TensorAlloy: a highly efficient atomistic neural network program for alloys

Xin Chen, Xing-Yu Gao, Ya-Fan Zhao, De-Ye Lin,[*] Wei-Dong Chu, and Hai-Feng Song[†]

Institute of Applied Physics and Computational Mathematics, 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 provide density functional theory level predictions within negligible time. Currently, the symmetry function descriptor based atomistic neural network is the most widely used model for modeling alloys. To precisely describe complex potential energy surfaces, 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 technical challenges. Thus, deriving analytical atomic force and 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 efforts 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 constructing machine learning interaction potentials for both *molecules* and *solids*.

[*] lindeye0716@163.com

[†] song_haifeng@iapcm.ac.cn

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] or its variants (modified embedded-atom method[5–8], angular-dependent interaction potential[9–11], etc), 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[5, 10, 12].

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[12–18]. 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. The symmetry function based atomistic neural network (ANN) model, first proposed by Parinello and Behler in 2007[19–23], is still the most widely used method for *solids* and its effectness and accuracy has been proven by several works[24–27]. Until now, various ML implementations[17, 18, 28] have been published by researchers. The Amp[28] package, developed by Peterson's group, is the first open-source framework that implements the symmetry function descriptor. Amp can be used to learn interaction potentials for both *molecules* and *solids* and it has successfully powered quite a few researches since 2016[29, 30]. ANI[17] and TensorMol[18] 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 is challenging and the efficiency can not be guaranteed.

Recently, machine learning platforms (TensorFlow[31], PyTorch[32], MXNet[33], 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. Implementations of the core algorithms and some trained models can be obtained from GitHub (https://github.com/Bismarrck/vap) freely.

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, i.e. 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[19], 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 $\mathbf{G}^{(4)}$:

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

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

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

$$g_4(\beta, \gamma, \zeta, i, j, k, \mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3) = (1 + \gamma \cos\theta)^{\zeta} \exp\left\{-\frac{\beta(r_{ij\mathbf{n_1}}^2 + r_{ik\mathbf{n_2}}^2 + r_{jk\mathbf{n_3}}^2)}{r_c^2}\right\} \times f_c(r_{ij\mathbf{n_1}}) f_c(r_{ik\mathbf{n_2}}) f_c(r_{jk\mathbf{n_3}}) \tag{10}$$

where η, ω, 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. 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

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

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

The derivations are theoretically straightforward but their implementations are challenging [28]. Later we will provide a new approach to address this problem.

The 3×3 virial stress tensor ϵ is an important metric to describe solids under deformation or external pressure. For arbitrary many-body interaction potentials, ϵ can be calculated with the following equation [34]:

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

where V is the volume, $\otimes$ indicates tensor product, f_i is the total force acting on atom i and $F'_{i\mathbf{n}}$ is the partial force. Until now, most machine learning force-field models focus on fitting total energy and atomic forces, very few of them[35–37] 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:

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

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 the program design and the core algorithm: the virtual atom approach (VAP). The most noticable feature of the TensorAlloy program is that the calculation of symmetry function descriptors is finally implemented in the compuation graph with the help of VAP. Thus, we can utilize the **AutoGrad** routine of TensorFlow to derive atomic forces and virial stress automatically and also efficiently.

The Ni-Mo binary alloy system will be used as the example to show this algorithm. For simplicity, only radial symmetry function related algorithms are visualized in the figures. The associated codes (Python3.7) are given in the appendix. A demo script can be downloaded from GitHub freely (https://github.com/Bismarrck/vap).

Fig 1 shows the general design of TensorAlloy. TensorAlloy has two phases: 1) is the training phase and 2) represents the prediction phase. These two phases have slightly different focusing points:

In the training phase, efficiency is the major concern. To accelerate model training, batching (or mini-batch training) is one of the most commonly used techniques. Batching requires aligned inputs. However, hundreds of different stoichiometries may exist in one dataset. To batch such structures, an universal approach of expressing different structures in a single reference system (the global symbol list or GSL) is necessary.

In the prediction phase, things may be a bit different. The stoichiometry of the input structure may not be included in the training dataset and it can vary significantly. Taking the example of Ni-Mo, the trained model must be capable of predicting any $\mathrm{Ni}_x\mathrm{Mo}_y$ solid while x and y can be either as large as 50000 or just zero. Hence, VAP must be flexible enough to handle these situations.

In the following subsection, we will discuss the details of VAP.

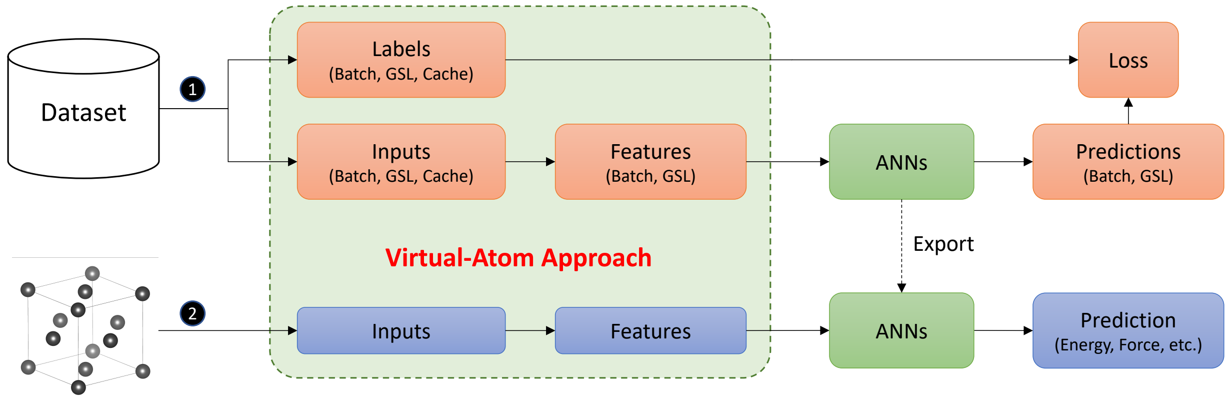

FIG. 1. The general design of TensorAlloy. **(1)** demonstrates the training phase while **(2)** represents the prediction phase.

A. Preparation

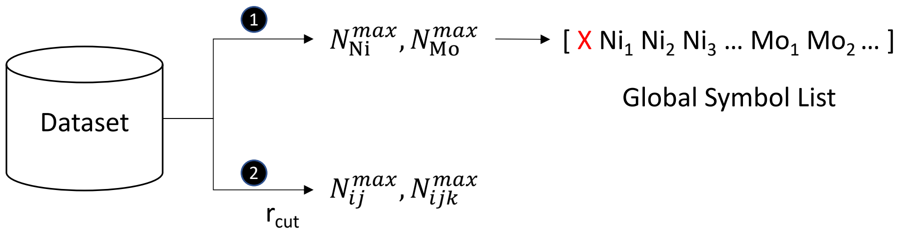
1
$N^{max}_{\mathrm{Ni}}, N^{max}_{\mathrm{Mo}}$
[X Ni$_1$ Ni$_2$ Ni$_3$ ... Mo$_1$ Mo$_2$ ...]
Global Symbol List
Dataset
2
$N^{max}_{ij}, N^{max}_{ijk}$
r$_{cut}$

FIG. 2. The initialization step. **(1)** describes the detections of $N^{\max}_{\mathrm{Ni}}$, $N^{\max}_{\mathrm{Ni}}$ and the global symbol list (GSL). The red 'X' represents the inserted virtual atom. **(2)** shows the detections of $N^{\max}_{ij}$ and $N^{\max}_{ijk}$ given r_{cut}.

Fig 2 shows the initialization of VAP. The cutoff radius r_{cut} must be fixed. The initialization involves two steps. The first step is to build the global symbol list (GSL), which acts as the universal reference system. A virtual atom is inserted at the very first position of this list. The benefits will be discussed later. To build GSL, all $N^{\max}_{\mathrm{el}}$ must be obtained. $N^{\max}_{\mathrm{el}}$

indicates the maximum appearances of an element in any structure and N_{element} represents the total number of unique chemical elements:

$$N^{\text{vap}} = 1 + \sum_{\text{el}} N_{\text{el}}^{\max} \tag{20}$$

For the Ni-Mo dataset, el $\in$ [Ni, Mo] and $N_{\text{element}} = 2$. Table I gives some examples of the GSL mapping. 'X' denotes the inserted virtual atom and $\text{Ni}_{108(54)}$ means the first 54 Ni atoms are mapped from the local stoichiometry. In the training phase, all $N_{\text{el}}^{\max}$ are constant. In the prediction phase, $N_{\text{el}}^{\max}$ shall be determined dynamically. One should note that the minimum value of $N_{\text{el}}^{\max}$ should be 1 due to technical reason (so that the computation graph of TensorFlow can be properly executed). The codes to get $N_{\text{el}}^{\max}$ and GSL are given in Appendix(C). With GSL setup, atomic positions and forces can be transformed as well. Appendix(D) gives the implementation of the virtual-atom mapping mechanism.

	Stoichiometry	$N_{\text{Ni}}^{\max}$	$N_{\text{Mo}}^{\max}$	N^{vap}	Stoichiometry (GSL)
	Ni_{54}				$\text{XNi}_{108(54)}\text{Mo}_{176(0)}$
Training	$\text{Ni}_{54}\text{Mo}_{12}$	108	176	285	$\text{XNi}_{108(54)}\text{Mo}_{176(12)}$
	Mo_{12}				$\text{XNi}_{108(0)}\text{Mo}_{176(12)}$
	Ni_{54}	54	1	56	$\text{XNi}_{54(54)}\text{Mo}_{1(0)}$
Prediction	$\text{Ni}_{54}\text{Mo}_{12}$	54	12	68	$\text{XNi}_{54(54)}\text{Mo}_{12(12)}$
	Mo_{12}	1	12	14	$\text{XNi}_{1(0)}\text{Mo}_{12(12)}$

TABLE I. The mappings of local stoichiometries to their GSL forms, taking examples of the SNAP/Ni-Mo dataset.

