# Amp Documentation

***Release 0.5***

**Andrew A. Peterson, Alireza Khorshidi**

February 24, 2017

Contents

Amp is an open-source package designed to easily bring machine-learning to atomistic calculations. This project is being developed at Brown University in the School of Engineering, primarily by **Andrew Peterson** and **Alireza Khorshidi**, and is released under the GNU General Public License.

Please see the project's git repository for the latest version, news, or a place to report an issue.

You can read about Amp in the below paper; if you find Amp useful, we would appreciate if you cite this work:

> Khorshidi & Peterson, "Amp: A modular approach to machine learning in atomistic simulations", *Computer Physics Communications* 207:310-324, 2016.

**Manual**:

CHAPTER 1

# Introduction

Amp is an open-source package designed to easily bring machine-learning to atomistic calculations. This allows one to predict (or really, interpolate) calculations on the potential energy surface, by first building up a regression representation from a "training set" of atomic images. The Amp calculator works by first learning from any other calculator (usually quantum mechanical calculations) that can provide energy and forces as a function of atomic coordinates. Depending upon the model choice, the predictions from Amp can take place with arbitrary accuracy, approaching that of the original calculator.

Amp is designed to integrate closely with the Atomic Simulation Environment (ASE). As such, the interface is in pure python, although several compute-heavy parts of the underlying codes also have fortran versions to accelerate the calculations. The close integration with ASE means that any calculator that works with ASE - including EMT, GPAW, DACAPO, VASP, NWChem, and Gaussian - can easily be used as the parent method.

CHAPTER 2

# Theory

According to the Born-Oppenheimer approximation, the ground-state potential energy of an atomic configuration is dictated solely by the nuclear coordinates (under certain conditions, such as the absence of external fields and constant charge). The potential energy is in general a very complicated function of the nuclear coordinates; it in theory can be calculated by directly solving the Schrodinger equation. However, in practice, an exact analytical solution to the many-body Schrodinger equation is very difficult (if not impossible), and most electronic structure codes provide a point-by-point approximation to the ground-state potential energy for given nuclear configurations.

Given enough example calculations from any electronic structure calculator, the idea is then to approximate the potential energy with a regression model:

$$\mathbf{R} \xrightarrow{\text{regression}} E = E(\mathbf{R}),$$

where $\mathbf{R}$ is the position of atoms in the system.

## Atomic representation of potential energy

In order to have a potential function which is simultaneously applicable to systems of different sizes, the total potential energy of the system can to be broken up into atomic energy contributions:

$$E(\mathbf{R}) = \sum_{\text{atom}=1}^{N} E_{\text{atom}}(\mathbf{R}).$$

The above expansion can be justified by assembling the atomic configuration by bringing atoms close to each other one by one. Then the atomic energy contributions (instead of the energy of the whole system at once) can be approximated using a regression method:

$$\mathbf{R} \xrightarrow{\text{regression}} E_{\text{atom}} = E_{\text{atom}}\left(\mathbf{R}\right).$$

## Descriptor

A better interpolation can be achieved if an appropriate symmetry function $\mathbf{G}$ of atomic coordinates, approximating the functional dependence of local energetics, is used as the input of the regression operator:

$$\mathbf{R} \xrightarrow{\mathbf{G}} \mathbf{G}\left(\mathbf{R}\right) \xrightarrow{\text{regression}} E_{\text{atom}} = E_{\text{atom}}\left(\mathbf{G}\left(\mathbf{R}\right)\right).$$

## Gaussian

A Gaussian descriptor **G** as a function of pair-atom distances and three-atom angles has been suggested by Behler [1], and is implemented within Amp. Radial fingerprints of the Gaussian type capture the interaction of atom $i$ with all atoms $j$ as the sum of Gaussians with width $\eta$ and center $R_s$,

$$G_i^I = \sum_{\substack{\text{atoms j within } R_c \\ \text{distance of atom i} \\ j \neq i}} e^{-\eta(R_{ij}-R_s)^2/R_c^2} f_c\left(R_{ij}\right).$$

By specifying many values of $\eta$ and $R_s$ we can begin to build a feature vector for regression.

The next type is the angular fingerprint accounting for three-atom interactions. The Gaussian angular fingerprints are computed for all triplets of atoms $i$, $j$, and $k$ by summing over the cosine values of the angles $\theta_{ijk} = \cos^{-1}\left(\frac{\mathbf{R}_{ij}.\mathbf{R}_{ik}}{R_{ij}R_{ik}}\right)$, ($\mathbf{R}_{ij} = \mathbf{R}_i - \mathbf{R}_j$), centered at atom $i$, according to

$$G_i^{II} = 2^{1-\zeta} \sum_{\substack{\text{atoms j, k within } R_c \\ \text{distance of atom i} \\ j,\ k \neq i \\ (j \neq k)}} \left(1+\lambda\cos\theta_{ijk}\right)^{\zeta} e^{-\eta\left(R_{ij}^2+R_{ik}^2+R_{jk}^2\right)/R_c^2} f_c\left(R_{ij}\right) f_c\left(R_{ik}\right) f_c\left(R_{jk}\right),$$

with parameters $\lambda$, $\eta$, and $\zeta$, which again can be chosen to build more elements of a feature vector.

The cutoff function $f_c\left(R_{ij}\right)$ in the above equations defines the energetically relevant local environment with value one at $R_{ij} = 0$ and zero at $R_{ij} = R_c$, where $R_c$ is the cutoff radius. In order to have a continuous force-field, the cutoff function $f_c\left(R_{ij}\right)$ as well as its first derivative should be continuous in $R_{ij} \in [0, \infty)$. One possible expression for such a function as proposed by Behler [1] is

$$f_c\left(r\right) == \begin{cases} 0.5\left(1+\cos\left(\pi\frac{r}{R_c}\right)\right) & \text{for } r \leq R_c, \\ 0 & \text{for } r > R_c. \end{cases}$$

Another more general choice for the cutoff function is the following polynomial [5]:

$$f_c\left(r\right) = \begin{cases} 1+\gamma\cdot\left(r/R_c\right)^{\gamma+1} - \left(\gamma+1\right)\left(r/R_c\right)^{\gamma} & \text{if } r \leq R_c, \\ 0 & \text{if } r > R_c, \end{cases}$$

with a user-specified parameter $\gamma$ that determines the rate of decay of the cutoff function as it extends from $r = 0$ to $r = R_c$.

The figure below shows how components of the fingerprints $\mathbf{G}_i^I$ and $\mathbf{G}_i^{II}$ change with, respectively, distance $R_{ij}$ between the pair of atoms $i$ and $j$ and the valence angle $\theta_{ijk}$ between the triplet of atoms $i$, $j$, and $k$ with central atom $i$:

## Zernike

A three-dimensional Zernike descriptor is also available inside Amp, and can be used as the atomic environment descriptor. The Zernike-type descriptor has been previously used in the machine-learning community extensively, but it has been suggested here for the first time for representing the local chemical environment. Zernike moments are basically a tensor product between spherical harmonics (complete and orthogonal on the surface of the unit sphere), and Zernike polynomials (complete and orthogonal within the unit sphere). Zernike descriptor components for each integer degree are then defined as the norm of Zernike moments with the same corresponding degree. For more details on the Zernike descriptor the reader is referred to the nice paper of Novotni and Klein [2].

Inspired by Bartok et. al. [3], to represent the local chemical environment of atom $i$, an atomic density function $\rho_i(\mathbf{r})$ is defined for each atomic local environment as the sum of delta distributions shifted to atomic positions:

$$\rho_i(\mathbf{r}) = \sum_{\substack{\text{atoms j within } R_c \\ \text{distance of atom i} \\ j \neq i}} \eta_j \delta\left(\mathbf{r} - \mathbf{R}_{ij}\right) f_c\left(\|\mathbf{R}_{ij}\|\right),$$

Next, components of the Zernike descriptor are computed from Zernike moments of the above atomic density destribution for each atom $i$.

The figure below shows how components of the Zernike descriptor vary with pair-atom distance, three-atom angle, and four-atom dehidral angle. It is important to note that components of the Gaussian descriptor discussed above are non-sensitive to the four-atom dehidral angle of the following figure.

### Bispectrum

Bispectrum of four-dimensional spherical harmonics have been suggested by Bartok et al. [3] to be invariant under rotation of the local atomic environment. In this approach, the atomic density distribution defined above is first mapped onto the surface of unit sphere in four dimensions. Consequently, Bartok et al. have shown that the bispectrum of this mapping can be used as atomic environment descriptor. We refer the reader to the original paper [3] for mathematical details. This approach of describing local environment is also available inside Amp.

## Regression Model

The general purpose of the regression model $x \xrightarrow{\text{regression}} y$ with input $x$ and output $y$ is to approximate the function $y = f(x)$ by using sample training data points $(x_i, y_i)$. The intent is to later use the approximated $f$ for input data $x_j$ (other than $x_i$ in the training data set), and make predictions for $y_j$. Typical regression models include Gaussian processes, support vector regression, and neural network.

### Neural network model

A neural network model is basically a very simple model of how the nervous system processes information. The first mathematical model was developed in 1943 by McCulloch and Pitts [4] for classification purposes; biological neurons either send or do not send a signal to the neighboring neuron. The model was soon extended to do linear and nonlinear regression, by replacing the binary activation function with a continuous function. The basic functional unit of a neural network is called “node”. A number of parallel nodes constitute a layer. A feed-forward neural network consists of at least an input layer plus an output layer. When approximating the PES, the output layer has just one neuron representing the potential energy. For a more robust interpolation, a number of “hidden layers” may exist in the neural network as well; the word “hidden” refers to the fact that these layers have no physical meaning. A schematic of a typical feed-forward neural network is shown below. In each node a number of inputs is multiplied by the corresponding weights and summed up with a constant bias. An activation function then acts upon the summation and an output is generated. The output is finally sent to the neighboring neuron in the next layer. Typically used activation functions are hyperbolic tangent, sigmoid, Gaussian, and linear functions. The unbounded linear activation function is particularly useful in the last hidden layer to scale neural network outputs to the range of reference values. For our purpose, the output of neural network represents energy of atomic system.

**References:**

1. “Atom-centered symmetry functions for constructing high-dimensional neural network potentials”, J. Behler, J. Chem. Phys. 134(7), 074106 (2011)
2. “Shape retrieval using 3D Zernike descriptors”, M. Novotni and R. Klein, Computer-Aided Design 36(11), 1047–1062 (2004)
3. “Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons”, A.P. Bart’ok, M.C. Payne, R. Kondor and G. Csanyi, Physical Review Letters 104, 136403 (2010)
4. “A logical calculus of the ideas immanent in nervous activity”, W.S. McCulloch, and W.H. Pitts, Bull. Math. Biophys. 5, 115–133 (1943)
5. “Amp: A modular approach to machine learning in atomistic simulations”, A. Khorshidi, and A.A. Peterson, Comput. Phys. Commun. 207, 310–324 (2016)

CHAPTER 3

# Credits

## People

This project is developed primarily by **Andrew Peterson** and **Alireza Khorshidi** in the Brown University School of Engineering. Specific credits:

- Andrew Peterson: lead, PI, many modules
- Alireza Khorshidi: many modules, Zernike descriptor
- Zack Ulissi: tensorflow version of neural network
- Muammar El Khatib: general contributions

We are also indebted to Nongnuch Artrith (MIT) and Pedro Felzenszwalb (Brown) for inspiration and technical discussion.

## Citations

We would appreciate if you cite the below publication for any use of Amp or its methods:

> Khorshidi & Peterson, "Amp: A modular approach to machine learning in atomistic simulations", *Computer Physics Communications* 207:310-324, 2016.

If you use Amp for saddle-point searches or nudged elastic bands, please also cite:

> Peterson, "Acceleration of saddle-point searches with machine learning", *Journal of Chemical Physics*, 145:074106, 2016.

CHAPTER 4

# Release notes / Revision history

## 0.5

Release date: February 24, 2017

The code has been significantly restructured since the previous version, in order to increase the modularity; much of the code structure has been changed since v0.4. Specific changes below:

- A parallelization scheme allowing for fast message passing with ZeroMQ.
- A simpler database format based on files, which optionally can be compressed to save diskspace.
- Incorporation of an experimental neural network model based on google's TensorFlow package. Requires TensorFlow version 0.11.0.
- Incorporation of an experimental bootstrap module for uncertainty analysis.

## 0.4

Release date: February 29, 2016

Corresponds to the publication of Khorshidi, A; Peterson*, AA. Amp: a modular approach to machine learning in atomistic simulations. Computer Physics Communications 207:310-324, 2016. http://dx.doi.org/10.1016/j.cpc.2016.05.010

Permanently available at https://doi.org/10.5281/zenodo.46737

## 0.3

Release date: July 13, 2015

First release under the new name "Amp" (Atomistic Machine-Learning Package/Potentials).

Permanently available at https://doi.org/10.5281/zenodo.20636

## 0.2

Release date: July 13, 2015

Last version under the name “Neural: Machine-learning for Atomistics”. Future versions are named “Amp”.

Available as the v0.2 tag in https://bitbucket.org/andrewpeterson/neural/commits/tag/v0.2

## 0.1

Release date: November 12, 2014

(Package name: Neural: Machine-Learning for Atomistics)

Permanently available at https://doi.org/10.5281/zenodo.12665.

First public bitbucket release: September, 2014.

CHAPTER 5

# Installation

AMP is python-based and is designed to integrate closely with the Atomic Simulation Environment (ASE). In its most basic form, it has few requirements:

- Python, version 2.7 is recommended.
- ASE.
- NumPy.
- SciPy.

To get more features, such as parallelization in training, a few more packages are recommended:

- Pexpect (or pxssh)
- ZMQ (or PyZMQ, the python version ofØMQ).

Certain advanced modules may contain dependencies that will be noted when they are used; for example Tensorflow for the tflow module or matplotlib for the plotting modules.

Basic installation instructions follow.

## Install ASE

We always test against the latest version (svn checkout) of ASE, but slightly older versions (>=3.9) are likely to work as well. Follow the instructions at the ASE website. ASE itself depends upon python with the standard numeric and scientific packages. Verify that you have working versions of NumPy and SciPy. We also recommend matplotlib in order to generate plots.

## Check out the code

As a relatively new project, it may be preferable to use the development version rather than "stable" releases, as improvements are constantly being made and features added. We run daily unit tests to make sure that our development code works as intended. We recommend checking out the latest version of the code via the project's bitbucket page. If you use git, check out the code with:

```
$ cd ~/path/to/my/codes
$ git clone git@bitbucket.org:andrewpeterson/amp.git
```

where you should replace '~/path/to/my/codes' with wherever you would like the code to be located on your computer. If you do not use git, just download the code as a zip file from the project's download page, and extract it

into '~/path/to/my/codes'. Please make sure that the folder '~/path/to/my/codes/amp' includes subdirectories 'amp', 'docs', 'tests', and 'tools'.

## Set the environment

You need to let your python version know about the existence of the amp module. Add the following line to your '.bashrc' (or other appropriate spot), with the appropriate path substituted for '~/path/to/my/codes':

```
$ export PYTHONPATH=~/path/to/my/codes/amp:$PYTHONPATH
```

You can check that this works by starting python and typing the below command, verifying that the location listed from the second command is where you expect:

```
>>> import amp
>>> print(amp.__file__)
```

See also the section on parallel processing for any issues that arise in making the environment work with Amp in parallel.

