}^{N_b}\left(E_i - E_i^{\mathrm{dft}}\right)^2} \\ & + \chi_{\mathrm{f}}\sqrt{\frac{1}{3\sum_i^{N_b} N_i}\sum_{i}^{N_b}\sum_{j}^{N_i}\sum_{\alpha}\left(f_{ij\alpha} - f_{ij\alpha}^{\mathrm{dft}}\right)^2} \\ & + \chi_{\mathrm{s}}\sqrt{\frac{1}{6N_b}\sum_{i}^{N_b}\sum_{j}^{6}\left(\epsilon_j^{\mathrm{voigt}} - \epsilon_j^{\mathrm{voigt,dft}}\right)^2} \end{aligned} \tag{22}$$

where N_b is the mini-batch size, N_i is the number of atoms of structure i and χ_{f} and χ_{s} are weights of force and stress losses. Typically both χ_{f} and χ_{s} range within $[1, 10]$. Stress tensors are converted to Voigt vectors. All energies are in $\mathbf{eV}$, all forces are in $\mathbf{eV/\AA}$ and all stress components are in $\mathbf{eV/\AA^3}$. In most cases, we set N_b to 50 or 100 and we use the Adam[33] optimizer with exponentially-decayed learning rate to minimize the loss function. The activation function is Leaky ReLU [34]:

$$\sigma(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x \geq 0 \end{cases} \tag{23}$$

and α is set to 0.2 in **TensorAlloy**. The kernel weights of all ANNs are initialized with the Xvaier[35] method and kernel biases are initialized with zeros.

IV. DISCUSSIONS

In this section we demonstrate two training experiments of **TensorAlloy**. All the results are obtained on a workstation with two Intel Xeon E5-2687v4 CPUs (18 cores per CPU, 2.3 GHz) and one NVIDIA GTX 1080 Ti GPU.

A. QM7

QM7[36, 37] is a publicly available benchmark dataset calculated at the DFT level. The QM7 dataset contains 7165 stable organic molecules (C, H, N, O, S) with 176 unique stoichiometries. The total energies range from -95 eV to -17 eV and the structure sizes range from 5 to 23. 1000 structures were randomly selected as test set before training.

Figure 4 demonstrates the performances of **TensorAlloy** on QM7 dataset. In this figure, 'Radial' means only radial symmetry functions (Equation 7) are used and 'Angular' indicates both radial and angular symmetry functions are used. The cutoff radius is 6.5 Åand the corresponding $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ are 506 and 5313, indicating the calculation of angular descriptors is much more expensive. The η for radial symmetry functions are 0.1, 0.5, 1, 2, 4, 8, 12, 16, 20, 40 and β (0.1, 0.5), γ (1, -1) and ζ (1, 4) are used for angular functions. The initial learning rate is 0.01 and the it is exponentially decayed with rate 0.99 for every 5000 steps. Each atomistic neural network has two hidden layers with 128 and 64 neurons.

Figure 4a compares the training speed (number of processed structures per second) with different batch sizes (number of structures per step). It shows that CPU perfers fewer descriptors (radial only) and smallber batch size while GPU is suitable for larger batch size and more complicated descriptors (radial + angular). Figure 4b shows the curves of mean absolute errors (MAEs, meV/atom) on the test set. This figure clearly illustrate another benefit of using larger batch size: models can be trained much faster. For the best case (batch size 100, angular symmetry functions included), the test MAE will reach 5 meV/atom (1.5 kcal/mol per structure) within just one GPU hour.

Figure 4c compares out residual model (Equation 21) with traditional ANN model (Equation 1). In fact, both models can converge to similar final results. However, the residual model has a much faster convergence speed because the introduction of the *static* part can reduce output values of ANNs close to (-1, 1) —the ideal output range of neural networks.