Then we can determine another two key constants: $N_{ij}^{\max}$ and $N_{ijk}^{\max}$. Here $N_{ij}^{\max}$ represents the maximum number of neighbor pairs in arbitrary structure and the subscript 'ijk' denotes triples. $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ only depend on r_{cut} and the associated codes are given in Appendix(B). Table II summarizes $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ at different r_{cut} of the two tested datasets: QM7 and SNAP/Ni-Mo.

B. Interatomic distances

Both the radial (Equation 9) and the angular (Equation 10) functions only interact with interatomic distances. In order to compute symmetry function descriptors, local atomic

Dataset	r_{cut}(Å)	$N_{ij}^{\max}$	$N_{ijk}^{\max}$
QM7	6.5	506	5313
SNAP/Ni-Mo	4.6	7200	176400
	6.0	12384	526320
	6.5	16284	965338

TABLE II. $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ of SNAP/Ni-Mo and QM7 at different r_{cut}.

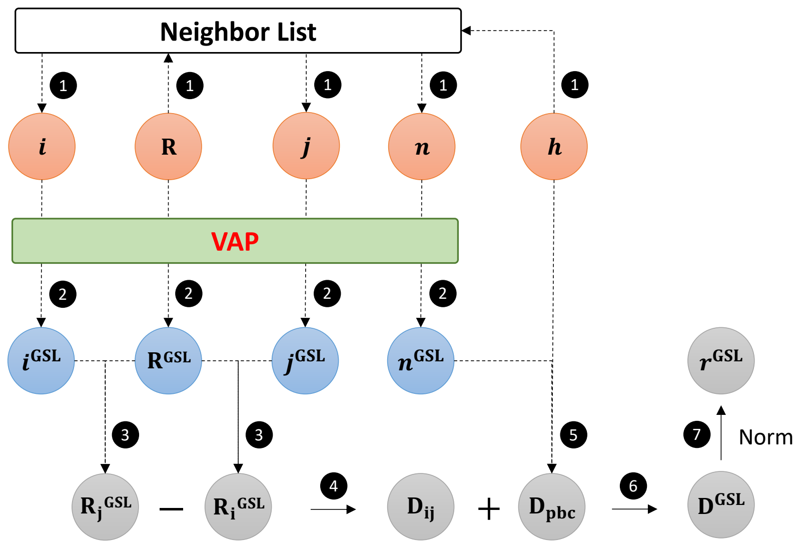

FIG. 3. The pseudo computation graph with execution orders for calculating interatomic distances **r** from atomic positions **R** and lattice tensor **h**.

positions shall be transformed to GSL interatomic distances first. Fig 3 demonstrates the pseudo computation graph for calculating interatomic distances $\mathbf{r}^{\text{GSL}}$ for radial descriptors:

1. Find all (i,j) neighbor pairs using the routine *neighbor_list* implemented in ASE[38]. **R** represents local atomic positions, **h** is the lattice tensor. **i** and **j** are vectors of length N_{ij} and **n** is the $N_{\text{ij}} \times 3$ periodic boundary shift matrix. In the training phase $N_{\text{ij}} \leq N_{ij}^{\max}$; in the prediction phase, $N_{ij}^{\max}$ is exactly equal to N_{ij}.

2. Map **R**, **i**, **j** and **n** to their corresponding GSL forms with padding zeros. One should note that these GSL arrays can be cached for training.

3. Broadcast $\mathbf{R}^{\text{GSL}}$ to $\mathbf{R}_i^{\text{GSL}}$ and $\mathbf{R}_j^{\text{GSL}}$ with $\mathbf{i}^{\text{GSL}}$ and $\mathbf{j}^{\text{GSL}}$ respectively.

4. Compute the displacements in the original cell: $\mathbf{D}_{ij} = \mathbf{j}^{\text{GSL}} - \mathbf{i}^{\text{GSL}}$

5. Compute the total periodic displacements: $\mathbf{D}_{pbc} = \mathbf{n}^{\text{GSL}}\mathbf{h}$

6. Compute the overall GSL displacements: $\mathbf{D}^{\text{GSL}} = \mathbf{D}_{pbc} + \mathbf{D}_{ij}$

7. Compute the interatomic distances $\mathbf{r}^{\text{GSL}}$.

Steps 1-2 are implemented in the function *get_g2_map* in Appendix(E). Steps 3-7 are implemented in the method *get_rij* of class *SymmetryFunction* in Appendix(G).

C. Radial descriptors

Now we can start the last step: computing radial symmetry descriptors $\mathbf{G}^{(2)}$ from $\mathbf{r}^{\text{GSL}}$. For simplicity, we will use the prediction phase to demonstrate this final section.

Suppose we have N_τ different (η, ω) pairs for Equation 7 and $(\eta, \omega)_\tau (0 \leq \tau < N_\tau)$ represents the $(\tau + 1)$-th pair. In the prediction phase, $\mathbf{r}^{\text{GSL}}$ should be a vector of length $N_{ij}^{\max}$ and the desired output $\mathbf{G}^{(2)}$ will be a matrix of shape $[N^{\text{vap}}, (N_{\text{element}})^2 \cdot N_\tau]$, as shown in Fig 5**d**:

1. The total number of rows is N^{vap} (Equation 20) and each row represents the complete radial symmetry function descriptors of an atom. The first row always corresponds to the inserted virtual atom.

2. The total number of columns is $N_{\text{element}} \cdot N_{\text{element}} \cdot N_\tau$. Here $N_{\text{element}} \cdot N_{\text{element}}$ indicates the number of element-element interaction types and each interaction is described by N_τ feature values.

Assume ν is a vector of length N_ν, we can define an integer matrix f_ν^S for mapping ν to a new array g of shape S satisfying the following conditions:

$$g(p) = \sum_{\tau=0}^{\tau<N_\nu} c_\tau(p) \cdot \nu(\tau) \tag{21}$$

$$c_\tau(p) = \begin{cases} 1 & {f^S}_\nu(\tau) = p \\ 0 & \text{else} \end{cases} \tag{22}$$

where p is a coordinate vector. The matrix f_ν^S shall have N_ν rows and N_S columns where N_S represents the dimension of g. Fig 4 visualizes this f_ν^S mapping scheme. This f_ν^S mapping plays a key role in transforming $\mathbf{r}^{\text{GSL}}$ to $\mathbf{G}^{(2)}$.

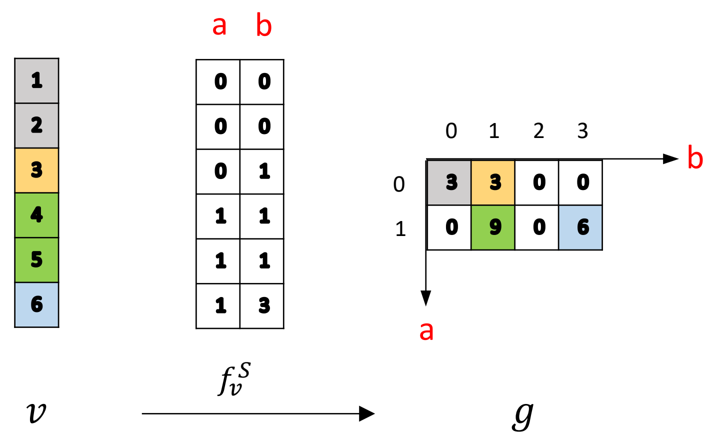

FIG. 4. The visualization of the mapping f_v^S. In this example $S = [2, 4]$ and p can be any (i, j) if $0 \le i < 2$ and $0 \le j < 4$.

Fig 5 demonstrates the calculation of $\mathbf{G}^{(2)}((\eta,\omega)_\tau)$ from $\mathbf{r}^{\mathrm{GSL}}$. $\mathbf{G}^{(2)}((\eta,\omega)_\tau)$ is the partial descriptors contributed by the τ-th (η,ω) combination. The overall $\mathbf{G}^{(2)}$ is just the sum of all partial contributions:

$$\mathbf{G}^{(2)} = \sum_{\tau=0}^{\tau<N_\tau} \mathbf{G}^{(2)}((\eta,\omega)_\tau) \tag{23}$$

This calculation can be easily implemented with a *For* loop. For each τ, there will be three major steps:

1. The first step is applying Equation 9 on $\mathbf{r}^{\mathrm{GSL}}$ with the given τ-th (η,ω) pair to obtain raw descriptors $g_2^{\mathrm{GSL}}((\eta,\omega)_\tau)$. $g_2^{\mathrm{GSL}}((\eta,\omega)_\tau)$ is a vector of length $N_{ij}^{\max}$ and its last $(N_{ij}^{\max} - N_{ij})$ components are dummy values.

