Contents

1 Introduction

All enclosed documentation and data is available from Github at: https://github.com/jboes/amp-tutorial.git

Similarly, the PDF can be found here: https://github.com/jboes/amp-tutorial/blob/master/workbook.pdf

All of the code described within is open source with the exception of the Vienna *ab initio* simulation package (VASP).

Have questions or are interested in learning more? Feel free to contact me at jboes@andrew.cmu.edu.

2 Theory

2.1 Density Functional Theory

In the Kitchin group, we use the VASP suit to perform Density functional theory (DFT) calculations.

- Solves the Kohn-Sham Equations which approximate the Schrodinger equation with electron density.
- VASP is ideal for bulk system since it uses plane waves to estimate the electron density.

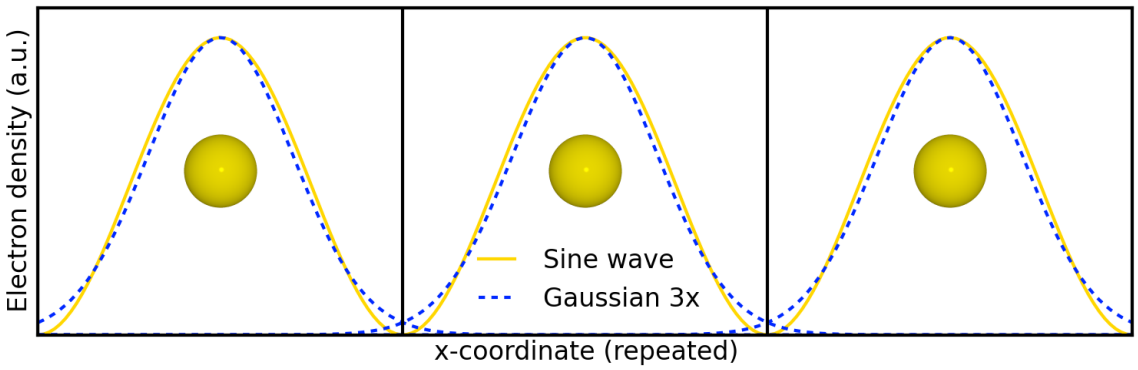


Figure 1: Example of a system using plane waves

- Although powerful and accurate DFT is too slow for more advanced application, such as molecular dynamics (MD) and larger unit cells.
- MD requires a large number of calculations in series, which DFT is poorly suited for.

2.2 Neural Networks (NN)

Neural networks are a machine learning technique which can be use to relate an input to

- This has been successfully implemented to predict potential energies of atomic structures
- A basic feed-forward neural networks is demonstrated in Figure 2 for a 2 atom system.

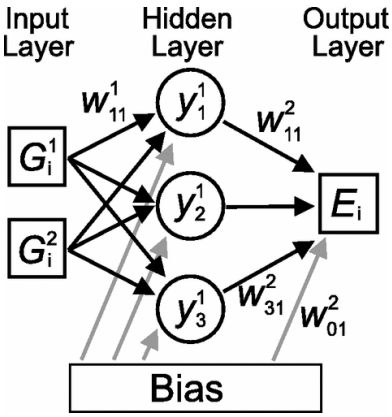


Figure 2: A basic neural network framework for a 2 atom system.

- Feed-forward neural networks are limited by their construction. Atoms cannot be added or disordered using this method.

2.3 Behler-Parrinello descriptors

To use NNs on variable systems of atoms we need a different set of inputs than atomic positions.

- There are many ways to describe various atomic configurations other than Cartesian coordinates.
- Whatever descriptor we use needs to be accessible without performing a DFT calculation.

- Behler and Parrinello suggested a cutoff radius and "symmetry functions".

The cutoff radius R_c us useful because it limits the size of the symmetry function required. This is demonstrated in Figure 3.

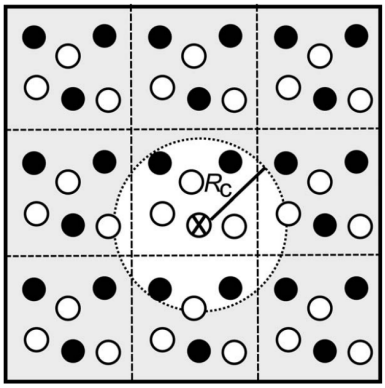
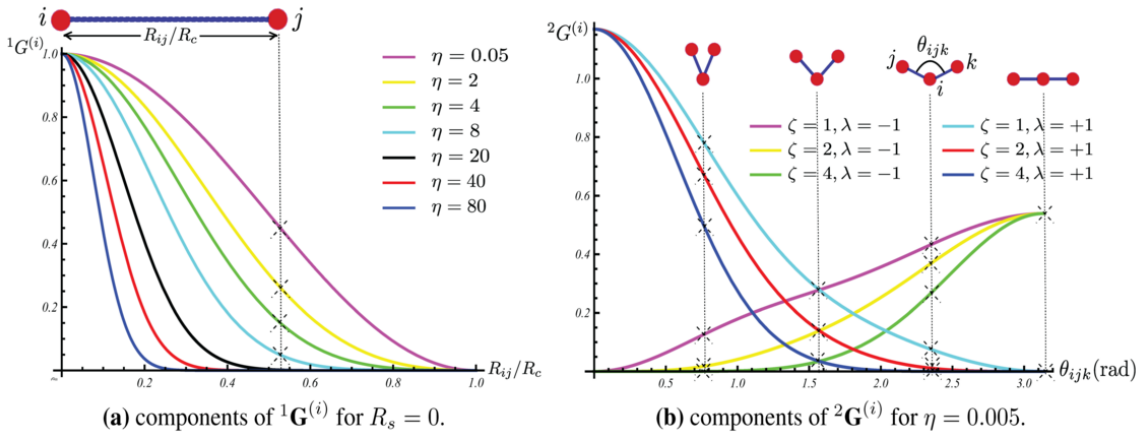


Figure 3: Demonstration of the cutoff radius in

- Choice of R_c is critical! We assume no interactions occur beyond this cutoff.
- This is not suitable for systems with long-range interactions.

For each atom, we construct a "symmetry functions" made of various Behler-Parrinello descriptors. Some of these descriptors are demonstrated in Figure 4.



(a) components of $^1\mathbf{G}^{(i)}$ for $R_s = 0$. **(b)** components of $^2\mathbf{G}^{(i)}$ for $\eta = 0.005$.

Figure 4: Visualization of the 1G and 2G Behler descriptors.

- We can use as many descriptors as needed to define the system.

- Fewer is better since less variables makes the function smaller, which in turn computes faster.

Finally, for each atom in a system, we calculate the "symmetry function" and pass it to a general feed-forward NN as shown in Figure 5.

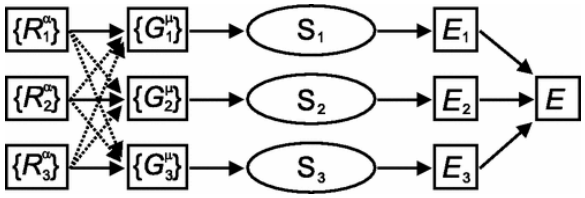


Figure 5: A Behler-Parrinello neural network for a 3 atom system.

- Then we sum the energy contributions from each atom to get the total energy.
- More information: http://dx.doi.org/10.1103/PhysRevLett.98.146401

3 Convergence calculations

First, we need to determine an appropriate level of convergence for our calculations. I usually use the natural bulk configuration of a metal for these studies. For Pd, this is fcc.

3.1 k-point convergence

First, we determine an appropriate k-point convergence. We will be performing many calculations, so a high level of accuracy is desirable, but not if the computational cost is too high. I use a high energy cutoff (400 eV) to make sure there are no effects from encut convergence to potentially skew the results.