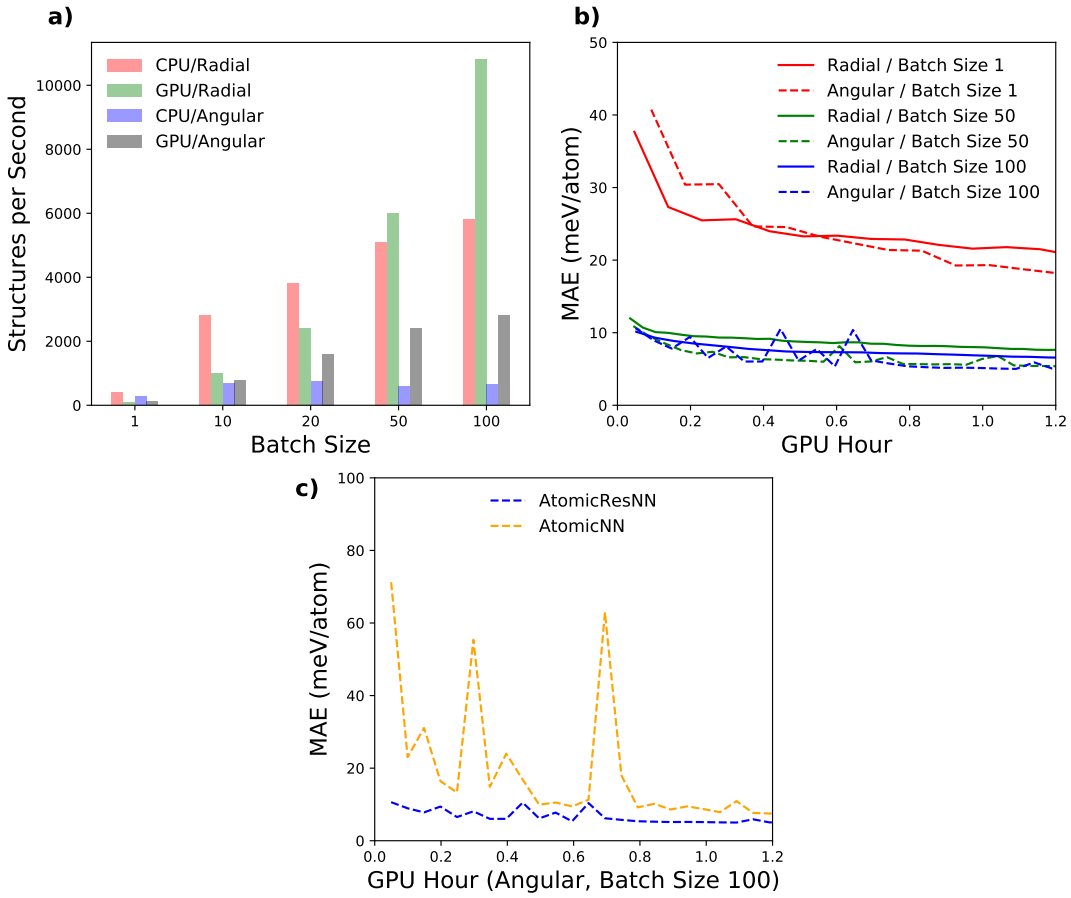

FIG. 4. **a)** The curves of training speed (structures per second) vs batch sizes. **b)** The curves of test MAE (meV/atom) vs training time (GPU hour) using different settings. **c)** The curves of test MAE (meV/atom) vs NN models.

B. Ni-Mo

SNAP/Ni-Mo[9, 10] is a publicly available dataset built by Shyue Ping Ong and co-workers. This dataset has 3971 Ni-Mo solids, including 461 pure Ni structures and 284 pure Mo structures. $N_{\mathrm{Ni}}^{\mathrm{max}}$ is 108 and $N_{\mathrm{Mo}}^{\mathrm{max}}$ equals 176. All DFT calculations were done by VASP[38] using the PBE[39] functional within the projector angumented-wave (PAW)[40] approach.

In this experiment, we trained four different models:

1. SFNN/Ni: this model uses 400 Ni structures for training and 61 for evaluation. The

loss function only includes energy and force. Hidden layer settings are 128,64,32.

2. SFNN/Mo: this model uses 250 Mo structures for training and 34 for evaluation. The loss function includes all three metrics. Hidden layer settings are also 128,64,32.

3. SFNN/Ni-Mo: this model uses 3600 randomly selected structures for training and the rest 373 structures are used for evaluation. The loss function includes only energy and force. Hidden layer settings are 64,32,32.

4. SFNN/Ni-Mo (esf): this model has exactly the same settings with SFNN/Ni-Mo except that its loss function also includes stress as training metrics. χ_s of Equation 22 is 160.

For all these models, the cutoff radius is set to 6.0 Å. The $N_{ij}^{\max}$ for Ni, Mo and Ni-Mo are 4678, 4504 and 7200, respectively. Only radial symmetry functions are used to compute atomic descriptors. The selected η are: 0.1, 0.5, 1, 2, 4, 8, 12, 16, 20 and 40. The learning rate starts from 0.01 and it will decay exponentially with rate 0.99 for every 5000 training steps. The maximum training steps is 500,000.