## Recommended step: Build fortran modules

Amp works in pure python, however, it will be annoyingly slow unless the associated Fortran 90 modules are compiled to speed up several parts of the code. The compilation of the Fortran 90 code and integration with the python parts is accomplished with f2py, which is part of NumPy. A Fortran 90 compiler will also be necessary on the system; a reasonable open-source option is GNU Fortran, or gfortran. This compiler will generate Fortran modules (.mod). gfortran will also be used by f2py to generate extension module fmodules.so on Linux or fmodules.pyd on Windows. In order to prepare the extension module the following steps need to be taken:

1. Compile model Fortran subroutines inside the model and descriptor folders by:

```
$ cd <installation-directory>/amp/model

$ gfortran -c neuralnetwork.f90

$ cd ../descriptor

$ gfortran -c cutoffs.f90
```

2. Move the modules "neuralnetwork.mod" and "cutoffs.mod" created in the last step, to the parent directory by:

```
$ cd ..

$ mv model/neuralnetwork.mod .

$ mv descriptor/cutoffs.mod .
```

3. Compile the model Fortran subroutines in companion with the descriptor and neuralnetwork subroutines by something like:

```
$ f2py -c -m fmodules model.f90 descriptor/cutoffs.f90 descriptor/gaussian.f90 descriptor/zernik
```

or on a Windows machine by:

```
$ f2py -c -m fmodules model.f90 descriptor/cutoffs.f90 descriptor/gaussian.f90 descriptor/zernike.f90
```

Note that if you update your code (e.g., with 'git pull origin master') and the fortran code changes but your version of fmodules.f90 is not updated, an exception will be raised telling you to re-compile your fortran modules.

## Recommended step: Run the tests

We include tests in the package to ensure that it still runs as intended as we continue our development; we run these tests on the latest build every night to try to keep bugs out. It is a good idea to run these tests after you install the package to see if your installation is working. The tests are in the folder *tests*; they are designed to run with nose. If you have nose installed, run the commands below:

```
$ mkdir /tmp/amptests
$ cd /tmp/amptests
$ nosetests ~/path/to/my/codes/amp/tests
```

CHAPTER 6

# Using Amp

If you are familiar with ASE, the use of Amp should be intuitive. At its most basic, Amp behaves like any other ASE calculator, except that it has a key extra method, called *train*, which allows you to fit the calculator to a set of atomic images. This means you can use Amp as a substitute for an expensive calculator in any atomistic routine, such as molecular dynamics, global optimization, transition-state searches, normal-mode analyses, phonon analyses, etc.

## Basic use

To use Amp, you need to specify a *descriptor* and a *model*. The below shows a basic example of training Amp with Gaussian descriptors and a neural network model—the Behler-Parinello scheme.

```
from amp import Amp
from amp.descriptor.gaussian import Gaussian
from amp.model.neuralnetwork import NeuralNetwork

calc = Amp(descriptor=Gaussian(), model=NeuralNetwork(),
           label='calc')
calc.train(images='my-images.traj')
```

After training is successful you can use your trained calculator just like any other ASE calculator (although you should be careful that you can only trust it within the trained regime). This will also result in the saving the calculator parameters to "<label>.amp", which can be used to re-load the calculator in a future session:

```
calc = Amp.load('calc.amp')
```

The modular nature of Amp is meant such that you can easily mix-and-match different descriptor and model schemes. See the theory section for more details.

## Adjusting convergence parameters

To control how tightly the energy is converged, you can adjust the *LossFunction*. Just insert before the *calc.train* line some code like:

```
from amp.model import LossFunction

convergence = {'energy_rmse': 0.02, 'force_rmse': 0.04}
calc.model.lossfunction = LossFunction(convergence=convergence)
```

You can see the adjustable parameters and their default values in the dictionary *LossFunction.default_parameters*:

```
>>> LossFunction.default_parameters
{'convergence': {'energy_rmse': 0.001, 'force_rmse': 0.005, 'energy_maxresid': None, 'force_maxresid
```

To change how the code manages the regression process, you can use the *Regressor* class. For example, to switch from the scipy's fmin_bfgs optimizer (the default) to scipy's basin hopping optimizer, try inserting the following lines before initializing training:

```
from amp.regression import Regressor
from scipy.optimize import basinhopping

regressor = Regressor(optimizer=basinhopping)
calc.model.regressor = regressor
```

## Turning on/off force training

Most electronic structure codes also give forces (in addition to potential energy) for each image, and you can get a much more predictive fit if you include this information while training. This is the default behavior in Amp. However, this can create issues: training will be much slower, convergence will be more difficult, and if there are inconsistencies in the training data (say if the calculator reports 0K-extrapolated energies rather than force-consistent ones), you might not be able to train at all. In these and many other cases you might want to turn off force training. To do this in the standard neural network model, you can do it through the *force_coefficient* keyword to the *LossFunction*; for example:

```
from amp.model import LossFunction

convergence = {'energy_rmse': 0.001}
calc.model.lossfunction = LossFunction(convergence=convergence,
                                       force_coefficient=None)
```

## Parallel processing

Most tasks in Amp are "embarrassingly parallel" and thus you should see a performance boost by specifying more cores. Our standard parallel processing approach requires the modules ZMQ (to pass messages between processes) and pxssh (to establish SSH connections across nodes, and is only needed if parallelizing on more than one node).

The code will try to automatically guess the parallel configuration from the environment variables that your batching system produces, using the function *amp.utilities.assign_cores*. (We only use SLURM on our system, so we welcome patches to get this utility working on other systems!) If you want to override the automatic guess, use the *cores* keyword when initializing Amp. To specify serial operation, use *cores=1*; to specify (for example) 8 cores on only a single node, use *cores=8* or *cores={'localhost': 8}*. For parallel operation, cores should be a dictionary where the keys are the hostnames and the values are the number of processors (cores) available on that node; e.g.,

```
cores = {'node241': 16,
         'node242': 16}
```

(One of the keys in the dictionary could also be *localhost*, as in the single-node example. Using *localhost* just prevents it from establishing an extra SSH connection.)

For this to work on multiple nodes, you need to be able to freely SSH between nodes on your system. Typically, this means that once you are logged in to your cluster you have public/private keys in use to ssh between nodes. If you can run *ssh localhost* without it asking you for a password, this is likely to work for you.

This also assumes that your environment is identical each time you SSH into a node; that is, all the packages such as ASE, Amp, ZMQ, etc., are available in the same version. Generally, if you are setting your environment with a .bashrc

or .modules file this will just work. However, if you need to set your environment variables on the machine that is being ssh'd to, you can do so with the *envcommand* keyword, which you might set to

```
envcommand = 'export PYTHONPATH=/path/to/amp:$PYTHONPATH'
```

This envcommand can be passed as a keyword to the initialization of the Amp class. Ultimately, Amp stores these and passes them around in a configuration dictionary called *parallel*, so if you are calling descriptor or model functions directly you may need to construct this dictionary, which has the form *parallel={'cores': ..., 'envcommand': ...}*.

## Advanced use

Under the hood, the train function is pretty simple; it just runs:

```
images = hash_images(images, ...)
self.descriptor.calculate_fingerprints(images, ...)
result = self.model.fit(images, self.descriptor, ...)
if result is True:
    self.save(filename)
```

- In the first line, the images are read and converted to a dictionary, addressed by a hash. This makes addressing the images simpler across modules and eliminates duplicate images. This also facilitates keeping a database of fingerprints, such that in future scripts you do not need to re-fingerprint images you have already encountered.
- In the second line, the descriptor converts the images into fingerprints, one fingerprint per image. There are two possible modes a descriptor can operate in: "image-centered" in which one vector is produced per image, and "atom-centered" in which one vector is produced per atom. That is, in atom-centered mode the image's fingerprint will be a list of lists. The resulting fingerprint is stored in *self.descriptor.fingerprints*, and the mode is stored in *self.parameters.mode*.
- In the third line, the model (e.g., a neural network) is fit to the data. As it is passed a reference to *self.descriptor*, it has access to the fingerprints as well as the mode. Many options are available to customize this in terms of the loss function, the regression method, etc.
- In the final pair of lines, if the target fit was achieved, the model is saved to disk.

## Re-training

If training is successful, Amp saves the parameters into an '<label>.amp' file (by default the label is 'amp', so this file is 'amp.amp'). You can load the pretrained calculator and re-train it further with tighter convergence criteria. You can specify if the pre-trained amp.amp will be overwritten by the re-trained one through the key word 'overwrite' (default is False).

```
calc = Amp.load('amp.amp')
calc.model.lossfunction = LossFunction(convergence=convergence)
calc.train(overwrite=True)
```

If training does not succeed, Amp raises a *TrainingConvergenceError*. You can use this within your scripts to catch when training succeeds or fails, for example:

```
from amp.utilities import TrainingConvergenceError

...

try:
    calc.train(images)
```

```
except TrainingConvergenceError:
    # Whatever you want to happen if training fails;
    # e.g., refresh parameters and train again.
```

## Global search in the parameter space

If the model is trained with minimizing a loss function which has a non-convex form, it might be desirable to perform a global search in the parameter space in prior to a gradient-descent optimization algorithm. That is, in the first step we do a random search in an area of parameter space including multiple basins (each basin has a local minimum). Next we take the parameters corresponding to the minimum loss function found, and start a gradient-descent optimization to find the local minimum of the basin found in the first step. Currently there exists a built-in global-search optimizer inside Amp which uses simulated-annealing algorithm. The module is based on the open-source simulated-annealing code of Wagner and Perry [1], but has been brought into the context of Amp. To use this module, the calculator object should be initiated as usual:

```
from amp import Amp
calc = Amp(descriptor=..., model=...)
images = ...
```

Then the calculator object and the images are passed to the *Annealer* module and the simulated-annealing search is performed by reducing the temperature from the initial maximum value *Tmax* to the final minimum value *Tmin* in number of steps *steps*:

```
from amp.utilities import Annealer
Annealer(calc=calc, images=images, Tmax=20, Tmin=1, steps=4000)
```

If *Tmax* takes a small value (greater than zero), then the algorithm reduces to the simple random-walk search. Finally the usual *train* module is called to continue from the best parameters found in the last step:

```
calc.train(images=images,)
```

**References:**

1. https://github.com/perrygeo/simanneal.

CHAPTER 7

# Example scripts

## A basic fitting script

The below script uses Gaussian descriptors with a neural network backend — the Behler-Parrinello approach — to train both energy and forces to a training set made by the script. Note this may take some time to run, which will depend upon the initial guess for the neural network parameters that is randomly generated. Try decreasing the *force_rmse* convergence parameter if you would like faster results.

```
"""Simple test of the Amp calculator, using Gaussian descriptors and neural
network model. Randomly generates data with the EMT potential in MD
simulations."""

import os
from ase import Atoms, Atom, units
import ase.io
from ase.calculators.emt import EMT
from ase.lattice.surface import fcc110
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase.md import VelocityVerlet
from ase.constraints import FixAtoms

from amp import Amp
from amp.descriptor.gaussian import Gaussian
from amp.model.neuralnetwork import NeuralNetwork
from amp.model import LossFunction


def generate_data(count, filename='training.traj'):
    """Generates test or training data with a simple MD simulation."""
    if os.path.exists(filename):
        return
    traj = ase.io.Trajectory(filename, 'w')
    atoms = fcc110('Pt', (2, 2, 2), vacuum=7.)
    atoms.extend(Atoms([Atom('Cu', atoms[7].position + (0., 0., 2.5)),
                        Atom('Cu', atoms[7].position + (0., 0., 5.))]))
    atoms.set_constraint(FixAtoms(indices=[0, 2]))
    atoms.set_calculator(EMT())
    atoms.get_potential_energy()
    traj.write(atoms)
    MaxwellBoltzmannDistribution(atoms, 300. * units.kB)
    dyn = VelocityVerlet(atoms, dt=1. * units.fs)
    for step in range(count - 1):
```

```
        dyn.run(50)
        traj.write(atoms)


generate_data(20)

calc = Amp(descriptor=Gaussian(),
           model=NeuralNetwork(hiddenlayers=(10, 10, 10)))
calc.model.lossfunction = LossFunction(convergence={'energy_rmse': 0.02,
                                                    'force_rmse': 0.02})
calc.train(images='training.traj')
```

Note you can monitor the progress of the training by typing *amp-plotconvergence amp-log.txt*, which will create a file called *convergence.pdf*.

## Examining fingerprints

With the modular nature, it's straightforward to analyze how fingerprints change with changes in images. The below script makes an animated GIF that shows how a fingerprint about the O atom in water changes as one of the O-H bonds is stretched. Note that most of the lines of code below are either making the atoms or making the figure; very little effort is needed to produce the fingerprints themselves—this is done in three lines.

```
# Make a series of images.
import numpy as np
from ase.structure import molecule
from ase import Atoms
atoms = molecule('H2O')
atoms.rotate('y', -np.pi/2.)
atoms.set_pbc(False)
displacements = np.linspace(0.9, 8.0, 20)
vec = atoms[2].position - atoms[0].position
images = []
for displacement in displacements:
    atoms = Atoms(atoms)
    atoms[2].position = (atoms[0].position + vec * displacement)
    images.append(atoms)

# Fingerprint using Amp.
from amp.utilities import hash_images
from amp.descriptor.gaussian import Gaussian
images = hash_images(images, ordered=True)
descriptor = Gaussian()
descriptor.calculate_fingerprints(images)

# Plot the data.
from matplotlib import pyplot

def barplot(hash, name, title):
    """Makes a barplot of the fingerprint about the O atom."""
    fp = descriptor.fingerprints[hash][0]
    fig, ax = pyplot.subplots()
    ax.bar(range(len(fp[1])), fp[1])
    ax.set_title(title)
    ax.set_ylim(0., 2.)
    ax.set_xlabel('fingerprint')
    ax.set_ylabel('value')
```

```
    fig.savefig(name)

for index, hash in enumerate(images.keys()):
    barplot(hash, 'bplot-%02i.png' % index,
            '%.2f$\\times$ equilibrium O-H bondlength'
            % displacements[index])

# For fun, make an animated gif.
import os
filenames = ['bplot-%02i.png' % index for index in range(len(images))]
command = ('convert -delay 100 %s -loop 0 animation.gif' %
           ' '.join(filenames))
os.system(command)
```

CHAPTER 8

# Analysis

## Convergence plots

You can use the tool called *amp-plotconvergence* to help you examine the output of an Amp log file. Run *amp-plotconvergence -h* for help at the command line.

You can also access this tool as *plot_convergence* from the *amp.analysis* module.

## Other plots

There are several other plotting tools within this module, including *plot_parity* for making parity plots, *plot_error* for making error plots, and *plot_sensitivity* for examining the sensitivity of the model output to the model parameters. These modules should produce plots like below; in the order parity, error, and sensitivity from left to right. See the module autodocumentation for details.

CHAPTER 9

# Neural network model

## Observer

When training the neural network, if you would like to monitor the progress you can do so with an observer. To do so, attach it before you train with something like

```
def observer(model, vector, loss):
    """This function can extract data during optimization.
    Full access is provided to the model, the vector of parameters, and
    the current value of the loss function.
    """
    pass

calc.model.observer = myobserver
```

Note that the observer must take exactly the three arguments specified above.

Your function "observer" will be called at each call to the loss function. For example, you can use this to print out values of specific parameter functions.

CHAPTER 10

# Building modules

Amp is designed to be modular, so if you think you have a great descriptor scheme or machine-learning model, you can try it out. This page describes how to add your own modules; starting with the bare-bones requirements to make it work, and building up with how to construct it so it integrates with respect to parallelization, etc.

