PyFit-FF User Manual

Author: J. Hickman

Version 1.0

NIST: Materials Science and Engineering Division

August 7, 2020

Contents

1 Introduction

PYFIT-FF is a tool for training artificial neural network (ANN) interatomic potentials. In this paradigm a neural network is trained to interpolate between various density functional theory (DFT) energy predictions to estimate bonding energies between atoms. The ANN training process is essentially a many-variable curve fitting optimization problem. In the case of PYFIT the optimization is done using PyTorch which is an open-source python based linear algebra, automatic differentiation, and optimization library. PyTorch includes various highly optimized vectorized operations which accelerate computation by utilizing hyper-threading on either a CPU or a GPU. The main benefits of PYFIT is that it is simple, highly portable, computationally efficient, flexible, and open source.

The following manual is written in an attempt to be useful to individuals spanning a wide range of backgrounds, anywhere from beginners to experts. The content assumes an elementary understanding of classical atomistic simulations but not necessarily an understanding of neural networks or ANN interatomic potentials. At the beginning of each section we provide a brief summary of the section's content and who the section is targeted at. Please refer to these initial paragraphs to determine which sections are most relevant to you and which can be safely skipped based on your background knowledge

2 Background

This section is targeted at individuals who have minimal background in neural networks or machine learning. The section provides a brief introduction to these topics. For the sake of brevity, we skip over many topics and only discuss content which is directly relevant to understanding PYFIT-FF and neural network potentials. If you are already familiar with these concepts then this section can be safely skipped.

2.1 Machine learning overview

Machine learning can be loosely divided into three subcategories, supervised learning, unsupervised learning, and reinforcement learning. The choice of which category a given problem falls into is largely dictated by the type of data present, knowledge about the inter-relations between the data points, and finally the algorithm's desired behavior. A brief description of these categories, along with a few common algorithm associated with each, is summarized in fig.1. In the present work, we are concerned with supervised-learning, where the data contains some known relationship between pairs inputs and known outputs.

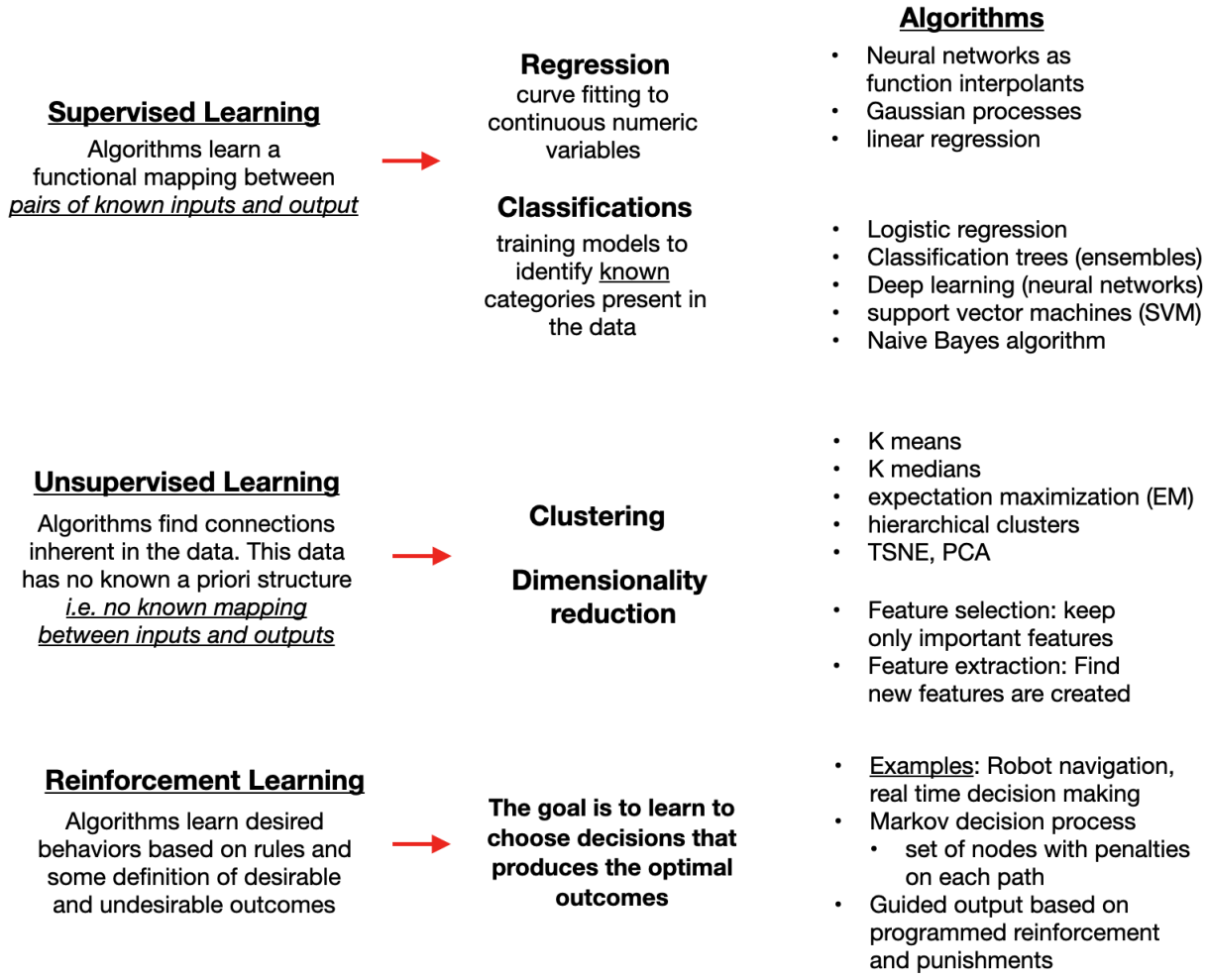


Figure 1: Brief summary of the three popular subdivisions of the field of machine learning

2.2 Regression

2.2.1 Overview

Regression refers to the process of finding an optimal parameterization $\mathbf{w}$ of some user specified model function $\mathbf{y} = \mathbf{F}(\mathbf{x}, \mathbf{w})$ which estimates the functional relationship ($\mathbb{R}^n \rightarrow \mathbf{F}(\mathbf{x},\mathbf{w}) \rightarrow \mathbb{R}^m$) between a collection of known inputs $\hat{\mathbf{x}}_i$ (in $\mathbb{R}^n$) and known outputs $\hat{\mathbf{y}}_i$ (in $\mathbb{R}^m$). The index i represents the i^{th} data point and the bold symbol denotes a vector. In reality, there is often some initial manipulation or transformation of raw data before it can be fed into the regression model. Therefore, in the parlance of machine learning, the inputs $\hat{\mathbf{x}}_i$ are often referred to as "descriptors" because they are a modified description of the raw data. Common choices for the regression function $\mathbf{F}$ include linear functions, logistic functions, and neural networks (refer to sec.2.3). Finding the optimal parameterization $\mathbf{w}$ of the function $\mathbf{F}$ is known as fitting or training. Training the model requires defining an objective function which quantifies the error ϵ between the model predictions $\mathbf{y}_i = \mathbf{F}(\hat{\mathbf{x}}_i, \mathbf{w})$ and the data $\hat{\mathbf{y}}_i$. Often the root mean squared error (RMSE) shown in fig.2 is used for the objective. The RMSE is a convenient choice because it provides a simple differentiable measure of the distance between the model predictions and the data. One then uses numerical optimization (see sec.2.2.2) to find the parametrization which minimizes the objective (error) function, i.e. the "tightest" fit. The optimization process starts with some "initial guess", often chosen at random, for the parameterization

w. The initial predictions are typically "non sense", however, as the algorithm progresses the model begins to "learn" the data. The regression process is exemplified in fig.2 for the simple case of a univariate quadratic model.