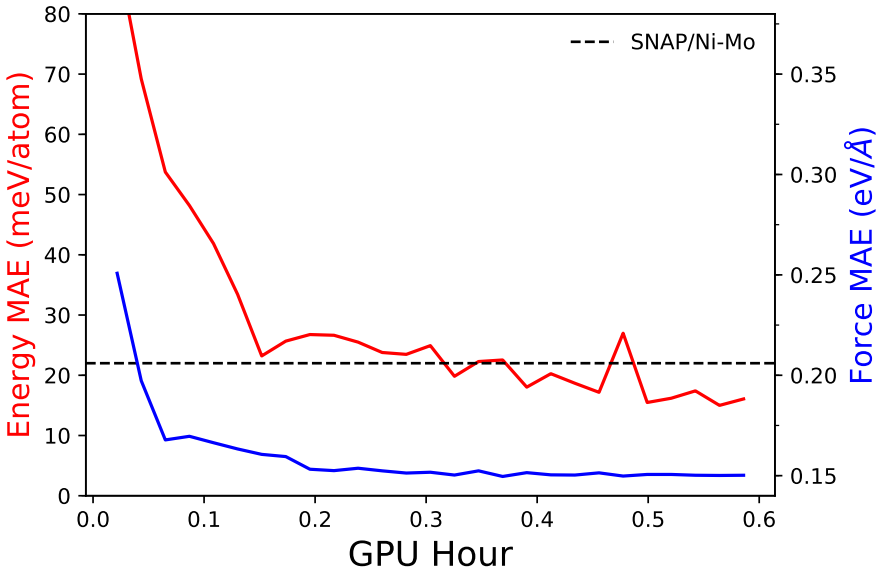

FIG. 5. The test MAEs of the SFNN/Ni-Mo model vs training time (GPU Hour). The black dotted line marks the performance of the SNAP/Ni-Mo model.

Table I summarizes the energy, force and stress prediction performances of **TensorAlloy**-optmized models compared with their corresponding SNAP models. The original SNAP

	Model	Mo	Ni_4Mo	Ni_3Mo	Ni_{Mo}	Mo_{Ni}	Ni	Overall
Energy (meV/atom)	Ni SNAP						**1.2**	
	Mo SNAP	13.2						
	Ni-Mo SNAP	16.2	4.0	5.2	22.7	33.9	7.9	22.5
	Ni SFNN						1.6	
	Mo SFNN	**9.8**						
	Ni-Mo SFNN	22.1	**3.3**	**4.1**	**11.3**	**13.7**	3.7	**11.0**
	Ni-Mo SFNN (esf)	30.0	6.1	9.0	16.6	26.2	7.8	19.1
Force (eV/Å)	Ni SNAP						0.05	
	Mo SNAP	0.25						
	Ni-Mo SNAP	0.29	0.14	0.16	0.13	0.55	0.11	0.23
	Ni SFNN						**0.05**	
	Mo SFNN	**0.20**						
	Ni-Mo SFNN	0.32	**0.09**	0.11	0.09	**0.15**	0.06	0.12
	Ni-Mo SFNN (esf)	0.36	0.10	**0.11**	**0.08**	0.16	0.06	**0.12**
Stress (GPa)	Mo SNAP	**0.87**						
	Mo SFNN	1.00						
	Ni-Mo SFNN	5.77	**1.72**	1.74	1.45	3.75	1.97	2.83
	Ni-Mo SFNN (esf)	**1.84**	2.05	**1.38**	**0.57**	**1.22**	**1.91**	**1.27**

TABLE I. Comparion of the MAEs in predicted energies (mev/atom), forces (eV/Å) and stress (GPa) relative to DFT for the **TensorAlloy** models and their corresponding SNAP models

formalism contains both radial and angular interactions. However, in most cases, the radial-only **TensorAlloy**-trained SFNN models can significantly outperform corresponding SNAP models. To precisely predict virial stress, integrating stress into total loss is undoubtedly necessary. For the pure Mo dataset, the SFNN/Mo model gives a MAE of 1.0 GPa on predicting stress while it will increase to 2 GPa if only using energy and force as training metrics. This phonomenon holds true for the Ni-Mo mixed models as well. If we include stress in the total loss function, we can obtain much more accurate stress predictions.

The trainings of **TensorAlloy** on these *solid* datasets are also very fast. Fig 5 shows the test MAE (meV/atom) vs training time (GPU Hour) of the SFNN/Ni-Mo model. **Ten-**

sorAlloy only needs less than **half an hour** to achieve equivalent performance of the SNAP/Ni-Mo model.