## Descriptor: minimal requirements

To build your own descriptor, it needs to have certain minimum requirements met, in order to play with *Amp*. The below code illustrates these minimum requirements:

```
from ase.calculators.calculator import Parameters

class MyDescriptor(object):

    def __init__(self, parameter1, parameter2):
        self.parameters = Parameters({'mode': 'atom-centered',})
        self.parameters.parameter1 = parameter1
        self.parameters.parameter2 = parameter2

    def tostring(self):
        return self.parameters.tostring()

    def calculate_fingerprints(self, images, cores, log):
        # Do the calculations...
        self.fingerprints = fingerprints  # A dictionary.
```

The specific requirements, illustrated above, are:

- Has a parameters attribute (of type *ase.calculators.calculator.Parameters*), which holds the minimum information needed to re-build your module. That is, if your descriptor has user-settable parameters such as a cutoff radius, etc., they should be stored in this dictionary. Additionally, it must have the keyword "mode"; which must be set to either "atom-centered" or "image-centered". (This keyword will be used by the model class.)
- Has a "tostring" method, which converts the minimum parameters into a dictionary that can be re-constructed using *eval*. If you used the ASE *Parameters* class above, this class is simple:

```
    def tostring():
        return self.parameters.tostring()
```

- Has a "calculate_fingerprints" method. The images argument is a dictionary of training images, with keys that are unique hashes of each image in the set produced with *amp.utilities.hash_images*. The log is a

*amp.utilities.Logger* instance, that the method can optionally use as *log('Message.')*. The cores keyword describes parallelization, and can safely be ignored if serial operation is desired. This method must save a sub-attribute *self.fingerprints* (which will be accessible in the main *Amp* instance as *calc.descriptor.fingerprints*) that contains a dictionary-like object of the fingerprints, indexed by the same keys that were in the images dictionary. Ideally, *descriptor.fingerprints* is an instance of *amp.utilities.Data*, but probably any mapping (dictionary-like) object will do.

A fingerprint is a vector. In **image-centered** mode, there is one fingerprint for each image. This will generally be just the Cartesian positions of all the atoms in the system, but transformations are possible. For example this could be accessed by the images key

```
>>> calc.descriptor.fingerprints[key]
>>> [3.223, 8.234, 0.0322, 8.33]
```

In **atom-centered** mode, there is a fingerprint for each atom in the image. Therefore, calling *calc.descriptor.fingerprints[key]* returns a list of fingerprints, in the same order as the atom ordering in the original ASE atoms object. So to access an individual atom's fingerprints one could do

```
>>> calc.descriptor.fingerprints[key][index]
>>> ('Cu', [8.832, 9.22, 7.118, 0.312])
```

**That is, the first item is the element of the atom, and the second is a 1-dimensional array which is that atom's fingerprint.** Thus, *calc.descriptor.fingerprints[hash]* gives a list of fingerprints, in the same order the atoms appear in the image they were fingerprinted from.

If you want to train your model to forces also (besides energies), your "calculate_fingerprints" method needs to calculate derivatives of the fingerprints with respect to coordinates as well. This is because forces (as the minus of coordinate-gradient of the potential energy) can be written, according to the chain rule of calculus, as the derivative of your model output (which represents energy here) with respect to model inputs (which is fingerprints) times the derivative of fingerprints with respect to spatial coordinates. These derivatives are calculated for each image for each possible pair of atoms (within the cutoff distance in the **atom-centered** mode). They can be calculated either analytically or simply numerically with finite-difference method. If a piece of code is written to calculate coordinate-derivatives of fingerprints, then the "calculate_fingerprints" method can save it as a sub-attribute *self.fingerprintprimes* (which will be accessible in the main *Amp* instance as *calc.descriptor.fingerprintprimes*) along with *self.fingerprints*. *self.fingerprintprimes* is a dictionary-like object, indexed by the same keys that were in the images dictionary. Ideally, *descriptor.fingerprintprimes* is an instance of *amp.utilities.Data*, but probably any mapping (dictionary-like) object will do.

Calling *calc.descriptor.fingerprintprimes[key]* returns the derivatives of fingerprints for the image key of interest. This is a dictionary where each key is a tuple representing the indices of the derivative, and each value is a list of finperprintprimes. (This list has the same length as the fingerprints.) For example, to retrieve derivatives of the fingerprints of atom indexed 2 (which is say Pt) with respect to $x$ coordinate of atom indexed 1 (which is say Cu), we should do

```
>>> calc.descriptor.fingerprintprimes[key][(1, 'Cu', 2, 'Pt', 0)]
>>> [-1.202, 0.130, 4.511, -0.721]
```

Or to retrieve derivatives of the fingerprints of atom indexed 1 with respect to $z$ coordinate of atom indexed 1, we do

```
>>> calc.descriptor.fingerprintprimes[key][(1, 'Cu', 1, 'Cu', 2)]
>>> [3.48, -1.343, -2.561, -8.412]
```

# Descriptor: standard practices

The below describes standard practices we use in building modules. It is not necessary to use these, but it should make your life easier to follow standard practices. And, if your code is ultimately destined to be part of an Amp release, you should plan to make it follow these practices unless there is a compelling reason not to.

We have an example of a minimal descriptor in *amp.descriptor.example*; it's probably easiest to copy this file and modify it to become your new descriptor. For a complete example of a working descriptor, see *amp.descriptor.gaussian*.

## The Data class

The key element we use to make our lives easier is the *Data* class. It should be noted that, in the development version, this is still a work in progress. The *Data* class acts like a dictionary in that items can be accessed by key, but also saves the data to disk (it is persistent), enables calculation of missing items, and can even parallelize these calculations across cores and nodes.

It is recommended to first construct a pure python version that fits with the *Data* scheme for 1 core, then expanding it to work with multiple cores via the following procedure. See the Gaussian descriptor for an example of implementation.

### Basic data addition

To make the descriptor work with the *Data* class, the *Data* class needs a keyword *calculator*. The simplest example of this is our *NeighborlistCalculator*, which is basically a wrapper around ASE's Neighborlist class:

```
class NeighborlistCalculator:
    """For integration with .utilities.Data
    For each image fed to calculate, a list of neighbors with offset
    distances is returned.
    """

    def __init__(self, cutoff):
        self.globals = Parameters({'cutoff': cutoff})
        self.keyed = Parameters()
        self.parallel_command = 'calculate_neighborlists'

    def calculate(self, image, key):
        cutoff = self.globals.cutoff
        n = NeighborList(cutoffs=[cutoff / 2.] * len(image),
                         self_interaction=False,
                         bothways=True,
                         skin=0.)
        n.update(image)
        return [n.get_neighbors(index) for index in range(len(image))]
```

Notice there are two categories of parameters saved in the init statement: *globals* and *keyed*. The first are parameters that apply to every image; here the cutoff radius is the same regardless of the image. The second category contains data that is specific to each image, in a dictionary format keyed by the image hash. In this example, there are no keyed parameters, but in the case of the fingerprint calculator, the dictionary of neighborlists is an example of a *keyed* parameter. The class must have a function called *calculate*, which when fed an image and its key, returns the desired value: in this case a neighborlist. Structuring your code as above is enough to make it play well with the *Data* container in serial mode. (Actually, you don't even need to worry about dividing the parameters into globals and keyed in serial mode.) Finally, there is a *parallel_command* attribute which can be any string which describes what this function does, which will be used later.

### Parallelization

The parallelization should work provided the scheme is embarassingly parallel; that is, each image's fingerprint is independent of all other images' fingerprints. We implement this in building the *amp.utilities.Data* dictionaries, using a scheme of establishing SSH sessions (with pxssh) for each worker and passing messages with ZMQ.

The *Data* class itself serves as the master, and the workers are instances of the specific module; that is, for the Gaussian scheme the workers are started with *python -m amp.descriptor.gaussian id hostname:port* where id is a unique identifier number assigned to each worker, and hostname:port is the socket at which the workers should open the connection to the mater (e.g., "node243:51247"). The master expects the worker to print two messages to the screen: "<amp-connect>" which confirms the connection is established, and "<stderr>"; the text that is between them alerts the master (and the user's log file) where the worker will write its standard error to. All messages after this are passed via ZMQ. I.e., the bottom of the module should contain something like:

```
if __name__ == "__main__":
    import sys
    import tempfile

    hostsocket = sys.argv[-1]
    proc_id = sys.argv[-2]

    print('<amp-connect>')
    sys.stderr = tempfile.NamedTemporaryFile(mode='w', delete=False,
                                             suffix='.stderr')
    print('stderr written to %s<stderr>' % sys.stderr.name)
```

After this, the worker communicates with the master in request (from the worker) / reply (from the master) mode, via ZMQ. (It's worth checking out the ZMQ Guide; (ZMQ Guide examples). Each request from the worker needs to take the form of a dictionary with three entries: "id", "subject", and (optionally) "data". These are easily created with the *amp.utilities.MessageDictionary* class. The first thing the worker needs to do is establish the connection to the master and ask its purpose:

```
import zmq
from ..utilities import MessageDictionary
msg = MessageDictionary(proc_id)

# Establish client session via zmq; find purpose.
context = zmq.Context()
socket = context.socket(zmq.REQ)
socket.connect('tcp://%s' % hostsocket)
socket.send_pyobj(msg('<purpose>'))
purpose = socket.recv_pyobj()
```

In the final line above, the master has sent a string with the *parallel_command* attribute mentioned above. You can have some if/elif statements to choose what to do next, but for the calculate_neighborlist example, the worker routine is as simple as requesting the variables, performing the calculations, and sending back the results, which happens in these few lines. This is all that is needed for parallelization (in pure python):

```
# Request variables.
socket.send_pyobj(msg('<request>', 'cutoff'))
cutoff = socket.recv_pyobj()
socket.send_pyobj(msg('<request>', 'images'))
images = socket.recv_pyobj()

# Perform the calculations.
calc = NeighborlistCalculator(cutoff=cutoff)
neighborlist = {}
while len(images) > 0:
```

```
    key, image = images.popitem()  # Reduce memory.
    neighborlist[key] = calc.calculate(image, key)

# Send the results.
socket.send_pyobj(msg('<result>', neighborlist))
socket.recv_string() # Needed to complete REQ/REP.
```

CHAPTER 11

# More on descriptors

## Fingerprint ranges

It is often useful to examine your fingerprints more closely. There is a utility that can help with that, an example of its use is below. This assumes you have open a calculator called "calc.amp" and you want to examine the fingerprint ranges for your training data.

```
from ase import io
from amp.descriptor.analysis import FingerprintPlot
from amp import Amp

calc = Amp.load('calc.amp')
images = io.read('training.traj', index=':')

fpplot = FingerprintPlot(calc)
fpplot(images)
```

This will create a plot that looks something like below, here showing the fingerprint ranges for the specified element.

You can also overlay a specific image's fingerprint on to the fingerprint plot by using the *overlay* keyword when calling fpplot.

CHAPTER 12

# More on models

## Visualizing neural network outputs

It can be useful to visualize the neural network model to see how it is behaving. For example, you may find nodes that are effectively shut off (e.g., always giving a constant value like 1) or that are acting as a binary switch (e.g., only returning 1 or -1). There is a tool to allow you to visualize the node outputs of a set of data.

```
from amp.model.neuralnetwork import NodePlot

nodeplot = NodePlot(calc)
nodeplot.plot(images, filename='nodeplottest.pdf')
```

This will create a plot that looks something like below. Note that one such series of plots is made for each element. Here, Layer 0 is the input layer, from the fingerprints. Layer 1 and Layer 2 are the hidden layers. Layer 3 is the output layer; that is, the contribution of Pt to the potential energy (before it is multiplied by and added to a parameter to bring it to the correct magnitude).

## Calling an observer during training

It can be useful to call a function known as an "observer" during the training of the model. In the neural network implementation, this can be accomplished by attaching an observer directly to the model. The observer is executed at each call to *model.get_loss*, and is fed the arguments (self, vector, loss). An example of using the observer to print out one component of the parameter vector is shown below:

```
def observer(model, vector, loss):
    """Prints out the first component of the parameter vector."""
    print(vector[0])

calc.model.observer = observer
calc.train(images)
```

With this approach, all kinds of fancy tricks are possible, like calling *another* Amp model that reports the loss function on a test set of images. This could be useful to implement training with early stopping, for example.

CHAPTER 13

# TensorFlow

Google has released an open-source version of its machine-learning software named Tensorflow, which can allow for efficient backpropagation of neural networks and utilization of GPUs for extra speed.

We have incorporated an experimental module that uses a tensorflow back-end, which may provide an acceleration particularly through access to GPU systems. As of this writing, the tensorflow code is in flux (with version 1.0 anticipated shortly).

## Dependencies

This package requires google's TensorFlow 0.11.0. You can install it as shown below:

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.11.0-cp27-none-
pip install -U --upgrade $TF_BINARY_URL
```

If you want more information, please see tensorflow's website for instructions for installation on your system.

## Example

```
#!/usr/bin/env python
"""Simple test of the Amp calculator, using Gaussian descriptors and neural
network model. Randomly generates data with the EMT potential in MD
simulations."""

from ase.calculators.emt import EMT
from ase.lattice.surface import fcc110
from ase import Atoms, Atom
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase import units
from ase.md import VelocityVerlet
from ase.constraints import FixAtoms

from amp import Amp
from amp.descriptor.gaussian import Gaussian
from amp.model.tflow import NeuralNetwork


def generate_data(count):
    """Generates test or training data with a simple MD simulation."""
    atoms = fcc110('Pt', (2, 2, 2), vacuum=7.)
```

```
    adsorbate = Atoms([Atom('Cu', atoms[7].position + (0., 0., 2.5)),
                       Atom('Cu', atoms[7].position + (0., 0., 5.))])
    atoms.extend(adsorbate)
    atoms.set_constraint(FixAtoms(indices=[0, 2]))
    atoms.set_calculator(EMT())
    MaxwellBoltzmannDistribution(atoms, 300. * units.kB)
    dyn = VelocityVerlet(atoms, dt=1. * units.fs)
    newatoms = atoms.copy()
    newatoms.set_calculator(EMT())
    newatoms.get_potential_energy()
    images = [newatoms]
    for step in range(count - 1):
        dyn.run(50)
        newatoms = atoms.copy()
        newatoms.set_calculator(EMT())
        newatoms.get_potential_energy()
        images.append(newatoms)
    return images


def train_test():
    label = 'train_test/calc'
    train_images = generate_data(2)
    convergence = {
            'energy_rmse': 0.02,
            'force_rmse': 0.02
            }

    calc = Amp(descriptor=Gaussian(),
               model=NeuralNetwork(hiddenlayers=(3, 3), convergenceCriteria=convergence),
               label=label,
               cores=1)

    calc.train(images=train_images,)
    for image in train_images:
        print "energy =", calc.get_potential_energy(image)
        print "forces =", calc.get_forces(image)


if __name__ == '__main__':
    train_test()
```

## Known issues

- *tflow* module does not work for versions different from 0.11.0.

## About

This module was contributed by Zachary Ulissi (Department of Chemical Engineering, Stanford University, zulissi@gmail.com) with help, testing, and discussions from Andrew Doyle (Stanford) and the Amp development team.

CHAPTER 14

# Fingerprint databases

Often, a user will want to train multiple calculators to a common set of images. This may be just in routine development of a trained calculator (e.g., trying different neural network sizes), in using multiple training instances trying to find a good initial guess of parameters, or in making a committee of calculators. In this case, it can be a waste of computational time to calculate the fingerprints (and more expensively, the fingerprint derivatives) more than once.

To deal with this, Amp saves the fingerprints to a database, the location of which can be specified by the user. If you want multiple calculators to avoid re-fingerprinting the same images, just point them to the same database location.

## Format

The database format is custom for Amp, and is designed to be as simple as possible. Amp databases end in the extension *.ampdb*. In its simplest form, it is just a directory with one file per image; that is, you will see something like below:

```
label-fingerprints.ampdb/
    loose/
        f60b3324f6001d810afbab9f85a6ea5f
        aeaaa21e5faccc62bae94c5c48b04031
```

In the above, each file in the directory "loose" is the hash of an image, and contains that image's fingerprint. We use a file-based "database" to avoid conflicts with multiple processes accessing a database at the same time, which can cause conflicts.

However, for large training sets this can lead to lots of loose files, which can eat up a lot of memory, and with the large number of files slow down indexing jobs (like backups and scans). Therefore, you can compress the database with the *amp-compress* tool, described below.

## Compress

To save disk space, you may periodically want to run the utility *amp-compress* (contained in the *tools* directory of the amp package; this should be on your path for normal installations). In this case, you would run *amp-compress* <*filename*>, which would result in the above *.ampdb* file being changed to:

```
label-fingerprints.ampdb/
    archive.tar.gz
    loose/
```

That is, the two fingerprints that were in the "loose" directory are now in the file "archive.tar.gz".

You can also use the *–recursive* (or *-r*) flag to compress all ampdb files in or below the specified directory.

When Amp reads from the above database, it first looks in the "loose" directory for the fingerprint. If it is not there, it looks in "archive.tar.gz". If it is not there, it calculates the fingerprint and adds it to the "loose" directory.

## Future

We plan to make the amp-compress tool more automated. If the user does not supply a separate *dblabel* keyword, then we assume that their process is the only process using the database, and it is safe to compress the database at the end of their training job. This would automatically clean up the loose files at the end of the job.

CHAPTER 15

# Development

This page contains standard practices for developing Amp, focusing on repositories and documentation.

## Repositories and branching

The main Amp repository lives on bitbucket, andrewpeterson/amp . We employ a branching model where the *master* branch is the main development branch, containing day-to-day commits from the core developers and honoring merge requests from others. From time to time, we create a new branch that corresponds to a release. This release branch contains only the tagged release and any bug fixes.

## Contributing

You are welcome to contribute new features, bug fixes, better documentation, etc. to Amp. If you would like to contribute, please create a private fork and a branch for your new commits. When it is ready, send us a merge request. We follow the same basic model as ASE; please see the ASE documentation for complete instructions.

As good coding practice, make sure your code passes both the pyflakes and pep8 tests. (On linux, you should be able to run *pyflakes file.py* and *pep8 file.py*, and then correct it by *autopep8 –in-place file.py*.) If adding a new feature: consider adding a (very brief) test to the tests folder to ensure your new code continues to work, and also be sure to write clear documentation.

It is also a good idea to send us an email if you are planning something complicated.

## Documentation

This documentation is built with sphinx. (Mkdocs doesn't seem to support autodocumentation.) To build a local copy, cd into the docs directory and try a command such as

```
sphinx-build . /tmp/ampdocs
firefox /tmp/ampdocs/index.html &  # View the local copy.
```

This uses the style "bizstyle"; if you find this is missing on your system, you can likely install it with

```
pip install --user sphinxjp.themes.bizstyle
```

You should then be able to update the documentation rst files and see changes on your own machine. For line breaks, please use the style of containing each sentence on a new line.

## Releases

To create a release, we go through the following steps.

- Create a new branch on the bitbucket repository with the version name, as in *v0.5*. (Don't create a separate branch if this is a bugfix release, e.g., v0.5.1 — just add those to the v0.5 branch.) All subsequent work is in the new branch.
- Change *docs/conf.py*'s version information to match the new version number.
- Change the version that prints out in the Amp headers by changing the *_ampversion* variable in *amp/__init__.py*.
- Add the version to readthedocs' available versions.
- Change the nightly tests to test this version as the stable build.
- Tag the release with the release number, e.g., v0.5-release or v0.5.1-release, the latter being for bug fixes. Do this on a local machine (on the correct branch) with *git tag -a v0.5-release*, followed by *git push origin –tags*. Note we need to add the "-release" part to the tag to prevent a naming conflict with the branch name in git. (See Issue #43 on the project's bitbucket page for discussion.)
- Create a DOI for the release and a copy on Zenodo.

**Module autodocumentation**:

CHAPTER 16

# Main

This module is the main part of the Amp package.

## Module contents

**class** amp.**Amp**(*descriptor*, *model*, *label='amp'*, *dblabel=None*, *cores=None*, *envcommand=None*, *logging=True*, *atoms=None*)

Bases: ase.calculators.calculator.Calculator, object

Atomistic Machine-Learning Potential (Amp) ASE calculator

**Parameters**

- **descriptor** (*object*) – Class representing local atomic environment.
- **model** (*object*) – Class representing the regression model. Can be only NeuralNetwork for now. Input arguments for NeuralNetwork are hiddenlayers, activation, weights, and scalings; for more information see docstring for the class NeuralNetwork.
- **label** (*str*) – Default prefix/location used for all files.
- **dblabel** (*str*) – Optional separate prefix/location for database files, including fingerprints, fingerprint derivatives, and neighborlists. This file location can be shared between calculator instances to avoid re-calculating redundant information. If not supplied, just uses the value from label.
- **cores** (*int*) – Can specify cores to use for parallel training; if None, will determine from environment
- **envcommand** (*string*) – For parallel processing across nodes, a command can be supplied here to load the appropriate environment before starting workers.
- **logging** (*boolean*) – Option to turn off logging; e.g., to speed up force calls.
- **atoms** (*object*) – ASE atoms objects with positions, symbols, energy, and forces in ASE format.

**calculate**(*atoms*, *properties*, *system_changes*)

Calculation of the energy of system and forces of all atoms.

**cores**

Get or set the cores for the parallel environment.

**Parameters** **cores** (*int or dictionary*) – Parallel configuration. If cores is an integer, parallelizes over this many processes on machine localhost. cores can also be a dictionary of

the type {'node324': 16, 'node325': 16}. If not specified, tries to determine from environment, using amp.utilities.assign_cores.

**descriptor**
Get or set the atomic descriptor.

**Parameters** **descriptor** (*object*) – Class instance representing the local atomic environment.

**implemented_properties = ['energy', 'forces']**

**classmethod load** (*Cls*, *file*, *Descriptor=None*, *Model=None*, ***kwargs*)
Attempts to load calculators and return a new instance of Amp.

Only a filename or file-like object is required, in typical cases.

If using a home-rolled descriptor or model, also supply uninstantiated classes to those models, as in Model=MyModel. (Not as Model=MyModel()!)

Any additional keyword arguments (such as label or dblabel) can be fed through to Amp.

**Parameters**

- **file** (*str*) – Name of the file to load data from.
- **Descriptor** (*object*) – Class representing local atomic environment.
- **Model** (*object*) – Class representing the regression model.

**model**
Get or set the machine-learning model.

**Parameters** **model** (*object*) – Class instance representing the regression model.

**save** (*filename*, *overwrite=False*)
Saves the calculator in a way that it can be re-opened with load.

**Parameters**

- **filename** (*str*) – File object or path to the file to write to.
- **overwrite** (*bool*) – If an output file with the same name exists, overwrite it.

**set** (***kwargs*)
Function to set parameters.

For now, this doesn't do anything as all parameters are within the model and descriptor.

**set_label** (*label*)
Sets label, ensuring that any needed directories are made.

**Parameters** **label** (*str*) – Default prefix/location used for all files.

**train** (*images*, *overwrite=False*)
Fits the model to the training images.

**Parameters**

- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **overwrite** (*bool*) – If an output file with the same name exists, overwrite it.

amp.**get_git_commit** (*ampdirectory*)
Attempts to get the last git commit from the amp directory.

amp.**importhelper**(*importname*)
Manually compiled list of available modules.

This is to prevent the execution of arbitrary (potentially malicious) code.

However, since there is an *eval* statement in string2dict maybe this is silly.

CHAPTER 17

# Descriptor

The descriptor module contains methods for describing the local atomic environment; that is, feature fectors that can be fed to machine-learning modules.

## Gaussian

**class** amp.descriptor.gaussian.**FingerprintCalculator**(*neighborlist*, *Gs*, *cutoff*, *fortran*)

For integration with .utilities.Data

**Parameters**

- **neighborlist** (*list of str*) – List of neighbors.
- **Gs** (*dict*) – Dictionary of symbols and lists of dictionaries for making symmetry functions. Either auto-genetrated, or given in the following form, for example:

```
>>> Gs = {"O": [{"type":"G2", "element":"O", "eta":10.},
...             {"type":"G4", "elements":["O", "Au"],
...              "eta":5., "gamma":1., "zeta":1.0}],
...       "Au": [{"type":"G2", "element":"O", "eta":2.},
...              {"type":"G4", "elements":["O", "Au"],
...               "eta":2., "gamma":1., "zeta":5.0}]}
```

- **cutoff** (*float*) – Radius above which neighbor interactions are ignored.
- **fortran** (*bool*) – If True, will use fortran modules, if False, will not.

**calculate**(*image*, *key*)

Makes a list of fingerprints, one per atom, for the fed image.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – key of the image after being hashed.

**get_fingerprint**(*index*, *symbol*, *neighborsymbols*, *neighborpositions*)

Returns the fingerprint of symmetry function values for atom specified by its index and symbol.

neighborsymbols and neighborpositions are lists of neighbors' symbols and Cartesian positions, respectively.

**Parameters**

- **index** (*int*) – Index of the center atom.

- **symbol** (*str*) – Symbol of the center atom.
- **neighborsymbols** (*list of str*) – List of neighbors' symbols.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions.

**Returns** **symbol, fingerprint** – fingerprints for atom specified by its index and symbol.

**Return type** list of float

**class** amp.descriptor.gaussian.**FingerprintPrimeCalculator**(*neighborlist*, *Gs*, *cutoff*, *fortran*)

For integration with .utilities.Data

**Parameters**

- **neighborlist** (*list of str*) – List of neighbors.
- **Gs** (*dict*) – Dictionary of symbols and lists of dictionaries for making symmetry functions. Either auto-genetrated, or given in the following form, for example:

```
>>> Gs = {"O": [{"type":"G2", "element":"O", "eta":10.},
...             {"type":"G4", "elements":["O", "Au"],
...              "eta":5., "gamma":1., "zeta":1.0}],
...       "Au": [{"type":"G2", "element":"O", "eta":2.},
...              {"type":"G4", "elements":["O", "Au"],
...               "eta":2., "gamma":1., "zeta":5.0}]}
```

- **cutoff** (*float*) – Radius above which neighbor interactions are ignored.
- **fortran** (*bool*) – If True, will use fortran modules, if False, will not.

**calculate**(*image*, *key*)

Makes a list of fingerprint derivatives, one per atom, for the fed image.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – key of the image after being hashed.

**get_fingerprintprime**(*index*, *symbol*, *neighborindices*, *neighborsymbols*, *neighborpositions*, *m*, *l*)

Returns the value of the derivative of G for atom with index and symbol with respect to coordinate x_{l} of atom index m.

neighborindices, neighborsymbols and neighborpositions are lists of neighbors' indices, symbols and Cartesian positions, respectively.

**Parameters**

- **index** (*int*) – Index of the center atom.
- **symbol** (*str*) – Symbol of the center atom.
- **neighborindices** (*list of int*) – List of neighbors' indices.
- **neighborsymbols** (*list of str*) – List of neighbors' symbols.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions.
- **m** (*int*) – Index of the pair atom.
- **l** (*int*) – Direction of the derivative; is an integer from 0 to 2.

**Returns** **fingerprintprime** – The value of the derivative of the fingerprints for atom with index and symbol with respect to coordinate x_{l} of atom index m.

**Return type** list of float

**class** amp.descriptor.gaussian.**Gaussian**(*cutoff=<Cosine cutoff with Rc=6.500 from amp.descriptor.cutoffs>*, *Gs=None*, *dblabel=None*, *elements=None*, *version=None*, *fortran=True*, *mode='atom-centered'*)

Bases: object

Class that calculates Gaussian fingerprints (i.e., Behler-style).

**Parameters**

- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.
- **Gs** (*dict*) – Dictionary of symbols and lists of dictionaries for making symmetry functions. Either auto-genetrated, or given in the following form, for example:

```
>>> Gs = {"O": [{"type":"G2", "element":"O", "eta":10.},
...             {"type":"G4", "elements":["O", "Au"],
...              "eta":5., "gamma":1., "zeta":1.0}],
...        "Au": [{"type":"G2", "element":"O", "eta":2.},
...               {"type":"G4", "elements":["O", "Au"],
...                "eta":2., "gamma":1., "zeta":5.0}]}
```

- **dblabel** (*str*) – Optional separate prefix/location for database files, including fingerprints, fingerprint derivatives, and neighborlists. This file location can be shared between calculator instances to avoid re-calculating redundant information. If not supplied, just uses the value from label.
- **elements** (*list*) – List of allowed elements present in the system. If not provided, will be found automatically.
- **version** (*str*) – Version of fingerprints.
- **fortran** (*bool*) – If True, will use fortran modules, if False, will not.
- **mode** (*str*) – Can be either 'atom-centered' or 'image-centered'.

**Raises** RuntimeError

**calculate_fingerprints**(*images*, *parallel=None*, *log=None*, *calculate_derivatives=False*)

Calculates the fingerpints of the images, for the ones not already done.

**Parameters**

- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **parallel** (*dict*) – Configuration for parallelization. Should be in same form as in amp.Amp.
- **log** (*Logger object*) – Write function at which to log data. Note this must be a callable function.

- **calculate_derivatives** (*bool*) – Decides whether or not fingerprintprimes should also be calculated.

**tostring**()
Returns an evaluatable representation of the calculator that can be used to restart the calculator.

amp.descriptor.gaussian.**Kronecker**(*i*, *j*)
Kronecker delta function.

**Parameters**

- **i** (*int*) – First index of Kronecker delta.
- **j** (*int*) – Second index of Kronecker delta.

**Returns** The value of the Kronecker delta.

**Return type** int

**class** amp.descriptor.gaussian.**NeighborlistCalculator**(*cutoff*)
For integration with .utilities.Data

For each image fed to calculate, a list of neighbors with offset distances is returned.

**Parameters** **cutoff** (*float*) – Radius above which neighbor interactions are ignored.

**calculate**(*image*, *key*)
For integration with .utilities.Data

For each image fed to calculate, a list of neighbors with offset distances is returned.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – key of the image after being hashed.

amp.descriptor.gaussian.**calculate_G2**(*neighborsymbols*, *neighborpositions*, *G_element*, *eta*, *cutoff*, *Ri*, *fortran*)
Calculate G2 symmetry function.

Ideally this will not be used but will be a template for how to build the fortran version (and serves as a slow backup if the fortran one goes uncompiled). See Eq. 13a of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **neighborsymbols** (*list of str*) – List of symbols of all neighbor atoms.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions.
- **G_element** (*dict*) – Symmetry functions of the center atom.
- **eta** (*float*) – Parameter of Gaussian symmetry functions.
- **cutoff** (*float*) – Radius above which neighbor interactions are ignored.
- **Ri** (*int*) – Index of the center atom.
- **fortran** (*bool*) – If True, will use the fortran subroutines, else will not.

**Returns** **ridge** – G2 fingerprint.

**Return type** float

amp.descriptor.gaussian.**calculate_G2_prime**(*neighborindices*, *neighborsymbols*, *neighborpositions*, *G_element*, *eta*, *cutoff*, *i*, *Ri*, *m*, *l*, *fortran*)

Calculates coordinate derivative of G2 symmetry function for atom at index i and position Ri with respect to coordinate x_{1} of atom index m.

See Eq. 13b of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **neighborindices** (*list of int*) – List of int of neighboring atoms.
- **neighborsymbols** (*list of str*) – List of symbols of neighboring atoms.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions of neighboring atoms.
- **G_element** (*dict*) – Symmetry functions of the center atom.
- **eta** (*float*) – Parameter of Behler symmetry functions.
- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.
- **i** (*int*) – Index of the center atom.
- **Ri** (*float*) – Position of the center atom.
- **m** (*int*) – Index of the atom force is acting on.
- **l** (*int*) – Direction of force.
- **fortran** (*bool*) – If True, will use the fortran subroutines, else will not.

**Returns** **ridge** – Coordinate derivative of G2 symmetry function for atom at index a and position Ri with respect to coordinate x_{1} of atom index m.

**Return type** float

amp.descriptor.gaussian.**calculate_G4**(*neighborsymbols*, *neighborpositions*, *G_elements*, *gamma*, *zeta*, *eta*, *cutoff*, *Ri*, *fortran*)

Calculate G4 symmetry function.

Ideally this will not be used but will be a template for how to build the fortran version (and serves as a slow backup if the fortran one goes uncompiled). See Eq. 13c of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **neighborsymbols** (*list of str*) – List of symbols of neighboring atoms.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions of neighboring atoms.
- **G_elements** (*dict*) – Symmetry functions of the center atom.
- **gamma** (*float*) – Parameter of Gaussian symmetry functions.
- **zeta** (*float*) – Parameter of Gaussian symmetry functions.
- **eta** (*float*) – Parameter of Gaussian symmetry functions.
- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are

ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.

- **Ri** (*int*) – Index of the center atom.
- **fortran** (*bool*) – If True, will use the fortran subroutines, else will not.

**Returns** **ridge** – G4 fingerprint.

**Return type** float

amp.descriptor.gaussian.**calculate_G4_prime**(*neighborindices*, *neighborsymbols*, *neighborpositions*, *G_elements*, *gamma*, *zeta*, *eta*, *cutoff*, *i*, *Ri*, *m*, *l*, *fortran*)

Calculates coordinate derivative of G4 symmetry function for atom at index i and position Ri with respect to coordinate x_{1} of atom index m.

See Eq. 13d of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **neighborindices** (*list of int*) – List of int of neighboring atoms.
- **neighborsymbols** (*list of str*) – List of symbols of neighboring atoms.
- **neighborpositions** (*list of list of float*) – List of Cartesian atomic positions of neighboring atoms.
- **G_elements** (*dict*) – Symmetry functions of the center atom.
- **gamma** (*float*) – Parameter of Behler symmetry functions.
- **zeta** (*float*) – Parameter of Behler symmetry functions.
- **eta** (*float*) – Parameter of Behler symmetry functions.
- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.
- **i** (*int*) – Index of the center atom.
- **Ri** (*float*) – Position of the center atom.
- **m** (*int*) – Index of the atom force is acting on.
- **l** (*int*) – Direction of force.
- **fortran** (*bool*) – If True, will use the fortran subroutines, else will not.

**Returns** **ridge** – Coordinate derivative of G4 symmetry function for atom at index i and position Ri with respect to coordinate x_{1} of atom index m.

**Return type** float

amp.descriptor.gaussian.**dCos_theta_ijk_dR_ml**(*i*, *j*, *k*, *Ri*, *Rj*, *Rk*, *m*, *l*)

Returns the derivative of Cos(theta_{ijk}) with respect to x_{1} of atomic index m.

See Eq. 14f of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **i** (*int*) – Index of the center atom.
- **j** (*int*) – Index of the first atom.
- **k** (*int*) – Index of the second atom.

- **Ri** (*float*) – Position of the center atom.
- **Rj** (*float*) – Position of the first atom.
- **Rk** (*float*) – Position of the second atom.
- **m** (*int*) – Index of the atom force is acting on.
- **l** (*int*) – Direction of force.

**Returns** **dCos_theta_ijk_dR_ml** – Derivative of Cos(theta_{ijk}) with respect to x_{l} of atomic index m.

**Return type** float

amp.descriptor.gaussian.**dRij_dRml**(*i*, *j*, *Ri*, *Rj*, *m*, *l*)
Returns the derivative of the norm of position vector R_{ij} with respect to coordinate x_{l} of atomic index m.

See Eq. 14c of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **i** (*int*) – Index of the first atom.
- **j** (*int*) – Index of the second atom.
- **Ri** (*float*) – Position of the first atom.
- **Rj** (*float*) – Position of the second atom.
- **m** (*int*) – Index of the atom force is acting on.
- **l** (*int*) – Direction of force.

**Returns** **dRij_dRml** – The derivative of the noRi of position vector R_{ij} with respect to x_{l} of atomic index m.

**Return type** list of float

amp.descriptor.gaussian.**dRij_dRml_vector**(*i*, *j*, *m*, *l*)
Returns the derivative of the position vector R_{ij} with respect to x_{l} of itomic index m.

See Eq. 14d of the supplementary information of Khorshidi, Peterson, CPC(2016).

**Parameters**

- **i** (*int*) – Index of the first atom.
- **j** (*int*) – Index of the second atom.
- **m** (*int*) – Index of the atom force is acting on.
- **l** (*int*) – Direction of force.

**Returns** The derivative of the position vector R_{ij} with respect to x_{l} of atomic index m.

**Return type** list of float

amp.descriptor.gaussian.**make_symmetry_functions**(*elements*)
Makes symmetry functions as in Nano Letters function by Artrith.

Elements is a list of the elements, as in: [”C”, “O”, “H”, “Cu”]. G[0] = {“type”:”G2”, “element”: “O”, “eta”: 0.0009} G[40] = {“type”:”G4”, “elements”: [”O”, “Au”], “eta”: 0.0001, “gamma”: 1.0, “zeta”: 1.0}

If G (a list) is fed in, this will add to it and return an expanded version. If not, it will create a new one.

**Parameters** **elements** (*list of str*) – List of symbols of all atoms.

**Returns** **G** – Symmetry functions if not given by the user.

**Return type** dict of lists

## Zernike

**class** amp.descriptor.zernike.**FingerprintCalculator**(*neighborlist*, *Gs*, *nmax*, *cutoff*, *fortran*)

For integration with .utilities.Data

**calculate**(*image*, *key*)

Makes a list of fingerprints, one per atom, for the fed image.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – Key of the image after being hashed.

**get_fingerprint**(*index*, *symbol*, *n_symbols*, *Rs*)

Returns the fingerprint of symmetry function values for atom specified by its index and symbol.

n_symbols and Rs are lists of neighbors' symbols and Cartesian positions, respectively.

**Parameters**

- **index** (*int*) – Index of the center atom.
- **symbol** (*str*) – Symbol of the center atom.
- **n_symbols** (*list of str*) – List of neighbors' symbols.
- **Rs** (*list of list of float*) – List of Cartesian atomic positions of neighbors.

**Returns** **symbols, fingerprints** – Fingerprints for atom specified by its index and symbol.

**Return type** list of float

**class** amp.descriptor.zernike.**FingerprintPrimeCalculator**(*neighborlist*, *Gs*, *nmax*, *cutoff*, *fortran*)

For integration with .utilities.Data

**calculate**(*image*, *key*)

Makes a list of fingerprint derivatives, one per atom, for the fed image.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – Key of the image after being hashed.

**get_fingerprintprime**(*index*, *symbol*, *n_indices*, *n_symbols*, *Rs*, *p*, *q*)

Returns the value of the derivative of G for atom with index and symbol with respect to coordinate x_{i} of atom index m. n_indices, n_symbols and Rs are lists of neighbors' indices, symbols and Cartesian positions, respectively.

**Parameters**

- **index** (*int*) – Index of the center atom.
- **symbol** (*str*) – Symbol of the center atom.
- **n_indices** (*list of int*) – List of neighbors' indices.
- **n_symbols** (*list of str*) – List of neighbors' symbols.
- **Rs** (*list of list of float*) – List of Cartesian atomic positions.

- **p** (*int*) – Index of the pair atom.
- **q** (*int*) – Direction of the derivative; is an integer from 0 to 2.

**Returns** **fingerprint_prime** – The value of the derivative of the fingerprints for atom with index and symbol with respect to coordinate x_{i} of atom index m.

**Return type** list of float

amp.descriptor.zernike.**Kronecker**(*i*, *j*)

Kronecker delta function.

**i** [int] First index of Kronecker delta.

**j** [int] Second index of Kronecker delta.

**Returns** **Kronecker delta**

**Return type** int

**class** amp.descriptor.zernike.**NeighborlistCalculator**(*cutoff*)

For integration with .utilities.Data

For each image fed to calculate, a list of neighbors with offset distances is returned.

**Parameters** **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.

**calculate**(*image*, *key*)

For integration with .utilities.Data For each image fed to calculate, a list of neighbors with offset distances is returned.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – Key of the image after being hashed.

**class** amp.descriptor.zernike.**Zernike**(*cutoff=<Cosine cutoff with Rc=6.500 from amp.descriptor.cutoffs>*, *Gs=None*, *nmax=5*, *dblabel=None*, *elements=None*, *version='2016.02'*, *mode='atom-centered'*, *fortran=True*)

Bases: object

Class that calculates Zernike fingerprints.

**Parameters**

- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.
- **Gs** (*dict*) – Dictionary of symbols and dictionaries for making symmetry functions. Either auto-genetrated, or given in the following form, for example:

```
>>> Gs = {"Au": {"Au": 3., "O": 2.}, "O": {"Au": 5., "O": 10.}}
```

- **nmax** (*integer or dict*) – Maximum degree of Zernike polynomials that will be included in the fingerprint vector. Can be different values for different species fed as a dictionary with chemical elements as keys.

- **dblabel** (*str*) – Optional separate prefix/location for database files, including fingerprints, fingerprint derivatives, and neighborlists. This file location can be shared between calculator instances to avoid re-calculating redundant information. If not supplied, just uses the value from label.
- **elements** (*list*) – List of allowed elements present in the system. If not provided, will be found automatically.
- **version** (*str*) – Version of fingerprints.
- **mode** (*str*) – Can be either 'atom-centered' or 'image-centered'.
- **fortran** (*bool*) – If True, will use fortran modules, if False, will not.

**Raises** RuntimeError, TypeError

**calculate_fingerprints**(*images*, *parallel=None*, *log=None*, *calculate_derivatives=False*)
Calculates the fingerpints of the images, for the ones not already done.

**Parameters**

- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **parallel** (*dict*) – Configuration for parallelization. Should be in same form as in amp.Amp.
- **log** (*Logger object*) – Write function at which to log data. Note this must be a callable function.
- **calculate_derivatives** (*bool*) – Decides whether or not fingerprintprimes should also be calculated.

**tostring**()
Returns an evaluatable representation of the calculator that can be used to restart the calculator.

amp.descriptor.zernike.**binomial**(*n*, *k*, *factorial*)
Returns C(n,k) = n!/(k!(n-k)!).

amp.descriptor.zernike.**calculate_R**(*n*, *l*, *rho*, *factorial*)
Calculates R_{n}^{l}(rho) according to the last equation of wikipedia.

amp.descriptor.zernike.**calculate_Z**(*n*, *l*, *m*, *x*, *y*, *z*, *factorial*)
Calculates Z_{nl}^{m}(x, y, z) according to the unnumbered equation afer Eq. (11) of "3D Zernike Descriptors for Content Based Shape Retrieval", Computer-Aided Design 36 (2004) 1047-1062.

amp.descriptor.zernike.**calculate_Z_prime**(*n*, *l*, *m*, *x*, *y*, *z*, *p*, *factorial*)
Calculates dZ_{nl}^{m}(x, y, z)/dR_{p} according to the unnumbered equation afer Eq. (11) of "3D Zernike Descriptors for Content Based Shape Retrieval", Computer-Aided Design 36 (2004) 1047-1062.

amp.descriptor.zernike.**calculate_q**(*nu*, *k*, *l*, *factorial*)
Calculates q_{kl}^{nu} according to the unnumbered equation afer Eq. (7) of "3D Zernike Descriptors for Content Based Shape Retrieval", Computer-Aided Design 36 (2004) 1047-1062.

amp.descriptor.zernike.**der_position**(*m*, *n*, *Rm*, *Rn*, *l*, *i*)
Returns the derivative of the norm of position vector R_{mn} with respect to x_{i} of atomic index l.

**Parameters**

- **m** (*int*) – Index of the first atom.
- **n** (*int*) – Index of the second atom.

- **Rm** (*float*) – Position of the first atom.
- **Rn** (*float*) – Position of the second atom.
- **l** (*int*) – Index of the atom force is acting on.
- **i** (*int*) – Direction of force.

**Returns** **der_position** – The derivative of the norm of position vector R_{mn} with respect to x_{i} of atomic index l.

**Return type** list of float

amp.descriptor.zernike.**generate_coefficients**(*elements*)

Automatically generates coefficients if not given by the user.

**Parameters** **elements** (*list of str*) – List of symbols of all atoms.

**Returns** **G**

**Return type** dict of dicts

## Bispectrum

**class** amp.descriptor.bispectrum.**Bispectrum**(*cutoff=<Cosine cutoff with Rc=6.500 from amp.descriptor.cutoffs>, Gs=None, jmax=5, dblabel=None, elements=None, version='2016.02', mode='atom-centered'*)

Bases: object

Class that calculates spherical harmonic bispectrum fingerprints.

**Parameters**

- **cutoff** (*object or float*) – Cutoff function, typically from amp.descriptor.cutoffs. Can be also fed as a float representing the radius above which neighbor interactions are ignored; in this case a cosine cutoff function will be employed. Default is a 6.5-Angstrom cosine cutoff.
- **Gs** (*dict*) – Dictionary of symbols and dictionaries for making fingerprints. Either autogenetrated, or given in the following form, for example:

```
>>> Gs = {"Au": {"Au": 3., "O": 2.}, "O": {"Au": 5., "O": 10.}}
```

- **jmax** (*integer or half-integer or dict*) – Maximum degree of spherical harmonics that will be included in the fingerprint vector. Can be also fed as a dictionary with chemical species as keys.
- **dblabel** (*str*) – Optional separate prefix/location for database files, including fingerprints, fingerprint derivatives, and neighborlists. This file location can be shared between calculator instances to avoid re-calculating redundant information. If not supplied, just uses the value from label.
- **elements** (*list*) – List of allowed elements present in the system. If not provided, will be found automatically.
- **version** (*str*) – Version of fingerprints.
- **Raises** –
- **-------** – RuntimeError, TypeError

**calculate_fingerprints**(*images*, *parallel=None*, *log=None*, *calculate_derivatives=False*)
Calculates the fingerpints of the images, for the ones not already done.

**Parameters**

- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **parallel** (*dict*) – Configuration for parallelization. Should be in same form as in amp.Amp.
- **log** (*Logger object*) – Write function at which to log data. Note this must be a callable function.
- **calculate_derivatives** (*bool*) – Decides whether or not fingerprintprimes should also be calculated.

**tostring**()
Returns an evaluatable representation of the calculator that can be used to restart the calculator.

amp.descriptor.bispectrum.**CG**(*a*, *alpha*, *b*, *beta*, *c*, *gamma*, *factorial*)
Clebsch-Gordan coefficient C_{a alpha b beta}^{c gamma} is calculated acoording to the expression given in Varshalovich Eq. (3), Section 8.2, Page 238.

**class** amp.descriptor.bispectrum.**FingerprintCalculator**(*neighborlist*, *Gs*, *jmax*, *cutoff*)
For integration with .utilities.Data

**calculate**(*image*, *key*)
Makes a list of fingerprints, one per atom, for the fed image.

**Parameters**

- **image** (*object*) – ASE atoms object.
- **key** (*str*) – key of the image after being hashed.

**get_fingerprint**(*index*, *symbol*, *n_symbols*, *Rs*)
Returns the fingerprint of symmetry function values for atom specified by its index and symbol.

n_symbols and Rs are lists of neighbors' symbols and Cartesian positions, respectively.

**Parameters**

- **index** (*int*) – Index of the center atom.
- **symbol** (*str*) – Symbol of the center atom.
- **n_symbols** (*list of str*) – List of neighbors' symbols.
- **Rs** (*list of list of float*) – List of Cartesian atomic positions of neighbors.

**Returns** **symbols, fingerprints** – fingerprints for atom specified by its index and symbol.

**Return type** list of float

**class** amp.descriptor.bispectrum.**NeighborlistCalculator**(*cutoff*)
For integration with .utilities.Data

For each image fed to calculate, a list of neighbors with offset distances is returned.

**calculate**(*image*, *key*)

amp.descriptor.bispectrum.**U**(*j*, *m*, *mp*, *omega*, *theta*, *phi*, *factorial*)
Calculates rotation matrix U_{MM'}^{J} in terms of rotation angle omega as well as rotation axis angles theta and phi, according to Varshalovich, Eq. (3), Section 4.5, Page 81. j, m, mp, and mpp here are J, M, M', and M" in Eq. (3).

amp.descriptor.bispectrum.**WignerD**(*j*, *m*, *mp*, *alpha*, *beta*, *gamma*, *factorial*)
Returns the Wigner-D matrix. alpha, beta, and gamma are the Euler angles.

amp.descriptor.bispectrum.**binomial**(*n*, *k*, *factorial*)
Returns C(n,k) = n!/(k!(n-k)!).

amp.descriptor.bispectrum.**calculate_B**(*j1*, *j2*, *j*, *G_element*, *cutoff*, *cutofffn*, *factorial*, *n_symbols*, *rs*, *psis*, *thetas*, *phis*)
Calculates bi-spectrum B_{j1, j2, j} according to Eq. (5) of "Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons", Phys. Rev. Lett. 104, 136403.

amp.descriptor.bispectrum.**calculate_c**(*j*, *mp*, *m*, *G_element*, *cutoff*, *cutofffn*, *factorial*, *n_symbols*, *rs*, *psis*, *thetas*, *phis*)
Calculates c^{j}_{m'm} according to Eq. (4) of "Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons", Phys. Rev. Lett. 104, 136403

amp.descriptor.bispectrum.**generate_coefficients**(*elements*)
Automatically generates coefficients if not given by the user.

**Parameters** **elements** (*list of str*) – List of symbols of all atoms.

**Returns** G

**Return type** dict of dicts

amp.descriptor.bispectrum.**m_values**(*j*)
Returns a list of m values for a given j.

## Cutoff functions

This script contains different cutoff function forms.

Note all cutoff functions need to have a "todict" method to support saving/loading as an Amp object.

All cutoff functions also need to have an *Rc* attribute which is the maximum distance at which properties are calculated; this will be used in calculating neighborlists.

**class** amp.descriptor.cutoffs.**Cosine**(*Rc*)
Bases: object

Cosine functional form suggested by Behler.

**Parameters** **Rc** (*float*) – Radius above which neighbor interactions are ignored.

**__call__**(*Rij*)

**Parameters** **Rij** (*float*) – Distance between pair atoms.

**Returns** The vaule of the cutoff function.

**Return type** float

**prime**(*Rij*)
Derivative of the Cosine cutoff function.

**Parameters** **Rij** (*float*) – Distance between pair atoms.

**Returns** The vaule of derivative of the cutoff function.

**Return type** float

**todict**()

**class** amp.descriptor.cutoffs.**Polynomial**(*Rc*, *gamma=4*)
Bases: object

Polynomial functional form suggested by Khorshidi and Peterson.

**Parameters**

- **gamma** (*float*) – The power of polynomial.
- **Rc** (*float*) – Radius above which neighbor interactions are ignored.

**__call__**(*Rij*, *gamma*)

**Parameters** **Rij** (*float*) – Distance between pair atoms.

**Returns** **value** – The vaule of the cutoff function.

**Return type** float

**prime**(*Rij*, *gamma*)
Derivative of the Cosine cutoff function.

**Parameters**

- **Rc** (*float*) – Radius above which neighbor interactions are ignored.
- **Rij** (*float*) – Distance between pair atoms.

**Returns** The vaule of derivative of the cutoff function.

**Return type** float

**todict**()

amp.descriptor.cutoffs.**dict2cutoff**(*dct*)
This function converts a dictionary (which was created with the to_dict method of one of the cutoff classes) into an instantiated version of the class. Modeled after ASE's dict2constraint function.

CHAPTER 18

# Model

This module is designed to include machine-learning models for interpolating energies and forces from either an atom-centered or image-centered fingerprint description.

## Model

**class** amp.model.**LossFunction**(*energy_coefficient=1.0*, *force_coefficient=0.04*, *convergence=None*, *parallel=None*, *overfit=0.0*, *raise_ConvergenceOccurred=True*, *log_losses=True*, *d=None*)

Basic loss function, which can be used by the model.get_loss method which is required in standard model classes.

This version is pure python and thus will be slow compared to a fortran/parallel implementation.

If parallel is None, it will pull it from the model itself. Only use this keyword to override the model's specification.

Also has parallelization methods built in.

See self.default_parameters for the default values of parameters specified as None.

**Parameters**

- **energy_coefficient** (*float*) – Coefficient of the energy contribution in the loss function.
- **force_coefficient** (*float*) – Coefficient of the force contribution in the loss function. Can set to None as shortcut to turn off force training.
- **convergence** (*dict*) – Dictionary of keys and values defining convergence. Keys are 'energy_rmse', 'energy_maxresid', 'force_rmse', and 'force_maxresid'. If 'force_rmse' and 'force_maxresid' are both set to None, force training is turned off and force_coefficient is set to None.
- **parallel** (*dict*) – Parallel configuration dictionary. Will pull from model itself if not specified.
- **overfit** (*float*) – Multiplier of the weights norm penalty term in the loss function.
- **raise_ConvergenceOccurred** (*bool*) – If True will raise convergence notice.
- **log_losses** (*bool*) – If True will log the loss function value in the log file else will not.

- **d** (*None or float*) – If d is None, both loss function and its gradient are calculated analytically. If d is a float, then gradient of the loss function is calculated by perturbing each parameter plus/minus d.

**attach_model**(*model*, *fingerprints=None*, *fingerprintprimes=None*, *images=None*)
Attach the model to be used to the loss function.

fingerprints and training images need not be supplied if they are already attached to the model via model.trainingparameters.

**Parameters**

- **model** (*object*) – Class representing the regression model.
- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.
- **fingerprintprimes** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprint derivatives as values.
- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.

**calculate_loss**(*parametervector*, *lossprime*)
Method that calculates the loss, derivative of the loss with respect to parameters (if requested), and max_residual.

**Parameters**

- **parametervector** (*list*) – Parameters of the regression model in the form of a list.
- **lossprime** (*bool*) – If True, will calculate and return dloss_dparameters, else will only return zero for dloss_dparameters.

**check_convergence**(*loss*, *energy_loss*, *force_loss*, *energy_maxresid*, *force_maxresid*)
Check convergence

Checks to see whether convergence is met; if it is, raises ConvergenceException to stop the optimizer.

**Parameters**

- **loss** (*float*) – Value of the loss function.
- **energy_loss** (*float*) – Value of the energy contribution of the loss function.
- **force_loss** (*float*) – Value of the force contribution of the loss function.
- **energy_maxresid** (*float*) – Maximum energy residual.
- **force_maxresid** (*float*) – Maximum force residual.

**default_parameters = {'convergence': {'energy_rmse': 0.001, 'force_rmse': 0.005, 'energy_maxresid': None, 'force_**

**get_loss**(*parametervector*, *lossprime*)
Returns the current value of the loss function for a given set of parameters, or, if the energy is less than the energy_tol raises a ConvergenceException.

**Parameters**

- **parametervector** (*list*) – Parameters of the regression model in the form of a list.
- **lossprime** (*bool*) – If True, will calculate and return dloss_dparameters, else will only return zero for dloss_dparameters.

**process_parallels** (*vector*, *server*, *n_pids*, *keys*, *args*)

**Parameters**

- **vector** (*list*) – Parameters of the regression model in the form of a list.
- **server** (*object*) – Master session of parallel processing.
- **processes** (*list of objects*) – Worker sessions for parallel processing.
- **keys** (*list*) – List of images keys for worker processes.
- **args** (*dict*) – Dictionary containing arguments of the method to be called on each worker process.

**class** amp.model.**Model**

Bases: object

Class that includes common methods between different models.

**calculate_dEnergy_dParameters** (*fingerprints*)

Calculates a list of floats corresponding to the derivative of model-predicted energy of an image with respect to model parameters.

**Parameters** **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.

**calculate_dForces_dParameters** (*fingerprints*, *fingerprintprimes*)

Calculates an array of floats corresponding to the derivative of model-predicted atomic forces of an image with respect to model parameters.

**Parameters**

- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.
- **fingerprintprimes** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprint derivatives as values.

**calculate_energy** (*fingerprints*)

Calculates the model-predicted energy for an image, based on its fingerprint.

**Parameters** **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.

**calculate_forces** (*fingerprints*, *fingerprintprimes*)

Calculates the model-predicted forces for an image, based on derivatives of fingerprints.

**Parameters**

- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.
- **fingerprintprimes** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprint derivatives as values.

**calculate_numerical_dEnergy_dParameters** (*fingerprints*, *d=1e-05*)

Evaluates dEnergy_dParameters using finite difference.

This will trigger two calls to calculate_energy(), with each parameter perturbed plus/minus d.

**Parameters**

- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.

- **d** (*float*) – The amount of perturbation in each parameter.

**calculate_numerical_dForces_dParameters** (*fingerprints*, *fingerprintprimes*, *d=1e-05*)
Evaluates dForces_dParameters using finite difference. This will trigger two calls to calculate_forces(), with each parameter perturbed plus/minus d.

**Parameters**

- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.
- **fingerprintprimes** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprint derivatives as values.
- **d** (*float*) – The amount of perturbation in each parameter.

**log**
Method to set or get a logger. Should be an instance of amp.utilities.Logger.

**Parameters** **log** (*Logger object*) – Write function at which to log data. Note this must be a callable function.

**tostring**()
Returns an evaluatable representation of the calculator that can be used to re-establish the calculator.

amp.model.**calculate_fingerprints_range** (*fp*, *images*)
Calculates the range for the fingerprints corresponding to images, stored in fp. fp is a fingerprints object with the fingerprints data stored in a dictionary-like object at fp.fingerprints. (Typically this is a .utilties.Data structure.) images is a hashed dictionary of atoms for which to consider the range.

In image-centered mode, returns an array of (min, max) values for each fingerprint. In atom-centered mode, returns a dictionary of such arrays, one per element.

amp.model.**ravel_data** (*train_forces*, *mode*, *images*, *fingerprints*, *fingerprintprimes*)
Reshapes data of images into lists.

**Parameters**

- **train_forces** (*bool*) – Determining whether forces are also trained or not.
- **mode** (*str*) – Can be either 'atom-centered' or 'image-centered'.
- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This is the training set of data. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **fingerprints** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprints as values.
- **fingerprintprimes** (*dict*) – Dictionary with images hashs as keys and the corresponding fingerprint derivatives as values.

amp.model.**send_data_to_fortran** (*_fmodules*, *energy_coefficient*, *force_coefficient*, *overfit*, *train_forces*, *num_atoms*, *num_images*, *actual_energies*, *actual_forces*, *atomic_positions*, *num_images_atoms*, *atomic_numbers*, *raveled_fingerprints*, *num_neighbors*, *raveled_neighborlists*, *raveled_fingerprintprimes*, *model*, *d*)
Function that sends images data to fortran code. Is used just once on each core.

## Neural Network

**class** amp.model.neuralnetwork.**NeuralNetwork**(*hiddenlayers=(5, 5)*, *activation='tanh'*, *weights=None*, *scalings=None*, *fprange=None*, *regressor=None*, *mode=None*, *lossfunction=None*, *version=None*, *fortran=True*, *checkpoints=100*)

Bases: *amp.model.Model*

Class that implements a basic feed-forward neural network.

**Parameters**

- **hiddenlayers** (*dict*) – Dictionary of chemical element symbols and architectures of their corresponding hidden layers of the conventional neural network. Number of nodes of last layer is always one corresponding to energy. However, number of nodes of first layer is equal to three times number of atoms in the system in the case of no descriptor, and is equal to length of symmetry functions of the descriptor. Can be fed using tuples as:

```
>>> hiddenlayers = (3, 2,)
```

  for example, in which a neural network with two hidden layers, the first one having three nodes and the second one having two nodes is assigned (to the whole atomic system in the no descriptor case, and to each chemical element in the atom-centered mode). When setting only one hidden layer, the dictionary can be fed as:

```
>>> hiddenlayers = (3,)
```

  In the atom-centered mode, neural network for each species can be assigned seperately, as:

```
>>> hiddenlayers = {"O":(3,5), "Au":(5,6)}
```

  for example.

- **activation** (*str*) – Assigns the type of activation funtion. "linear" refers to linear function, "tanh" refers to tanh function, and "sigmoid" refers to sigmoid function.
- **weights** (*dict*) – In the case of no descriptor, keys correspond to layers and values are two dimensional arrays of network weight. In the atom-centered mode, keys correspond to chemical elements and values are dictionaries with layer keys and network weight two dimensional arrays as values. Arrays are set up to connect node i in the previous layer with node j in the current layer with indices w[i,j]. The last value for index i corresponds to bias. If weights is not given, arrays will be randomly generated.
- **scalings** (*dict*) – In the case of no descriptor, keys are "intercept" and "slope" and values are real numbers. In the fingerprinting scheme, keys correspond to chemical elements and values are dictionaries with "intercept" and "slope" keys and real number values. If scalings is not given, it will be randomly generated.
- **fprange** (*dict*) – Range of fingerprints of each chemical species. Should be fed as a dictionary of chemical species and a list of minimum and maximun, e.g.:

```
>>> fprange={"Pd": [0.31, 0.59], "O":[0.56, 0.72]}
```

- **regressor** (*object*) – Regressor object for finding best fit model parameters, e.g. by loss function optimization via amp.regression.Regressor.
- **mode** (*str*) – Can be either 'atom-centered' or 'image-centered'.

- **lossfunction** (*object*) – Loss function object, if at all desired by the user.
- **version** (*object*) – Version of this class.
- **fortran** (*bool*) – If True, allows for extrapolation, if False, does not allow.
- **checkpoints** (*int*) – Frequency with which to save parameter checkpoints upon training. E.g., 100 saves a checpoint on each 100th training setp. Specify None for no checkpoints.
- **note** (**.**) – with hiddenlayers.

**Raises** RuntimeError, NotImplementedError

**calculate_atomic_energy** (*afp*, *index*, *symbol*)
Given input to the neural network, output (which corresponds to energy) is calculated about the specified atom. The sum of these for all atoms is the total energy (in atom-centered mode).

**Parameters**

- **afp** (*list*) – Atomic fingerprints in the form of a list to be used as input to the neural network.
- **index** (*int*) – Index of the atom for which atomic energy is calculated (only used in the atom-centered mode).
- **symbol** (*str*) – Symbol of the atom for which atomic energy is calculated (only used in the atom-centered mode).

**Returns** Energy.

**Return type** float

**calculate_dAtomicEnergy_dParameters** (*afp*, *index=None*, *symbol=None*)
Returns the derivative of energy square error with respect to variables.

**Parameters**

- **afp** (*list*) – Atomic fingerprints in the form of a list to be used as input to the neural network.
- **index** (*int*) – Index of the atom for which atomic energy is calculated (only used in the atom-centered mode)
- **symbol** (*str*) – Symbol of the atom for which atomic energy is calculated (only used in the atom-centered mode)

**Returns** The value of the derivative of energy square error with respect to variables.

**Return type** list of float

**calculate_dForce_dParameters** (*afp*, *derafp*, *direction*, *nindex=None*, *nsymbol=None*)
Returns the derivative of force square error with respect to variables.

**Parameters**

- **afp** (*list*) – Atomic fingerprints in the form of a list to be used as input to the neural network.
- **derafp** (*list*) – Derivatives of atomic fingerprints in the form of a list to be used as input to the neural network.
- **direction** (*int*) – Direction of force.
- **nindex** (*int*) – Index of the neighbor atom which force is acting at. (only used in the atom-centered mode)

- **nsymbol** (*str*) – Symbol of the neighbor atom which force is acting at. (only used in the atom-centered mode)

**Returns** The value of the derivative of force square error with respect to variables.

**Return type** list of float

**calculate_force**(*afp*, *derafp*, *direction*, *nindex=None*, *nsymbol=None*)
Given derivative of input to the neural network, derivative of output (which corresponds to forces) is calculated.

**Parameters**

- **afp** (*list*) – Atomic fingerprints in the form of a list to be used as input to the neural network.
- **derafp** (*list*) – Derivatives of atomic fingerprints in the form of a list to be used as input to the neural network.
- **direction** (*int*) – Direction of force.
- **nindex** (*int*) – Index of the neighbor atom which force is acting at. (only used in the atom-centered mode)
- **nsymbol** (*str*) – Symbol of the neighbor atom which force is acting at. (only used in the atom-centered mode)

**Returns** Force.

**Return type** float

**fit**(*trainingimages*, *descriptor*, *log*, *parallel*, *only_setup=False*)
Fit the model parameters such that the fingerprints can be used to describe the energies in trainingimages. log is the logging object. descriptor is a descriptor object, as would be in calc.descriptor.

**Parameters**

- **trainingimages** (*dict*) – Hashed dictionary of training images.
- **descriptor** (*object*) – Class representing local atomic environment.
- **log** (*Logger object*) – Write function at which to log data. Note this must be a callable function.
- **parallel** (*dict*) – Parallel configuration dictionary. Takes the same form as in amp.Amp.
- **only_setup** (*bool*) – only_setup is primarily for debugging. It initializes all variables but skips the last line of starting the regressor.

**forcetraining**
Returns true if forcetraining is turned on (as determined by examining the convergence criteria in the loss function), else returns False.

**get_loss**(*vector*)
Method to be called by the regression master.

Takes one and only one input, a vector of parameters. Returns one output, the value of the loss (cost) function.

**Parameters** **vector** (*list*) – Parameters of the regression model in the form of a list.

**get_lossprime**(*vector*)
Method to be called by the regression master.

Takes one and only one input, a vector of parameters. Returns one output, the value of the derivative of the loss function with respect to model parameters.

**Parameters** **vector** (*list*) – Parameters of the regression model in the form of a list.

**lossfunction**
Allows the user to set a custom loss function.

For example, >>> from amp.model import LossFunction >>> lossfxn = LossFunction(energy_tol=0.0001) >>> calc.model.lossfunction = lossfxn

**Parameters** **lossfunction** (*object*) – Loss function object, if at all desired by the user.

**vector**
Access to get or set the model parameters (weights, scaling for each network) as a single vector, useful in particular for regression.

**Parameters** **vector** (*list*) – Parameters of the regression model in the form of a list.

**class** amp.model.neuralnetwork.**NodePlot**(*calc*)
Creates plots to visualize the output of the nodes in the neural networks.

initialize with a calculator that has parameters; e.g. a trained calculator or else one in which fit has been called with the setup_only flag turned on.

Call with the 'plot' method, which takes as argment a list of images

**plot**(*images*, *filename='nodeplot.pdf'*)
Creates a plot of the output of each node, as a violin plot.

**class** amp.model.neuralnetwork.**Raveler**(*weights*, *scalings*)
Class to ravel and unravel variable values into a single vector.

This is used for feeding into the optimizer. Feed in a list of dictionaries to initialize the shape of the transformation. Note no data is saved in the class; each time it is used it is passed either the dictionaries or vector. The dictionaries for initialization should be two levels deep.

weights, scalings are the variables to ravel and unravel

**to_dicts**(*vector*)
Puts the vector back into weights and scalings dictionaries of the form initialized. vector must have same length as the output of unravel.

**to_vector**(*weights*, *scalings*)
Puts the weights and scalings embedded dictionaries into a single vector and returns it. The dictionaries need to have the identical structure to those it was initialized with.

amp.model.neuralnetwork.**calculate_dOutputs_dInputs**(*parameters*, *derafp*, *outputs*, *nsymbol*)

**Parameters**

- **parameters** (*dict*) – ASE dictionary object.
- **derafp** (*list*) – Derivatives of atomic fingerprints in the form of a list to be used as input to the neural network.
- **outputs** (*dict*) – Outputs of neural network nodes.
- **nsymbol** (*str*) – Symbol of the atom for which atomic energy is calculated (only used in the atom-centered mode)

**Returns** Derivatives of outputs of neural network nodes w.r.t. inputs.

**Return type** dict

amp.model.neuralnetwork.**calculate_nodal_outputs**(*parameters*, *afp*, *symbol*)
Given input to the neural network, output (which corresponds to energy) is calculated about the specified atom. The sum of these for all atoms is the total energy (in atom-centered mode).

**Parameters**

- **parameters** (*dict*) – ASE dictionary object.
- **afp** (*list*) – Atomic fingerprints in the form of a list to be used as input to the neural network.
- **symbol** (*str*) – Symbol of the atom for which atomic energy is calculated (only used in the atom-centered mode)

**Returns** Outputs of neural network nodes

**Return type** dict

amp.model.neuralnetwork.**calculate_ohat_D_delta**(*parameters*, *outputs*, *W*)
Calculates extra matrices ohat, D, delta needed in mathematical manipulations.

Notations are consistent with those of 'Rojas, R. Neural Networks - A Systematic Introduction. Springer-Verlag, Berlin, first edition 1996'

**Parameters**

- **parameters** (*dict*) – ASE dictionary object.
- **outputs** (*dict*) – Outputs of neural network nodes.
- **W** (*dict*) – The same as weight dictionary, but the last rows associated with biases are deleted in W.

amp.model.neuralnetwork.**get_random_scalings**(*images*, *activation*, *elements=None*)
Generates initial scaling matrices, such that the range of activation is scaled to the range of actual energies.

**images** [dict] ASE atoms objects (the training set).

**activation: str** Assigns the type of activation funtion. "linear" refers to linear function, "tanh" refers to tanh function, and "sigmoid" refers to sigmoid function.

**elements: list of str** List of atom symbols; used in the atom-centered mode only.

**Returns** scalings

**Return type** float

amp.model.neuralnetwork.**get_random_weights**(*hiddenlayers*, *activation*, *no_of_atoms=None*, *fprange=None*)
Generates random weight arrays from variables.

**hiddenlayers: dict** Dictionary of chemical element symbols and architectures of their corresponding hidden layers of the conventional neural network. Number of nodes of last layer is always one corresponding to energy. However, number of nodes of first layer is equal to three times number of atoms in the system in the case of no descriptor, and is equal to length of symmetry functions in the atom-centered mode. Can be fed as:

```
>>> hiddenlayers = (3, 2,)
```

for example, in which a neural network with two hidden layers, the first one having three nodes and the second one having two nodes is assigned (to the whole atomic system in the case of no descriptor, and to each chemical element in the atom-centered mode). In the atom-centered mode, neural network for each species can be assigned seperately, as:

```
>>> hiddenlayers = {"O":(3,5), "Au":(5,6)}
```

for example.

**activation** [str] Assigns the type of activation funtion. "linear" refers to linear function, "tanh" refers to tanh function, and "sigmoid" refers to sigmoid function.

**no_of_atoms** [int] Number of atoms in atomic systems; used only in the case of no descriptor.

**fprange** [dict] Range of fingerprints of each chemical species. Should be fed as a dictionary of chemical species and a list of minimum and maximun, e.g:

```
>>> fprange={"Pd": [0.31, 0.59], "O":[0.56, 0.72]}
```

**Returns** weights

**Return type** float

## Tensorflow Neural Network

A work in progress, this module *amp.model.tflow* uses Google's TensorFlow package to implement a neural network, which may provide GPU acceleration and other advantages.

**class** amp.model.tflow.**NeuralNetwork**(*hiddenlayers=(5, 5)*, *activation='tanh'*, *keep_prob=1.0*, *maxTrainingEpochs=10000*, *importname=None*, *batchsize=2*, *initialTrainingRate=0.0001*, *miniBatch=False*, *tfVars=None*, *saveVariableName=None*, *parameters=None*, *sess=None*, *energy_coefficient=1.0*, *force_coefficient=0.04*, *scikit_model=None*, *convergenceCriteria=None*, *optimizationMethod='l-BFGS-b'*, *input_keep_prob=0.8*, *ADAM_optimizer_params={'beta1': 0.9}*, *regularization_strength=None*, *numTrainingImages={}*, *elementFingerprintLengths=None*, *fprange=None*, *weights=None*, *scalings=None*, *unit_type='float'*, *preLoadTrainingData=True*, *relativeForceCutoff=None*)

TensorFlow-based Neural Network model.

Uses Google's machine-learning code to construct a neural network. This method also allows for GPU acceleration.

**Parameters**

- **hiddenlayers** – Structure of the neural network. Can either be in the format (int,int,int), where each element represnts the size of a layer and there and the length of the list is the number of layers, or dictionary format of the network structure for each element type. E.g. {'Cu': (5, 5), 'O': (10, 5)}
- **activation** – Activation type. (XXX Provide list of possibilities.)
- **keep_prob** (*float*) – Dropout rate for the neural network to reduce overfitting. (keep_prob=1. uses all nodes, keep_prob~0.5-0.8 better for training)
- **maxTrainingEpochs** (*int*) – Maximum number of times to loop through the training data before giving up.
- **batchsize** (*int*) – Batch size for minibatch (if miniBatch is set to True).

- **initialTrainingRate** – Initial training rate for SGD optimizers like ADAM. See the TF documentation for choose this value. Likely between 1e-2 and 1e-5, depending on use case, whether mini-batch is on, etc.
- **miniBatch** (*bool*) – Whether to use minibatches in training.
- **tfVars** – Tensorflow variables (used if restoring from a previous save).
- **saveVariableName** (*str*) – Name used for the internal tensorflow variable naming scheme. If variables have the same name as another model in the same tensorflow session, there will be collisions.
- **parameters** – Dictionary of parameters to be used in initialization. Mostly these are the same keywords as the keyword arguments in this function. This is primarily used to make saving/loading easier.
- **sess** – tensorflow session to use (None means start a new session)
- **maxAtomsForces** (*int*) – Number of atoms to be used in the force training. It sets the upper bound on the number of atoms that can be used to calculate the force for. E.g., if maxAtomsForces=40, then forces can only be calculated for images with less than 40 atoms.
- **energy_coefficient** (*float*) – Used to adjust the loss function; this is the weight applied to the energy component.
- **force_coefficient** (*float or None*) – Used to adjust the loss function; this is the weight applied to the force component. Note you can turn off force training by setting this to None.
- **convergenceCriteria** (*dict*) – Dictionary of convergence criteria, analagous to the main AMP convergence criteria dictionary.
- **optimizationMethod** (*string*) – Set the optimization method for the NN parameters. Currently either 'ADAM' for the ADAM optimizer in tensorflow, of 'l-BFGS-b' for the deterministic l-BFGS-b method. ADAM is usually faster per training step, has all of the benefits of being a stochastic optimizer, and allows for mini-batch operation, but has more tunable parameters and can be harder to get working well. l-BFGS-b usually works for small/moderate network sizes.
- **input_keep_prob** – Dropout ratio on the first layer (from fingerprints to the neural network. Rule of thumb is this should be 0 to 0.2. Only applies when using a SGD optimizer like ADAM. BFGS ignores this.
- **ADAM_optimizer_params** – Dictionary of parameters to pass to the ADAM optimizer. See https://www.tensorflow.org/versions/r0.11/api_docs/python/train.html#AdamOptimizer for documentation
- **regularization_strength** – Weight for L2-regularization in the cost function
- **fprange** (*dict*) – This is a dictionary that contains the minimum and maximum values seen for each fingerprint of each element. These
- **weights** (*np array*) – Input that allows the NN weights (and biases) to be set directly. This is only used for verifying that the calculation is working correctly in the CuOPd test case. In general, don't use this except for testing the code. This argument is analagous to the original AMP NeuralNetwork module.
- **scalings** – Input that allows the NN final scaling o be set directly. This is only used for verifying that the calculation is working correctly in the CuOPd test case. In general, don't use this except for testing the code. This argument is analagous to the original AMP NeuralNetwork module.

- **unit_type** (*string*) – Sets the internal datatype of the tensorflow model. Either "float" for 32-bit FP precision, or "double" for 64-bit FP precision.
- **preLoadTrainingData** (*bool*) – Decides whether to run the training by preloading all training data into tensorflow. Doing so results in faster training if the entire dataset can fit into memory. This only works when not using mini-batch.
- **relativeForceCutoff** (*float*) – Parameter for controlling whether the force contribution to the trained cost function is absolute (just differences of force compared to training forces) or relative for large values of the force. This basically sets the upper limit on the forces that should be fitted (e.g. if the force is >A, then the force is scaled). This helps when a small number of images have very large forces that don't need to be reconstructed perfectly.

**calculate_energy**(*fingerprint*)
Get the energy by feeding in a list to the get_list version (which is more efficient for anything greater than 1 image).

**calculate_forces**(*fingerprint*, *derfingerprint*)

**constructModel**(*sess*, *graph*, *preLoadData=False*, *numElements=None*, *numTrainingImages=None*, *num_dgdx_Eindices=None*, *numTrainingAtoms=None*)
Sets up the tensorflow neural networks for each atom type.

**constructSessGraphModel**(*tfVars*, *sess*, *trainOnly=False*, *numElements=None*, *numTrainingImages=None*, *num_dgdx_Eindices=None*, *numTrainingAtoms=None*)

**fit**(*trainingimages*, *descriptor*, *parallel*, *log=None*)
Fit takes a bunch of training images (which are assumed to have a working calculator attached), and fits the internal variables to the training images.

**generateFeedInput**(*curinds*, *energies*, *atomArraysAll*, *dgdx*, *dgdx_Eindices*, *dgdx_Xindices*, *nAtomsDict*, *atomsIndsReverse*, *batchsize*, *trainingrate*, *keepprob*, *inputkeepprob*, *natoms*, *forcesExp=0.0*, *forces=False*, *energycoefficient=1.0*, *forcecoefficient=None*, *training=True*)
Generates the input dictionary that maps various inputs on the python side to placeholders for the tensorflow model.

**getVariance**(*fingerprint*, *nSamples=10*, *l=1.0*)

**get_energy_list**(*hashs*, *fingerprintDB*, *fingerprintDerDB=None*, *keep_prob=1.0*, *input_keep_prob=1.0*, *forces=False*, *nsamples=1*)
Methods to get the energy and forces for a set of configurations.

**initializeVariables**()
Resets all of the variables in the current tensorflow model.

**preLoadFeed**(*feedinput*)

**setWeightsScalings**(*feedinput*, *weights*, *scalings*)

**tostring**()
Dummy tostring to make things work.

amp.model.tflow.**bias_variable**(*shape*, *name*, *unit_type*, *a=0.1*)
Helper functions taken from the MNIST tutorial to generate weight and bias variables with random initial weights.

amp.model.tflow.**generateBatch**(*curinds*, *elements*, *atomArraysAll*, *nAtomsDict*, *atomsIndsReverse*, *dgdx*, *dgdx_Eindices*, *dgdx_Xindices*)
This method generates batches from a large dataset using a set of selected indices curinds.

amp.model.tflow.**generateTensorFlowArrays**(*fingerprintDB*, *elements*, *keylist*, *fingerprintDerDB=None*)

This function generates the inputs to the tensorflow graph for the selected images. The essential problem is that each neural network is associated with a specific element type. Thus, atoms in each ASE image need to be sent to different networks.

Inputs:

fingerprintDB: a database of fingerprints, as taken from the descriptor

elements: a list of element types (e.g. 'C','O', etc)

**keylist: a list of hashs into the fingerprintDB that we want to create** inputs for

**fingerprintDerDB: a database of fingerprint derivatives, as taken from the** descriptor

maxAtomsForces: the maximum length of the atoms

Outputs:

**atomArraysAll: a dictionary of fingerprint inputs to each element's neural** network

**nAtomsDict: a dictionary for each element with lists of the number of** atoms of each type in each image

**atomsIndsReverse: a dictionary that contains the index of each atom into** the original keylist

nAtoms: the number of atoms in each image

**atomArraysAllDerivs: dictionary of fingerprint derivates for each** element's neural network

amp.model.tflow.**model**(*x*, *segmentinds*, *keep_prob*, *input_keep_prob*, *batchsize*, *neuronList*, *activationType*, *fplength*, *mask*, *name*, *dgdx*, *dgdx_Xindices*, *dgdx_Eindices*, *element*, *unit_type*, *totalNumAtoms*)

Generates a multilayer neural network with variable number of neurons, so that we have a template for each atom's NN.

amp.model.tflow.**reorganizeForces**(*forces*, *natoms*)

amp.model.tflow.**weight_variable**(*shape*, *name*, *unit_type*, *stddev=0.1*)

Helper functions taken from the MNIST tutorial to generate weight and bias variables with random initial weights.

CHAPTER 19

# Regression

This module includes a regressor object used to optimize the parameters of the machine-learning model.

## Module contents

**class** amp.regression.**Regressor**(*optimizer='BFGS'*, *optimizer_kwargs=None*, *lossprime=True*)

Class to manage the regression of a generic model. That is, for a given parameter set, calculates the cost function (the difference in predicted energies and actual energies across training images), then decides how to adjust the parameters to reduce this cost function. Global optimization conditioners (e.g., simulated annealing, etc.) can be built into this class.

**Parameters**

- **optimizer** (*str*) – The optimizer to use. Several defaults are available including 'L-BFGS-B', 'BFGS', 'TNC', or 'NCG'. Alternatively, any function can be supplied which behaves like scipy.optimize.fmin_bfgs.
- **optimizer_kwargs** (*dict*) – Optional keywords for the corresponding optimizer.
- **lossprime** (*boolean*) – Decides whether or not the regressor needs to be fed in by gradient of the loss function as well as the loss function itself.

**regress**(*model*, *log*)

Performs the regression. Calls model.get_loss, which should return the current value of the loss function until convergence has been reached, at which point it should raise a amp.utilities.ConvergenceException.

**Parameters**

- **model** (*object*) – Class representing the regression model.
- **log** (*str*) – Name of script to log progress.

CHAPTER 20

# Utilities

This module contains utilities for use with various aspects of the Amp calculator.

## Module contents

**class** amp.utilities.**Annealer**(*calc*, *images*, *Tmax=None*, *Tmin=None*, *steps=None*, *updates=None*)

Bases: object

Inspired by the simulated annealing implementation of Richard J. Wagner <wagnerr@umich.edu> and Matthew T. Perry <perrygeo@gmail.com> at https://github.com/perrygeo/simanneal.

Performs simulated annealing by calling functions to calculate loss and make moves on a state. The temperature schedule for annealing may be provided manually or estimated automatically.

Can be used by something like:

```
>>> from amp import Amp
>>> from amp.descriptor.gaussian import Gaussian
>>> from amp.model.neuralnetwork import NeuralNetwork
>>> calc = Amp(descriptor=Gaussian(), model=NeuralNetwork())
```

which will initialize tha calc object as usual, and then

```
>>> from amp.utilities import Annealer
>>> Annealer(calc=calc, images=images)
```

which will perform simulated annealing global search in parameters space, and finally

```
>>> calc.train(images=images)
```

for gradient descent optimization.

**Tmax = 20.0**

**Tmin = 2.5**

**anneal**()

Minimizes the loss of a system by simulated annealing.

**Parameters** **state** – An initial arrangement of the system

**Returns** The best state and loss found.

**Return type** state, loss

**auto** (*minutes*, *steps=2000*)
Minimizes the loss of a system by simulated annealing with automatic selection of the temperature schedule.

Keyword arguments: state – an initial arrangement of the system minutes – time to spend annealing (after exploring temperatures) steps – number of steps to spend on each stage of exploration

Returns the best state and loss found.

**copy_state** (*state*)
Returns an exact copy of the provided state Implemented according to self.copy_strategy, one of

•deepcopy : use copy.deepcopy (slow but reliable)

•slice: use list slices (faster but only works if state is list-like)

•method: use the state's copy() method

**copy_strategy = 'copy'**

**get_loss** (*state*)
Calculate state's loss

**move** (*state*)
Create a state change

**static round_figures** (*x*, *n*)
Returns x rounded to n significant figures.

**save_state** (*fname=None*)
Saves state

**save_state_on_exit = False**

**set_schedule** (*schedule*)
Takes the output from *auto* and sets the attributes

**set_user_exit** (*signum*, *frame*)
Raises the user_exit flag, further iterations are stopped

**steps = 10000**

**static time_string** (*seconds*)
Returns time in seconds as a string formatted HHHH:MM:SS.

**update** (*step*, *T*, *L*, *acceptance*, *improvement*)
Prints the current temperature, loss, acceptance rate, improvement rate, elapsed time, and remaining time.

The acceptance rate indicates the percentage of moves since the last update that were accepted by the Metropolis algorithm. It includes moves that decreased the loss, moves that left the loss unchanged, and moves that increased the loss yet were reached by thermal excitation.

The improvement rate indicates the percentage of moves since the last update that strictly decreased the loss. At high temperatures it will include both moves that improved the overall state and moves that simply undid previously accepted moves that increased the loss by thermal excititation. At low temperatures it will tend toward zero as the moves that can decrease the loss are exhausted and moves that would increase the loss are no longer thermally accessible.

**updates = 50**

**user_exit = False**

**exception** amp.utilities.**ConvergenceOccurred**
Bases: exceptions.Exception

Kludge to decide when scipy's optimizers are complete.

**class** amp.utilities.**Data**(*filename*, *db=<class amp.utilities.FileDatabase>*, *calculator=None*)
Serves as a container (dictionary-like) for (key, value) pairs that also serves to calculate them.

Works by default with python's shelve module, but something that is built to share the same commands as shelve will work fine; just specify this in dbinstance.

Designed to hold things like neighborlists, which have a hash, value format.

This will work like a dictionary in that items can be accessed with data[key], but other advanced dictionary functions should be accessed with through the .d attribute:

```
>>> data = Data(...)
>>> data.open()
>>> keys = data.d.keys()
>>> values = data.d.values()
```

**calculate_items**(*images*, *parallel*, *log=None*)
Calculates the data value with 'calculator' for the specified images.

images is a dictionary, and the same keys will be used for the current database.

**close**()
Safely close the database.

**open**(*mode='r'*)
Open the database connection with mode specified.

**class** amp.utilities.**FileDatabase**(*filename*)
Using a database file, such as shelve or sqlitedict, that can handle multiple processes writing to the file is hard.

Therefore, we take the stupid approach of having each database entry be a separate file. This class behaves essentially like shelve, but saves each dictionary entry as a plain pickle file within the directory, with the filename corresponding to the dictionary key (which must be a string).

Like shelve, this also keeps an internal (memory dictionary) representation of the variables that have been accessed.

Also includes an archive feature, where files are instead added to a file called 'archive.tar.gz' to save disk space. If an entry exists in both the loose and archive formats, the loose is taken to be the new (correct) value.

**archive**()
Cleans up to save disk space and reduce huge number of files.

That is, puts all files into an archive. Compresses all files in <path>/loose and places them in <path>/archive.tar.gz. If archive exists, appends/modifies.

**close**()
Only present for compatibility with shelve.

**keys**()
Return list of keys, both of in-memory and out-of-memory items.

**classmethod open**(*Cls*, *filename*, *flag=None*)
Open present for compatibility with shelve. flag is ignored; this format is always capable of both reading and writing.

**update**(*newitems*)

**class** amp.utilities.**Logger**(*file*)
Logger that can also deliver timing information.

**Parameters** **file** (*str*) – File object or path to the file to write to. Or set to None for a logger that does nothing.

**__call__**(*message*, *toc=None*, *tic=False*)
Writes message to the log file.

**Parameters**

- **message** (*str*) – Message to be written.
- **toc** (*bool or str*) – If toc=True or toc=label, it will append timing information in minutes to the timer.
- **tic** (*bool or str*) – If tic=True or tic=label, will start the generic timer or a timer associated with label. Equivalent to self.tic(label).

**tic**(*label=None*)
Start a timer.

**Parameters** **label** (*str*) – Label for managing multiple timers.

**class** amp.utilities.**MessageDictionary**(*process_id*)
Standard container for all messages (typically requests, via zmq.context.socket.send_pyobj) sent from the workers to the master.

This returns a simple dictionary. This is roughly email format. Initialize with process id (e.g., 'from'). Call with subject and data (body).

**class** amp.utilities.**MetaDict**
Bases: dict

Dictionary that can also store metadata. Useful for images dictionary so that images can still be iterated by keys.

**metadata = {}**

**exception** amp.utilities.**TrainingConvergenceError**
Bases: exceptions.Exception

Error to be raised if training does not converge.

amp.utilities.**assign_cores**(*cores*, *log=None*)
Tries to guess cores from environment.

If fed a log object, will write its progress.

amp.utilities.**hash_image**(*atoms*)
Creates a unique signature for a particular ASE atoms object.

This is used to check whether an image has been seen before. This is just an md5 hash of a string representation of the atoms object.

**Parameters** **atoms** (*ASE dict*) – ASE atoms object.

**Returns**

**Return type** Hash key of 'atoms'.

amp.utilities.**hash_images**(*images*, *log=None*, *ordered=False*)
Converts input images – which may be a list, a trajectory file, or a database – into a dictionary indexed by their hashes.

Returns this dictionary. If ordered is True, returns an OrderedDict. When duplicate images are encountered (based on encountering an identical hash), a warning is written to the logfile. The number of duplicates of each

image can be accessed by examinging dict_images.metadata['duplicates'], where dict_images is the returned dictionary.

amp.utilities.**importer**(*name*)
Handles strange import cases, like pxssh which might show up in eithr the package pexpect or pxssh.

amp.utilities.**make_filename**(*label*, *base_filename*)
Creates a filename from the label and the base_filename which should be a string.

Returns None if label is None; that is, it only saves output if a label is specified.

**Parameters**

- **label** (*str*) – Prefix.
- **base_filename** (*str*) – Basic name of the file.

amp.utilities.**make_sublists**(*masterlist*, *n*)
Randomly divides the masterlist into n sublists of roughly equal size.

The intended use is to divide a keylist and assign keys to each task in parallel processing. This also destroys the masterlist (to save some memory).

amp.utilities.**now**(*with_utc=False*)

**Returns**

**Return type** String of current time.

amp.utilities.**randomize_images**(*images*, *fraction=0.8*)
Randomly assigns 'fraction' of the images to a training set and (1 - 'fraction') to a test set. Returns two lists of ASE images.

**Parameters**

- **images** (*list or str*) – List of ASE atoms objects in ASE format. This can also be the path to an ASE trajectory (.traj) or database (.db) file.
- **fraction** (*float*) – Portion of train_images to all images.

**Returns** **train_images, test_images** – Lists of train and test images.

**Return type** list

amp.utilities.**setup_parallel**(*parallel*, *workercommand*, *log*)
Starts the worker processes and the master to control them.

This makes an SSH connection to each node (including the one the master process runs on), then creates the specified number of processes on each node through its SSH connection. Then sets up ZMQ for efficienty communication between the worker processes and the master process.

Uses the parallel dictionary as defined in amp.Amp. log is an Amp logger. module is the name of the module to be called, which is usually given by self.calc.__module, etc. workercommand is stub of the command used to start the servers, typically like "python -m amp.descriptor.gaussian". Appended to this will be " <pid> <serversocket> &" where <pid> is the unique ID assigned to each process and <serversocket> is the address of the server, like 'node321:34292'.

**Returns**

**server** – The ssh connections (pxssh instances; if these objects are destroyed pxssh will close the sessions)

the pid_count, which is the total number of workers started. Each worker can be communicated directly through its PID, an integer between 0 and pid_count

**Return type** (a ZMQ socket)

amp.utilities.**start_workers**(*process_ids*, *workerhostname*, *workercommand*, *log*, *envcommand*)
A function to start a new SSH session and establish processes on that session.

amp.utilities.**string2dict**(*text*)
Converts a string into a dictionary.

Basically just calls *eval* on it, but supplies words like OrderedDict and matrix.

CHAPTER 21

# Analysis

Tools for analysis of output exist here.

## Module contents

amp.analysis.**plot_convergence**(*logfile*, *plotfile='convergence.pdf'*)

Makes a plot of the convergence of the cost function and its energy and force components.

**Parameters**

- **logfile** (*str*) – Name or path to the log file.
- **plotfile** (*str*) – Name or path to the plot file.

amp.analysis.**plot_error**(*load*, *images*, *label='error'*, *dblabel=None*, *plot_forces=True*, *plotfile=None*, *color='b.'*, *overwrite=False*, *returndata=False*, *energy_coefficient=1.0*, *force_coefficient=0.04*)

Makes an error plot of Amp energies and forces versus real energies and forces.

**Parameters**

- **load** (*str*) – Path for loading an existing ".amp" file. Should be fed like 'load="filename.amp"'.
- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **label** (*str*) – Default prefix/location used for all files.
- **dblabel** (*str*) – Optional separate prefix/location of database files, including fingerprints, fingerprint primes, and neighborlists, to avoid calculating them. If not supplied, just uses the value from label.
- **plot_forces** (*bool*) – Determines whether or not forces should be plotted as well.
- **plotfile** (*Object*) – File for the plot.
- **color** (*str*) – Plot color.
- **overwrite** (*bool*) – If a plot or an script containing values found overwrite it.
- **returndata** (*bool*) – Whether to return a reference to the figures and their data or not.
- **energy_coefficient** (*float*) – Coefficient of energy loss in the total loss function.
- **force_coefficient** (*float*) – Coefficient of force loss in the total loss function.

amp.analysis.**plot_parity**(*load*, *images*, *label='parity'*, *dblabel=None*, *plot_forces=True*, *plotfile=None*, *color='b.'*, *overwrite=False*, *returndata=False*, *energy_coefficient=1.0*, *force_coefficient=0.04*)

Makes a parity plot of Amp energies and forces versus real energies and forces.

**Parameters**

- **load** (*str*) – Path for loading an existing ".amp" file. Should be fed like 'load="filename.amp"'.
- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **label** (*str*) – Default prefix/location used for all files.
- **dblabel** (*str*) – Optional separate prefix/location of database files, including fingerprints, fingerprint primes, and neighborlists, to avoid calculating them. If not supplied, just uses the value from label.
- **plot_forces** (*bool*) – Determines whether or not forces should be plotted as well.
- **plotfile** (*Object*) – File for the plot.
- **color** (*str*) – Plot color.
- **overwrite** (*bool*) – If a plot or an script containing values found overwrite it.
- **returndata** (*bool*) – Whether to return a reference to the figures and their data or not.
- **energy_coefficient** (*float*) – Coefficient of energy loss in the total loss function.
- **force_coefficient** (*float*) – Coefficient of force loss in the total loss function.

amp.analysis.**plot_sensitivity**(*load*, *images*, *d=0.0001*, *label='sensitivity'*, *dblabel=None*, *plotfile=None*, *overwrite=False*, *energy_coefficient=1.0*, *force_coefficient=0.04*)

Returns the plot of loss function in terms of perturbed parameters.

Takes the load file and images. Any other keyword taken by the Amp calculator can be fed to this class also.

**Parameters**

- **load** (*str*) – Path for loading an existing ".amp" file. Should be fed like 'load="filename.amp"'.
- **images** (*list or str*) – List of ASE atoms objects with positions, symbols, energies, and forces in ASE format. This can also be the path to an ASE trajectory (.traj) or database (.db) file. Energies can be obtained from any reference, e.g. DFT calculations.
- **d** (*float*) – The amount of perturbation in each parameter.
- **label** (*str*) – Default prefix/location used for all files.
- **dblabel** (*str*) – Optional separate prefix/location of database files, including fingerprints, fingerprint primes, and neighborlists, to avoid calculating them. If not supplied, just uses the value from label.
- **plotfile** (*Object*) – File for the plot.
- **overwrite** (*bool*) – If a plot or an script containing values found overwrite it.
- **energy_coefficient** (*float*) – Coefficient of energy loss in the total loss function.
- **force_coefficient** (*float*) – Coefficient of force loss in the total loss function.

amp.analysis.**read_trainlog**(*logfile*)
Reads the log file from the training process, returning the relevant parameters.

**Parameters** **logfile** (*str*) – Name or path to the log file.

**Indices and tables**

- genindex
- modindex
- search

## a

## Symbols



## A



## B



## C

## D



## F



## G

## U



## V



## W



## Z