2. The second step is generating the mapping $f_{g,\tau}^S$. In the Ni-Mo case, $S = [285, 4N_\tau]$. $f_{g,\tau}^S$ is a matrix of shape $[N_{ij}^{\max}, 2]$ and $f_{g,\tau}^S$ is constructed from the base mapping array f_g^S by adding τ to its second column. f_g^S is also a $N_{ij}^{\max} \times 2$ matrix. f_g^S stores fundamental properties of $\mathbf{r}^{\mathrm{GSL}}$: the center atom (e.g. Ni_1) and type (e.g. Ni-Mo) of each atom-atom interaction:

   a. Its first column is directly copied i^{GSL}, representing the GSL indices of the center atoms.

   b. The values of its second column are multiples of N_τ indicating the offsets, or horizontal positions in $\mathbf{G}^{(2)}$. These values are obtained by multiplying N_τ with

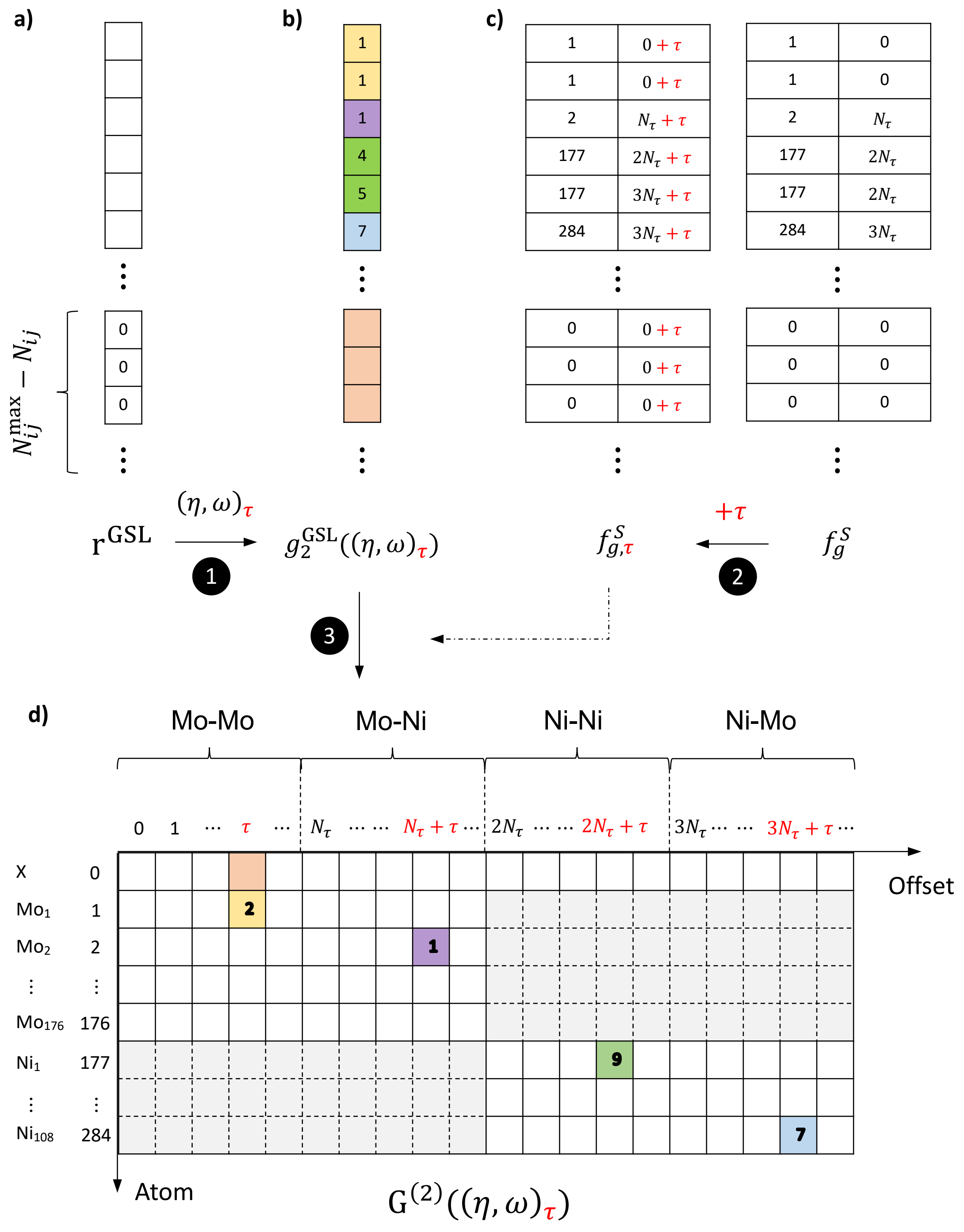

FIG. 5. The definition of $f_\nu^{(2,4)}$: map the vector ν to array g of shape [2, 4], or mathematically $g(i,j) = \sum_k^{f_\nu^{(2,4)}(k)=(i,j)} \nu(k)$.

the type indices of $\mathbf{r}^{\mathrm{GSL}}$. For example, the Ni-Mo dataset has two unique elements: Ni, Mo. Hence, there will be four types of interactions: Mo-Mo (0), Mo-Ni (1), Ni-Ni (2) and Ni-Mo (3). Then for each value (atom-atom interaction) of $\mathbf{r}^{\mathrm{GSL}}$, its type index can be determined with $\mathbf{i}^{\mathrm{GSL}}$ and $\mathbf{j}^{\mathrm{GSL}}$ based on the pre-built GSL.

3. Compute $\mathbf{G}^{(2)}((\eta, \omega)_\tau)$ by applying $f^S_{g,\tau}$ on $g^{\mathrm{GSL}}_2((\eta, \omega)_\tau)$. The padded zeros in $\mathbf{r}^{\mathrm{GSL}}$ are finally mapped to the inserted virtual atom. Hence, the zero-padding of $\mathbf{r}^{\mathrm{GSL}}$ can be easily separated and elinimated (just remove the first row of $\mathbf{G}^{(2)}$). That's why TensorAlloy can use arbitrary number of structures without any stoichiometry restriction to do batch training.

The mapping scheme f^S_ν can be directly achieved using routine *tf.scatter_nd* implemented by TensorFlow. The calculation of the base mapping array f^S_g is implemented in Appendix(E). f^S_g is also cacheable. The calculation of $\mathbf{G}^{(2)}((\eta, \omega)_\tau)$ is implemented in method *get_g2_op_for_tau* of class *SymmetryFunction* in Appendix(G).

The workflow for the batch training phase is almost identical to the prediction phase introduced above. There are still some noticable diiferences. Assume N_b represents the batch size:

1. Most of the variables ($\mathbf{r}^{\mathrm{GSL}}$, $\mathbf{i}^{\mathrm{GSL}}$, $\mathbf{R}^{\mathrm{GSL}}$, etc.) have one more dimension. For example, $\mathbf{R}^{\mathrm{GSL}}$ should be an array of shape $[N_b, N^{\max}_{ij}, 3]$.

2. The base mapping f^S_g becomes a matrix of shape $[N^{\max}_{ij}, 3]$. Its last two columns remain unchanged. However, the newly inserted first column —representing the batch indices —is **not cacheable** because batching is a random action. The first column should be generated dynamically.

3. τ should be added to the third column of f^S_g.

4. In the training phase, N^{vap} is a constant determined before training. In the contrast, N^{vap} becomes a tensor that should be manually specified in the prediction phase.

5. In the training phase, the reference atomic forces should be transformed to its corresponding GSL array because the NN-derived force array $\mathbf{F}^{\mathrm{NN}}$ is in GSL reference:

$$\mathbf{F}^{\mathrm{NN}} = -\frac{\partial E}{\partial \mathbf{R}^{\mathrm{GSL}}} \tag{24}$$

Hence, in the prediction phase, a reverse VAP mapping is required to convert $\mathbf{F}^{\mathrm{NN}}$ to its local form.

The training phase is implemented in *BatchSymmetryFunction* which inherits most of methods from its parent class *SymmetryFunction*. The dynamic batching is implemented in method *get_v2g_map_batch_indexing_matrix*. The method *get_g_shape* will return a fixed array instead. The function *map_gsl_array_to_local* of *VirtualAtomMap* is used to do the reverse mapping.

D. Angular descriptors

The same algorithm can be applied to compute angular symmetry function descriptors. The triple list shall be constructed from the neighbor list (Fig 3) and $N_{ij}^{\max}$ shall be replaced by $N_{ijk}^{\max}$. Typically $N_{ijk}^{\max} \gg N_{ij}^{\max}$.

Appendix(F) demonstrates how to compute the base mapping array. The calculation of $\mathbf{G}^{(4)}(\beta, \gamma, \zeta)_\tau$ is implemented in method *get_g4_op_for_tau* of class *SymmetryFunction* in Appendix(G).

E. Min-max normalization

Raw values of $\mathbf{G}^{(2)}$ and $\mathbf{G}^{(4)}$ may range from zero to several hundreds. So a min-max normalization routine is recommended. For each column $\mathbf{G}(:, \tau)$ where $0 \leq \tau < N_\tau$, we just normalize its values to range [0, 1]:

$$\mathbf{G}(:, \tau) = \frac{\mathbf{G}(:, \tau) - \min(\mathbf{G}(:, \tau))}{\max(\mathbf{G}(:, \tau)) - \min(\mathbf{G}(:, \tau))} \tag{25}$$

where $\min(\mathbf{G}(:, \tau))$ and $\max(\mathbf{G}(:, \tau))$ should be determined in the training phase and remain constant in the prediction phase.

F. The residual model

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

$$\begin{aligned} E^{total} &= E^{residual} + E^{static} \\ &= \sum_{\text{el}} \mathbf{NN}_{el}\left(\mathbf{G}_{\text{el}}\right) + \sum_{\text{el}} n_{el} E_{el} \end{aligned} \tag{26}$$

where $\{E_{el}\}$ is a set of *trainable* scalar variables representing the *static* energy of each type of element[13]. The NN in Equation 26 just describes the atomistic interactions (the *residual* energy). The introduction of E^{static} can significantly limit value range of NN outputs, leading to faster convergence speed and higher training stability (Fig 6c). The initial $\{E_{el}\}$ are calculated by solving the linear system $Ax = b$ where A is a $N_{data} \times N_{\mathrm{element}}$ matrix and b is a vector of length N_{data}. N_{data} is the number of training examples. $A(i, j)$ is the number of j-th element in structure i and $b(i)$ is the corresponding total energy.