Figure 6 shows that a Monkhorst-pack grid of roughly (14, 14, 14) k-points is sufficient to each 1 meV convergence.

```
1 from ase.lattice.cubic import FaceCenteredCubic
2 import numpy as np
```

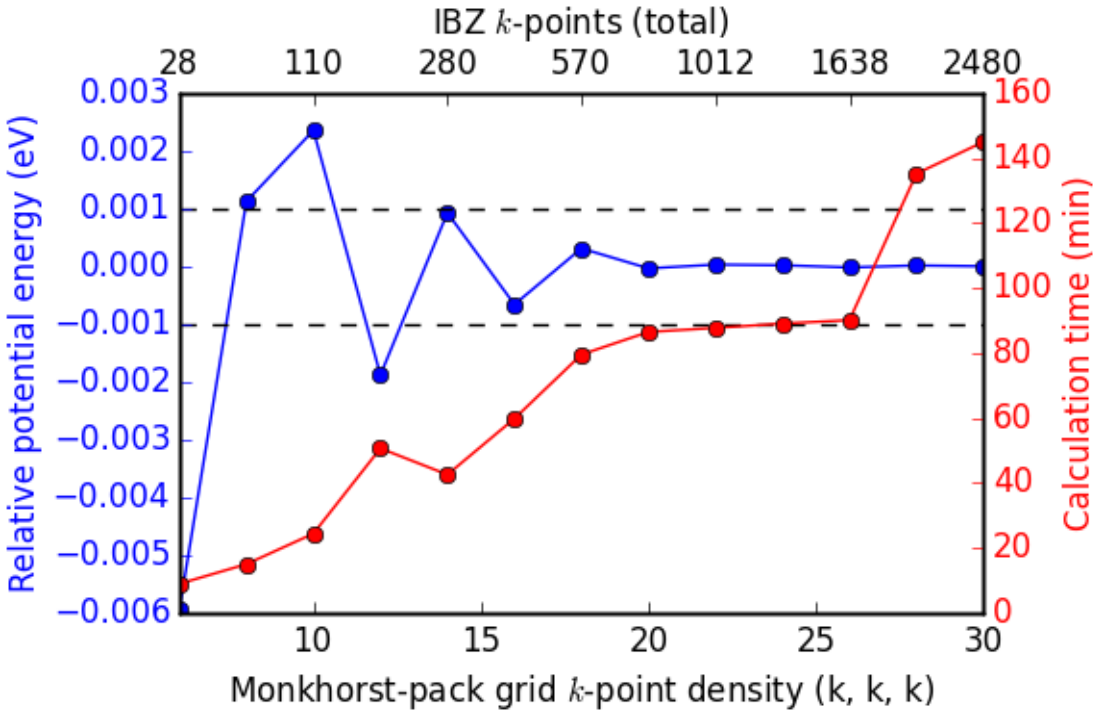


Figure 6: k-point convergence metrics for a single atom unit cell of fcc Pd

```
from jasp import *
import matplotlib.pyplot as plt
from ase.visualize import view
JASPRC['queue.walltime'] = '24:00:00'

# Define the atoms object of interest
atoms = FaceCenteredCubic('Pd',
                          directions=[[0, 1, 1],
                                      [1, 0, 1],
                                      [1, 1, 0]],
                          latticeconstant=3.939)

# Always a good idea to visualize your unit cell before starting
#view(atoms)

# We will sample a large range of k-points
kpts = np.linspace(6, 30, 13)

nrg, t, ibz = [], [], []
ready = True
```

```
for k in kpts:

    with jasp('DFT/structure=fcc/convergence=kpoints/kpoints={0}'.format(int(k)),
              xc='PBE',
              kpts=(int(k), int(k), int(k)),
              encut=400, # Choose a relatively large value
              ibrion=-1, # Perform a single-point calcuation
              atoms=atoms) as calc:
        try:
            atoms = calc.get_atoms()
            nrg += [atoms.get_potential_energy()]
            t += [calc.get_elapsed_time() / 60.0]
            ibz += [len(calc.read_ibz_kpoints())]
        except(VaspQueued, VaspSubmitted):
            ready = False

if ready:
    # Take all energies in reference to the last
    nrg = np.array(nrg) - nrg[-1]

    fig = plt.figure(figsize=(6, 4))
    ax1 = fig.add_subplot(111)
    ax1.plot(kpts, nrg, 'bo-')

    tol = 0.001
    ax1.plot([kpts.min(), kpts.max()], [tol, tol], 'k--')
    ax1.plot([kpts.min(), kpts.max()], [-tol, -tol], 'k--')

    ax1.set_xlim(kpts.min(), kpts.max())
    ax1.set_ylabel('Relative potential energy (eV)', color='b')
    ax1.tick_params(axis='y', colors='b')

    ax2 = ax1.twinx()

    ax2.plot(kpts, t, 'ro-')
    ax2.set_ylabel('Calculation time (min)', color='r')
    ax2.set_xlim(kpts.min(), kpts.max())
    ax2.tick_params(axis='y', colors='r')
    ax2.set_ylim(0, 160)

    ax3 = ax1.twiny()
```

```
64
65      ax3.set_xticks([0./ 24, 4./24, 8./24, 12./24, 16./24, 20./24, 24./24])
66      ax3.set_xticklabels([ibz[0], ibz[2], ibz[4], ibz[6], ibz[8], ibz[10], ibz[12],])
67      ax3.set_xlabel('IBZ $k$-points (total)')
68
69      ax1.set_xlabel('Monkhorst-pack grid $k$-point density (k, k, k)')
70      plt.tight_layout()
71      plt.savefig('images/conv-kpt.png')
```

3.2 encut convergence

Next, we look at energy cutoff convergence. Similarly, k-point density is fixed at (16, 16, 16) for these calculations to ensure no effects from lack of convergence.

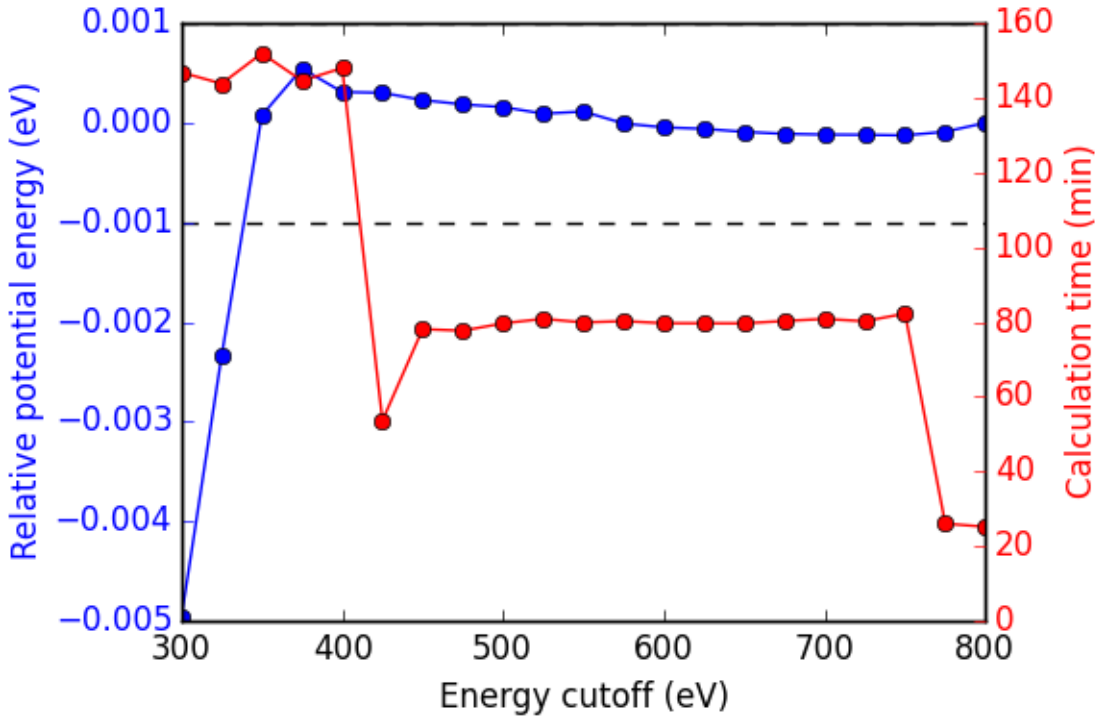


Figure 7: Energy cutoff convergence metrics for a single atom unit cell of fcc Pd.