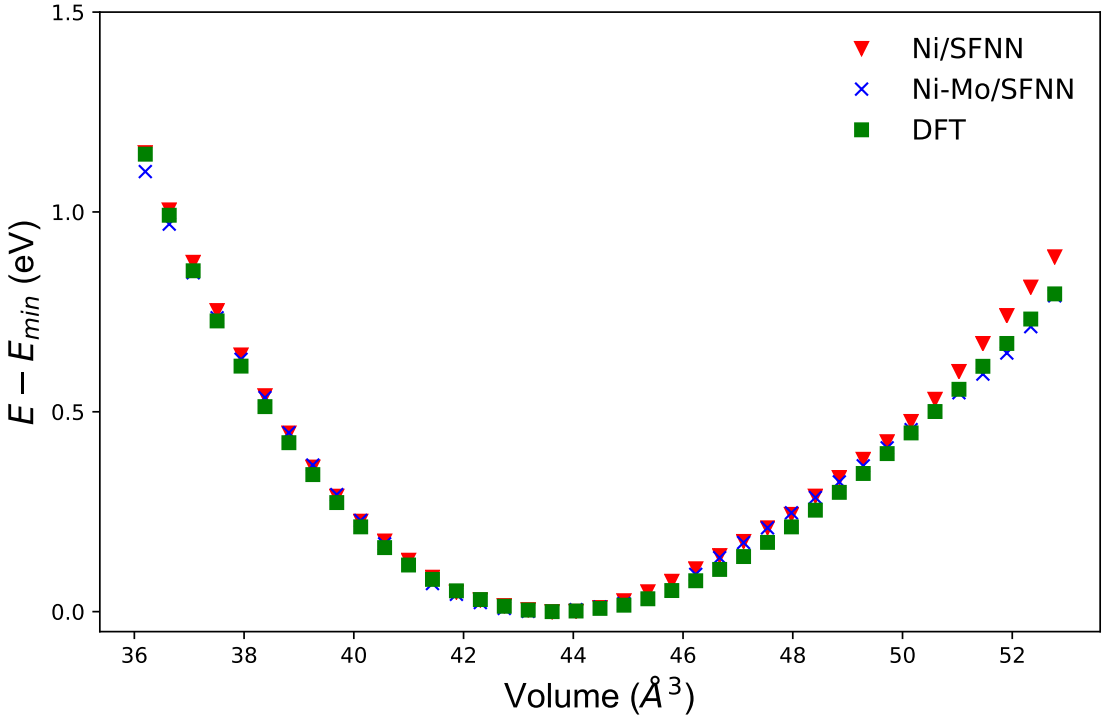

FIG. 6. The energy vs volume curves of fcc Ni for the DFT, Ni/SFNN and Ni-Mo/SFNN models.

To further validate our models, we also calculate the energy-volume curves of a conventional fcc Ni cell using SFNN/Ni and SFNN/Ni-Mo models. The results are plotted in Fig 6. Both curves overlap the DFT curve very well in the range of -17% to 21% from the equilibrium volume. The surface energy tests, as shown in Fig 7, also proves the accuracy of TensorAlloy-optimized models. PyMatGen[41, 42] is used to generate these surface slabs.

V. CONCLUSIONS

In this work, we proposed a new algorithm, namd the virtual atom approach, to construct symmetry function based atomistic neural networks. With this method, we can build direct computation graph from positions to total energy using modern machine learning frameworks (like TensorFlow), thus, making the technical barrier of calculating NN-derived atomic force and virial stress neglectable. We also derived an alternative form of the general virial

