Amp Documentation

Release 0.6

Andrew A. Peterson, Alireza Khorshidi

Jul 19, 2018

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.

The latest stable release of Amp is version 0.6, released on July 31, 2017; see the *Release notes* page for a download link. Please see the project's git repository for the latest development version or a place to report an issue.

You can read about Amp in the below paper; if you find this project 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.

News:

An amp-users mailing list has been started, for general discussions about the use and development of Amp. You can subscribe via listserv at:

https://listserv.brown.edu/?A0=AMP-USERS

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

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 (it also supports Python3).
- 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.

2.1 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.

2.2 Get the code

The latest stable release of Amp is version 0.5, which is permanently available at https://doi.org/10.5281/zenodo.322427. If installing version 0.5, you should follow ignore the rest of this page and follow the instructions included with the download (see docs/installation.rst or look for v0.5 on http://amp.readthedocs.io).

We are constantly improving *Amp* and adding features, so depending on your needs it may be preferable to use the development version rather than "stable" releases. We run daily unit tests to try 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'.

2.3 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.

2.4 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. We have included a *Make* file that automatizes the building of Fortran modules. To use it, install GNU Makefile on your Linux distribution or macOS. For Python2, then simply do:

```
$ cd <installation-directory>/amp/
$ make python2
```

For Python3:

```
$ cd <installation-directory>/amp/
$ make python3
```

If you do not have the GNU Makefile installed, you can prepare the Fortran extension modules manually in the following steps:

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/zernike.f90 model/neuralnetwork.f90
```

Note that for Python3, you need to use *f2py3* instead of *f2py*.

or on a Windows machine by:

```
$ f2py -c -m fmodules model.f90 descriptor/cutoffs.f90 descriptor/gaussian.f90␣
↪descriptor/zernike.f90 model/neuralnetwork.f90 --fcompiler=gnu95 --compiler=mingw32
```

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.

2.5 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 and GNU Makefile installed, simply do:

```
$ make py2tests      (for Python2)
$ make py3tests      (for Python3)
```

Otherwise, if you have only nose installed (and not GNU Makefile), run the commands below:

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

CHAPTER 3

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.

3.1 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 `NeuralNetwork` 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.

3.2 Adjusting convergence parameters

To control how tightly the energy and/or forces are 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 `default_parameters`:

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

Note that you can also set a maximum residual of any energy or force prediction with the appropriate keywords above.

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
```

3.3 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. However, this can create issues: training will tend to be slower than training energies only, 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, or if there are egg-box errors), you might not be able to train at all. For this reason, Amp defaults to energy-only training, but you can turn on force-training via the convergence dictionary as noted above. Note that there is a *force_coefficient* keyword also fed to the `LossFunction` class which can control the relative weighting of the energy and force RMSEs used in the path to convergence.

```
from amp.model import LossFunction

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

3.4 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': ...}*.

3.5 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.

3.6 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.
```

3.7 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()` method 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 4

Community

4.1 Mailing list

An amp-users listserv is available for general discussion, troubleshooting, suggestions, etc. It is available at

https://listserv.brown.edu/?A0=AMP-USERS

The archives of this list are also available to members of the list.

4.2 Bugs and issues

To report bugs, issues, works-in-progress, or feature requests (although those might best be first discussed on amp-users), please use our Issue Tracker on the repository page. It is available at

https://bitbucket.org/andrewpeterson/amp/issues

4.3 Contributions

You are welcome to contribute to this project. See the *Development* page.

CHAPTER 5

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 **R** is the position of atoms in the system.

5.1 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:

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

5.2 Descriptor

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

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

5.2.1 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 η and center R_s,

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

By specifying many values of η 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 $\theta_{ijk} = \cos^{-1}\left(\dfrac{\mathbf{R}_{ij}.\mathbf{R}_{ik}}{R_{ij}R_{ik}}\right)$, ($\mathbf{R}_{ij} = \mathbf{R}_i - \mathbf{R}_j$), centered at atom i, according to

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

with parameters λ, η, and ζ, 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

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

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 γ 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 θ_{ijk} between the triplet of atoms i, j, and k with central atom i:

5.2.2 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:

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

Next, components of the Zernike descriptor are computed from Zernike moments of the above atomic density distribution 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.

5.2.3 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.

5.3 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.

5.3.1 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 6

Credits

6.1 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.

6.2 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 7

Release notes

7.1 0.6.1

Release date: July 19, 2018

- Updated to allow installation via pip.

7.2 0.6

Release date: July 31, 2017

- Python 3 compatibility. Following the release of python3-compatible ASE, we decided to jump on the wagon ourselves. The code should still work fine in python 2.7. (The exception is the tensorflow module, which still only lives inside python 2, unfortunately.)
- A community page has been added with resources such as the new mailing list and issue tracker.
- The default convergence parameters have been changed to energy-only training; force-training can be added by the user via the loss function. This makes convergence easier for new users.
- Convergence plots show maximum residuals as well as root mean-squared error.
- Parameters to make the Gaussian feature vectors are now output to the log file.
- The helper function `make_symmetry_functions()` has been added to more easily customize Gaussian fingerprint parameters.