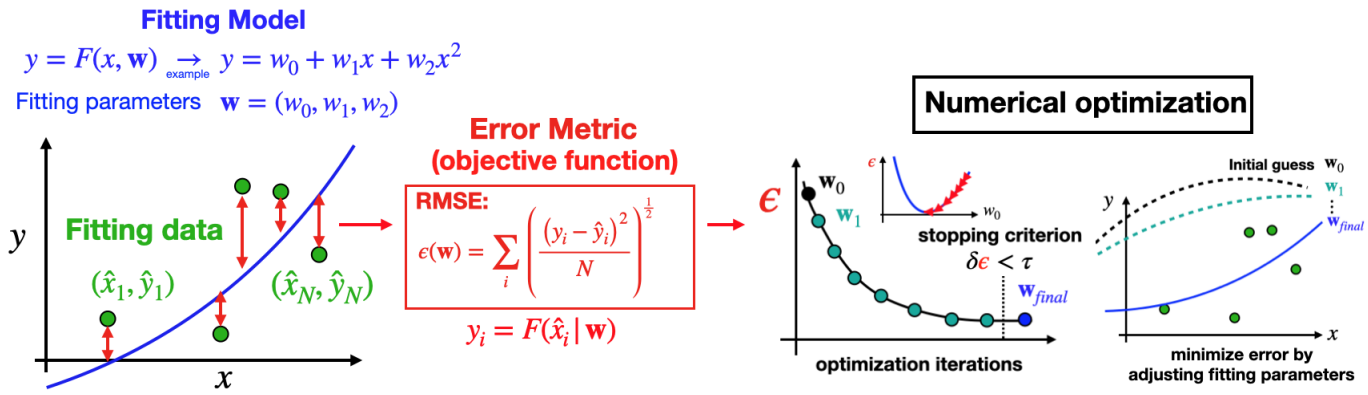


Figure 2: Regression example: univariate quadratic model

2.2.2 Numerical optimization

The goal of a numerical optimization algorithm is to find either local or global minima or maxima of an objective function. This process becomes very abstract and difficult in high dimensional spaces. Many numerical optimization algorithms exist, and many require computing the objective function's gradient. This can also be difficult or impractical for very complex functions. Fortunately there are packages such as PyTorch and TensorFlow which can track operations on select variables and automatically compute the gradients via the chain rule. This method is known as automatic differentiation. PyTorch and TensorFlow also have built in optimization libraries streamline the process of fitting neural networks or other regression models by seamlessly taking care of much of the differentiation and optimization process. Since PYFIT relies entirely on PyTorch to do the optimization "under the hood" we will not cover here.

2.2.3 Transferability and Overfitting

It is important to be aware of the range of applicability of your model when performing regression. Often this means knowing whether a given prediction is interpolating or extrapolating the data. Interpolation describes predictions made "inside" the model's training region whereas extrapolation implies predictions made "outside". If a model exhibits good extrapolations then it is said to be transferable. The meaning of interpolation and extrapolation become very abstract in high dimensional spaces but are still important.

Another important consideration is whether the model complexity is appropriate for the data, i.e. are you overfitting or under-fitting the data. Over/under fitting pertains primarily to the model's ability to interpolate although overly complex models may also suffer from poor transferability. Underfitting occurs when the model is too simple and cannot accurately estimate the data. Overfitting occurs when the model is over parameterized and can no longer reliably interpolate due to "wiggles" arising between data points, i.e. fitting to noise. To monitor overfitting it is important to use a validation set which is a random subset of the training data. If overfitting is occurring during the training process then the validation error will tend increase while the fitting error continues to decrease. When analyzing overfitting, it is often helpful to monitor mean absolute error (MAE) because it is less sensitive to outliers than the RMSE. A single un-trained point on the edge of the data-cloud

can act is an outlier and cause the appearance of slight overfitting in the RMSE when in reality it is just an untrained single outlier.

The concepts of transferability and overfitting are illustrated in fig.3 via a simple example in which a neural network approximates a univariate sine function using training data over the range $[0, 2\pi]$. Assuming a given model's complexity level is adequate and a given input is relatively "close" to the training region then you should expect decent interpolation. Transferability, on the other hand, is generally difficult to predict and depends on many factors. One way to achieve "transferability" is by using a-priori information about the data to correctly choose an appropriate model which captures the fundamental characteristics of the data. For instance, in the sine example shown in Fig.3, the user could have chosen $f(x) = Asin(\omega(x - x_o)) + C$ as their model rather than using a neural network. In this case the parameterization vector would be $\mathbf{w} = (A, \omega, x_o, C)$. If this model had been used then the user would get both excellent extrapolation and interpolation. Another way to achieve transferability is by capturing a fundamental characteristic of the model inputs, i.e. using good descriptors. For instance, if the individual knew that the data was periodic over the training interval then this could be incorporated into model by periodically remapping the input variable which would lead to increased transferability.

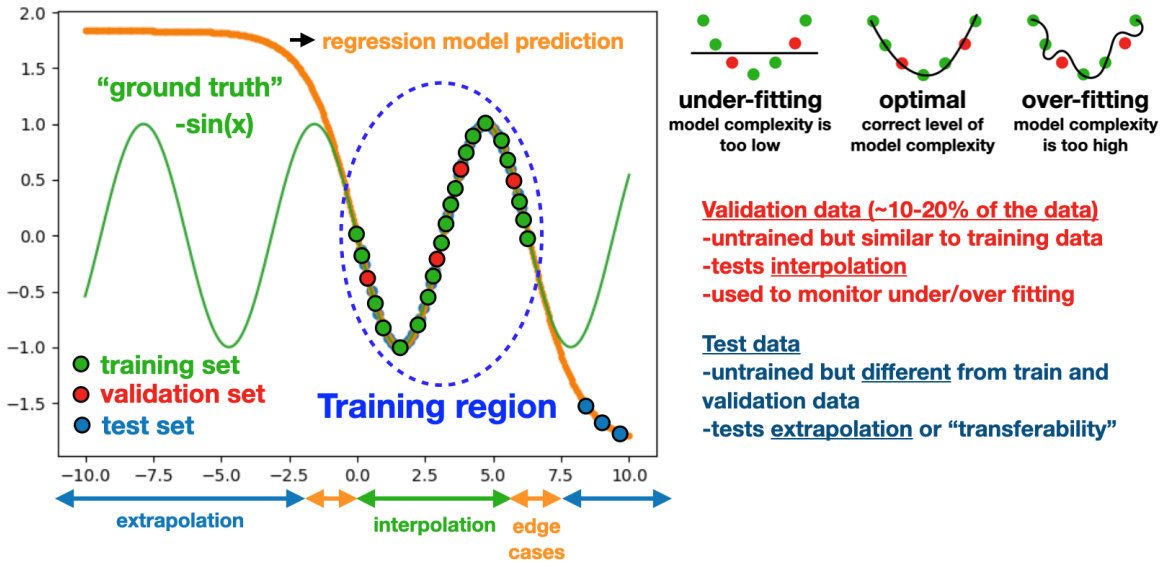


Figure 3: Regression considerations

2.2.4 Regularization

Overfitting (see sec.2.2.3) can be mitigated be using "regularization" techniques or even eliminated altogether by choosing a model with the appropriate level of complexity. In the case of neural networks, the regularization helps to ensure the the fitting parameters don't get too large during the training process, which in turn results in a "smoother" neural network function. There are many ways to regularize a model, however, the two common methods are L1 and L2 regularization. These methods introduce additional terms to the objective function, given by $L_1 = \sum_{i=1}^{n} \lambda |w_i|$ and $L_2 = \sum_{i=1}^{n} \lambda w_i^2$, which penalize large fitting parameters. Both types of regularization are available in PYFIT and can be controlled by the user by adjusting certain input parameters which will be discussed in more detail in subsequent sections.

2.3 Neural networks (NN)

This section provides an elementary introduction to how artificial neural networks work. The section is included to help the reader understand what PYFIT-FF is doing internally but is not a rigorous treatment of the theory of neural networks.

2.3.1 Artificial neuron

ANNs are a mathematical analog of biological central nervous systems (CNS) which are composed of many interconnected biological neurons. A given neuron receives electrical signals from surrounding neurons and either propagates the signal further along the network or terminates it. Similarly, ANNs are composed of various interconnected artificial neurons, also known as nodes, which can exchange information. Fig.4 shows a schematic of an artificial neuron. The neuron receives signals s_i from other neurons in the network, each of these inputs has an associated weight w_i. These weights either increase or decrease the strength of the respective signal based on its importance during the fitting process. Each neuron also has a single "bias" b which roughly defines the numerical threshold beyond which it is "activated". The weights and biases are adjustable fitting parameters used to train the neuron, similar to the coefficients in a polynomial model. The neuron's total input signal is the weighted and biased sum. This sum is then fed into an activation function which often takes the form of a sigmoid function. The output of the activation function is the output of the neuron, this signal then moves further along the network and the process repeats itself.