In this case, Figure 7 shows 350 eV energy cutoff is sufficient to achieve 1 meV convergence. Interestingly, the timing information suggests that 450 eV may be a better choice, or higher, but this is difficult to determine with a single run.

```
from ase.lattice.cubic import FaceCenteredCubic
import numpy as np
from jasp import *
import matplotlib.pyplot as plt
from ase.visualize import view
JASPRC['queue.walltime'] = '24:00:00'

# Define the atoms object of interest
atoms = FaceCenteredCubic('Pd',
                          directions=[[0, 1, 1],
                                      [1, 0, 1],
                                      [1, 1, 0]],
                          latticeconstant=3.939)

# Always a good idea to visualize your unit cell before starting
#view(atoms)

# We will sample a large range of encut
encut = np.linspace(300, 800, (800-300)/23)

nrg, t = [], []
ready = True
for k in encut:

    with jasp('DFT/structure=fcc/convergence=encut/encut={0}'.format(int(k)),
              xc='PBE',
              kpts=(16, 16, 16), # Choose a relatively large value
              encut=k,
              ibrion=-1, # Perform a single-point calcuation
              atoms=atoms) as calc:
        try:
            atoms = calc.get_atoms()
            nrg += [atoms.get_potential_energy()]
            t += [calc.get_elapsed_time() / 60.0]
        except(VaspQueued, VaspSubmitted):
            ready = False

if ready:
    # Take all energies in reference to the last
    nrg = np.array(nrg) - nrg[-1]
```

```
41
42      fig = plt.figure(figsize=(6, 4))
43      ax1 = fig.add_subplot(111)
44      ax1.plot(encut, nrg, 'bo-')
45
46      tol = 0.001
47      ax1.plot([encut.min(), encut.max()], [tol, tol], 'k--')
48      ax1.plot([encut.min(), encut.max()], [-tol, -tol], 'k--')
49
50      ax1.set_xlim(encut.min(), encut.max())
51      ax1.set_ylabel('Relative potential energy (eV)', color='b')
52      ax1.tick_params(axis='y', colors='b')
53
54      ax2 = ax1.twinx()
55
56      #ax2.plot(encut, t, 'ro-')
57      ax2.barv(encut, t, facecolor='r', alpha=0.25)
58      ax2.set_ylabel('Calculation time (min)', color='r')
59      ax2.set_xlim(encut.min(), encut.max())
60      ax2.tick_params(axis='y', colors='r')
61      ax2.set_ylim(0, 160)
62
63      ax1.set_xlabel('Energy cutoff (eV)')
64      plt.tight_layout()
65      plt.savefig('./images/conv-encut.png')
```

3.3 ediff convergence

Finally, we look at the effects of electronic convergence criteria on total energy convergence. For this study, k-points are fixed at (16, 16, 16) and encut at 400 eV.

Interestingly, Figure 8 shows that values less than 5e-3 eV (or 5 meV) have no effect on the convergence of the total energy. The calculation times suggest that the default of 1e-4 eV is a good choice.

```
1  from ase.lattice.cubic import FaceCenteredCubic
2  import numpy as np
3  from jasp import *
```

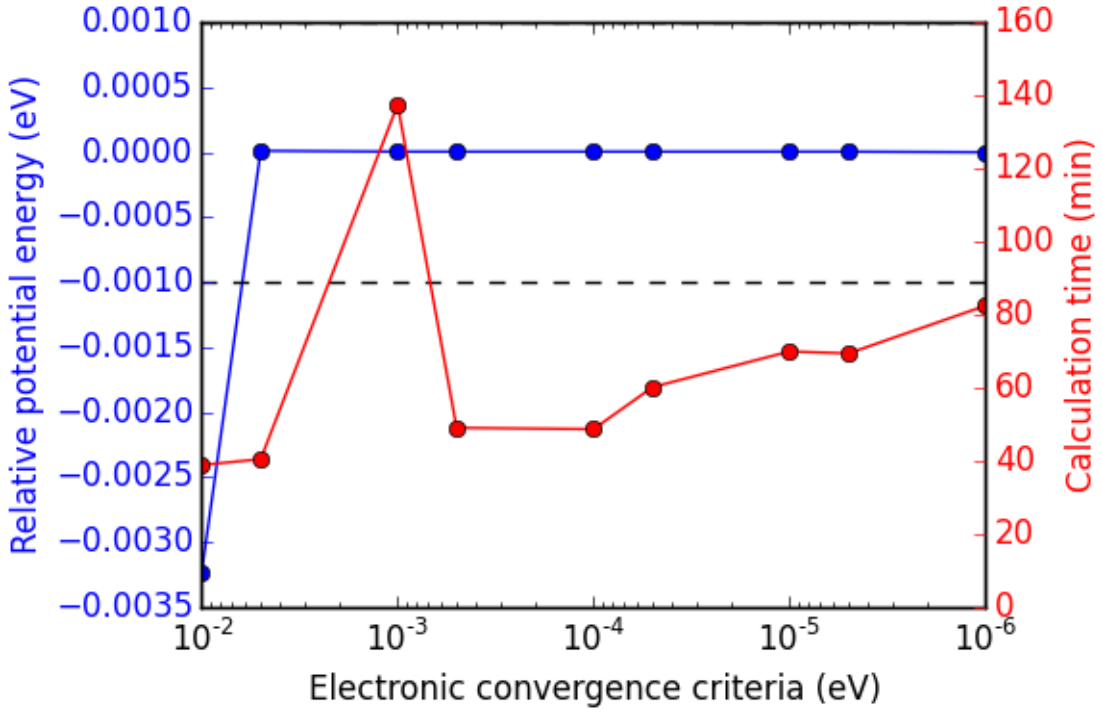


Figure 8: Electronic convergence criteria (ediff) convergence metrics for a single atom unit cell of fcc Pd.

```
4  import matplotlib.pyplot as plt
5  from ase.visualize import view
6  JASPRC['queue.walltime'] = '24:00:00'
7
8  # Define the atoms object of interest
9  atoms = FaceCenteredCubic('Pd',
10                           directions=[[0, 1, 1],
11                                       [1, 0, 1],
12                                       [1, 1, 0]],
13                           latticeconstant=3.939)
14
15 # Always a good idea to visualize your unit cell before starting
16 #view(atoms)
17
18 # We will sample a small range of ediff
19 ediff = np.array([1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5, 5e-6, 1e-6])
20
21 nrg, t = [], []
22 ready = True
```

```
for k in ediff:

    with jasp('DFT/structure=fcc/convergence=ediff/ediff={0:1.6f}'.format(k),
              xc='PBE',
              kpts=(16, 16, 16), # Choose a relatively large value
              encut=400,
              ediff=k,
              ibrion=-1, # Perform a single-point calcuation
              atoms=atoms) as calc:
        try:
            atoms = calc.get_atoms()
            nrg += [atoms.get_potential_energy()]
            t += [calc.get_elapsed_time() / 60.0]
        except(VaspQueued, VaspSubmitted):
            ready = False

if ready:
    # Take all energies in reference to the last
    nrg = np.array(nrg) - nrg[-1]

    fig = plt.figure(figsize=(6, 4))
    ax1 = fig.add_subplot(111)
    ax1.semilogx(ediff, nrg, 'bo-')

    tol = 0.001
    ax1.plot([ediff.min(), ediff.max()], [tol, tol], 'k--')
    ax1.plot([ediff.min(), ediff.max()], [-tol, -tol], 'k--')

    ax1.set_xlim(ediff.min(), ediff.max())
    ax1.set_ylabel('Relative potential energy (eV)', color='b')
    ax1.tick_params(axis='y', colors='b')
    ax1.invert_xaxis()

    ax2 = ax1.twinx()

    ax2.semilogx(ediff, t, 'ro-')
    ax2.set_ylabel('Calculation time (min)', color='r')
    ax2.set_xlim(ediff.min(), ediff.max())
    ax2.tick_params(axis='y', colors='r')
    ax2.invert_xaxis()
    ax2.set_ylim(0, 160)
```

```
64
65      ax1.set_xlabel('Electronic convergence criteria (eV)')
66      plt.tight_layout()
67      plt.savefig('./images/conv-ediff.png')
```

4 Equation of state

Next we use the convergence criteria to calculate Pd bulk fcc EOS at the desired level of accuracy. I have chosen (14, 14, 14) k-points, 400 eV encut, and 1e-4 eV ediff (default setting). We will need a good sized sample to fit the neural network. I have chosen a fine grid of 71 points about the expected minimum in energy, and 29 additional points to span the space leading to "infinite" separation. Figure 9 shows the resulting fit. The code block also generates an ASE database, which we will use from this point on for easy access to the data. It is attached here: (double-click to open).

