FF-project Documentation

Release June 2023

Qiang Zhu, Howard Yanxon, David Zagaceta

Aug 17, 2023

Content

The aim of PyXtalFF project is to provide an automated computational infrastructure to train the interatomic potentials for inorganic periodic systems from high-end quantum mechanical calculations.

The current version is `0.2.3` at GitHub.

Expect updates upon request by Qiang Zhu's group at University of Nevada Las Vegas.

CHAPTER 1

Installation

1.1 Dependencies

PyXtal_FF is entirely based on Python 3. *Thus Python 2 will not be supported!*. To make it work, several major Python libraries are required.

- NumPy>=1.13.3
- SciPy>=1.1.0
- Matplotlib>=2.0.0
- Sklearn>=0.20.0
- Numba>=0.50.1
- ase>=3.18.0
- Pytorch>=1.2

1.2 To install

To install it, one can simply type `pip install pyxtal_ff` or make a copy of the source code, and then install it manually.

```
$ git clone https://github.com/qzhu2017/PyXtal_FF.git
$ cd FF-project
$ python setup.py install
```

This will install the module. The code can be used within Python via

```
import pyxtal_ff
print(pyxtal_ff.__version__)
```

CHAPTER 2

Background and Theory

2.1 Overall Framework

PyXtalFF involves two important components: **descriptors** and **force field training**. Four types of descriptors are supported in the code, including,

- (Weighted) Behler-Parrinello Symmetry Functions,
- Embedded Atom Descriptors,
- SO(4) Bispectrum Components,
- Smooth SO(3) power spectrum.

For the force field training, the code consists of

- Artificial neural network,
- Generalized linear regressions.

2.2 Atomic Descriptors

In an atomic structure, Cartesian coordinates poorly describe the structural environment. While the energy of a crystal structure remains unchanged, the Cartesian coordinates change as translational or rotational operation is applied to the structure[1]. Thus, physically meaningful descriptor must withhold the energy change as the alterations are performed to the structural environment. In another words, the descriptor needs to be invariant with respect to translation and rotational operations, and the exchanges of any equivalent atom. To ensure the descriptor mapping from the atomic positions smoothly approaching zero beyond the R_c, a cutoff function (f_c) is included to most decriptor mapping schemes, here the exception is the Smooth SO(3) Power Spectrum:

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

[1] Albert P Bartok, Risi Kondor and Gabor Csanyi, "On representing chemical environments," Phys. Rev. B 87, 184115 (2013)

In addition to the cosine function, we also support other types of functions (see cutoff functions)

In the following, the types of descriptors will be explained in details.

2.2.1 Atom Centered Symmetry Function (ACSF)

Behler-Parrinello method—atom-centered descriptors—utilizes a set of symmetry functions[2]. The symmetry functions map two atomic Cartesian coordinates to a distribution of distances between atom (radial functions) or three atomic Cartesian coordinates to a distribution of bond angles (angular functions). These mappings are invariant with respect to translation, rotation, and permutation of atoms of the system. Therefore, the energy of the system will remain unchanged under these mapping of symmetry functions.

PyXtal_FF supports three types of symmetry functions:

$$G_i^{(2)} = \sum_{j \neq i} e^{-\eta(R_{ij}-\mu)^2} \cdot f_c(R_{ij})$$

$$G_i^{(4)} = 2^{1-\zeta} \sum_{j \neq i} \sum_{k \neq i,j} [(1+\lambda \cos\theta_{ijk})^{\zeta} \cdot e^{-\eta(R_{ij}^2+R_{ik}^2+R_{jk}^2)} \cdot f_c(R_{ij}) \cdot f_c(R_{ik}) \cdot f_c(R_{jk})]$$

$$G_i^{(5)} = 2^{1-\zeta} \sum_{j \neq i} \sum_{k \neq i,j} [(1+\lambda \cos\theta_{ijk})^{\zeta} \cdot e^{-\eta(R_{ij}^2+R_{ik}^2)} \cdot f_c(R_{ij}) \cdot f_c(R_{ik})]$$

where η and R_s are defined as the width and the shift of the symmetry function. As for $G^{(4)}$ and $G^{(5)}$, they are a few of many ways to capture the angular information via three-body interactions (θ_{ijk}). ζ determines the strength of angular information. Finally, λ values are set to +1 and -1, for inverting the shape of the cosine function.