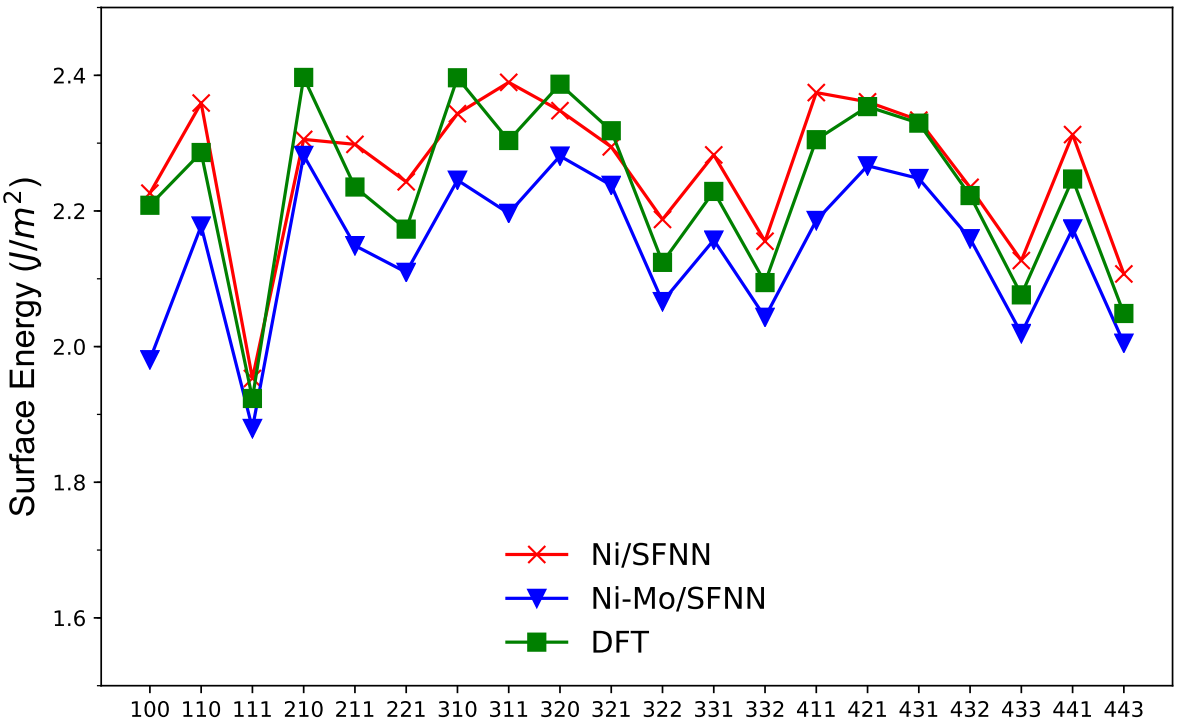

FIG. 7. Surface energies (J/m^2) of different Ni surfaces calculated with DFT, Ni/SFNN and Ni-Mo/SFNN models. The MAEs of the Ni/SFNN and Ni-Mo/SFNN relative to the DFT values are 0.09 and 0.05, respectively.

stress equation. The new formula can be implemented with simple matrix operations and is suitable for arbitrary machine learning platform based potentials. We developed a highly efficient Python program, **TensorAlloy**, for both *molecules* and *solids*. We tested our program on two public datasets, QM7 (molecules) and SNAP/Ni-Mo (solids). **TensorAlloy** can reach state-of-art performances within just one GPU Hour.

ACKNOWLEDGMENTS

This work was supported by. Computing resources were provided by.

[1] M. S. Daw and M. I. Baskes, Phys. Rev. Lett **50**, 1285 (1983).

[2] M. S. Daw and M. I. Baskes, Phys. Rev. B **29**, 6443 (1984).

[3] M. I. Baskes, Phys. Rev. B **46**, 2727 (1992).

[4] X. W. Zhou, R. A. Johnson, and H. N. G. Wadley, Phys. Rev. B **69**, 10.1103/PhysRevB.69.144113 (2004).

[5] B. Jelinek, S. Groh, M. F. Horstemeyer, J. Houze, S. G. Kim, G. J. Wagner, A. Moitra, and M. I. Baskes, Phys. Rev. B **85**, 10.1103/PhysRevB.85.245102 (2012).

[6] X. Chen, M. S. Jorgensen, J. Li, and B. Hammer, J. Chem. Theory Comput. **14**, 3933 (2018).

[7] K. T. Schutt, H. E. Sauceda, P. J. Kindermans, A. Tkatchenko, and K. R. Muller, J. Chem. Phys **148**, 241722 (2018).

[8] K. T. Schutt, F. Arbabzadah, S. Chmiela, K. R. Muller, and A. Tkatchenko, Nat. Commun **8**, 13890 (2017).

[9] C. Chen, Z. Deng, R. Tran, H. Tang, I.-H. Chu, and S. P. Ong, Phys. Rev. M **1**, 10.1103/PhysRevMaterials.1.043603 (2017).

[10] X. G. Li, C. Hu, C. Chen, Z. Deng, J. Luo, and S. P. Ong, Phys. Rev. B. **98**, 094104 (2018).

[11] J. Behler and M. Parrinello, Phys. Rev. Lett **98**, 146401 (2007).

[12] J. Behler, J. Chem. Phys **134**, 074106 (2011).

[13] J. Behler, J. Phys. Condens. Matter **26**, 183001 (2014).

[14] J. Behler, Phys. Chem. Chem. Phys **13**, 17930 (2011).

[15] J. Behler, Int. J. Quantum Chem. **115**, 1032 (2015).

[16] S.-D. Huang, C. Shang, X.-J. Zhang, and Z.-P. Liu, Chem. Sci. **8**, 6327 (2017).

[17] S. Hajinazar, J. Shao, and A. N. Kolmogorov, Phys. Rev. B **95**, 10.1103/PhysRevB.95.014114 (2017).

[18] B. Onat, E. D. Cubuk, B. D. Malone, and E. Kaxiras, Phys. Rev. B **97**, 10.1103/PhysRevB.97.094106 (2018).

[19] R. Kobayashi, D. Giofré, T. Junge, M. Ceriotti, and W. A. Curtin, Phys. Rev. M **1**, 10.1103/PhysRevMaterials.1.053604 (2017).

[20] A. Khorshidi and A. A. Peterson, Comput. Phys. Commun **207**, 310 (2016).

[21] E. L. Kolsbjerg, A. A. Peterson, and B. Hammer, Phys. Rev. B **97**, 10.1103/PhysRevB.97.195424 (2018).

[22] A. A. Peterson, J. Chem. Phys **145**, 0.1063/1.4960708 (2016).