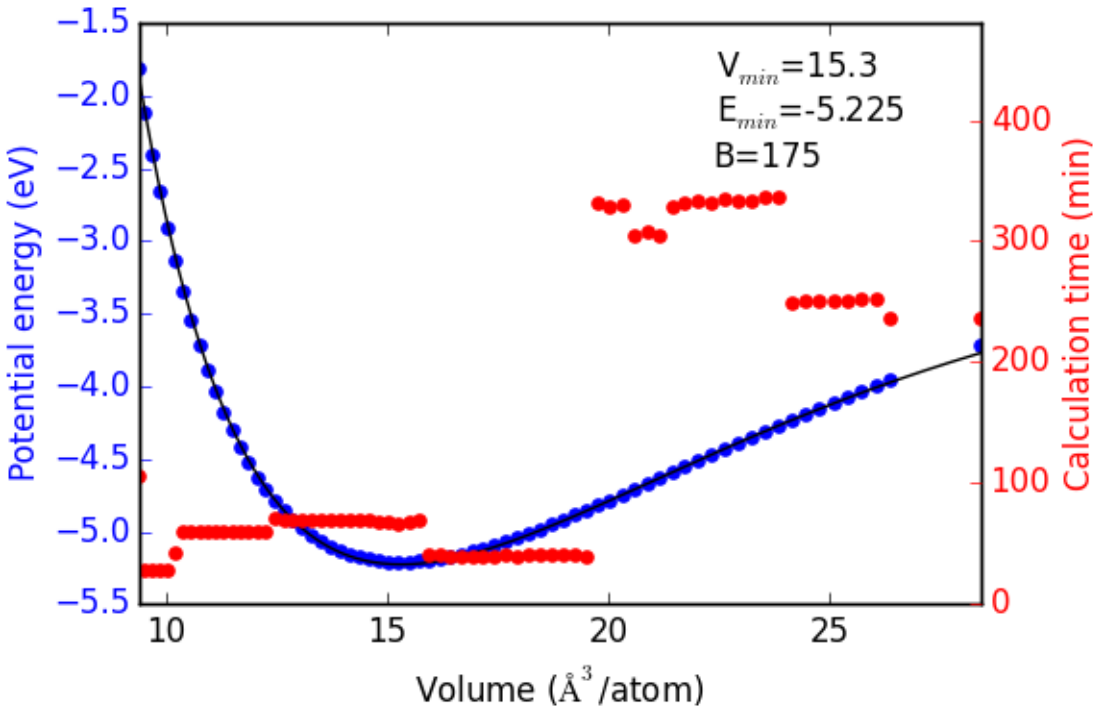


Figure 9: Equation of state for fcc Pd as calculated from DFT.

```
from ase.lattice.cubic import FaceCenteredCubic
import numpy as np
from jasp import *
import matplotlib.pyplot as plt
from ase.utils.eos import EquationOfState
from ase.units import kJ
JASPRC['queue.walltime'] = '24:00:00'

# Functions produced by Jacob Boes for work in computational catalysis
# These are freely available at: https://github.com/jboes/jbtools.git
import jbtools.gilgamesh as jb

# Fraction of equilibrium lattice constant to be calculated
factor = np.append(np.linspace(0.85, 1.2, 71),
                   np.linspace(1.23, 2.07, 29))

nrg, vol, t = [], [], []
ready = True
for x in factor:

    atoms = FaceCenteredCubic('Pd',
                              directions=[[0, 1, 1],
                                          [1, 0, 1],
                                          [1, 1, 0]],
                              latticeconstant=3.939)

    delta = np.array([[x, 0., 0.],
                      [0., x, 0.],
                      [0., 0., x]])
    atoms.set_cell(np.dot(atoms.get_cell(), delta),
                   scale_atoms=True)

    with jasp('DFT/structure=fcc/convergence=None/factor={0:1.3f}'.format(x),
              xc='PBE',
              kpts=(14, 14, 14), # Choose an appropriate value
              encut=400,
              ibrion=-1,
              atoms=atoms) as calc:
        try:
            atoms = calc.get_atoms()
```

```
            nrg += [atoms.get_potential_energy()]
            vol += [atoms.get_volume()]
            t += [calc.get_elapsed_time() / 60.0]
        except(VaspQueued, VaspSubmitted):
            ready = False

if ready:
    # Here we collect the data to an ASE database
    # for easy future manipulation
    jb.write_database('DFT/structure=fcc/convergence=None/')

    # We will use only the energies \pm 15 $\AA^{3}$ about
    # the minimum energy for the figure.
    min_nrg = vol[nrg.index(min(nrg))]
    ind = (np.array(vol) > min_nrg - 15) & (np.array(vol) < min_nrg + 15)
    vol = np.array(vol)[ind]
    nrg = np.array(nrg)[ind]
    t = np.array(t)[ind]

    # Fit the data to SJEOS
    eos = EquationOfState(vol, nrg)
    v0, e0, B, fit = eos.fit()

    x = np.linspace(vol.min(), vol.max(), 250)

    fig = plt.figure(figsize=(6, 4))
    ax1 = fig.add_subplot(111)
    ax1.scatter(vol, nrg, color='b')
    ax1.plot(x, fit(x**-(1.0 / 3)), 'k-')

    ax1.set_xlim(vol.min(), vol.max())
    ax1.set_ylabel('Potential energy (eV)', color='b')
    ax1.tick_params(axis='y', colors='b')

    ax1.text(vol.max() - 6, nrg.max(),
             'V$_{0}$={1:1.1f}'.format('{min}', v0),
             va='center', ha='left')
    ax1.text(vol.max() - 6, nrg.max() - 0.3,
             'E$_{0}$={1:1.3f}'.format('{min}', e0),
             va='center', ha='left')
    ax1.text(vol.max() - 6, nrg.max() - 0.6,
```

```
            'B={0:1.0f}'.format(B  / kJ * 1.0e24),
            va='center', ha='left')

    ax2 = ax1.twinx()

    ax2.scatter(vol, t, color='r')
    ax2.set_ylabel('Calculation time (min)', color='r')
    ax2.set_xlim(vol.min(), vol.max())
    ax2.tick_params(axis='y', colors='r')
    ax2.set_ylim(0, 480)

    ax1.set_xlabel('Volume ($\AA^{3}$/atom)')
    plt.tight_layout()
    plt.savefig('./images/eos.png')
```

5 Neural network

To train a neural network we will be using AMP (https://bitbucket.org/andrewpeterson/amp), a software package developed by the Peterson group at Brown University.