By default, ACSF splits each different atomic pair and triplets into different descriptors. For instance, a G2 function of SiO2 system for each Si has Si-Si, Si-O descriptors, while G4 has Si-Si-Si, Si-Si-O, O-Si-O. This is not convenient for its extension to multiple component systems. Therefore, an alterative solution is to assign the weight function to each G2 and G4 distances by the atomic number.

$$G_i^{(2)} = \sum_{j \neq i} Z_j e^{-\eta(R_{ij}-\mu)^2} \cdot f_c(R_{ij})$$

$$G_i^{(4)} = 2^{1-\zeta} \sum_{j \neq i} \sum_{k \neq i,j} Z_j Z_k [(1+\lambda \cos\theta_{ijk})^{\zeta} \cdot e^{-\eta(R_{ij}^2+R_{ik}^2+R_{jk}^2)} \cdot f_c(R_{ij}) \cdot f_c(R_{ik}) \cdot f_c(R_{jk})]$$

$$G_i^{(5)} = 2^{1-\zeta} \sum_{j \neq i} \sum_{k \neq i,j} Z_j Z_k [(1+\lambda \cos\theta_{ijk})^{\zeta} \cdot e^{-\eta(R_{ij}^2+R_{ik}^2)} \cdot f_c(R_{ij}) \cdot f_c(R_{ik})]$$

The above formula is called wACSF[3].

2.2.2 Embedded Atom Density

Embedded atom density (EAD) descriptor[4] is inspired by embedded atom method (EAM)—description of atomic bonding by assuming each atom is embedded in the uniform electron cloud of the neighboring atoms. The EAM generally consists of a functional form in a scalar uniform electron density for each of the "embedded" atom plus the short-range nuclear repulsion potential. Given the uniform electron gas model, the EAM only works for metallic

[2] Jorg Behler and Michele Parrinello, "Generalized neural-network representation of high-dimensional potential-energy surfaces," Phys. Rev. Lett. 98, 146401 (2007)

[3] M. Gastegger, L. Schwiedrzik, M. Bittermann, F. Berzsenyi and P. Marquetand, J. Chem. Phys. 148, 241709 (2018)

[4] Zhang, C. Hu, B. Jiang, "Embedded atom neural network potentials: Efficient and accurate machine learning with a physically inspired representation," The Journal of Physical Chemistry Letters 10 (17) (2019) 4962–4967 (2019).

systems, even so the EAM can severely underperform in predicting the metallic systems. Therefore, the density can be modified by including the square of the linear combination the atomic orbital components:

$$\rho_i(R_{ij}) = \sum_{l_x,l_y,l_z}^{l_x+l_y+l_z=L} \frac{L!}{l_x!l_y!l_z!}\left(\sum_{j\neq i}^{N} Z_j \Phi(R_{ij})\right)^2$$

where Z_j represents the atomic number of neighbor atom j. L is the quantized angular momentum, and $l_{x,y,z}$ are the quantized directional-dependent angular momentum. For example, $L = 2$ corresponds to the d orbital. Lastly, the explicit form of Φ is:

$$\Phi(R_{ij}) = x_{ij}^{l_x} y_{ij}^{l_y} z_{ij}^{l_z} \cdot e^{-\eta(R_{ij}-\mu)^2} \cdot f_c(R_{ij})$$

According to quantum mechanics, ρ follows the similar procedure in determining the probability density of the states, i.e. the Born rule.

Furthermore, EAD can be regarded as the improved Gaussian symmetry functions. EAD has no classification between the radial and angular term. The angular or three-body term is implicitly incorporated in when $L > 0$. By definition, the computation cost for calculating EAD is cheaper than angular symmetry functions by avoiding the extra sum of the k neighbors. In term of usage, the parameters η and μ are similar to the strategy used in the Gaussian symmetry functions, and the maximum value for L is 3, i.e. up to f orbital.