[23] J. S. Smith, O. Isayev, and A. E. Roitberg, Chem. Sci. **8**, 3192 (2017).

[24] K. Yao, J. E. Herr, D. Toth, R. McKintyre, and J. Parkhill, Chem. Sci. **9**, 2261 (2018).

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (USENIX Association, Savannah, GA, 2016) pp. 265–283.

[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS-W* (2017).

[27] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, arXiv e-prints , arXiv:1512.01274 (2015), arXiv:1512.01274 [cs.DC].

[28] A. P. Thompson, S. J. Plimpton, and W. Mattson, J. Chem. Phys **131**, 154107 (2009).

[29] L. Zhang, J. Han, H. Wang, R. Car, and W. E, Phys. Rev. Lett. **120**, 143001 (2018).

[30] L. Zhang, H. Wang, and W. E, J. Chem. Phys **148**, 124113 (2018).

[31] H. Wang, L. Zhang, J. Han, and W. E, Comput. Phys. Commun **228**, 178 (2018).

[32] A. H. Larsen, J. J. 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. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. 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, and K. W. Jacobsen, Journal of Physics: Condensed Matter **29**, 273002 (2017).

[33] D. P. Kingma and J. Ba, CoRR **abs/1412.6980** (2014), arXiv:1412.6980.

[34] A. L. Maas, A. Y. Hannun, and A. Y. Ng, in *Proceedings of the 30th International Conference on Machine Learning* (Atlanta, Georgia, USA, 2013).

[35] X. Glorot and Y. Bengio, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Y. W. Teh and M. Titterington (PMLR, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.

[36] L. C. Blum and J.-L. Reymond, J. Am. Chem. Soc **131**, 8732–8733 (2009).

[37] M. Rupp, A. Tkatchenko, K. R. Muller, and O. A. von Lilienfeld, Phys. Rev. Lett **108**, 058301 (2012).

[38] G. Kresse and J. Furthmüller, Phys. Rev. B **54**, 11169 (1996).

[39] J. P. Perdew, K. Burke, and M. Ernzerhof, Phys. Rev. Lett **77**, 3865 (1996).

[40] P. E. Blöchl, Phys. Rev. B **50**, 17953 (1994).

[41] S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, and G. Ceder, Comput. Mater. Sci **68**, 314 (2013).

[42] W. Sun and G. Ceder, Surf. Sci **617**, 53 (2013).

APPENDIX

A. Calculations of $N_{ij}^{\max}$ and $N_{ijk}^{\max}$

The following codes shows how to calculate $N_{ij}^{\max}$ and $N_{ijk}^{\max}$. Python-3.7 and ASE-3.17.0 are suggested. In this block dataset should be a list of ase.Atoms objects.

```
nij_max = 0
nijk_max = 0
for i in range(dataset):
    atoms = dataset[i]
    assert isinstance(atoms, ase.Atoms)
    ilist, jlist = ase.neighborlist.neighbor_list('ij', atoms, cutoff=rc)
    nij = len(ilist)
    nl = {}
    for i, atomi in enumerate(ilist):
        if atomi not in nl:
            nl[atomi] = []
        nl[atomi].append(jlist[i])
    nijk = 0
    for atomi, nlist in nl.items():
        n = len(nlist)
        nijk += (n - 1 + 1) * (n - 1) // 2
    nij_max = max(nij, nij_max)
    nijk_max = max(nijk, nijk_max)
```

B. Derivation and implementation of the virial stress equation

According to Equation 6 and Equation 5, Equation 4 can be expanded:

$$\begin{aligned} r_{ij\mathbf{n}} =& \|\mathbf{r}_i^{(0)} - \mathbf{r}_j^{(0)} + \mathbf{n}^T\mathbf{h}\| \\ =& \left[\left(r_{j,x}^{(0)} - r_{i,x}^{(0)} + \sum_\alpha n_\alpha h_{\alpha x}\right)^2 + \left(r_{j,y}^{(0)} - r_{i,y}^{(0)} + \sum_\alpha n_\alpha h_{\alpha y}\right)^2 + \right. \\ & \left.\left(r_{j,z}^{(0)} - r_{i,z}^{(0)} + \sum_\alpha n_\alpha h_{\alpha z}\right)^2\right]^{\frac{1}{2}} \end{aligned} \tag{24}$$

where $\alpha = x, y, z$. Thus, we can calculate the derivation of $r_{ij\mathbf{n}}$ with respect to $h_{\alpha\beta}$:

$$\frac{\partial r_{ij\mathbf{n}}}{\partial h_{\alpha\beta}} = \frac{1}{r_{ij\mathbf{n}}} \cdot \Delta_{ij\mathbf{n}\beta} \cdot n_\alpha \tag{25}$$

$$\Delta_{ij\mathbf{n}\beta} = r_{j,\beta}^{(0)} - r_{i,\beta}^{(0)} + \sum_\alpha n_\alpha h_{\alpha\beta} \tag{26}$$