To train the model, we use the following root mean-squared error (RMSE) total loss function:

$$\begin{aligned}
\mathbf{Loss} = & \sqrt{\frac{1}{N_b}\sum_{i=1}^{N_b}\left(E_i - E_i^{\mathrm{dft}}\right)^2} \\
& + \chi_{\mathrm{f}}\sqrt{\frac{1}{3N_bN^{\mathrm{vap}}}\sum_{i}^{N_b}\sum_{j}^{N^{\mathrm{vap}}}\sum_{\alpha}\left(f_{ij\alpha} - f_{ij\alpha}^{\mathrm{dft}}\right)^2} \\
& + \chi_{\mathrm{s}}\sqrt{\frac{1}{6N_b}\sum_{i}^{N_b}\sum_{j}^{6}\left(\epsilon_j^{\mathrm{voigt}} - \epsilon_j^{\mathrm{voigt,dft}}\right)^2}
\end{aligned} \tag{27}$$

where N_b is the mini-batch size and χ_{f} and χ_{s} are weights of force and stress losses. Stress tensors are converted to Voigt vectors. All energies are in **eV**, all forces are in **eV/Å** and all stress components are in $\mathbf{eV/Å^3}$. In most cases, we set N_b to 50 or 100 and we use the ADAM[39] optimizer with exponentially-decayed learning rate to minimize the loss function. The default activation function is softplus:

$$\sigma(x) = \log(1 + e^x) \tag{28}$$

The kernel weights of all NNs are initialized with the Xvaier[40] method and kernel biases are initialized with zeros.

IV. DISCUSSIONS

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

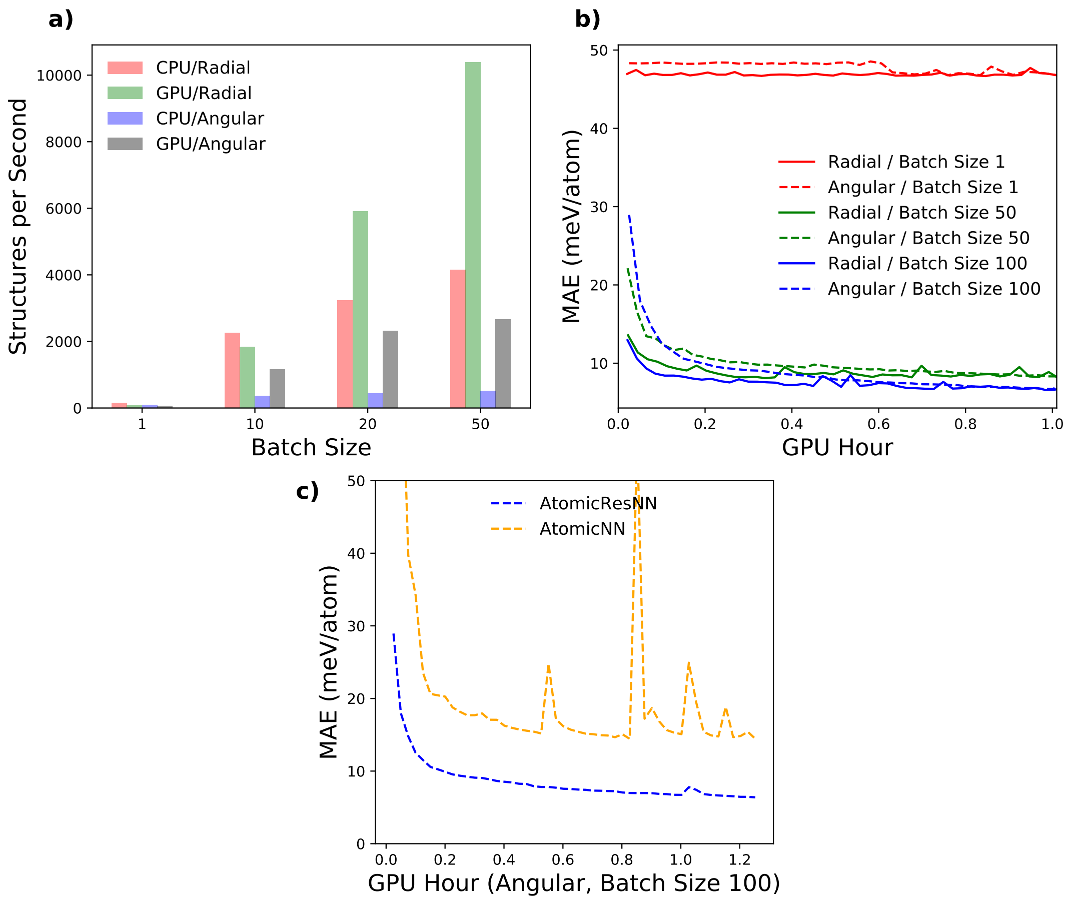

FIG. 6. **a)** The curves of training speed (structures per second) vs batch sizes. **b)** The curves of test MAE (meV/atom) vs training time (GPU hour) using different settings. **c)** The curves of test MAE (meV/atom) vs NN models.

A. QM7

QM7[41, 42] 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 6 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 Å.

The corresponding $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ are 506 and 5313, respectively. Values of 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. ω is fixed to 0. The initial learning rate is 0.01. Learning rate decay is disabled for the following tests. Min-max normalization is enabled. Each atomistic neural network has two hidden layers with 64 and 32 neurons. The force and stress losses of Equation 27 are disabled for this experiment.

Figure 6a 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 6b 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: the trainings can be 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) in just one GPU hour.

Figure 6c compares out residual model (Equation 26) with traditional ANN model (Equation 1). The proposed residual model has a much faster convergence rate and better stability. One exaplanation is that the *static* part acts as a normalization function. By introducing the *static* term, the output values of ANNs are much closer to the ideal output range (-1, 1) of neural networks.

B. Ni-Mo

SNAP/Ni-Mo[12, 16] is a publicly available dataset built by Shyue Ping Ong and coworkers. This dataset has 3971 Ni-Mo solids, including 461 pure Ni structures and 284 pure Mo structures. All DFT calculations were done by VASP[43] using the PBE[44] functional within the projector angumented-wave (PAW)[45] approach.

In this experiment, we trained three different models:

1. Ni: this model uses 400 Ni structures for training and 61 for evaluation. The loss function only includes energy and force contributions.

2. Mo: this model uses 250 Mo structures for training and 34 for evaluation. The loss function includes all three (energy, force, stress) types of contributions.

3. Ni-Mo: this model uses 3673 structures for training and 300 for evaluation. The loss function includes all three types of contributions.

For all these models, the cutoff radius is set to 6.5 Å. 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, 40 and ω are 0.0, 3.0. The hidden layer sizes are 128, 64, 32. The activation function is softplus. The learning rate starts from 0.01 and it will decay exponentially with rate 0.95 for every 3000 steps.

	Energy (meV/atom)			Force (eV/Å)			Stress (GPa)	
Model	Mo	Ni	Ni-Mo	Mo	Ni	Ni-Mo	Mo	Ni-Mo
Mo	4.9 (13.2)		16.5 (16.2)	0.19 (0.25)		0.28 (0.29)	0.24 (0.87)	0.75
Ni		1.2 (1.2)	4.5 (7.9)		0.04 (0.05)	0.07 (0.11)		0.83
Ni_4Mo			4.1 (4.0)			0.09 (0.14)		0.98
Ni_3Mo			4.5 (5.2)			0.11 (0.16)		1.19
Ni_{Mo}			12.9 (22.7)			0.09 (0.13)		0.34
Mo_{Ni}			12.9 (33.9)			0.12 (0.55)		0.45
Overall			10.8 (22.5)			0.11 (0.23)		0.59

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

Table III summarizes the energy, force and stress prediction performances of TensorAlloy-trained models compared with their corresponding SNAP benchmarks. The original SNAP formalism contains both radial and angular interactions. However, in most cases, the radial-only TensorAlloy-trained models can significantly outperform corresponding SNAP models. The trainings of these models are also very fast. All these models are trained within 2 GPU hours on our workstation.