2.2.3 SO(4) Bispectrum Components

The SO(4) bispectrum components[5,6] are another type of atom-centered descriptor based on triple correlation of the atomic neighbor density function on the 3-sphere. The distribution of atoms in an atomic environment can be represented as a sum of delta functions, this is known as the atomic neighbor density function.

$$\rho(\boldsymbol{r}) = \delta(\boldsymbol{r}) + \sum_i \delta(\boldsymbol{r} - \boldsymbol{r_i})$$

Then this function can mapped to the 3 sphere by mapping the atomic coordinates (x, y, z) to the 3-sphere by the following relations:

$$\theta = \arccos\left(\frac{z}{r}\right)$$

$$\phi = \arctan\left(\frac{y}{x}\right)$$

$$\omega = \pi\frac{r}{r_{cut}}$$

Using this mapping, the Atomic Neighbor Density Function is then expanded on the 3-sphere using the Wigner-D matrix elements, the harmonic functions on the 3-sphere. The resulting expansion coefficients are given by:

$$c^j_{m',m} = D^j_{m',m}(\mathbf{0}) + \sum_i D^j_{m',m}(\boldsymbol{r}_i)$$

The triple correlation of the Atomic Neighbor Density Function on the 3-sphere is then given by a third order product of the expansion coefficients by the Fourier theorem.

$$B_{j_1,j_2,j} = \sum_{m',m=-j}^{j} c^j_{m',m} \sum_{m'_1,m_1=-j_1}^{j_1} c^{j_1}_{m'_1,m_1} \times \sum_{m'_2,m_2=-j_2}^{j_2} c^{j_2}_{m'_2,m_2} C^{jj_1j_2}_{mm_1m_2} C^{jj_1j_2}_{m'm'_1m'_2},$$

Where C is a Clebsch-Gordan coefficient.

[5] Albert P Bartok, Mike C Payne, Risi Kondor and Gabor Csanyi, "Gaussian approximation potentials: The accuracy of quantum mechan-ics, without the electrons," Phys. Rev. Lett. 104, 136403 (2010)

[6] A.P. Thompson, L.P. Swiler, C.R. Trott, S.M. Foiles and G.J. Tucker, "Spectral neighbor analysis method for automated generation ofquantum-accurate interatomic potentials," J. Comput. Phys. 285, 316–330 (2015)

2.2.4 Smooth SO(3) Power Spectrum

Now instead of considering a hyperdimensional space, we can derive a similar descriptor by taking the auto correlation of the atomic neighbor density function through expansions on the 2-sphere and a radial basis on a smoothened atomic neighbor density function[6].

$$\rho' = \sum_i e^{-\alpha|\boldsymbol{r}-\boldsymbol{r}_i|^2}$$

This function is then expanded on the 2-sphere using Spherical Harmonics and a radial basis $g_n(r)$ orthonormalized on the interval $(0, r_{\text{cut}})$.

$$c_{nlm} = \langle g_n Y_{lm}|\rho'\rangle = 4\pi e^{-\alpha r_i^2} Y_{lm}^*(\boldsymbol{r}_i) \int_0^{r_{\text{cut}}} r^2 g_n(r) I_l(2\alpha r r_i) e^{-\alpha r^2} dr$$

Where I_l is a modified spherical bessel function of the first kind. The autocorrelation or power spectrum is obtained through the following sum.

$$p_{n_1 n_2 l} = \sum_{m=-l}^{+l} c_{n_1 lm} c_{n_2 lm}^*$$

2.3 Expression of Target Properties

For all of the regression techniques, the force field training involves fitting of energy, force, and stress simultaneously, although PyXtal_FF allows the fitting of force or stress to be optional. The energy can be written in the sum of atomic energies, in which is a functional (F) of the descriptor ($\boldsymbol{X}_i$):

$$E_{\text{total}} = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} F_i(\boldsymbol{X}_i)$$

Since neural network and generalized linear regressions have well-defined functional forms, analytic derivatives can be derived by applying the chain rule to obtain the force at each atomic coordinate, $\boldsymbol{r}_m$:

$$\boldsymbol{F}_m = -\sum_{i=1}^{N} \frac{\partial F_i(\boldsymbol{X}_i)}{\partial \boldsymbol{X}_i} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m}$$

Finally, the stress tensor is acquired through the virial stress relation:

$$\boldsymbol{S} = -\sum_{m=1}^{N} \boldsymbol{r}_m \otimes \sum_{i=1}^{N} \frac{\partial F_i(\boldsymbol{X}_i)}{\partial \boldsymbol{X}_i} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m}$$

2.4 Force Field Training

Here, we reveal the functional form (F) presented in equation above. The functional form is essentially regarded as the regression model. Each regression model is species-dependent, i.e. as the the number of species increases, the regression parameters will increase. This is effectively needed to describe the presence of other chemical types in complex system. Hence, explanation for the regression models will only consider single-species for the sake of simplicity.