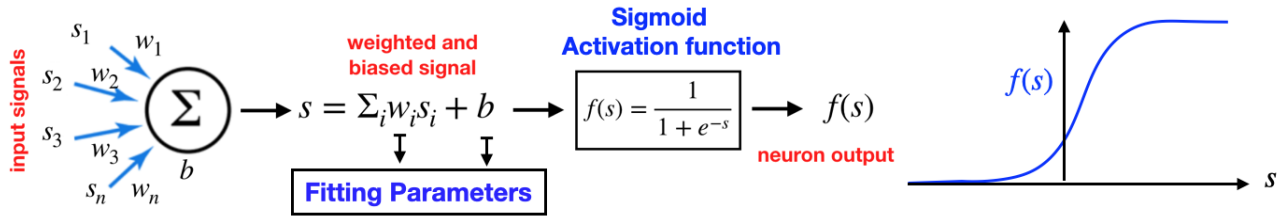


Figure 4: Artificial neuron schematic

2.3.2 Artificial neural networks

The primary utility of ANNs is their ability to act as a universal non-linear fitting functions. This makes them particularly useful as interpolating functions for high dimensional regression problems (see sec.2.2). To better understand ANNs it is instructive to consider a simple example as shown in fig.5. Even though this is a small network we can use it to illustrate several fundamental aspects of ANNs. These concepts all generalize to larger networks and include the following.

- Interconnection:
    - There are many ways to connect the neurons in an ANN. The various methods are appropriate for different types of data and applications, however, in the present work we focus on the case of fully connected feed-forward networks. In this paradigm the interconnection between the nodes take the form of consecutive layers. The signal propagates from left (the inputs side) to right (the output side) with every node in a given layer being connected to every node in the subsequent layer.

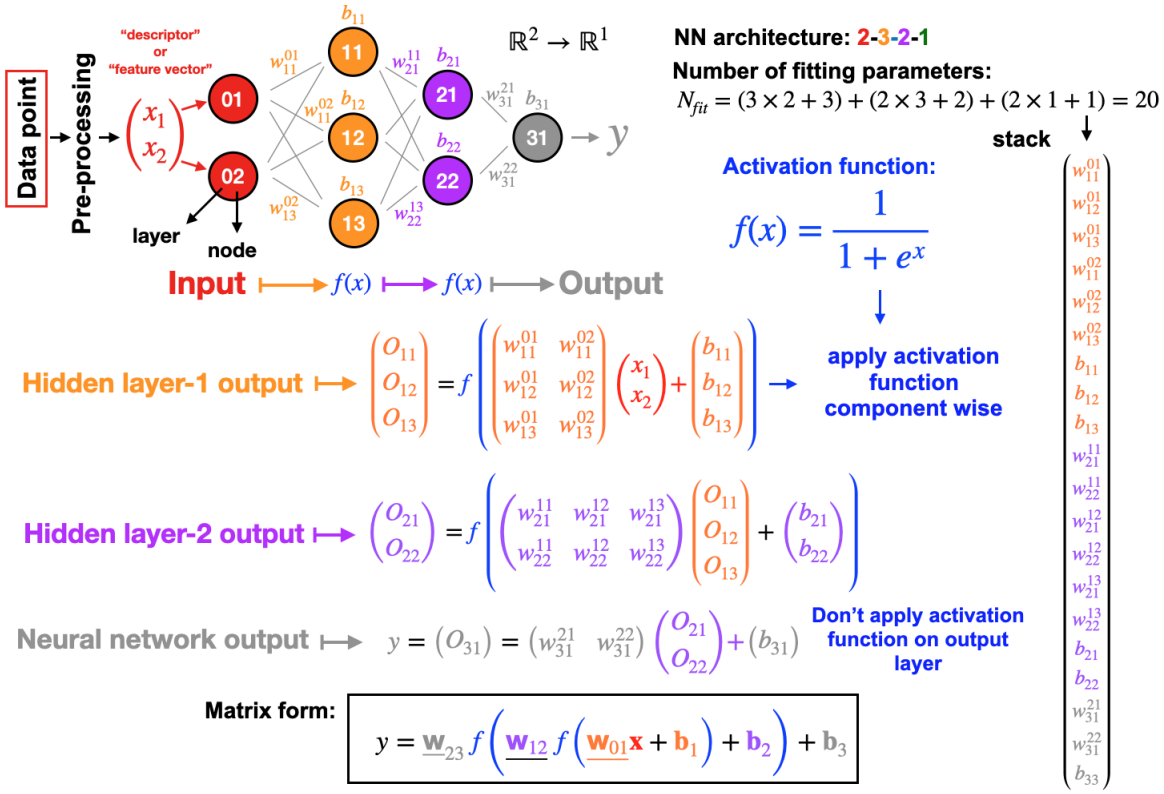


Figure 5: Simple artificial neural network example

- Architecture:
  - Neural networks are composed of an input layer, one or more "hidden" layers, and an output layer. The architecture can be specified by stating the number of nodes in each of theses layers, in fig.5 the architecture is 2-3-2-1.
  - The dimensionality of the input and output layers are dictated by the data. In fig.5 the ANN maps a two dimensional vector $\boldsymbol{x}$ to a scalar y, i.e. $\mathbb{R}^2 \to \mathbb{R}^1$. Therefore the input and output layers have dimensions 2 and 1 respectively. In general a neural network can map between arbitrary spaces, i.e $\mathbb{R}^N \to \mathbb{R}^M$.
  - The size and number of the hidden layers dictate the complexity of the ANN model. Neural networks with more hidden layers are said to be "deeper". However, the number of nodes in each layer also effects the model complexity.
- Activations function choice
  - As mentioned above the activation function is often a sigmoid function, however, there are many other common choices as well. Typically only the hidden layers are passed through the activation function. The input layer simply passes the input vector and output of the final layer is just the weighted and biased sum.
- Evaluation
  - The figure shows the various analytic steps for evaluating the ANN. Notice that this evaluations is simply the repeated application of consecutive linear matrix transformations which are then fed component wise into the non-linear activation function.

- Fitting parameters:
  - The collection of all weight and bias terms in the various matrices constitute the fitting parameters of the ANN. The total number of fitting parameters can be calculated by counting the components of the various weight matrices and bias vectors. In the figure, there are $(3 \times 2 + 3)$ fitting parameters describing the connection between the input layer and the first hidden layer. The number of fitting parameters in the other layers can be similarly calculated leading to the a total of $(3 \times 2 + 3) + (2 \times 3 + 2) + (2 \times 1 + 1) = 20$ fitting parameters. Notice that these matrices can be "unwound" and stacked into a single long vector. This is how PYFIT stores the ANN potential in its output files. This long vector can also be converted back into the various sub-matrices using knowledge of the ANN architecture .
- Training
  - In order for an ANN to be useful it must be trained to learn a desired functional relationship. This is the same as the optimization process for any other regression problem described in sec.2.2.

The neural network in fig.5 can also be reformulated into a "batch" implementation in which multiple inputs are evaluated simultaneously as shown in fig.6. This is achieved by transposing various matrices. This "batch" formalism is the method used by PYFIT.