The speed of the prediction phase is also a crucial metric for measuring interaction potentials. Fig 7 summarizes the time distribution of the Ni-Mo model in the prediction phase on the GPU workstation. Here 'prediction' is defined as calculating energy, forces and stress tensor of an arbitrary *ase.Atoms* object. In these tests, the primitive structure MoNi is obtained by replacing one of the two bcc Mo atoms with a Ni atom. The test subjects $(MoNi))_x$

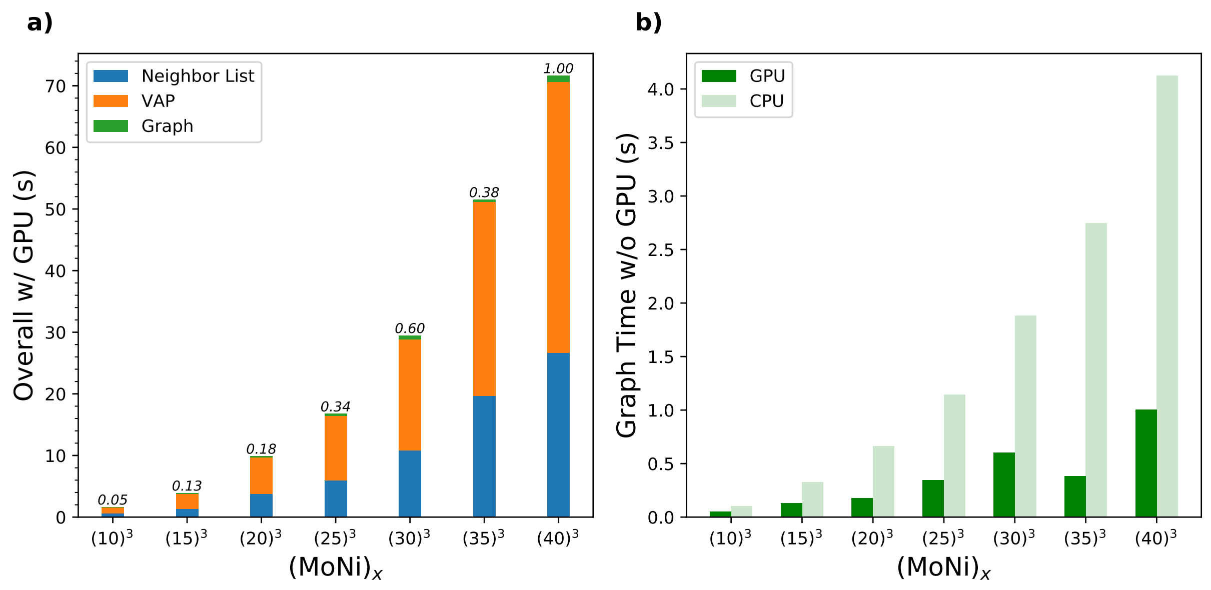

FIG. 7. Benchmarks of the Ni-Mo model in the prediction phase. The test structures are (MoNi)$_x$ where $x = 10^3, 15^3, 20^3, 25^3, 30^3, 35^3, 40^3$. The left figure **a)** demonstrates the overall time and the right figure **b)** compares the 'Graph' time with/out GPU. Blue bars indicate the elapsed time (in seconds) for finding neighbors, orange bars denote the time for initializing VAP-GSL arrays and green bars and the numbers above represents the graph execution time (used for obtaining energy, forces and stress).

are just supercells of the base structure. We choose $x \in [10^3, 15^3, 20^3, 25^3, 30^3, 35^3, 40^3]$ as a typical molecular dynamics simulation for studying alloy properties needs at least thousands of atoms. The elaspsed time is splitted to three parts: 'Neighbor' (blue) indicates the execution time of the *neighbor_list* routine of ASE for finding local neighbors (**(1)** of Fig 3), 'VAP' (orange) denotes the time to construct VAP-GSL arrays (**(2)** of Fig 3) and 'Graph' (green bars and the numbers above) represents the time of getting energy, forces and stress tensor by executing the computation graph. The smallest subject (2000 atoms) needs 1.6 seconds and the largest system (128000 atoms) requires 71.6 seconds on the workstation. Most of the time are spent on the Python-based single-thread 'Neighbor' and 'VAP' routines —which may be greatly optimized by utilizing efficient C/C++/Fortran codes (e.g. Lammps). For the 'Graph' part, the execution time can be as small as 1 second (with GPU) or 4 seconds (without GPU) for the 128000-atoms structure. According to these tests, it's fairly practical to use TensorAlloy-trained symmetry function interaction potential on studying large alloy systems.

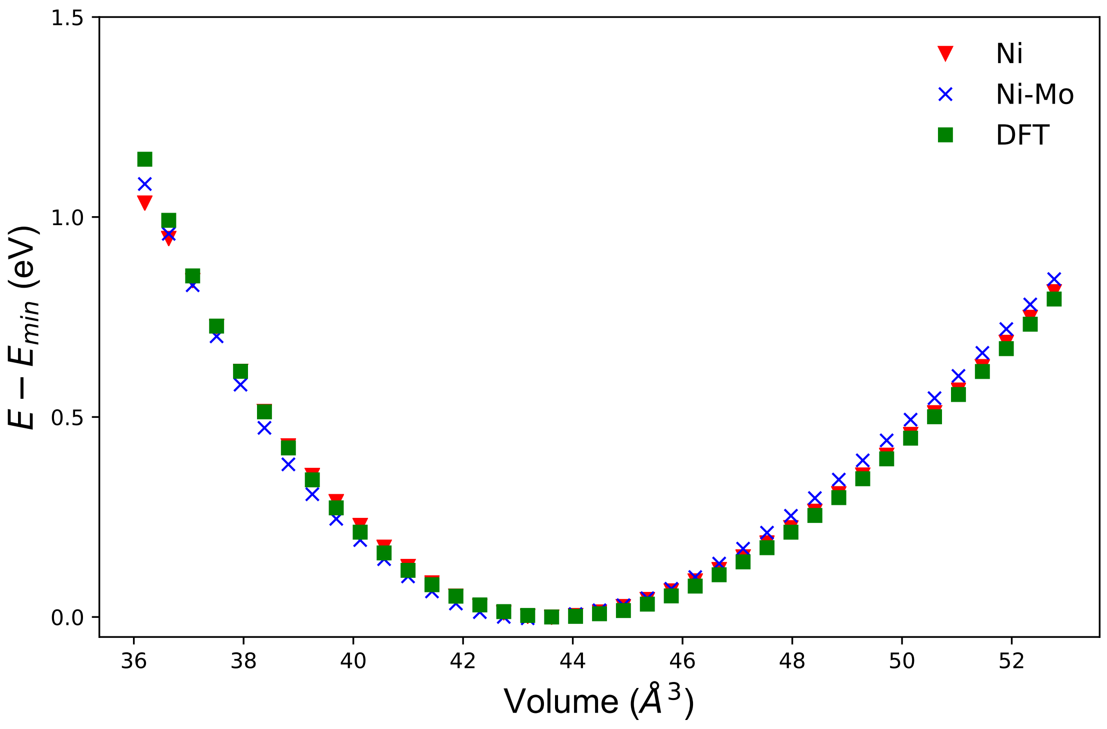

FIG. 8. The energy vs volume curves of fcc Ni for the DFT, Ni and Ni-Mo models.

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 8. 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 9, also proves the accuracy of TensorAlloy-optimized models. PyMatGen[46, 47] is used to generate these surface slabs.

The benchmark function and the trained models can be accessed from the VAP GitHub repo (https://github.com/Bismarrck/vap) freely.

V. CONCLUSIONS

In this work, we proposed a two-phases (training, prediction) approach (the virtual atom approach) to construct symmetry function based atomistic neural networks. With this method, we can build direct computation graph from atomic positions to total energy within TensorFlow, thus making the technical barrier of calculating NN-derived atomic forces and

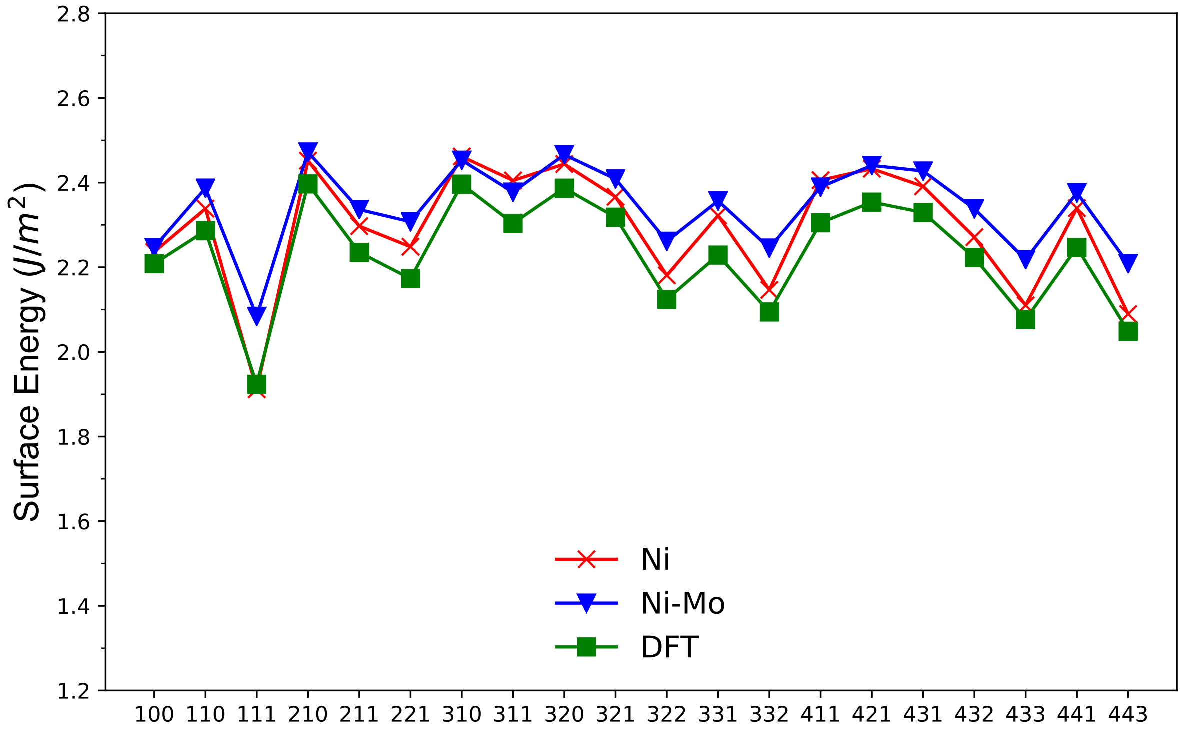

FIG. 9. Surface formation energies (J/m^2) of different Ni surfaces calculated by DFT, Ni and Ni-Mo models

virial stress neglectable. We also derived an alternative form of the general virial stress equation. This equation can be calculated with simple matrix operations, which makes it suitable for machine learning based interaction potentials. With these new features, we developed a Python program, TensorAlloy. This program can be used to learn interaction potentials of *molecules* or *solids*.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB0201203, the Science Challenge Project under Grant No. TZ2018002 and the National Natural Science Foundation of China under Grant No.