Furthermore, it is important to choose differentiable functional as well as its derivative due to the existence of force (F) and stress (S) contribution along with the energy (E) in the loss function:

$$\Delta = \frac{1}{2M} \sum_{i=1}^{M} \left[\left(\frac{E_i - E_i^{\text{Ref}}}{N_{\text{atom}}^i} \right)^2 + \frac{\beta_f}{3N_{\text{atom}}^i} \sum_{j=1}^{3N_{\text{atom}}^i} (F_{i,j} - F_{i,j}^{\text{Ref}})^2 + \frac{\beta_s}{6} \sum_{p=0}^{2} \sum_{q=0}^{p} (S_{pq} - S_{pq}^{\text{Ref}})^2 \right]$$

where M is the total number of structures in the training pool, and N_i^{atom} is the total number of atoms in the i-th structure. The superscript Ref corresponds to the target property. β_f and β_s are the force and stress coefficients respectively. They scale the importance between energy, force, and stress contribution as the force and stress information can overwhelm the energy information due to their sizes. Additionally, a regularization term can be added to induce penalty on the entire parameters preventing overfitting:

$$\Delta_{\text{p}} = \frac{\alpha}{2M} \sum_{i=1}^{m} (\boldsymbol{w}^i)^2$$

where α is a dimensionless number that controls the degree of regularization.

2.4.1 Generalized Linear Regression

This regression methodology is a type of polynomial regression. Essentially, the quantum-mechanical energy, forces, and stress can be expanded via Taylor series with atom-centered descriptors as the independent variables:

$$E_{\text{total}} = \gamma_0 + \boldsymbol{\gamma} \cdot \sum_{i=1}^{N} \boldsymbol{X}_i + \frac{1}{2} \sum_{i=1}^{N} \boldsymbol{X}_i^T \cdot \boldsymbol{\Gamma} \cdot \boldsymbol{X}_i$$

where N is the total atoms in a structure. γ_0 and $\boldsymbol{\gamma}$ are the weights presented in scalar and vector forms. $\boldsymbol{\Gamma}$ is the symmetric weight matrix (i.e. $\boldsymbol{\Gamma}_{12} = \boldsymbol{\Gamma}_{21}$) describing the quadratic terms. In this equation, we only restricted the expansion up to polynomial 2 due to to enormous increase in the weight parameters.

In consequence, the force on atom j and the stress matrix can be derived, respectively:

$$\boldsymbol{F}_m = -\sum_{i=1}^{N} \left(\boldsymbol{\gamma} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m} + \frac{1}{2} \left[\frac{\partial \boldsymbol{X}_i^T}{\partial \boldsymbol{r}_m} \cdot \boldsymbol{\Gamma} \cdot \boldsymbol{X}_i + \boldsymbol{X}_i^T \cdot \boldsymbol{\Gamma} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m} \right] \right)$$

$$\boldsymbol{S} = -\sum_{m=1}^{N} \boldsymbol{r}_m \otimes \sum_{i=1}^{N} \left(\boldsymbol{\gamma} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m} + \frac{1}{2} \left[\frac{\partial \boldsymbol{X}_i^T}{\partial \boldsymbol{r}_m} \cdot \boldsymbol{\Gamma} \cdot \boldsymbol{X}_i + \boldsymbol{X}_i^T \cdot \boldsymbol{\Gamma} \cdot \frac{\partial \boldsymbol{X}_i}{\partial \boldsymbol{r}_m} \right] \right)$$

Notice that the energy, force, and stress share the weights parameters $\{\gamma_0, \boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_N, \boldsymbol{\Gamma}_{11}, \boldsymbol{\Gamma}_{12}, ..., \boldsymbol{\Gamma}_{NN}\}$. Therefore, a reliable MLP must satisfy the three conditions in term of energy, force, and stress.