Permanently available at https://doi.org/10.5281/zenodo.836788

7.3 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.

Permanently available at https://doi.org/10.5281/zenodo.322427

7.4 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

7.5 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

7.6 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

7.7 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 8

Example scripts

8.1 A basic fitting script

The below script uses Gaussian descriptors with a neural network backend — the Behler-Parrinello approach — to train energies only to a training set made by the script. Note that most of the code is just generating the training data, and the training takes place in a couple of lines.

```
"""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


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]))
```

(continues on next page)

(continued from previous page)

```
    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.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*.

8.2 A basic script with forces

The below script trains both energy and forces to the same training set as above. 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())
```

(continues on next page)

(continued from previous page)

```
    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*.

8.3 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.descriptor.gaussian import Gaussian
descriptor = Gaussian()
from amp.utilities import hash_images
images = hash_images(images, ordered=True)
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."""
```

(continues on next page)

(continued from previous page)

```
    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 9

Analysis

9.1 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.

9.2 Other plots

There are several other plotting tools within the `amp.analysis` 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 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.

10.1 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]
```

10.2 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*.

10.2.1 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'))
```

(continues on next page)

(continued from previous page)

```
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

11.1 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

12.1 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).

12.2 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

Gaussian descriptor

13.1 Custom parameters

The Gaussian descriptor creates feature vectors based on the Behler scheme, and defaults to values used in Nano Letters 14:2670, 2014. You can specify custom parameters for the elements of the feature vectors as listed in the documentation of the `Gaussian` class.

There is also a helper function `make_symmetry_functions()` within the `amp.descriptor.gaussian` module to assist with this. An example of making a custom fingerprint is given below for a two-element system.

```
import numpy as np
from amp import Amp
from amp.descriptor.gaussian import Gaussian, make_symmetry_functions
from amp.model.neuralnetwork import NeuralNetwork

elements = ['Cu', 'Pt']
G = make_symmetry_functions(elements=elements, type='G2',
                            etas=np.logspace(np.log10(0.05), np.log10(80.),
                                             num=4))
G += make_symmetry_functions(elements=elements, type='G4',
                             etas=[0.005],
                             zetas=[1., 4.],
                             gammas=[+1., -1.])

G = {'Cu': G,
     'Pt': G}
calc = Amp(descriptor=Gaussian(Gs=G),
           model=NeuralNetwork())
```

CHAPTER 14

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).

14.1 Dependencies

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

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.
↪11.0-cp27-none-linux_x86_64.whl
pip install -U --upgrade $TF_BINARY_URL
```

or macOS:

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.
↪11.0-py2-none-any.whl
pip install -U --upgrade $TF_BINARY_URL
```

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

14.2 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."""

```

(continues on next page)

(continued from previous page)

```python
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__':
```

(continues on next page)

(continued from previous page)

```
    train_test()
```

14.3 Known issues

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

14.4 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 15

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.

15.1 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.

15.2 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.

15.3 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 16

Development

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

16.1 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.

16.2 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. Finally, to make users aware of your new feature or change, add a bullet point to the release notes page of the documentation under the Development version heading.

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

16.3 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.

16.4 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., 0.5.1 — just add those to the v0.5 branch.) All subsequent work is in the new branch. Note the branch name starts with "v", while the tag names will not, to avoid naming conflicts.
- 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*.
- Change revision history to include this release; generally the changes should have been catalogued under a "Development version" heading.
- Commit and push the changes to the new branch on bitbucket.
- Tag the release with the release number, e.g., '0.5' or '0.5.1', the latter being for bug fixes. Do this on a local machine (on the correct branch) with *git tag -a 0.5*, followed by *git push origin –tags*.
- Add the version to readthedocs' available versions; also set it as the default stable version.
- Change the nightly tests to test this branch as the "stable" build.
- Create a DOI for the release via zenodo.org. Note that all the ".git" files and folders should be removed from the files before uploading to Zenodo. The DOI can then be added to the development version's release notes. (I don't think there's a way to get it into the archival version on Zenodo!)

Module autodocumentation:

CHAPTER 17

Main

This module is the main part of the Amp package.

17.1 Module contents

CHAPTER 18

Descriptor

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

18.1 Gaussian

18.2 Zernike

18.3 Bispectrum

18.4 Cutoff functions

CHAPTER 19

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.

19.1 Model

19.2 Neural Network

19.3 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.

CHAPTER 20

Regression

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

20.1 Module contents

CHAPTER 21

Utilities

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

21.1 Module contents

CHAPTER 22

Analysis

Tools for analysis of output exist here.

22.1 Module contents

Indices and tables

- genindex
- modindex
- search