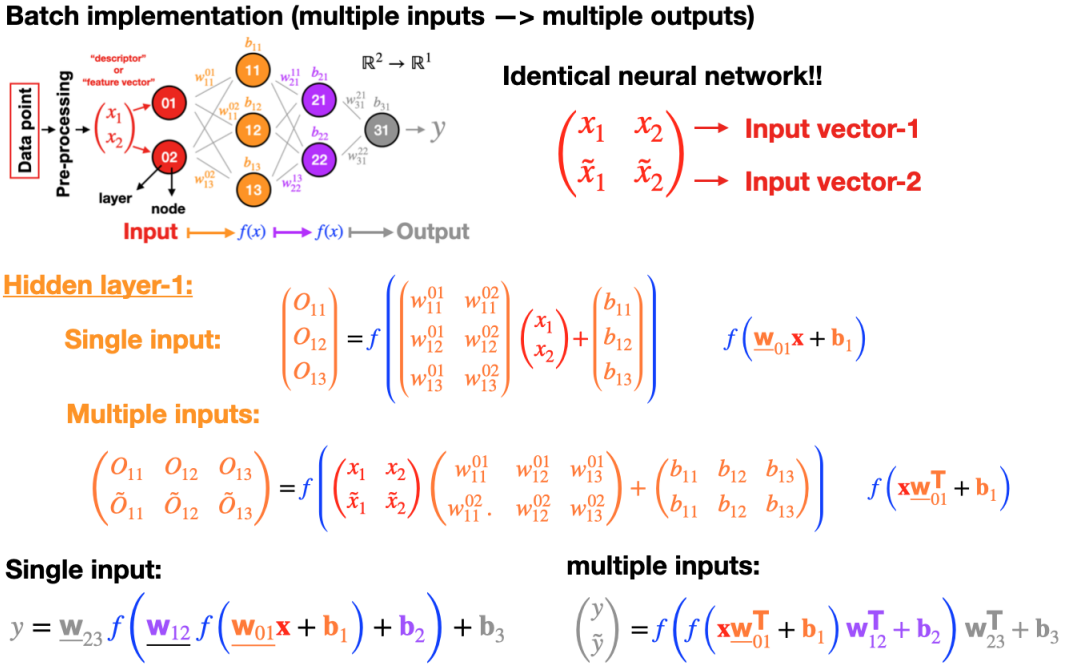


Figure 6: "Batch mode" artificial neural network example

2.4 Machine learning interatomic potentials

2.4.1 General idea

In the present discussion we will only provide a brief overview of how machine learning potentials work. A thorough discussion on the topic can be found in the literature. For a given material, the

general idea is to train a regression model (neural network, Gaussian process, etc) to estimate the material's potential energy surface (PES), denoted $U = U(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N)$. This function gives an energy prediction for any arbitrary configuration of atoms $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N$. In order to use a regression model one first needs to "fingerprint" the local environment around each atom. This is done by quantifying the environment around the i^{th} atom as a vector of fixed length, denoted $\boldsymbol{G_i}$. The vector $\boldsymbol{G_i}$ is computed by feeding the atom's neighbor list into a set of analytic functions which output the fingerprint vector $\boldsymbol{G_i}$. This is then used as the input to the regression model which outputs the atom's energy. The energies of all the atoms are summed and the model can be trained, by adjusting the fitting parameters, to reproduce DFT energy predictions. This workflow is summarized graphically in fig.7.

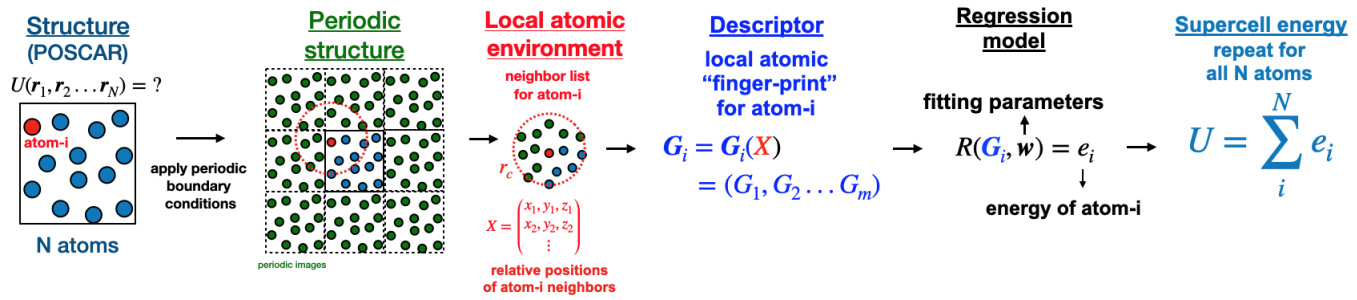


Figure 7: Machine learning potential flowchart

2.4.2 Training

This is the same as the regression optimization process described in sec.2.2. Specifically, the ANN fitting parameters are adjusted using a numerical optimization algorithm to minimizes an error metric to achieve a good fit of the data. At first the fitting parameters are randomized and the ANN outputs "non-sense" but as the optimization progresses the ANN eventually "learns" to reproduce the training data. This workflow is summarized graphically in fig.8.

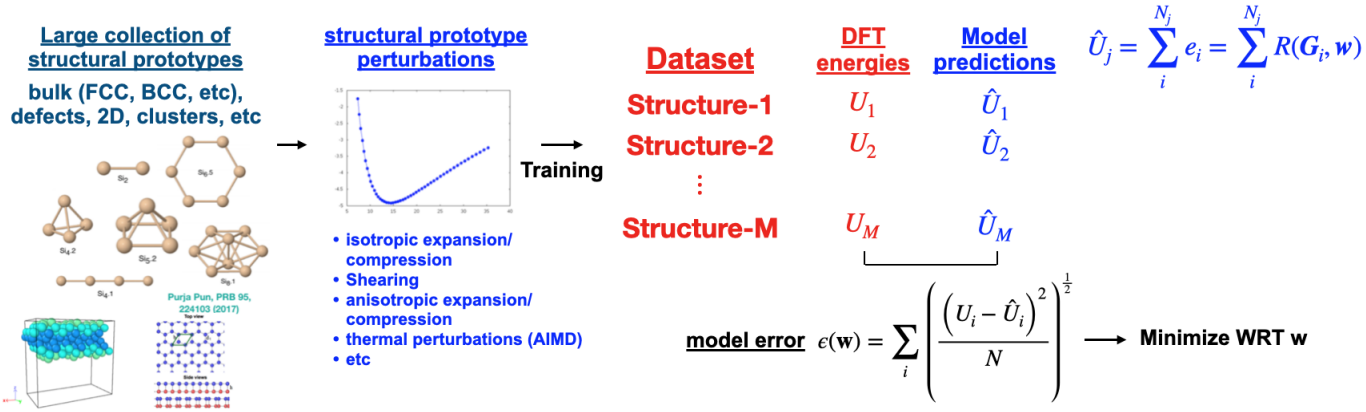


Figure 8: Machine learning potential flowchart

2.5 Local Atomic Structural Fingerprinting

As mentioned in the pervious section, one needs to quantify the local environment around a given atom in order to use the various regression techniques. The following sections outline this

fingerprinting process in more detail.

2.5.1 Method-1:

Currently the only atomic finger printing scheme coded in PYFIT is the method developed by Purja Pun and Mishin [2]. In this method the atomic environment surrounding neighboring atom is within r_c the the cutoff where a atom $\boldsymbol{r}_i$ is represented as a single vector denoted $\boldsymbol{g}_i$ which both radial and angular .. The radial component of $\boldsymbol{g}_i$, consists of a set of $\tilde{N}$ Gaussian functions given by Eq.1

$$f_n(r) = \frac{1}{r_n} e^{(r-r_n)^2/\sigma^2} f_c(r), \quad n = 0,\ 1,\ 2, \ldots \tilde{N} \tag{1}$$

Each of of the $\tilde{N}$ functions is shifted by a user specified parameter r_n which sets the location function's center. The term σmust also be specified by the user and controls the width of each of the $\tilde{N}$ functions. The scaling by r_n^{-1} ensures that atoms which are closer to $\boldsymbol{r}_i$ contribute more significantly to the local structure parameter. The various $f_n(r)$ functions can be intepretted as defining concentric search zones around $\boldsymbol{r}_i$ each of which is centered r_n which probe interatomic distances of the neighbors of $\boldsymbol{r}_i$. In Eq.1 the term $f_c(r)$ defines a cutoff function as shown in Eq.2which forces the radial terms smoothly to zero at r_c (where d is an additional parameter defining the cutoff range).

$$f_c(r) = \begin{cases} \dfrac{(r-r_c)^4}{d^4 + (r-r_c)^4} & r < r_c \\ 0 & r \geq r_c \end{cases} \tag{2}$$