2.4.2 Neural Network Regression

Another type of regression model is neural network regression. Due to the set-up of the algorithm, neural network is suitable for training large data sets. Neural network gains an upper hand from generalized linear regression in term of the flexibility of the parameters.

A mathematical form to determine any node value can be written as:

$$X_{n_i}^l = a_{n_i}^l \left(b_{n_i}^{l-1} + \sum_{n_j=1}^{N} W_{n_j,n_i}^{l-1,l} \cdot X_{n_j}^{l-1} \right)$$

The value of a neuron ($X_{n_i}^l$) at layer l can determined by the relationships between the weights ($W_{n_j,n_i}^{l-1,l}$), the bias ($b_{n_i}^{l-1}$), and all neurons from the previous layer ($X_{n_j}^{l-1}$). $W_{n_j,n_i}^{l-1,l}$ specifies the connectivity of neuron n_j at layer $l-1$ to the neuron n_i at layer l. $b_{n_i}^{l-1}$ represents the bias of the previous layer that belongs to the neuron n_i. These connectivity are summed based on the total number of neurons (N) at layer $l-1$. Finally, an activation function ($a_{n_i}^l$) is applied to the summation to induce non-linearity to the neuron ($X_{n_i}^l$). X_{n_i} at the output layer is equivalent to an atomic energy, and it represents an atom-centered descriptor at the input layer. The collection of atomic energy contributions are summed to obtain the total energy of the structure.

CHAPTER 3

Quick Start

Below is a quick example to quickly make a force field for silicon.

```
from pyxtal_ff import PyXtal_FF

train_data = "pyxtal_ff/datasets/Si/PyXtal/Si8.json"
descriptors = {'type': 'SOAP',
               'Rc': 5.0,
               'parameters': {'lmax': 4, 'nmax': 3},
               'N_train': 400,
              }
model = {'system': ['Si'],
         'hiddenlayers': [16, 16],
        }
ff = PyXtal_FF(descriptors=descriptors, model=model)
ff.run(mode='train', TrainData=train_data)
```

The script will first compute the SOAP descriptor . As long as the descritors are obtained, they will be fed to the neural network trainining. Below is an example output from this quick script.

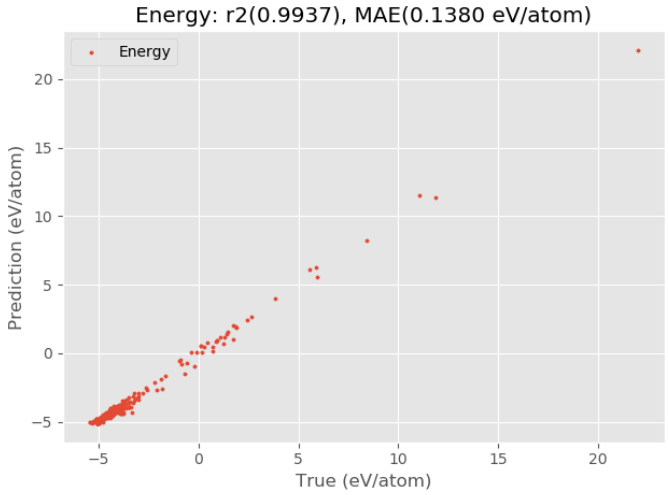
```
              _______          _     _           _          ________ ________
             (______ \       \ \   / /         | |        (________|________)
              ______) )    _ \ \/ / /_  _____| |         ______    ______
             |  _____/ | | | )   (|  _)/ _   | |        |  ___)  |  ___)
             | |     | |_| |/ /\ \ |_( ( |  | |_______| |      | |
             |_|      \___  /_/  \_\___)_||_|_(_______)_|      |_|
                         (_____/



================================ version 0.0.9 ================================

Descriptor parameters:
type         : SOAP
Rc           : 5.0
```