U1630250.

[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] M. I. Baskes and R. A. Johnson, Modelling Simul. Mater. Sci. Eng. **2**, 147 (1994).

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

[8] B.-J. Lee, J.-H. Shim, and M. I. Baskes, Phys. Rev. B **68**, 10.1103/PhysRevB.68.144112 (2003).

[9] Y. Mishin and A. Y. Lozovoi, Acta Mater. **54**, 5013 (2006).

[10] Y. Mishin, M. J. Mehl, and D. A. Papaconstantopoulos, Acta Mater. **53**, 4029 (2005).

[11] G. P. Purja Pun, K. A. Darling, L. J. Kecskes, and Y. Mishin, Acta Mater. **100**, 377 (2015).

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

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

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

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

[16] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] 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.

[32] 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).

[33] 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].

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

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

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

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

[38] 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).

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

[40] 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.

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

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

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

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

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

[46] 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).

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

APPENDIX

A. Derivation and implementation of the virial stress equation

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

$$\begin{aligned} r_{ij\mathbf{n}} =& \|\mathbf{r}_i^{(0)} - \mathbf{r}_j^{(0)} + \mathbf{n}^T\mathbf{h}\| \\ =& \left[\left(r_{j,x}^{(0)} - r_{i,x}^{(0)} + \sum_\alpha n_\alpha h_{\alpha x}\right)^2 + \left(r_{j,y}^{(0)} - r_{i,y}^{(0)} + \sum_\alpha n_\alpha h_{\alpha y}\right)^2 + \right. \\ & \left.\left(r_{j,z}^{(0)} - r_{i,z}^{(0)} + \sum_\alpha n_\alpha h_{\alpha z}\right)^2\right]^{\frac{1}{2}} \end{aligned} \tag{29}$$

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

$$\frac{\partial r_{ij\mathbf{n}}}{\partial h_{\alpha\beta}} = \frac{1}{r_{ij\mathbf{n}}} \cdot \Delta_{ij\mathbf{n}\beta} \cdot n_\alpha \tag{30}$$

$$\Delta_{ij\mathbf{n}\beta} = r_{j,\beta}^{(0)} - r_{i,\beta}^{(0)} + \sum_\alpha n_\alpha h_{\alpha\beta} \tag{31}$$

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

$$\frac{\partial E^{total}}{\partial h_{\alpha\beta}} = \sum_i^N \sum_l^{N_G} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial G_{il}}{\partial h_{\alpha\beta}} \tag{32}$$

If G_{il} is a radial symmetry function:

$$\frac{\partial G_{il}^{(2)}}{\partial h_{\alpha\beta}} = \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{1}{r_{ij\mathbf{n}}} \cdot \Delta_{ij\mathbf{n}\beta} \cdot n_\alpha \tag{33}$$

$$\frac{\partial G_{il}^{(2)}}{\partial h_{\gamma\alpha}} \cdot h_{\gamma\beta} = \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \cdot n_\gamma \cdot h_{\gamma\beta} \tag{34}$$

If G_{il} is an angular symmetry function:

$$\begin{aligned}
\frac{\partial G_{il}^{(4)}}{\partial h_{\alpha\beta}} &= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \left(\frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\partial r_{ij\mathbf{n_1}}}{h_{\alpha\beta}} + \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\partial r_{ik\mathbf{n_2}}}{h_{\alpha\beta}} + \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\partial r_{jk\mathbf{n_3}}}{h_{\alpha\beta}} \right) \\
&= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\Delta_{ij\mathbf{n_1}\beta}}{r_{ij\mathbf{n_1}}} \cdot n_{1,\alpha} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\Delta_{ik\mathbf{n_2}\beta}}{r_{ik\mathbf{n_2}}} \cdot n_{2,\alpha} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\Delta_{jk\mathbf{n_2}\beta}}{r_{jk\mathbf{n_3}}} \cdot n_{3,\alpha}
\end{aligned} \tag{35}$$

$$\begin{aligned}
\frac{\partial G_{il}^{(4)}}{\partial h_{\gamma\alpha}} \cdot h_{\gamma\beta} &= \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ij\mathbf{n_1}}} \cdot \frac{\Delta_{ij\mathbf{n_1}\alpha}}{r_{ij\mathbf{n_1}}} \cdot n_{1,\gamma} \cdot h_{\gamma\beta} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{ik\mathbf{n_2}}} \cdot \frac{\Delta_{ik\mathbf{n_1}\alpha}}{r_{ik\mathbf{n_2}}} \cdot n_{2,\gamma} \cdot h_{\gamma\beta} \\
&+ \sum_{j,k\neq j,k\neq i} \sum_{\mathbf{n_1}} \sum_{\mathbf{n_2}} \sum_{\mathbf{n_3}} \frac{\partial g_4}{\partial r_{jk\mathbf{n_3}}} \cdot \frac{\Delta_{jk\mathbf{n_1}\alpha}}{r_{jk\mathbf{n_3}}} \cdot n_{3,\gamma} \cdot h_{\gamma\beta}
\end{aligned} \tag{36}$$

Combining with Equation 32, 34 and 36, we can now start deriving the right part of Equation 19. To simplify the derivation, we just assume G_{il} represents a radial function:

$$\begin{aligned}
\left(\left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} \right)_{\alpha\beta} &= \sum_{\gamma} \frac{\partial E^{total}}{\partial h_{\gamma\alpha}} h_{\gamma\beta} \\
&= \sum_{\gamma} \sum_{i}^{N} \sum_{l}^{N_G} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \cdot n_{\gamma} h_{\gamma\beta} \\
&= \sum_{i}^{N} \sum_{l}^{N_G} \sum_{j\neq i} \sum_{\mathbf{n}} \frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \sum_{\gamma} \cdot n_{\gamma} h_{\gamma\beta} \\
&= -\sum_{i}^{N} \sum_{l}^{N_G} \sum_{j\neq i} \sum_{\mathbf{n}} f'_{ij\mathbf{n}\alpha} \sum_{\gamma} \cdot n_{\gamma} h_{\gamma\beta}
\end{aligned} \tag{37}$$

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

$$f'_{ij\mathbf{n}\alpha} = -\frac{\partial \mathbf{NN}_{el}(\mathbf{G}_i)}{\partial G_{il}} \cdot \frac{\partial g_2}{\partial r_{ij\mathbf{n}}} \cdot \frac{\Delta_{ij\mathbf{n}\alpha}}{r_{ij\mathbf{n}}} \tag{38}$$

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

$$\left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} = -\left(\sum_{\mathbf{n}} \mathbf{h}^T \mathbf{n} \otimes \sum_{i=1}^{N} F'_{i\mathbf{n}} \right)^T \tag{39}$$

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

$$\sum_{i=1}^{N} \mathbf{r}_i^{(0)} \otimes f_i = R^T F \tag{40}$$

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

$$\epsilon = -F^T R + \left(\frac{\partial E^{total}}{\partial \mathbf{h}} \right)^T \mathbf{h} \tag{41}$$

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
```

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

The following function returns N_{ij} and N_{ijk} of an ase.Atoms:

```
def get_nij_and_nijk(atoms, rc, angular=False):
    ilist, jlist = neighbor_list('ij', atoms, cutoff=rc)
    nij = len(ilist)
    if angular:
        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
    else:
        nijk = 0
    return nij, nijk
```

During the training phase, $N_{ij}^{\max}$ and $N_{ijk}^{\max}$ of a dataset (a collection of ase.Atoms objects) can be pre-computed with the following codes:

```
nij_max = 0
nijk_max = 0
for atoms in dataset:
    nij, nijk = get_nij_and_nijk(atoms, rc, angular)
    nij_max = max(nij, nij_max)
    nijk_max = max(nijk, nijk_max)
```

C. Calculations of $N_{\mathrm{element}}^{\max}$

The following function returns all $N_{\mathrm{element}}^{\max}$ (e.g., $N_{\mathrm{Ni}}^{\max}$) of a dataset:

```
def get_max_occurs(dataset):
    from collections import Counter
    max_occurs = Counter()
    for atoms in dataset:
        c = Counter(atoms.get_chemical_symbols())
        for e, n in c.items():
            max_occurs[e] = max(n, max_occurs[e])
    return max_occurs
```

D. Implementation of the Virtual-Atom Map

The following class implements the core of the Virtual-Atom Approach. The argument *max_occurs* just represents the Global Symbol List (GSL). The positions and forces can be transformed to their GSL forms by using the function *map_to_gsl_array*.