Before we begin creating out neural network, we need to separate about 10% of out data into a validation set. This will be useful later, when determining whether over fitting has occurred. There is functionality for this in AMP, but it does not provide with as much control as the following code.

```
from ase.db import connect
import os
import random
import numpy as np

db = connect('data.db')

n = db.count()
n_train = int(round(n * 0.9))

n_ids =  np.array(range(n)) + 1

```

```
# This will sudo-randomly select 10% of the calculations
# Which is useful for reproducing our results.
random.seed(256)
train_samples = random.sample(n_ids, n_train)
valid_samples = set(n_ids) - set(train_samples)

db.update(list(train_samples), train_set=True)
db.update(list(valid_samples), train_set=False)

db0 = connect('train.db')

for d in db.select(['train_set=True']):
    db0.write(d, key_value_pairs=d.key_value_pairs)
```

Now we have sudo-randomly labeled 10% of our calculations for validation, and the rest are waiting to be trained in the new train.db file. This file is also attached:  (double-click to open).

5.1 Training neural networks

For all of out neural networks, we will be using the Behler-Parenello (BP) framework for distinguishing between geometries of atoms. Little to no work is published on how to systematically chose an appropriate number of variables for your BP framework, so we simply use the default settings in AMP for now. However, it is worth mentioning that a single G1 type variable (simplest possible descriptor) could be used to describe the fcc EOS, if that is all we are interested in.

We also need to define a cutoff radius for our system which will determine the maximum distance that the BP framework considers atoms to be interacting. 6 Å is a typical value used in the literature for metals with no appreciable long range interactions, which we will be using here.

Finally, it is also often desirable to have multiple neural networks which are trained to the same level of accuracy, but with different frameworks. These frameworks are determined by the number of nodes and hidden layers used. In general, we want the smallest number

of nodes and layers possible to avoid the possibility of over fitting. However, too small a framework will be too rigid to properly fit complex potential energy surfaces.

These jobs can be run locally:

```
from amp import Amp
from amp.descriptor import *
from amp.regression import *
import os

for n in [2, 3]:
    label = '{0}-{0}'.format(n)
    wd = os.path.join(os.getcwd(), 'networks/' + label)

    if not os.path.exists(wd):
        os.makedirs(wd)

    calc = Amp(label="./networks/{0}/".format(label),
               descriptor=Behler(cutoff=6.0),
               regression=NeuralNetwork(hiddenlayers=(2, '{0}'.format(n))))

    calc.train("./train.db", # The training data
               cores=1,
               global_search=None, # not found the simulated annealing feature useful
               extend_variables=False) # feature does not work properly and will crash
```

We can also submit them to the queue on Gilgamesh:

```
import os
import subprocess
import time

home = os.getcwd()

# We will try an iteration for 2 and 3 nodes with 2 hidden layers.
for n in [2, 3]:

    label = '{0}-{0}'.format(n)
    wd = os.path.join(home, 'networks/' + label)

```

```
    if not os.path.exists(wd):
        os.makedirs(wd)
    os.chdir(wd)

    run_amp = '''#!/usr/bin/env python
from amp import Amp
from amp.descriptor import *
from amp.regression import *

calc = Amp(label="./",
           descriptor=Behler(cutoff=6.0),
           regression=NeuralNetwork(hiddenlayers=(2, {0})))

calc.train("../../train.db", # The training data
           cores=1,
           global_search=None, # not found the simulated annealing feature useful
           extend_variables=False) # feature does not work properly and will crash
'''.format(n)

    cmd = '''#!/bin/bash
#PBS -N {0}
#PBS -l nodes=1:ppn=1
#PBS -l walltime=24:00:00
#PBS -l mem=2GB
#PBS -joe
cd $PBS_O_WORKDIR
./submit.py
#end'''.format(wd)

    with open('submit.py', 'w') as f:
        f.write(run_amp)
    os.chmod('submit.py', 0777)

    with open('submit.sh', 'w') as f:
        f.write(cmd)

    subprocess.call(['qsub', 'submit.sh'])
    time.sleep(5)
    os.unlink('submit.sh')
    os.chdir(wd)
```

```
1305344.gilgamesh.cheme.cmu.edu
1305345.gilgamesh.cheme.cmu.edu
```

Once the calculations finish we can check their convergence using the code below. These are trivial networks to train, so convergence should not be an issue. If there is a problem, restart the calculation to try again. This can be a difficult and time consuming part of the process for more complex system.

```
import os
import json

print('|Hidden layers|Iteration|Time|Cost Function|Energy RMSE|Force RMSE|')
print('|-')

for r, d, f in os.walk('networks'):
    if 'train-log.txt' in f:
        with open(os.path.join(r, 'train-log.txt'), 'r') as fi:
            v = fi.readlines()[-3].split()

    if 'trained-parameters.json' in f:
        with open(os.path.join(r, 'trained-parameters.json'), 'r') as fi:
            p = json.load(fi)
        n = p['hiddenlayers']
        print('|{0}|{1}|{2}|{3}|{4}|{5}|'.format(n, v[0], v[1], v[2], v[3], v[4]))
```

Hidden layers	Iteration	Time	Cost Function	Energy RMSE	Force RMSE
{u'Pd': [2, 2]}	497	2015-11-18T15:59:22	8.921 (-05)	9.956 (-04)	0.000 (+00)
{u'Pd': [2, 3]}	266	2015-11-18T15:59:34	8.967 (-05)	9.982 (-04)	0.000 (+00)

The single atom unit cell enforces perfect symmetry. This results in cancellation of forces on the atom, meaning we will not be able to use our current neural networks for molecular dynamic simulation.

5.2 Validation of the network

Now we need to validate our results to ensure that no over fitting has occurred. First, we will look at the residuals to the training and validation data. Then we will see if the neural networks perform well for their intended purpose. For ease of access, we will add the neural network energy predictions to the database for each structure.

```
from ase.db import connect
from amp import Amp

db = connect('data.db')

calc2 = Amp('networks/2-2/')
calc3 = Amp('networks/3-3/')

for d in db.select():
    atoms = db.get_atoms(d.id)
    atoms.set_calculator(calc2)
    nrg2 = atoms.get_potential_energy()

    atoms.set_calculator(calc3)
    nrg3 = atoms.get_potential_energy()

    db.update(d.id, NN2=nrg2, NN3=nrg3)
```

5.2.1 Analysis of residuals

First we look at the residual errors of all the data in the database for each of our frameworks shown in Figure 10 and 11. For both fits, the validation set has lower RMSE than the training set. This is a good indication that neither has been over fit, which we can also observe for this simple example, since the validation points follow the same trends observed for the training set data. This is also a good example of how adding additional, unnecessary elements to the framework leads to lower overall fitting accuracy for.

```
import numpy as np
import matplotlib.pyplot as plt
```