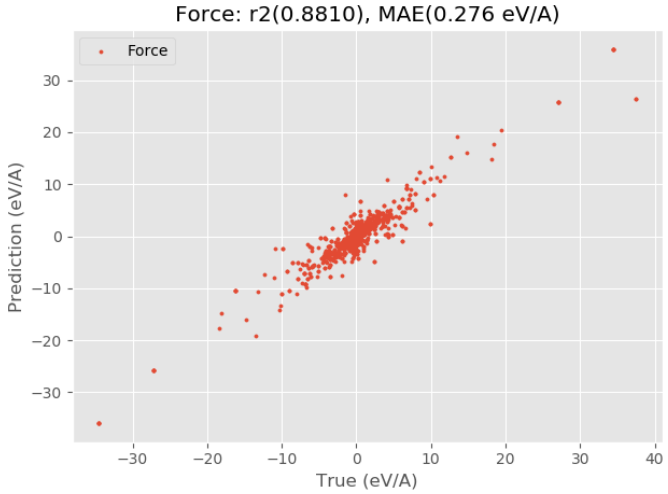
(continues on next page)

(continued from previous page)

```
nmax        : 3
lmax        : 4


2012 structures have been loaded.
Computing the descriptors...
 400 out of  400
Saving descriptor-feature data to Si-SOAP/Train_db.dat

=================================== Training ===================================

No of structures   : 400
No of descriptors  : 30
No of parameters   : 785
No of epochs       : 1
Optimizer          : lbfgs
Force_coefficient  : 0.03
Stress_coefficient : None
Batch_size         : None

Iteration   99:
eng_loss:   0.020505     force_loss:   0.022794   stress_loss:   0.000000
↪regularization:   0.000000
    Loss:   0.043299     Energy MAE:     0.1383     Force MAE:     0.2759     Stress
↪MAE:     0.0000

Iteration  100:
eng_loss:   0.020105     force_loss:   0.022543   stress_loss:   0.000000
↪regularization:   0.000000
    Loss:   0.042649     Energy MAE:     0.1380     Force MAE:     0.2756     Stress
↪MAE:     0.0000

The training time: 116.85 s
The Neural Network Potential is exported to Si-SOAP/16-16-checkpoint.pth


============================ Evaluating Training Set ============================

The results for energy:
    Energy R2      0.993670
    Energy MAE     0.138006
    Energy RMSE    0.200526
The energy figure is exported to: Si-SOAP/Energy_Train.png


The results for force:
    Force R2       0.880971
    Force MAE      0.275650
    Force RMSE     0.707787
The force figure is exported to: Si-SOAP/Force_Train.png
```

After the training is complete, the optimized weight information will be stored as `Si-SOAP/16-16-checkpoint.pth`, where `16-16` describes the neuron information. In the meantime, the code also provide graphicial output to facilitate the analysis.





If you feel that the quality of results are not satisfactory, you can contine the training from the previous run file (`Si-SOAP/16-16-checkpoint.pth`) with the `restart` option.

```
from pyxtal_ff import PyXtal_FF

train_data = "pyxtal_ff/datasets/Si/PyXtal/Si8.json"
descriptors = {'type': 'SOAP',
               'Rc': 5.0,
               'parameters': {'lmax': 4, 'nmax': 3},
               'N_train': 400,
              }
model = {'system': ['Si'],
         'hiddenlayers': [16, 16],
         'restart': 'Si-SOAP/16-16-checkpoint.pth',
        }
ff = PyXtal_FF(descriptors=descriptors, model=model)
ff.run(mode='train', TrainData=train_data)
```

```
The results for energy:
    Energy R2      0.997013
    Energy MAE     0.093162
```

(continues on next page)

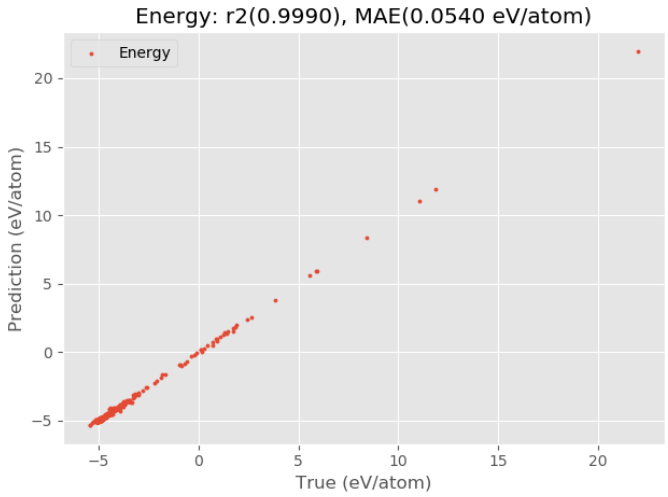
(continued from previous page)