Then we can derive $\partial E^{total}/\partial h_{\alpha\beta}$:

$$\frac{\partial E^{total}}{\partial h_{\alpha\beta}} = \sum_i^N \sum_l^{N_G} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial G_{il}}{\partial h_{\alpha\beta}} \tag{27}$$

If G_{il} is a radial symmetry function:

$$\frac{\partial G_{il}^{(2)}}{\partial h_{\alpha\beta}} = \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{1}{r_{ij\mathbf{n}}} \cdot \Delta_{ij\mathbf{n}\beta} \cdot n_\alpha \tag{28}$$

$$\frac{\partial G_{il}^{(2)}}{\partial h_{\gamma\alpha}} \cdot h_{\gamma\beta} = \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \cdot n_\gamma \cdot h_{\gamma\beta} \tag{29}$$

If G_{il} is an angular symmetry function:

$$\begin{aligned}
\frac{\partial G_{il}^{(4)}}{\partial h_{\alpha\beta}} &= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \left(\frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\partial r_{ij\mathbf{n_1}}}{h_{\alpha\beta}} + \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\partial r_{ik\mathbf{n_2}}}{h_{\alpha\beta}} + \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\partial r_{jk\mathbf{n_3}}}{h_{\alpha\beta}} \right) \\
&= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\Delta_{ij\mathbf{n_1}\beta}}{r_{ij\mathbf{n_1}}} \cdot n_{1,\alpha} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\Delta_{ik\mathbf{n_2}\beta}}{r_{ik\mathbf{n_2}}} \cdot n_{2,\alpha} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\Delta_{jk\mathbf{n_2}\beta}}{r_{jk\mathbf{n_3}}} \cdot n_{3,\alpha} \qquad (30) \\
\frac{\partial G_{il}^{(4)}}{\partial h_{\gamma\alpha}} \cdot h_{\gamma\beta} &= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\Delta_{ij\mathbf{n_1}\alpha}}{r_{ij\mathbf{n_1}}} \cdot n_{1,\gamma} \cdot h_{\gamma\beta} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\Delta_{ik\mathbf{n_1}\alpha}}{r_{ik\mathbf{n_2}}} \cdot n_{2,\gamma} \cdot h_{\gamma\beta} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}}\sum_{\mathbf{n_2}}\sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\Delta_{jk\mathbf{n_1}\alpha}}{r_{jk\mathbf{n_3}}} \cdot n_{3,\gamma} \cdot h_{\gamma\beta} \qquad (31)
\end{aligned}$$

Combining with Equation 27, 29 and 31, we can now start deriving the right part of Equation 19. To simplify the derivation, we just assume G_{il} represents a radial function:

$$\begin{aligned}
\left(\left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} \right)_{\alpha\beta} &= \sum_{\gamma} \frac{\partial E^{total}}{\partial h_{\gamma\alpha}} h_{\gamma\beta} \\
&= \sum_{\gamma}\sum_{i}^{N}\sum_{l}^{N_G} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \sum_{j\neq i}\sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \cdot n_\gamma h_{\gamma\beta} \\
&= \sum_{i}^{N}\sum_{l}^{N_G}\sum_{j\neq i}\sum_{\mathbf{n}} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \sum_{\gamma} \cdot n_\gamma h_{\gamma\beta} \\
&= -\sum_{i}^{N}\sum_{l}^{N_G}\sum_{j\neq i}\sum_{\mathbf{n}} f'_{ij\mathbf{n}} \sum_{\gamma} \cdot n_\gamma h_{\gamma\beta} \qquad (32)
\end{aligned}$$

where $f'_{ij\mathbf{n}}$ is the partial force:

$$f'_{ij\mathbf{n}} = -\frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \qquad (33)$$

If G_{il} is an angular symmetry function, we can also get a similar expression. Thus,

$$\left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} = -\left(\sum_{\mathbf{n}} \mathbf{h}^T \mathbf{n} \otimes \sum_{i=1}^{N} F'_{i\mathbf{n}} \right)^T \qquad (34)$$

The left part of Equation 19 can be calculated with simple matrix multiplacation:

$$\sum_{i=1}^{N} \mathbf{r}_i^{(0)} \otimes f_i = R^T F \tag{35}$$

So finally we can derive the vectorized expression of virial stress:

$$\epsilon = -F^T R + \left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} \tag{36}$$