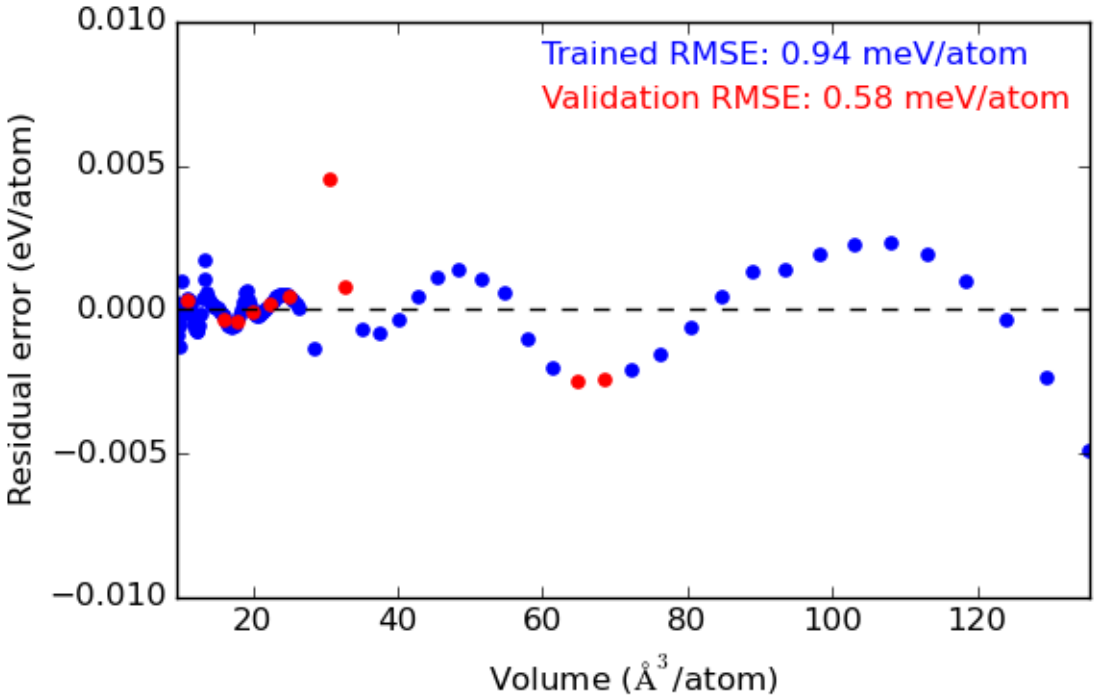


Figure 10: Residual errors to the 2-2 framework neural network.

```
3  from ase.db import connect
4  from amp import Amp
5  import os
6
7  db = connect('data.db')
8
9  for n in [2, 3]:
10
11     Qe, Ne, vol, ind = [], [], [], []
12     for d in db.select():
13
14         Qe += [d.energy]
15         vol += [d.volume]
16
17         Ne += [d['NN{0}'.format(n)]]
18         ind += [d.train_set]
19
20     res = np.array(Ne) - np.array(Qe)
21     mask = np.array(ind)
22     valid = np.ma.masked_array(res, mask)
```

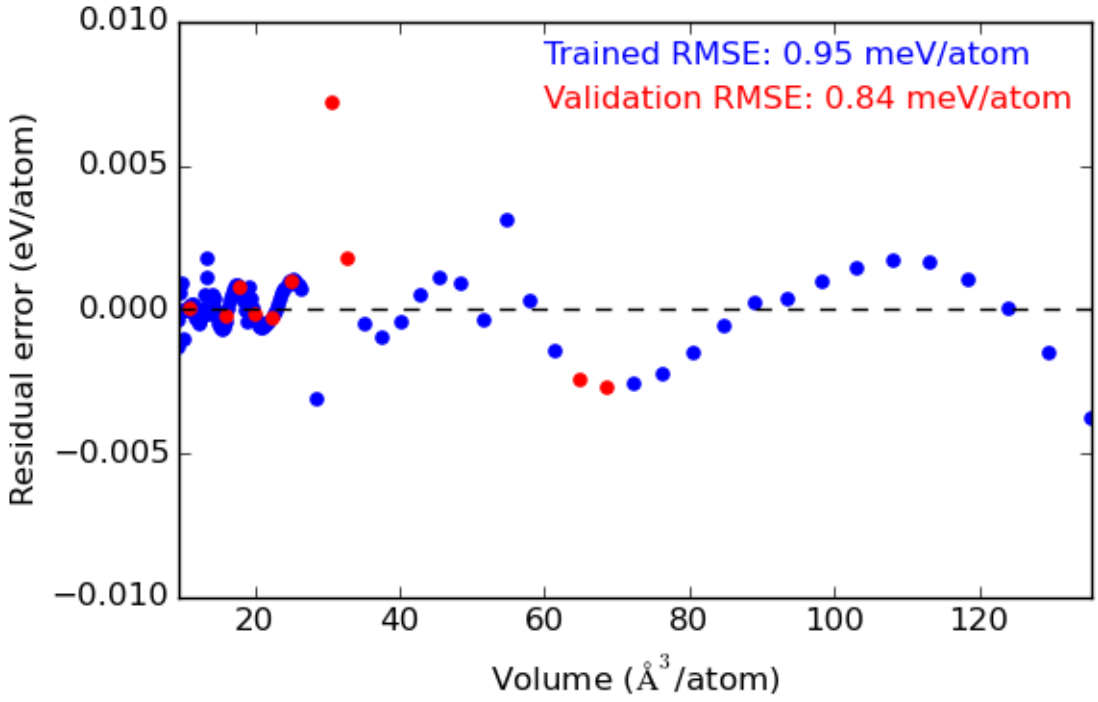


Figure 11: Residual errors to the 3-3 framework neural network.

```
23      train = np.ma.masked_array(res, ~mask)
24      vRMSE = np.sqrt(np.sum(valid ** 2)/ len(valid))
25      tRMSE = np.sqrt(np.sum(train ** 2)/ len(train))
26
27      plt.figure(figsize=(6, 4))
28
29      plt.text(60, 0.0085,
30               'Trained RMSE: {0:1.2f} meV/atom'.format(tRMSE * 1000),
31               color='b', ha='left')
32      plt.text(60, 0.0070,
33               'Validation RMSE: {0:1.2f} meV/atom'.format(vRMSE * 1000),
34               color='r', ha='left')
35
36      plt.scatter(vol, train, color='b')
37      plt.scatter(vol, valid, color='r')
38      plt.plot([min(vol), max(vol)], [0, 0], 'k--')
39      plt.xlim(min(vol), max(vol))
40      plt.ylim(-0.01, 0.01)
41      plt.xlabel('Volume ($\AA^{3}$/atom)')
42      plt.ylabel('Residual error (eV/atom)')
```

```
43      plt.tight_layout()
44      plt.savefig('./images/residuals-NN{0}.png'.format(n))
```

5.2.2 Recreate the equation of state

Next, we recreate the equation of state using both of the neural networks and the same methodology as with DFT. The results are shown in Figures 12 and 13 for the 2-2 and 3-3 frameworks, respectively.

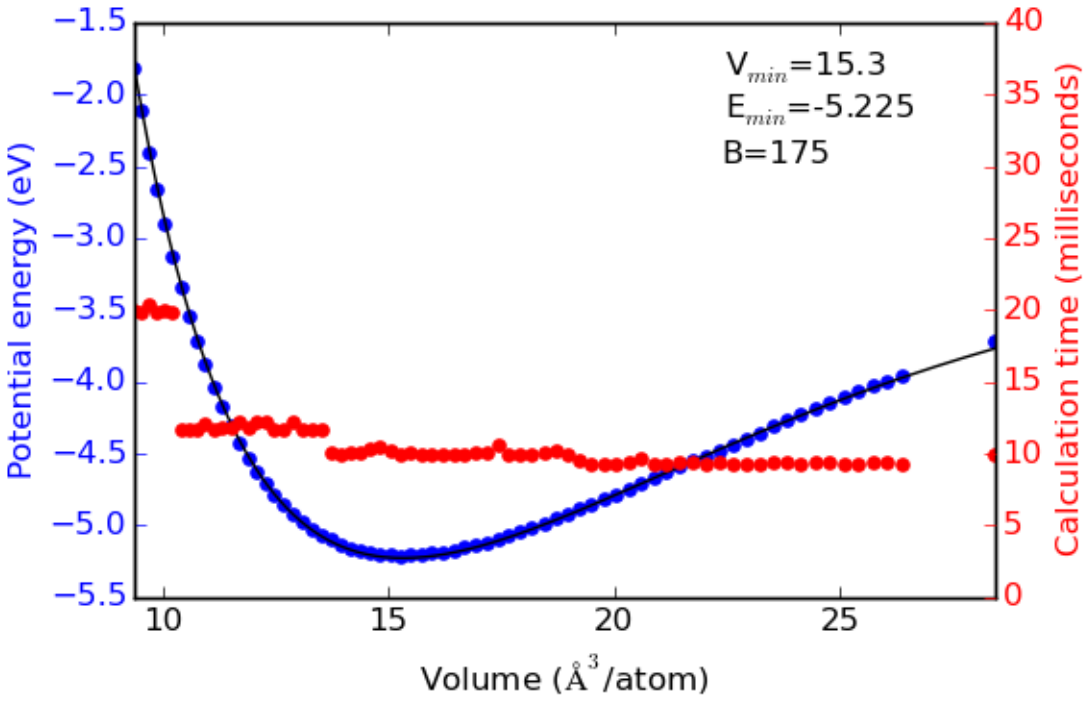


Figure 12: Equation of state for fcc Pd as calculated from a neural network with 2-2 framework.