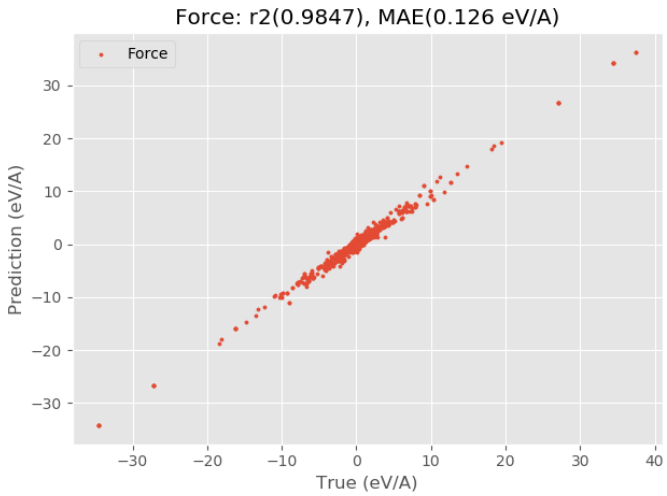
```
    Energy RMSE    0.137752
The energy figure is exported to: Si-SOAP/Energy_Train.png


The results for force:
    Force R2       0.951379
    Force MAE      0.200881
    Force RMSE     0.452365
The force figure is exported to: Si-SOAP/Force_Train.png
```

Clearly, running another 100 training steps notably reduces the MAE values. Thereforce, we can continue to train it by specifying the `epoch` option.

```
from pyxtal_ff import PyXtal_FF

train_data = "pyxtal_ff/datasets/Si/PyXtal/Si8.json"
descriptors = {'type': 'SOAP',
               'Rc': 5.0,
               'parameters': {'lmax': 4, 'nmax': 3},
               'N_train': 400,
              }
model = {'system': ['Si'],
         'hiddenlayers': [16, 16],
         'restart': 'Si-SOAP/16-16-checkpoint.pth',
         'epoch': 600,
        }
ff = PyXtal_FF(descriptors=descriptors, model=model)
ff.run(mode='train', TrainData=train_data)
```

Below are the results after 1000 steps of training.




Force: r2(0.9847), MAE(0.126 eV/A)
Force
Prediction (eV/A)
True (eV/A)
30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30
40

CHAPTER 4

Step by Step instruction

This page illustrates how to run the simulation from the scratch.

```
from pyxtal_ff import PyXtal_FF
```

4.1 Define the source of data

```
TrainData = "pyxtal_ff/datasets/SiO2/OUTCAR_SiO2"
```

At the moment, we accept the various formats:

- ase.db
- json
- OUTCAR

In principle, one can easily write a utility function to follow the style as shown in the utility section.

Among all different formats, we recommend the use of ase.db. Following ase db, you use need to add the following additional tags to each atoms object,

```
from ase.db import connect

# Suppose you have the following variables
# - struc: ase atoms objects
# - eng: total DFT energy
# - forces: DFT Forces: N*3 array
# - stress: DFT Stress: 1*6 stress [in GPa, xx, yy, zz, xy, xz, yz]
# - db_name: the filename to store the information and pass to pyxtal_ff

data = {'dft_energy': eng,
        'dft_force': forces,
        'dft_stress': stress,
```

(continues on next page)

(continued from previous page)

```
        #'group': group,
       }

with connect(db_name) as db:
    db.write(struc, data=data)
```

Note that different codes arrange the stress tensor in different order and unit. For PyXtal_FF, we strictly use `GPa` and the order of `[xx, yy, zz, xy, xz, yz]`.

4.2 Choosing the descriptor

Four types of descriptors are available (see Atomic Descriptors). Each of them needs some additional parameters to be defined as follows.