```
class VirtualAtomMap:
    REAL_ATOM_START = 1

    def __init__(self, max_occurs: Counter, symbols: List[str]):
        istart = VirtualAtomMap.REAL_ATOM_START
        self.max_occurs = max_occurs
        self.symbols = symbols
        self.max_vap_n_atoms = sum(max_occurs.values()) + istart
        elements = sorted(max_occurs.keys())
        offsets = np.cumsum([max_occurs[e] for e in elements])[:-1]
        offsets = np.insert(offsets, 0, 0)
        delta = Counter()
        index_map = {}
        mask = np.zeros(self.max_vap_n_atoms, dtype=bool)
        for i, symbol in enumerate(symbols):
            i_ele = elements.index(symbol)
            i_old = i + istart
            i_new = offsets[i_ele] + delta[symbol] + istart
            index_map[i_old] = i_new
            delta[symbol] += 1
            mask[i_new] = True
        reverse_map = {v: k - 1 for k, v in index_map.items()}
        index_map[0] = 0
        reverse_map[0] = -1
        self.atom_masks = mask
        self.index_map = index_map
        self.reverse_map = reverse_map
        self.splits = np.array([1, ] + [max_occurs[e] for e in elements])

    def map_array_to_gsl(self, array: np.ndarray):
        rank = np.ndim(array)
        if rank == 2:
            array = array[np.newaxis, ...]
        elif rank <= 1 or rank > 3:
            raise ValueError("The rank should be 2 or 3")
        if array.shape[1] == len(self.symbols):
            array = np.insert(
                array, 0, np.asarray(0, dtype=array.dtype), axis=1)
        else:
            shape = (array.shape[0], len(self.symbols), array.shape[2])
            raise ValueError(f"The shape should be {shape}")
        indices = []
        for i in range(self.max_vap_n_atoms):
            indices.append(
                self.reverse_map.get(i, -1) + self.REAL_ATOM_START)
        output = array[:, indices]
        if rank == 2:
            output = np.squeeze(output, axis=0)
        return output
    def reverse_map_gsl_to_local(self, array: np.ndarray):
        rank = np.ndim(array)
        if rank == 2:
            array = array[np.newaxis, ...]
        assert array.shape[1] == self.max_vap_n_atoms
        istart = self.REAL_ATOM_START
        indices = []
        for i in range(istart, istart + len(self.symbols)):
            indices.append(self.index_map[i])
        output = array[:, indices]
        if rank == 2:
            output = np.squeeze(output, axis=0)
        return output
```

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

This function shows the calculation of g^2_{map}. The returned dict is organized as follows:

- 'g2.v2g_map' : g^2_{map}
- 'g2.ilist' : $i^{(0)}_{+,\mathrm{GSL}}$
- 'g2.jlist' : $j^{(0)}_{+,\mathrm{GSL}}$
- 'g2.shift' : $\mathbf{n}_+$

$N_{ij}^{\max}$ 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]*.

```
def get_g2_map(atoms: Atoms,
               rc: float,
               nij_max: int,
               interactions: list,
               vap: VirtualAtomMap,
               offsets: np.ndarray,
               for_prediction=False):
    if for_prediction:
        iaxis = 0
    else:
        iaxis = 1
    g2_map = np.zeros((nij_max, iaxis + 2), dtype=np.int32)
    tlist = np.zeros(nij_max, dtype=np.int32)
    symbols = atoms.get_chemical_symbols()
    tic = time.time()
    ilist, jlist, n1 = 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.pad(ilist + 1, (0, nij_max - nij), 'constant')
    jlist = np.pad(jlist + 1, (0, nij_max - nij), 'constant')
    n1 = np.pad(n1, ((0, nij_max - nij), (0, 0)), 'constant')
    n1 = n1.astype(np.float32)
    for count in range(len(ilist)):
        if ilist[count] == 0:
            break
        ilist[count] = vap.index_map[ilist[count]]
        jlist[count] = vap.index_map[jlist[count]]
    g2_map[:, iaxis + 0] = ilist
    g2_map[:, iaxis + 1] = offsets[tlist]
    return {"g2.v2g_map": g2_map, "g2.ilist": ilist, "g2.jlist": jlist,
            "g2.shift": n1}
```

F. Implementation of the $g^4_{\rm map}$

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

One must note that 'g2.ilist' contains GSL indices while the symbol list of the target *Atoms* is in local order. So the *reverse* mapping is necessary.

```
def get_g4_map(atoms: Atoms,
               g2_map: dict,
               interactions: list,
               offsets: np.ndarray,
               vap: VirtualAtomMap,
               nijk_max: int,
               for_prediction=True):
    if for_prediction:
        iaxis = 0
    else:
        iaxis = 1
    g4_map = np.zeros((nijk_max, iaxis + 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, atom_gsl_i in enumerate(g2_map["g2.ilist"]):
        if atom_gsl_i == 0:
            break
        if atom_gsl_i not in indices:
            indices[atom_gsl_i] = []
            vectors[atom_gsl_i] = []
        indices[atom_gsl_i].append(g2_map["g2.jlist"][i])
        vectors[atom_gsl_i].append(g2_map["g2.shift"][i])
```

```
count = 0
for atom_gsl_i, nl in indices.items():
    atom_local_i = vap.reverse_map[atom_gsl_i]
    symboli = symbols[atom_local_i]
    prefix = '{}'.format(symboli)
    for j in range(len(nl)):
        atom_vap_j = nl[j]
        atom_local_j = vap.reverse_map[atom_vap_j]
        symbolj = symbols[atom_local_j]
        for k in range(j + 1, len(nl)):
            atom_vap_k = nl[k]
            atom_local_k = vap.reverse_map[atom_vap_k]
            symbolk = symbols[atom_local_k]
            interaction = '{}{}'.format(
                prefix, ''.join(sorted([symbolj, symbolk])))
            ijk[count] = atom_gsl_i, atom_vap_j, atom_vap_k
            n1[count] = vectors[atom_gsl_i][j]
            n2[count] = vectors[atom_gsl_i][k]
            n3[count] = vectors[atom_gsl_i][k] - vectors[atom_gsl_i][j]
            index = interactions.index(interaction)
            g4_map[count, iaxis + 0] = atom_gsl_i
            g4_map[count, iaxis + 1] = offsets[index]
            count += 1
return {"g4.v2g_map": g4_map,
        "g4.ij.ilist": ijk[:, 0], "g4.ij.jlist": ijk[:, 1],
        "g4.ik.ilist": ijk[:, 0], "g4.ik.klist": ijk[:, 2],
        "g4.jk.jlist": ijk[:, 1], "g4.jk.klist": ijk[:, 2],
        "g4.shift.ij": n1, "g4.shift.ik": n2, "g4.shift.jk": n3}
```

G. Compute the descriptors

Now we can implements the symmetry function descriptor. Here *SymmetryFunction* is the base class and *BatchSymmetryFunction* is its subclass for batch training.

The return value of the method *build_graph* is a dict and its keys are chemical elements and values are correspond atomic descriptors and atom masks. Taking the example of the Ni-Mo dataset with $r_c = 4.6$, batch size 10 and only 10 radial symmetry functions, the corresponding $N_{ij}^{\max}$, $N_{\mathrm{Ni}}^{\max}$ and $N_{\mathrm{Mo}}^{\max}$ are 108, 176 and 7200, respectively. Thus, the return dict should be:

- 'Mo': ($\mathbf{G}^{\mathrm{Mo}}$, δ^{Mo})
- 'Ni': ($\mathbf{G}^{\mathrm{Ni}}$, δ^{Ni})

where $\mathbf{G}^{\mathrm{Mo}}$ is a float tensor of shape [10, 176, 40], δ^{Mo} is a float tensor of shape [10, 176], $\mathbf{G}^{\mathrm{Ni}}$ is a float tensor of shape [10, 108, 40] and δ^{Ni} is a float tensor of shape [10, 108] during the training phase.

The demo script can be found on GitHub: https://github.com/Bismarrck/vap.

```
class SymmetryFunction:
    gather_fn = staticmethod(tf.gather)

    def __init__(self, rc, elements, eta=np.array([0.05, 4.0, 20.0, 80.0]),
                 omega=np.array([0.0]), beta=np.array([0.005, ]),
                 gamma=np.array([1.0, -1.0]), zeta=np.array([1.0, 4.0]),
                 angular=True, periodic=True):
        all_kbody_terms, kbody_terms, elements = \
            get_kbody_terms(elements, angular=angular)
        ndim, kbody_sizes = compute_dimension(
            all_kbody_terms, len(eta), len(omega), len(beta), len(gamma),
            len(zeta))
        self.rc = rc
        self.all_kbody_terms = all_kbody_terms
        self.kbody_terms = kbody_terms
        self.elements = elements
        self.n_elements = len(elements)
        self.periodic = periodic
        self.angular = angular
        self.kbody_sizes = kbody_sizes
        self.ndim = ndim
        self.kbody_index = {
            kbody_term: self.all_kbody_terms.index(kbody_term)
            for kbody_term in self.all_kbody_terms}
        self.offsets = np.insert(np.cumsum(kbody_sizes), 0, 0)
        self.radial_indices_grid = ParameterGrid({
            'eta': np.arange(len(eta), dtype=int),
            'omega': np.arange(len(omega), dtype=int)})
        self.angular_indices_grid = ParameterGrid({
            'beta': np.arange(len(beta), dtype=int),
            'gamma': np.arange(len(gamma), dtype=int),
            'zeta': np.arange(len(zeta), dtype=int)})
        self.initial_values = {'eta': eta, 'omega': omega, 'gamma': gamma,
                               'beta': beta, 'zeta': zeta}

    @staticmethod
    def get_pbc_displacements(shift, cells, dtype=tf.float32):
        return tf.matmul(shift, cells, name='displacements')

    def get_rij(self, positions, cells, ilist, jlist, shift, name):
        with tf.name_scope(name):
            dtype = positions.dtype
            Ri = self.gather_fn(positions, ilist, 'Ri')
            Rj = self.gather_fn(positions, jlist, 'Rj')
            Dij = tf.subtract(Rj, Ri, name='Dij')
            if self.periodic:
                pbc = self.get_pbc_displacements(shift, cells, dtype=dtype)
                Dij = tf.add(Dij, pbc, name='pbc')
            with tf.name_scope("safe_norm"):
                eps = tf.constant(1e-8, dtype=dtype, name='eps')
                rij = tf.sqrt(tf.reduce_sum(
                    tf.square(Dij, name='Dij2'), axis=-1) + eps)
                return rij, Dij

    def get_v2g_map(self, features: dict, prefix: str):
        return tf.identity(features[f"{prefix}.v2g_map"], name='v2g_map')