The angular contribution to the structural environment is accounted for using three body angles between pairs of neighboring atoms where θ_{ijk} gives the angle between $\mathbf{r}_j$ and $\mathbf{r}_k$ (with $\theta_{ijk} \in [0, \pi]$). The cosine of θ_{ijk} is feed into Legendre polynomials $P_m(x)$ of varying orders m where $P_0(x) = 1$ and $P_1(x) = x$ and all higher orders can be obtained from the recursion relation $(m+1)P_{m+1}(x) = (2m+1)xP_m(x) - mP_{m-1}(x)$. The total number of Legendre polynomials M used in the model as well as the choice of which polynomial orders are used represent additional hyper-parameters which the potential developer needs to specify. This choice of angular term is inspired from concepts in electrostatics and are composed of are chosen to mono-pole ... . The individual components of $\boldsymbol{g}_i$ are computed by summing the product of radial and angular contributions over all pairs of neighboring atoms as shown in Eq.3

$$\tilde{g}_i^{m,n} = \sum_j \sum_k P_m(\cos\theta_{ijk}) f_n(r_{ij}) f_n(r_{ik}). \tag{3}$$

In Eq.3 for each of the $\tilde{N}$ values of n there are M Legendre polynomials orders m and therefore the total number structural descriptors in the $\boldsymbol{g}_i$ vector is given by $L = \tilde{N}M$. Finally to ensure the more desirable range for neural networks regression modeling (i.e. $g_i^{m,n} \approx [-10, 10]$). The final step of the process is to apply the transformation shown Eq.4 in to the $\tilde{g}_i^{m,n}$ components

$$g_i^{m,n} = sinh^{-1}(\tilde{g}_i^{m,n}) \tag{4}$$

3 PYFIT-FF :

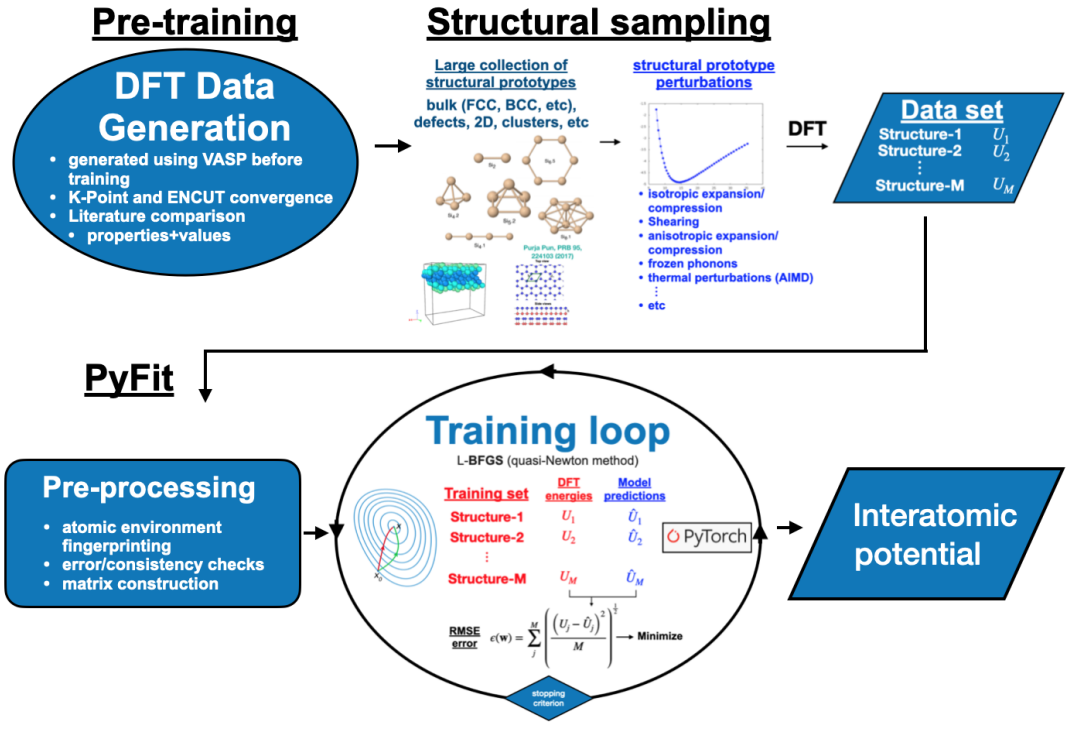
The following subsections are a practical overview on how use PYFIT. These sections are indented for anyone who is new to PYFIT, regardless of their background. The primary focus is on how to download the code, the structure and meaning of the code's inputs and outputs files, and how to run the included example. These sections assume that you are already familiar with the content from section(2) and section(3).

3.1 Functionality

Because this is the first iteration of the code the current functionality is relatively limited, however, we are currently working on adding the additional functionality to PYFIT.

- Current functionality
  - Single component mathematical ANN training using the local atomic environment descriptors developed by Purja-Pun and Mishin
- Beta version
  - Single component PINN training using the local atomic environment descriptors developed by Purja-Pun and Mishin
- Future plans
  - Add Behler-Parrinello LSP for the case of single component ANN
  - Multi-component ANN training
  - Multi-component PINN training

3.2 Workflow



3.3 Getting started

The instructions can also be found in the README file found on the GitHub page . For completeness we also include the instructions here. Currently PyFit-FF has been tested and found to work efficiently on *macOS and Ubuntu-Linux*. The code has not yet been tested on Windows, however, it should work provided the required dependancies are met.

3.3.1 Dependencies

The following dependencies are required:

- PyTorch
- Python 3.x
- NumPy

The instructions below are for Linux and Mac systems. Instructions for Windows machines will be added in the future.

- **Installation option-1 (recommended)**:
  - Condais a popular open source software package management system. The following method will create a "self-contained" Conda environment to install python3.7 and the various dependencies for PyFit. Because the environment is self contained it will not effect your system's default python distribution.
  - **Step-1: Install Conda**
    - The Conda installation process is well documented at the following link
    - https://docs.conda.io/projects/conda/en/latest/user-guide/install/
  - **Step-2: Create a dedicated Conda environment for PyFit**
    - Executing the following commands from the command line will create a conda environment named TORCH3.7 and will install the required dependencies. Be sure to answer "yes" to the various prompts. The entire process should only take a few minutes and take up roughly 2.5 GB of disk space.
      - `conda deactivate`
      - `conda create -n TORCH3.7 python=3.7`
      - `conda activate TORCH3.7`
      - `conda install -c pytorch pytorch`
    - **Note**: NumPy will automatically be installed during the PyTorch installation
- **Installation option-2 (manual):**
  - **Note**: If you already have have python3.X installed on your system then you can install the PyTorch package using the following pip command. Use caution, if NumPy is already installed, this method will likely re-install NumPy to a different version which could potentially 'break' things.
    - sudo pip3 install torch numpy

3.3.2 Downloading PyFit-FF

Once the dependencies are met then PYFIT can be downloaded from Github using the following command. This will make a directory called "PYFIT-FF" on your machine.

```
git clone https://github.com/jfh3/PYFIT-FF
```

Alternatively you can manually download the folder using the green "code" button on https://github.com/jfh3/PYFIT-FF and clicking "Download Zip"