Each neural network creats are an excellent fit to the DFT data, and we see that the calculation speed has improved by up to 6 orders of magnitude in the most extreme cases. For this application the choice of framework seems to have little effect on the equation of state produced.

```
1   import numpy as np
2   import matplotlib.pyplot as plt
```

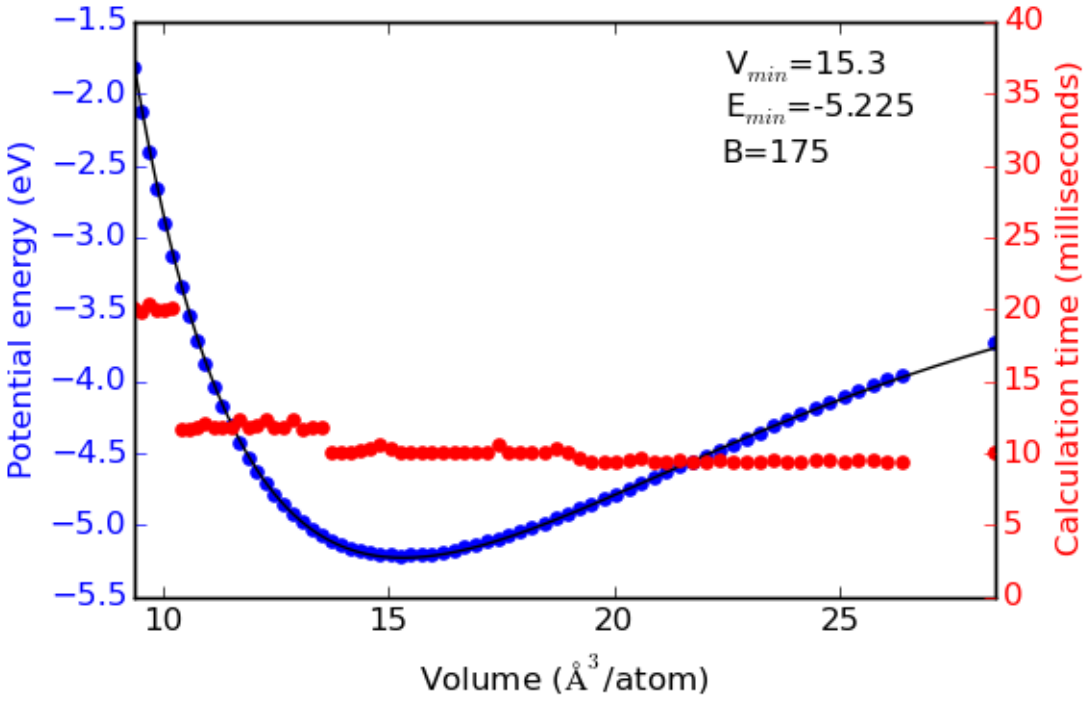


Figure 13: Equation of state for fcc Pd as calculated from a neural network with 3-3 framework.

```
from ase.utils.eos import EquationOfState
from ase.db import connect
from amp import Amp
from ase.visualize import view
import os
import json
import time
from ase.units import kJ

db = connect('data.db')

for r, d, f in os.walk('networks'):
    if 'trained-parameters.json' in f:
        calc = Amp(load=r + '/')

        with open(os.path.join(r, 'trained-parameters.json'), 'r') as fi:
            p = json.load(fi)
        n = p['hiddenlayers'].values()[0]

```

```
nrg, vol, t = [], [], []
for d in db.select():
    atoms = db.get_atoms(d.id)
    atoms.set_calculator(calc)

    time1 = time.time()
    energy = atoms.get_potential_energy()
    time2 = time.time()

    nrg += [energy]
    vol += [d.volume]
    t += [(time2 - time1) * 1000]

min_nrg = vol[nrg.index(min(nrg))]
ind = (np.array(vol) > min_nrg - 15) & (np.array(vol) < min_nrg + 15)
vol = np.array(vol)[ind]
nrg = np.array(nrg)[ind]
t = np.array(t)[ind]

# Fit the data to SJEOS
eos = EquationOfState(vol, nrg)
v0, e0, B, fit = eos.fit()

x = np.linspace(vol.min(), vol.max(), 250)

fig = plt.figure(figsize=(6, 4))
ax1 = fig.add_subplot(111)
ax1.scatter(vol, nrg, color='b')
ax1.plot(x, fit(x**-(1.0 / 3)), 'k-')

ax1.set_xlim(vol.min(), vol.max())
ax1.set_ylabel('Potential energy (eV)', color='b')
ax1.tick_params(axis='y', colors='b')

ax1.text(vol.max() - 6, nrg.max(),
         'V$_{0}$={1:1.1f}'.format('{min}', v0),
         va='center', ha='left')
ax1.text(vol.max() - 6, nrg.max() - 0.3,
         'E$_{0}$={1:1.3f}'.format('{min}', e0),
         va='center', ha='left')
ax1.text(vol.max() - 6, nrg.max() - 0.6,
```

```
                'B={0:1.0f}'.format(B  / kJ * 1.0e24),
                va='center', ha='left')

    ax2 = ax1.twinx()

    ax2.scatter(vol, t, color='r')
    ax2.set_ylabel('Calculation time (milliseconds)', color='r')
    ax2.set_xlim(vol.min(), vol.max())
    ax2.tick_params(axis='y', colors='r')
    ax2.set_ylim(0, 40)

    ax1.set_xlabel('Volume ($\AA^{3}$/atom)')
    plt.tight_layout()
    plt.savefig('./images/eos-NN{0}.png'.format(n[-1]))
```

5.3 Applications

Now we can try and apply our neural networks to things it was not fit to.

For this, we will use or two neural networks jointly which will save us a good amount of time validating the networks as we begin to extrapolate. This is demonstrated in the next section.

5.3.1 Geometry optimization

First, we expand the region of equation of state to see how well it extrapolates. In Figure 14, we expand the region of the original equation of state beyond the black dashed lines.

At extreme stretch (factor $> 2.07\%$) both neural networks agree because we have trained it nearly to the cutoff radius of 6.0 Å.

As soon as we strain the lattice below the trained region, the network predictions quickly diverge. This indicates that the training set is not useful for predictions in this region.

We performed 1,000 calculations to produce this figure. To have validated all 1,000 points with DFT would be too time consuming. Instead, we rely on disagreement between neural networks with different framework to probe poorly fitted regions.