    @staticmethod
    def get_v2g_map_delta(tau):
        return tf.constant([0, tau], dtype=tf.int32, name='delta')

    def get_g_shape(self, features: dict):
        return [features['n_atoms_vap'], self.ndim]

    def get_g2_op_for_tau(self, shape, tau, r, rc2, fc_r, base_v2g_map):
        with tf.name_scope(f"Grid{tau}"):
            grid = self.radial_indices_grid[tau]
            etai = grid['eta']
            omegai = grid['omega']
            eta = tf.constant(self.initial_values['eta'][etai], r.dtype)
            omega = tf.constant(
                self.initial_values['omega'][omegai], r.dtype)
            delta = self.get_v2g_map_delta(tau)
            r2c = tf.math.truediv(tf.square(r - omega), rc2, name='r2c')
            v = tf.exp(-tf.multiply(eta, r2c, 'eta_r2c')) * fc_r
            v2g_map_tau = tf.add(base_v2g_map, delta, f'v2g_map_{tau}')
            return tf.scatter_nd(v2g_map_tau, v, shape, f"g{tau}")

    def get_g2_op(self, features: dict):
```

```
    with tf.variable_scope("G2"):
        r = self.get_rij(features['positions'],
                          features['cells'],
                          features['g2.ilist'],
                          features['g2.jlist'],
                          features['g2.shift'],
                          name='rij')[0]
        rc2 = tf.constant(self.rc**2, dtype=r.dtype, name='rc2')
        fc_r = cosine_cutoff(r, rc=self.rc, name='fc_r')
        base_v2g_map = self.get_v2g_map(features, prefix='g2')
        shape = self.get_g_shape(features)
        values = []
        for tau in range(len(self.radial_indices_grid)):
            values.append(
                self.get_g2_op_for_tau(
                    shape, tau, r, rc2, fc_r, base_v2g_map))
        return tf.add_n(values, name='g')

def get_g4_op_for_tau(self, shape, tau: int, cos_theta, r2c, fc_r,
                      base_v2g_map):
    with tf.name_scope(f"Grid{tau}"):
        grid = self.angular_indices_grid[tau]
        betai = grid['beta']
        gammai = grid['gamma']
        zetai = grid['zeta']
        beta = tf.constant(
            self.initial_values['beta'][betai], r2c.dtype)
        gamma = tf.constant(
            self.initial_values['gamma'][gammai], r2c.dtype)
        zeta = tf.constant(
            self.initial_values['zeta'][zetai], r2c.dtype)
        delta = self.get_v2g_map_delta(tau)
        one = tf.constant(1.0, dtype=r2c.dtype, name='one')
        two = tf.constant(2.0, dtype=r2c.dtype, name='two')
        gt = tf.math.multiply(gamma, cos_theta, name='gt')
        gt1 = tf.add(gt, one, name='gt1')
        gt1z = tf.pow(gt1, zeta)
        z1 = tf.math.subtract(one, zeta, name='z1')
        z12 = tf.pow(two, z1)
        c = tf.math.multiply(gt1z, z12, name='c')
        v = tf.multiply(c * tf.exp(-beta * r2c), fc_r, f'v_{tau}')
        v2g_map_tau = tf.add(
            base_v2g_map, delta, name=f'v2g_map_{tau}')
        return tf.scatter_nd(v2g_map_tau, v, shape, f'g{tau}')

def get_g4_op(self, features: dict):
    with tf.variable_scope("G4"):
        rij = self.get_rij(features['positions'],
                           features['cells'],
                           features['g4.ij.ilist'],
                           features['g4.ij.jlist'],
                           features['g4.shift.ij'],
                           name='rij')[0]
        rik = self.get_rij(features['positions'],
                           features['cells'],
                           features['g4.ik.ilist'],
                           features['g4.ik.klist'],
                           features['g4.shift.ik'],
                           name='rik')[0]
        rjk = self.get_rij(features['positions'],
                           features['cells'],
                           features['g4.jk.jlist'],
                           features['g4.jk.klist'],
                           features['g4.shift.jk'],
                       name='rjk')[0]
        rij2 = tf.square(rij, name='rij2')
        rik2 = tf.square(rik, name='rik2')
        rjk2 = tf.square(rjk, name='rjk2')
        rc2 = tf.constant(self.rc**2, dtype=rij.dtype, name='rc2')
        r2 = tf.add_n([rij2, rik2, rjk2], name='r2')
        r2c = tf.math.truediv(r2, rc2, name='r2_rc2')
        with tf.name_scope("CosTheta"):
            upper = tf.subtract(rij2 + rik2, rjk2, name='upper')
            lower = tf.multiply(2.0 * rij, rik, name='lower')
            cos_theta = tf.math.truediv(upper, lower, name='theta')
        with tf.name_scope("Cutoff"):
            fc_rij = cosine_cutoff(rij, rc=self.rc, name='fc_rij')
            fc_rik = cosine_cutoff(rik, rc=self.rc, name='fc_rik')
            fc_rjk = cosine_cutoff(rjk, rc=self.rc, name='fc_rjk')
            fc_r = tf.multiply(fc_rij, fc_rik * fc_rjk, 'fc_r')
        base_v2g_map = self.get_v2g_map(features, prefix='g4')
        shape = self.get_g_shape(features)
        values = []
        for tau in range(len(self.angular_indices_grid)):
            values.append(
                self.get_g4_op_for_tau(
                    shape, tau, cos_theta, r2c, fc_r, base_v2g_map))
        return tf.add_n(values, name='g')

    def get_row_split_sizes(self, features: dict):
        return features['row_splits']

    @staticmethod
    def get_row_split_axis():
        return 0
```

```
def get_column_split_sizes(self):
    column_splits = {}
    for i, element in enumerate(self.elements):
        column_splits[element] = [len(self.elements), i]
    return column_splits

@staticmethod
def get_column_split_axis():
    return 1

def split_descriptors(self, descriptors, features: dict):
    with tf.name_scope("Split"):
        row_split_sizes = self.get_row_split_sizes(features)
        row_split_axis = self.get_row_split_axis()
        column_split_sizes = self.get_column_split_sizes()
        column_split_axis = self.get_column_split_axis()
        splits = tf.split(
            descriptors, row_split_sizes, axis=row_split_axis,
            name='rows')[1:]
        atom_masks = tf.split(
            features['atom_masks'], row_split_sizes,
            axis=row_split_axis,
            name='atom_masks')[1:]
        if len(self.elements) > 1:
            blocks = []
            for i in range(len(splits)):
                element = self.elements[i]
                size_splits, idx = column_split_sizes[element]
                block = tf.split(splits[i],
                                 size_splits,
                                 axis=column_split_axis,
                                 name='{}_block'.format(element))[idx]
                blocks.append(block)
        else:
            blocks = splits
        return dict(zip(self.elements, zip(blocks, atom_masks)))

    def build_graph(self, features: dict):
        with tf.variable_scope("Behler"):
            descriptors = self.get_g2_op(features)
            if self.angular:
                descriptors += self.get_g4_op(features)
        return self.split_descriptors(descriptors, features)



class BatchSymmetryFunction(SymmetryFunction):
    gather_fn = staticmethod(tf.batch_gather)

    def __init__(self, rc, max_occurs: Counter, nij_max: int,
                 nijk_max: int, batch_size: int,
                 eta=np.array([0.05, 4.0, 20.0, 80.0]),
                 omega=np.array([0.0]), beta=np.array([0.005, ]),
                 gamma=np.array([1.0, -1.0]), zeta=np.array([1.0, 4.0]),
                 angular=True, periodic=True):
        elements = sorted(list(max_occurs.keys()))

        super(BatchSymmetryFunction, self).__init__(
            rc=rc, elements=elements, eta=eta, beta=beta, gamma=gamma,
            zeta=zeta, omega=omega, angular=angular, periodic=periodic)
        self._max_occurs = max_occurs
        self._max_n_atoms = sum(max_occurs.values())
        self._nij_max = nij_max
        self._nijk_max = nijk_max
        self._batch_size = batch_size
```

```
@staticmethod
def get_pbc_displacements(shift, cells, dtype=tf.float32):
    with tf.name_scope("Einsum"):
        shift = tf.convert_to_tensor(shift, dtype=dtype, name='shift')
        cells = tf.convert_to_tensor(cells, dtype=dtype, name='cells')
        return tf.einsum(
            'ijk,ikl->ijl', shift, cells, name='displacements')

def get_g_shape(self, _):
    n_atoms_vap = self._max_n_atoms + 1
    return [self._batch_size, n_atoms_vap, self.ndim]

def get_v2g_map_batch_indexing_matrix(self, prefix='g2'):
    if prefix == 'g2':
        ndim = self._nij_max
    else:
        ndim = self._nijk_max
    indexing_matrix = np.zeros(
        (self._batch_size, ndim, 3), dtype=np.int32)
    for i in range(self._batch_size):
        indexing_matrix[i] += [i, 0, 0]
    return indexing_matrix

@staticmethod
def get_v2g_map_delta(tau):
    return tf.constant([0, 0, tau], tf.int32, name='delta')

def get_v2g_map(self, features: dict, prefix="g2"):
    indexing = self.get_v2g_map_batch_indexing_matrix(prefix=prefix)
    return tf.add(
        features[f"{prefix}.v2g_map"], indexing, name='v2g_map')
```

```
def get_row_split_sizes(self, _):
    row_splits = [1, ]
    for i, element in enumerate(self.elements):
        row_splits.append(self._max_occurs[element])
    return row_splits

@staticmethod
def get_row_split_axis():
    return 1

@staticmethod
def get_column_split_axis():
    return 2
```