3.3.3 Closer look inside

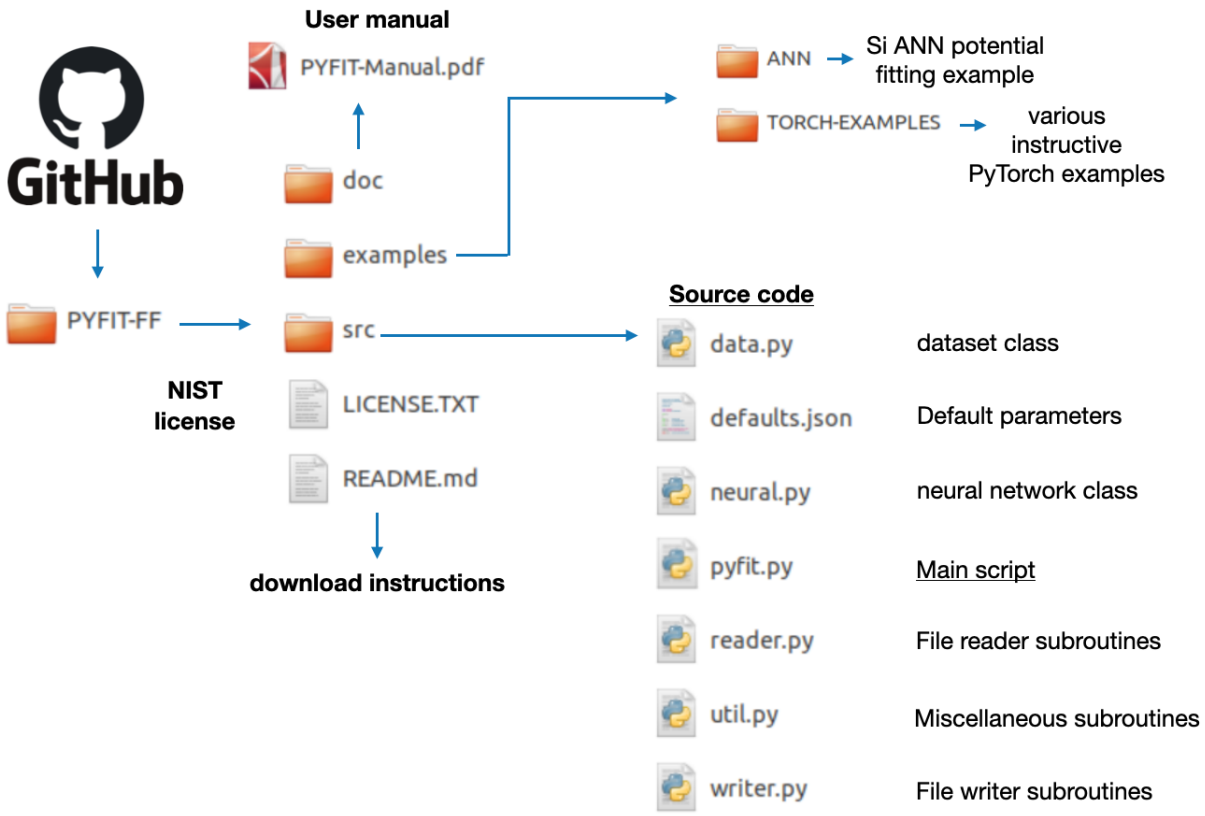


Figure 9: PYFIT-FF file directory obtained from downloading from Github.

3.3.4 Running the code

PyFit is simply a collection of python scripts and therefore it can be run just like any other python code. However, the primary script is pyfit.py. This script runs the program and calls the other subroutines. The first line of pyfit.py is '#!/usr/bin/env python3'. This line allows the file to run like an executable from anywhere on your computer as long as you point to it. The code will run and output files from inside the directory from which it was called but will also 'see' and utilize the various subroutines in PYFIT-FF/src. The code does require a single input json file. This input file is discussed in detail in the 'PYFIT-FF/doc/PYFIT-Manual.pdf' file. When you run PyFit you need to include the path to the input file as a command line argument (see below).

Option-1 (recommended):

To make PyFit accessible from anywhere on your computer simply find a permanent location for the PYFIT-FF directory and put a link to the pyfit.py file in ~/bin/. This will make the file part of your system's PATH and allow you to easily access it. I typically keep the PYFIT-FF directory in ~/bin/ and create the link using the following commands:

```
mv PYFIT-FF ~/bin
cd ~/bin/
ln -s PYFIT-FF/src/pyfit.py  pyfit
```

To run the example, simply navigate to "PYFIT-FF/examples/MNN" and run the following commands

```
cd PYFIT-FF/examples/MNN
pyfit input.json
```

Notice that the command "pyfit input.json" points to the link in ~/bin/pyfit and executes the program. If everything is working the code should start and begin to write output to the screen and to various log files.

Option-2:

Alternatively if you're just testing PyFit-FF then you can simply navigate to the example directory and point to the pyfit.py script using the relative path "../../src/pyfit.py".

```
cd PYFIT-FF/examples/MNN
../../src/pyfit.py input.json
```

These commands are equivalent to the more familiar

```
cd PYFIT-FF/examples/MNN
python3.7 ../../src/pyfit.py input.json
```

3.3.5 Removing PyFit

To permanently remove PyFit from your system simply delete the Conda environment and the code directory.

- rm -rf PYFIT-FF
- conda remove --name TORCH3.7 --all

3.4 Inputs File:

In order to run the code requires various input files which tell it how to carry out the training run. In the following sub-sections we discuss the meaning and formats of these files in detail.

3.4.1 NN file:

The NN file contains the neural network's weights and bias values, or a flag signaling their randomization, and the LSP hyper-parameters. This file has a particular format and the meaning of the various lines are described in fig.10. We also include a small "toy" NN to show the ordering convention

for the weights and biases and the hardcoded ordering of the LSP input vector. The meaning of the LSP hyper-parameters is discussed in more detail in sec.2.5.

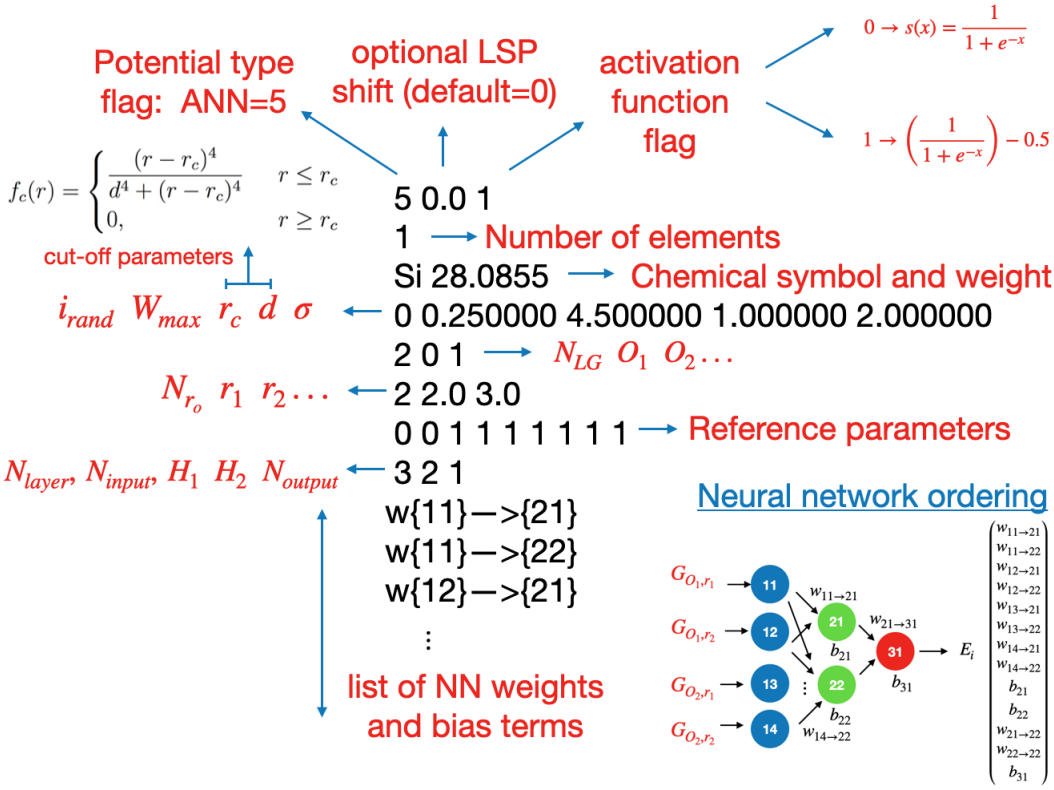


Figure 10: Neural network potential file format description

The meaning of many of the file's values are obvious from the captions in Fig.10 but certain quantities require further discussion.