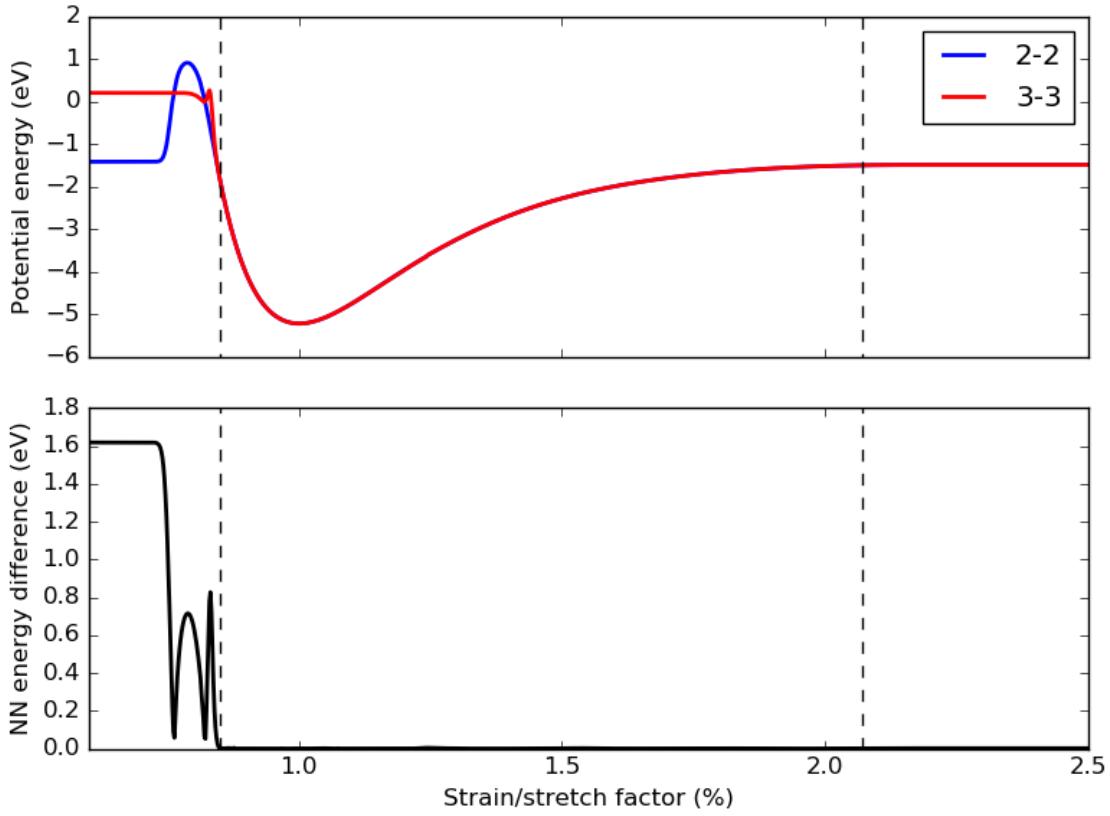


Figure 14: Expansion of the equation of state beyond the region incorporated into the training set.

```
from amp import Amp
import numpy as np
from ase.lattice.cubic import FaceCenteredCubic
import matplotlib.pyplot as plt
import collections

D = {}
for calc in ['networks/2-2/',
             'networks/3-3/']:

    D[calc[-2]] = collections.OrderedDict()
    for x in np.linspace(0.60, 2.5, 1000.):

        atoms = FaceCenteredCubic('Pd',
                                  directions=[[0, 1, 1],
                                              [1, 0, 1],
```

```
                                          [1, 1, 0]],
                                 latticeconstant=3.939)

        delta = np.array([[x, 0., 0.],
                          [0., x, 0.],
                          [0., 0., x]])
        atoms.set_cell(np.dot(atoms.get_cell(), delta),
                       scale_atoms=True)

        atoms.set_calculator(Amp(calc))

        D[calc[-2]][x] = atoms.get_potential_energy()

res = abs(np.array(D['3'].values()) - np.array(D['2'].values()))

f, ax = plt.subplots(2, 1, sharex=True)
ax[0].plot(D['2'].keys(), D['2'].values(), 'b', lw=2, label='2-2')
ax[0].plot(D['3'].keys(), D['3'].values(), 'r', lw=2, label='3-3')
ax[0].plot([0.85, 0.85], [2, -6], 'k--')
ax[0].plot([2.07, 2.07], [2, -6], 'k--')
ax[0].set_ylabel('Potential energy (eV)')
ax[0].set_xlim(0.6, 2.5)
ax[0].legend(loc='best')

ax[1].plot([0.85, 0.85], [0, 1.8], 'k--')
ax[1].plot([2.07, 2.07], [0, 1.8], 'k--')
ax[1].plot(D['2'].keys(), res, 'k', lw=2)
ax[1].set_ylabel('NN energy difference (eV)')
ax[1].set_ylim(0, 1.8)
ax[1].set_xlabel('Strain/stretch factor (%)')
plt.tight_layout(w_pad=0.0)
plt.savefig('./images/app-eos.png')
```

5.3.2 More complex calculations

Here we attempt to calculate the vacancy formation energy for fcc Pd. This is calculated as shown in Equation 1.

$$E_v = E_f - \frac{n_i - 1}{n_i} E_i \tag{1}$$

from the literature (http://www.cs.sandia.gov/~aematts/pdffiles/PRB66214110.pdf), we know that DFT-GGA should predict a vacancy formation energy of about 1.50 eV.

- Vacancy formation energy with 2-2 framework NN: 4.170 eV
- Vacancy formation energy with 3-3 framework NN: 0.411 eV

neither network does a good job predicting the vacancy formation energy. This is because the networks do not know how to calculate the energy of an fcc lattice with a missing atom.

```
from amp import Amp
import numpy as np
from ase.lattice.cubic import FaceCenteredCubic
import matplotlib.pyplot as plt
from ase.visualize import view
from ase.optimize import BFGS

for calc in ['networks/2-2/',
             'networks/3-3/']:
    atoms = FaceCenteredCubic('Pd',
                              directions=[[0, 1, 1],
                                          [1, 0, 1],
                                          [1, 1, 0]],
                              latticeconstant=3.939)
    atoms.set_calculator(Amp(calc))
    atoms *= (3, 3, 3)

    nrg0 = atoms.get_potential_energy()

    del atoms[0]
    dyn = BFGS(atoms)
    dyn.run(fmax=0.05)

    nrg1 = atoms.get_potential_energy()
```

```
    fw = calc.split('/')[-2]
    ve = nrg1 - (26/27.)*nrg0

    print 'Vacancy formation energy with {0} framework NN: {1:1.3f} eV'.format(fw, ve)
```

```
BFGS:    0  23:22:14     -131.319326       0.6402
BFGS:    1  23:22:19     -131.359079       0.4868
BFGS:    2  23:22:24     -131.407465       0.1290
BFGS:    3  23:22:29     -131.408432       0.1178
BFGS:    4  23:22:34     -131.409158       0.0511
BFGS:    5  23:22:39     -131.406953       0.0411
Vacancy formation energy with 2-2 framework NN: 4.170 eV
BFGS:    0  23:22:44     -135.182786       0.0558
BFGS:    1  23:22:49     -135.182910       0.0554
BFGS:    2  23:22:54     -135.183387       0.0320
Vacancy formation energy with 3-3 framework NN: 0.411 eV
```

5.3.3 Molecular dynamics

Finally, we try an MD simulation.

```
from __future__ import print_function
from ase.lattice.cubic import FaceCenteredCubic
from ase.md.langevin import Langevin
from ase.io.trajectory import Trajectory
from ase import units
from amp import Amp

# Set up a crystal
atoms = FaceCenteredCubic('Pd',
                          directions=[[0, 1, 1],
                                      [1, 0, 1],
                                      [1, 1, 0]],
                          latticeconstant=3.939,
```

```
                          size=(3, 3, 3))

# Describe the interatomic interactions with the Effective Medium Theory
atoms.set_calculator(Amp('networks/2-2/'))

# We want to run MD with constant energy using the Langevin algorithm
# with a time step of 5 fs, the temperature T and the friction
# coefficient to 0.02 atomic units.
dyn = Langevin(atoms, 5 * units.fs, 900 * units.kB, 0.002)


def printenergy(a=atoms):  # store a reference to atoms in the definition.
    """Function to print the potential, kinetic and total energy."""
    epot = a.get_potential_energy() / len(a)
    ekin = a.get_kinetic_energy() / len(a)

dyn.attach(printenergy, interval=10)

# We also want to save the positions of all atoms after every time step.
traj = Trajectory('MD.traj', 'w', atoms)
dyn.attach(traj.write, interval=10)

# Now run the dynamics
dyn.run(2000)
```

```
from ase.io.trajectory import Trajectory
from amp import Amp
import matplotlib.pyplot as plt
from ase.visualize import view

traj = Trajectory('MD.traj', 'r')


E2, E3 = [], []
for atoms in traj:
   E2 += [atoms.get_potential_energy()]

   atoms.set_calculator(Amp('networks/3-3/'))
   E3 += [atoms.get_potential_energy()]

# We can use this to visualize every 10th step of the MD simulation.
```

```
17  view(traj)
18
19  #plt.figure(figsize=(6, 4))
20  #plt.plot(range(len(E2)), E2)
21  #plt.plot(range(len(E3)), E3)
22  #plt.show()
```