- `BehlerParrinello` (ACSF, wACSF)

```
parameters = {'G2': {'eta': [0.003214, 0.035711, 0.071421,
                             0.124987, 0.214264, 0.357106],
                     'Rs': [0]},
              'G4': {'lambda': [-1, 1],
                     'zeta': [1],
                     'eta': [0.000357, 0.028569, 0.089277]}
             }

descriptor = {'type': 'ACSF',
              'parameters': parameters,
              'Rc': 5.0,
             }
```

The `wACSF` is also supported. In this case, the number of descriptors will linearly dependent on the number of atoms in the system.

- `EAD`

```
parameters = {'L': 2, 'eta': [0.36],
              'Rs': [0.  , 0.75, 1.5 , 2.25, 3.  , 3.75, 4.5]}

descriptor = {'type': 'EAD',
              'parameters': parameters,
              'Rc': 5.0,
              }
```

- `SO4`

```
descriptor = {'type': 'SO4',
              'Rc': 5.0,
              'parameters': {'lmax': 3},
             }
```

- `SO3`

```
descriptor = {'type': 'SO3',
              'Rc': 5.0,
              'parameters': {'lmax': 4, 'nmax': 3},
             }
```

4.3 Defining your optimizer

The optimizer is defined by a dictionary which contains 2 keys:

- `method`
- `parameters`

Currently, the `method` options are

- `L-BFGS-B`
- `SGD`
- `ADAM`

If `SGD` or `ADAM` is chosen, the default learning rate is 1e-3. Usually, one only needs to specify the `method`. If no optimizer is defined, `L-BFGS-B` will be used.

4.4 Setting the NN parameters

```
model = {'system' : ['Si','O'],
         'hiddenlayers': [30, 30],
         'activation': ['tanh', 'tanh', 'linear'],
         'batch_size': None,
         'epoch': 1000,
         'force_coefficient': 0.05,
         'alpha': 1e-5,
         'path': 'SiO2-BehlerParrinello/',
         'restart': None, #'SiO2-BehlerParrinello/30-30-checkpoint.pth',
         'optimizer': {'method': 'lbfgs'},
         }
```

- `system`: a list of elements involved in the training, *list*, e.g., ['Si', 'O']
- `hiddenlayers`: the nodes information used in the training, *list or dict*, default: [6, 6],
- `activation`: activation functions used in each layer, *list or dict*, default: ['tanh', 'tanh', 'linear'],
- `batch_size`: the number of samples (structures) used for each iteration of NN; *int*, default: all structures,
- `force_coefficient`: parameter to scale the force contribution relative to the energy in the loss function; *float*, default: 0.03,
- `stress_coefficient`: balance parameter to scale the stress contribution relative to the energy. *float*, default: None,
- `alpha`: L2 penalty (regularization term) parameter; *float*, default: 1e-5,
- `restart`: dcontinuing Neural Network training from where it was left off. *string*, default: None.
- `optimizer`: optimizers used in NN training.
- `epoch`: A measure of the number of times all of the training vectors are used once to update the weights. *int*, default: 100.

Note that a lot of them have the default parameters. So the simplest case to define the model is to just define the `system` key:

```
model = {'system' : ['Si','O']}
```

Also, you can just pick the values from a previous run by defining the `restart` key:

```
model = {'restart': 'Si-O-BehlerParrinello/30-30-parameters.json'}
```

4.5 Setting the linear regression models

```
model = {'algorithm': 'PR',
         'system' : ['Si'],
         'force_coefficient': 1e-4,
         'order': 1,
         'alpha': 0,
        }
```

- `alpha`: L2 penalty (regularization term) parameter; *float*, default: 1e-5,
- `order`: linear regression (1) or quadratic fit (2)

4.6 Invoking the simulation

Finally, one just need to load the defined data, descriptors and NN model to PyXtal_FF and execute the `run` function.

```
ff = PyXtal_FF(descriptors=descriptor, model=model)
ff.run(TrainData=TrainData, TestData=TestData)
```

CHAPTER 5

Examples

We provide a series of examples of using PyXtal_FF for different material systems (see link to GitHub). We hope they can give the users a sence about the parameters setup and running time.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search