Now, the virial stress can be calculated with just a few lines of codes within arbitrary Machine Learning framework (TensorFlow, PyTorch, etc). The following codes are written in Python-3.7/TensorFlow-1.12, where *energy* and *volume* are scalar tensors, *cell* is a 3×3 tensor, *positions* is a $N \times 3$ tensor and *forces* (the total forces on these atoms) is also a $N \times 3$ tensor:

```
def get_virial_stress_tensor(energy, cell, volume, positions, forces):
    dEdh = tf.gradients(energy, cell, name='dEdh')[0]
    right = tf.matmul(tf.transpose(dEdh, name='dEdhT'), cell)
    left = tf.matmul(tf.transpose(forces), positions)
    stress = tf.add(tf.negative(left), right, name='stress')
    stress = tf.div(stress, volume, name='virial')
    return stress
```

C. Implementation of the g^2_{map}

This function shows the calculation of g^2_{map}. The returned dict contains g^2_{map}, $i^{(0)}_{+,\mathrm{g}}$, $j^{(0)}_{+,\mathrm{g}}$ and $\mathbf{n_g}$. $N^{\max}_{ij}$ should be provided in advance. interactions is a list of string representing the **ordered** interactions (e.g. ['NiNi','NiMo','MoMo','MoNi']). offsets is a list of integers marking the starting indices of the interactions. As an example, if N_η is 8, the offsets corresponding to ['NiNi','NiMo','MoMo','MoNi'] should be [0, 8, 16, 24]. local_to_global_map is a dict mapping local indices to global indices.

```
def get_g2_map(atoms: Atoms, nij_max: int, interactions: list,
               local_to_global_map: dict, offsets: list):
    g2_map = np.zeros((nij_max, 2), dtype=np.int32)
    tlist = np.zeros(nij_max, dtype=np.int32)
    symbols = atoms.get_chemical_symbols()
    ilist, jlist, Slist = neighbor_list('ijS', atoms, rc)
    nij = len(ilist)
    tlist.fill(0)
    for i in range(nij):
        symboli = symbols[ilist[i]]
        symbolj = symbols[jlist[i]]
        tlist[i] = interactions.index('{}{}'.format(symboli, symbolj))
    ilist = np.resize(ilist + 1, (nij_max, ))
    jlist = np.resize(jlist + 1, (nij_max, ))
    Slist = np.resize(Slist, (nij_max, 3))
    for count in range(len(ilist)):
        if ilist[count] == 0:
            break
        ilist[count] = local_to_global_map[ilist[count]]
        jlist[count] = local_to_global_map[jlist[count]]
    g2_map[:, 0] = ilist
    g2_map[:, 1] = offsets[tlist]
    return {"g2_map": g2_map, "ilist": ilist, "jlist": jlist,
            "shift": np.asarray(Slist)}
```

D. Implementation of the g^4_{map}

The construction of g^4_{map} is similar to g^2_{map}. $N^{\max}_{ijk}$ should be provided in advance. global_to_local_map is the reverse dict of local_to_global_map.

```
def get_g4_map(atoms: Atoms, g2_map: dict, interactions: list,
               offsets: list, global_to_local_map: dict, nijk_max: int):
    g4_map = np.zeros((nijk_max, 2), dtype=np.int32)
    ijk = np.zeros((nijk_max, 3), dtype=np.int32)
    n1 = np.zeros((nijk_max, 3), dtype=np.float32)
    n2 = np.zeros((nijk_max, 3), dtype=np.float32)
    n3 = np.zeros((nijk_max, 3), dtype=np.float32)
    symbols = atoms.get_chemical_symbols()
    indices = {}
    vectors = {}
    for i, atomi in enumerate(g2["ilist"]):
        if atomi == 0:
            break
        if atomi not in indices:
            indices[atomi] = []
            vectors[atomi] = []
        indices[atomi].append(g2["jlist"][i])
        vectors[atomi].append(g2["shift"][i])
    count = 0
    for atomi, nl in indices.items():
        indexi = global_to_local_map[atomi]
        symboli = symbols[indexi]
        prefix = '{}'.format(symboli)
        for j in range(len(nl)):
            atomj = nl[j]
            indexj = global_to_local_map[atomj]
            symbolj = symbols[indexj]
            for k in range(j + 1, len(nl)):
```

```
            atomk = nl[k]
            indexk = global_to_local_map[atomk]
            symbolk = symbols[indexk]
            interaction = '{}{}'.format(
                prefix, ''.join(sorted([symbolj, symbolk])))
            ijk[count] = atomi, atomj, atomk
            n1[count] = vectors[atomi][j]
            n2[count] = vectors[atomi][k]
            n3[count] = vectors[atomi][k] - vectors[atomi][j]
            index = interactions.index(interaction)
            g4_map[count, 0] = atomi
            g4_map[count, 1] = offsets[index]
            count += 1
return {"g4_map": g4_map, "ij": ij, "ik": ik, "jk": jk,
        "n1": n1, "n2": n2, "n3": n3}
```