- i_{rand}**:** This parameter tells the code whether or not to randomize the NN at beginning or the run. If $i_{rand} = 0$ the code will not randomize but instead read the pre-existing weights and biases stored in the file. If $i_{rand} = 0$ then the code will randomize a NN while reading this file.
- W_{max}**:** This parameter is the magnitude of the maximum (and minimum) possible weight and bias generated during the NN randomization. This parameter is ignored if $i_{rand} = X$
- r_c, h_c**:** These parameters define the cutoff radius and range for the cutoff function $f_c(r)$
- σ**:** This controls the width of the Gaussian functions which define the radial term in the LSP calculations
- N_{LG}**:** This specifies the number of Legendre polynomials to be used in the LSP calculations. The following N_{LG} numbers specify the order of the polynomials to be used
- N_{r_o}**:** This specifies the number of Gaussian functions to be used in the LSP calculations. The following N_{r_o} numbers are the locations of the centers for these Gaussian functions
- The remainder of the file is the neural network weights and bias values stored in a long column. Again we include the "toy" NN to show the meaning of the ordering of theses values.

3.4.2 Input.json:

Each bullet point represents one of PYFITS option parameters along with the default value to the right of the parameter. The order of these parameters in the input.json file does not matter. The following parameters can optionally be added to the input file, if they are not included the code with use the default values stored in "/PYFIT-FF/src/defaults.json".

- **"pot_type" : "NN"**
    - As the name suggests, this parameters tells the code what type of inter-atomic potential to train. Currently the default is "NN" is which is also the only option. This tells the code to train a purely mathematical neural network, however, in the future additional options such as PINN will be added.
- **"pot_file" : "NN1.dat"**
    - This parameter is the path to the initial neural network file. The format of that file is described in detail in sec.3.4.1, but essentially it provides the code with various hyper-parameters to use to quantify the local structure as well as the neural network architecture. By default the code will look for a file called "NN1.dat" is the directory from which the code was run, however, one can also modify the input.json file to tell the code to look elsewhere.
- **"dataset_path" : "train.dat"**
    - This provides the code with the path to the dataset file where the various POSCAR structures and energies needed to train the neural network are stored. The format of this file is described in detail in section X. By default the code will look for a file called "train.dat" in the directory from which the code was run, however, one can also modify the input.json file to tell the code to look elsewhere.
- **"re_randomize" : true**
    - This parameter tells the code to randomize the neural network weights and bias terms at the beginning of the training run (i.e. to start "fresh"). This randomization can also be achieved through parameters in the initial neural network file as described in sec.3.4.1, however, the 're_randomize' parameter can be used to override the options in the NN file and force an initial randomization.
- **"write_lsp" : false**
    - This tells the code whether or not to write a file containing the local structural fingerprints for each atom during the code's setup phase.
- **"dump_poscars" : false**
    - This parameter tells the code whether to write an individual POSCAR file for each structure in the dataset file. This is useful for visualizing the data set in OVITIO. The default value is "false" because the POSCAR files tend to clutter the output directory and are only needed when visualization is required.
- **"use_cuda" : false**

- Specify whether or not to use a GPU for multi-thread operations. By default the parameter is "false" and the code will do multi-threading using the CPU level. If this parameter is set to "true" but the code can't find a GPU it will default back to using the CPU cores.

- **"constrain_WB" : 0.0**

  - This parameter is used to set an upper/lower limit on the maximum possible value of the neural network's fitting parameters (i.e. weights and bias) which can have a regularizing effect. This is done using a smooth and continuous function to avoid discontinuities which may arise with a sharper cutoff, at present this function is given by $\tilde{w} =$ where w are the unconstrained weights and A='constrain_WB'. If 'constrain_WB'==0 then the neural network is unconstrained and the weights and biases terms can take on any value, however, if the 'constrain_WB'>0 then this value specifies the maximum/minimum possible weight and bias.

- **"save_every1" : 100**

  - This parameter specifies the number of optimization iterations between logging various statistical quantities associated with the fitting process. The data controlled by this parameter takes up a negligible amount of disk space and therefore it is not a problem to report frequently, however, reporting does require a small amount of extra computation and therefore is good to avoid reporting at every single step which may slow the fitting (i.e. don't use 'save_every1'=1). At the end of the fitting loop the code will always save the final state using the step=111111.

- **"save_every2" : 1000**

  - This parameter specifies the number of optimization iterations between writing the the current NN as well as various diagnostic files. These files take up space on the disk and clutter the output directory. Therefore, it is good not to write too frequently. At the end of the fitting loop the code will always save the final state using the step=111111 to record the final state before exiting.

- **"cnst_final_bias" : true**

  - Sometimes it is beneficial to manually control the value of the neural network's output layer bias vector (i.e. make it constant and no longer treat it as a fitting parameter). If 'cnst_final_bias'=true then this will occur with the final vector being manually set using the 'final_bias' option. If 'cnst_final_bias'=false then the 'final_bias' option is ignored and the NN with be trained in the normal sense with the final layer being used as a bias term.

- **"final_bias" : 0.0**

  - This is the manually specified value of the neural network's output layer bias vector which is used by the code if 'cnst_final_bias'=true (see "cnst_final_bias')

- **"train_edges" : true,**

  - When training neural networks it can be beneficial to include data points on the "edge" of the "data cloud". This is because data sampling is often less dense in these regions. If "train_edges"=true the code always include the points with minimum and maximum volume from each structural group in the training set rather than using them or validation.

- **"max_iter" : 100000**
    - Specify the maximum number of LBFGS iterations (the code will exit once this is reached).
- **"rmse_tol" : 0.000001,**
    - This is the stopping criterion or training tolerance. It can be used to automatically stop the code when the training "bottoms out" in a local minima during the training run. The code will compare the current RMSE with the value from 10 steps ago and if the absolute change is less then "rmse_tol" then the code will save the final state of the neural network and exit. If this parameter is set to 0.0 then the code will continue until "max_iter" is reached.
- **"ramp_LR" : true**
    - When restarting a training run from a pre-trained initial neural network it can be beneficial to slowly increase the learning rate from a small value (i.e. small search steps) to a larger one (i.e. larger steps with faster convergence). This is done so that the optimizer doesn't immediately jump out of the current minima and into an entirely different region of the search space. If "ramp_LR"=true then this ramping occurs and obeys function INSERT where the hyper-parameters of this function are specified by the option "LR_o","LR_f", and 'mid_ramp'. If "ramp_LR"=false then the learning rate is constant throughout the entire fitting run and is set at the value specified by "LR_f".
- **"mid_ramp" : 50**
    - This specifies the number of steps after which the learning rate ramping process will have reached its midpoint. Small values of "mid_ramp" result in rapid ramping of the learning rate whereas large values of "mid_ramp" result in gradual increases (see "ramp_LR" for more details).
- **"LR_o" : 0.001**
    - This is the initial learning rate at the beginning of the "ramping" stage (see "ramp_LR" for more details).
- **"LR_f" : 0.10**
    - This is the final learning rate at the end of the ramp in stage (see "ramp_LR" for more details). If "ramp_LR"=false then the learning rate is constant throughout the entire fitting run and is set at the value specified by "LR_f".
- **"fraction_train" : 0.8**
    - This parameter tells the code how to split the dataset into the training and validation sets, as the name suggests the parameter specifies the fraction of the entire read data set to be used for training. The default is 0.8 which means use a 80/20 training and validation split. If this parameter is set to one then training is preformed on the entire set and no validation occurs.
- **"lbfgs_max_iter" : 20**

- PYFIT uses the LBFGS optimization method, this method uses a line search approach during the minimization. One of the method's parameters is how many points to carry out during the line search. The default for PyTorch is 20 which is also the default for PYFIT and seems to be a good choice, however, the user can also decrease the value using the PYFIT input file if desired.

- **"u_shift" : 0.0**

  - This parameter specifies constant per-atom energy shift which is applied to all DFT energies read by the code. The value is in units eV/atom and can be used to ensure the DFT cohesive energy matches the experimental value.

3.4.3 Dataset file:

The dataset files contains the atomic structures and the associated energies needed for the training, validation, and test sets (see sec.2.2 for more discussion of these data sets). This information needs to be stored as a simple a sequence of POSCAR files followed by the associated DFT energies (in eV) (more information on the POSCAR format can be found online at the following link). It is important to note that the DFT energies in the file (denoted here as U_{DFT}) are shifted by the parameter "u_shift" defined in the input.json file such that the total potential energy of a given structure used by PYFIT is $U = U_{DFT} + N \times u_{shift}$ (where N is the number of atoms in the structure). An example of a simple dataset file with only three structures is shown in Fig.11. For back-compatibility with other ANN training codes the PYFIT-FF dataset file is currently hardcoded to only accept cartesian type POSCAR files using the format shown in shown in Fig.11. There are error checks in the code but it is important that users stick directly to this format. One other important thing to note is the comment line of the POSCAR files. This line is particularly important in PYFIT because it is used to identify different "structural groups" present in the dataset (see sec.9). The code identifies the first string in the comment line and uses it as the structure's group ID (GID). Each time the code finds a unique GID it will define a new structural group, however, if the GID has already been encountered then the code simply adds the structure to the appropriate group. The GID tags in the comment lines can be arbitrary strings, however, it is good to come up with some code so you can distinguish between groups. One way to do this is by defining GIDs based on bulk atomic structure (FCC, DC, BCC, HCP, LIQ, AMORP .. etc), types of deviations from equilibrium (ANISO, ISO, AIMD ... etc), or whether the structure contains defects (SF=Stacking fault, PD=Point defect, SURF, ... etc). These groups are diagnostic for trying to troubleshoot why certain properties of the potential deviate from the expected data values. For example, if a particular structural group is not fitted tight enough then the predicted properties associated with that group will likely not match the DFT.

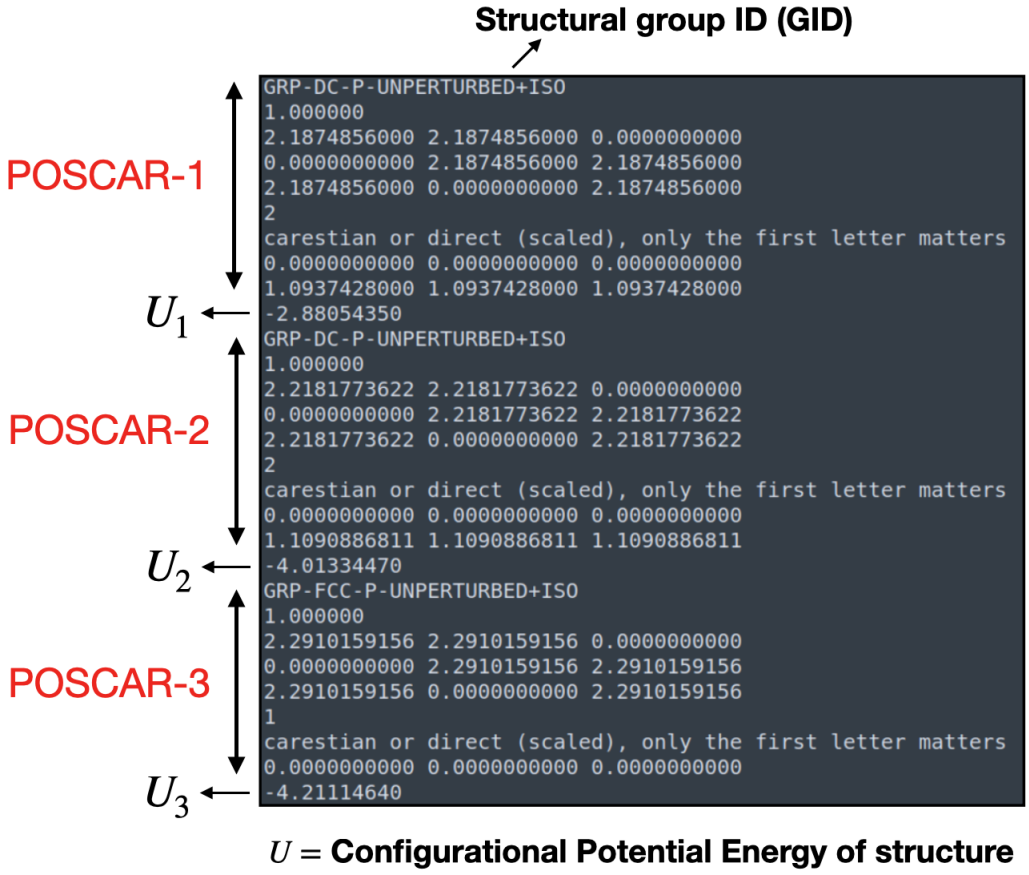


Figure 11: Example data set file with just three structures. The first two structures belong to the same group, i.e. which represents defect free diamond structures with their atoms in the equilibrium positions but isotropically expanded or compressed from the equilibrium lattice constant. As well as a third structure which if FCC and similarly deviated from equilibrium.

3.5 Outputs Files:

3.5.1 Description:

The code sequentially outputs various files to monitor the optimizer's progress. The output files all begin with a characteristic string, denoted here as PREFIX. The first portion of the PREFIX is the string "PF", to denote PYFIT, this is followed by a string specifying the date and time at which the run started, for example, PREFIX="PF-2020-06-04-H15-M13-S40". The "PF" makes it easy to quickly delete the files from a previous run if desired, i.e "rm PF-*". The use of the timestamp ensures that the code outputs files with unique names and therefore doesn't overwrite previous files when multiple runs are carried out from the same directory. The prefix is hardcoded, however, if you want to change it then modify line-9 of "src/writer.py". A description of each output file, along with its format, is summarized in the following bullets.

- **log.dat file:**
  - The code writes a single file with the naming conventions PREFIX-log.dat. This file sequentially reports what the code is doing, along with the parameters it is encountering. This file has no particular format, although it is somewhat similar to a JSON file. The information in the log file is more or less self-explanatory and it is primarily used to log the run parameters and for debugging purposes.

    - ◇ Note: The quantities which are written to the log file are also output on the screen if PYFIT is run directly from the command line.

- **SB.json**
    - The code accumulates all of the run parameters in a "snowball" (SB) dictionary. One of python's nice attributes is how it handles the scope of dictionaries. In python dictionaries are objects which can be passed between functions and edited inside those functions without re-defining and returning a new dictionary, i.e. the function can edit the original dictionary that it was passed. In that sense the dictionary acts almost like a global object which are useful for sharing information between modules and functions. This dictionary is saved as a JSON file denoted PREFIX-SB.json. The file contains similar information as the log file, however, the use of the JSON format enables more seamless interfacing with external codes.

- **NN files:**
    - The code periodically writes the current state of the neural network to a "NN" file using the naming convention "PREFIX-NN-STEP.dat" where STEP refers the training iteration, e.g. "PF-2020-06-04-H16-M45-S23-NN-100.dat". These files have the same format as those described in section.3.4.1. They serve as checkpoints for the algorithm and can also be used as input files to molecular dynamics simulators such as in LAMMPS [1] and PGMC. The number of iterations between this file save for this file is controlled by "save_every_1" in the input.json file, see sec.3.4.2.

- e_vs_v **files:**
    - As the name suggests these files contain volume vs energy data for the various structures. Each dataset, i.e. training, validation, and test, will get its own file. The naming convention is , for example 00-e_vs_v-test-1000.dat. The first column is the volume per atom in the structure (V/N), the second column is the energy per atom DFT energy (E_{DFT}/N).

- **err file:**
    - This file is a simple columnar text file which reports various terms of the objective function as function of the iteration step. This file only reports metrics associated with the training set, metrics associated with other data sets are reported in the stat files described below. The frequency at which the values are written is controlled by the save_every parameter in the input.json file, see sec.3.4.2. If a particular term in the objective function is not being used then it will be reported as a zero in the file.

- **stat files:**
    - These files describe the statistical state of the training process and contain various error metrics of the various data sets; training, validation, test.

- **LSP file:**
    - This file is only written if "write_lsp" equals "True" in the input.json file. The file is primarily for debugging purposes and therefore it's not written unless requested since the file is often relatively large and rarely used. It is written before the data is partitioned into different subsets and therefore each line corresponds to a particular atom in the total dataset.

References

- [1] Steve Plimpton. *Fast parallel algorithms for short-range molecular dynamics*. Journal of computational physics, 117(1):1–19, 1995. 3.5.1

  [2] G P Purja Pun, R Batra, R Ramprasad, and Y Mishin. *Physically informed artificial neural networks for atomistic modeling of materials*. Nature communications, 10, 2019. URL https://www.nature.com/articles/s41467-019-10343-5.pdf?origin=ppub. 2.5.1