

Contents

1	Convergence calculations	1
1.1	k-point convergence	1
1.2	encut convergence	3
1.3	ediff convergence	5
2	Equation of state	7
3	Creating a neural network	9
3.1	training neural networks	10

1 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.

1.1 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 1 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
3 from jasp import *
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
```

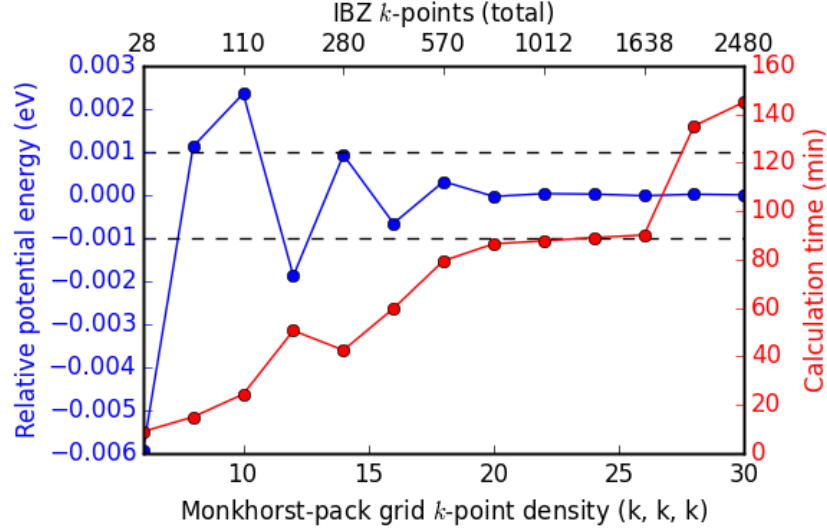


Figure 1: k -point convergence metrics for a single atom unit cell of fcc Pd

```

18 # We will sample a large range of k-points
19 kpts = np.linspace(6, 30, 13)
20
21 nrg, t, ibz = [], [], []
22 ready = True
23 for k in kpts:
24
25     with jasp('DFT/structure=fcc/convergence=kpoints/kpoints={0}'.format(int(k)),
26              xc='PBE',
27              kpts=(int(k), int(k), int(k)),
28              encut=400, # Choose a relatively large value
29              ibrion=-1, # Perform a single-point calculation
30              atoms=atoms) as calc:
31
32         try:
33             atoms = calc.get_atoms()
34             nrg += [atoms.get_potential_energy()]
35             t += [calc.get_elapsed_time() / 60.0]
36             ibz += [len(calc.read_ibz_kpoints())]
37         except (VaspQueued, VaspSubmitted):
38             ready = False
39
40 if ready:
41     # Take all energies in reference to the last
42     nrg = np.array(nrg) - nrg[-1]
43
44 fig = plt.figure(figsize=(6, 4))
45 ax1 = fig.add_subplot(111)
46 ax1.plot(kpts, nrg, 'bo-')

```

```

47     tol = 0.001
48     ax1.plot([kpts.min(), kpts.max()], [tol, tol], 'k--')
49     ax1.plot([kpts.min(), kpts.max()], [-tol, -tol], 'k--')
50
51     ax1.set_xlim(kpts.min(), kpts.max())
52     ax1.set_ylabel('Relative potential energy (eV)', color='b')
53     ax1.tick_params(axis='y', colors='b')
54
55     ax2 = ax1.twinx()
56
57     ax2.plot(kpts, t, 'ro-')
58     ax2.set_ylabel('Calculation time (min)', color='r')
59     ax2.set_xlim(kpts.min(), kpts.max())
60     ax2.tick_params(axis='y', colors='r')
61     ax2.set_ylim(0, 160)
62
63     ax3 = ax1.twinx()
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')

```

1.2 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.

In this case, Figure 2 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.

```

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

```

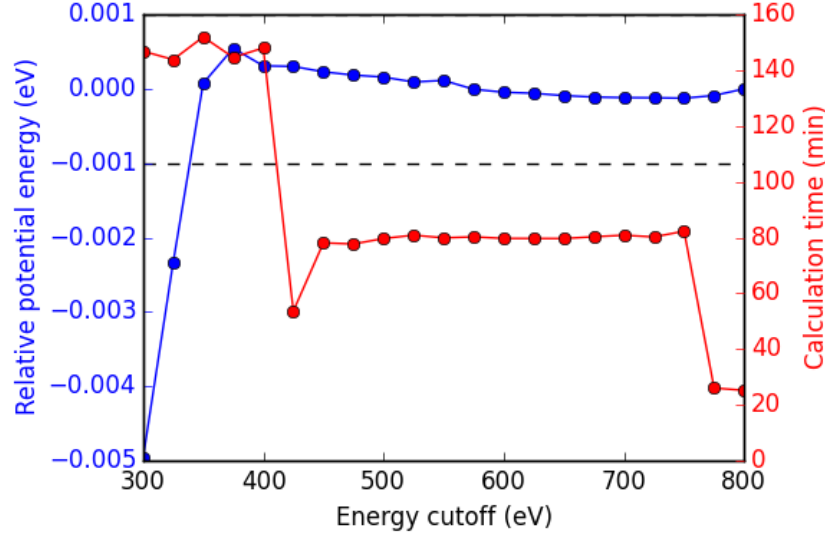


Figure 2: Energy cutoff convergence metrics for a single atom unit cell of fcc Pd.

```

16 #view(atoms)
17
18 # We will sample a large range of encut
19 encut = np.linspace(300, 800, (800-300)/23)
20
21 nrg, t = [], []
22 ready = True
23 for k in encut:
24
25     with jasp('DFT/structure=fcc/convergence=encut/encut={0}'.format(int(k)),
26              xc='PBE',
27              kpts=(16, 16, 16), # Choose a relatively large value
28              encut=k,
29              ibrion=-1, # Perform a single-point calculation
30              atoms=atoms) as calc:
31
32         try:
33             atoms = calc.get_atoms()
34             nrg += [atoms.get_potential_energy()]
35             t += [calc.get_elapsed_time() / 60.0]
36         except(VaspQueued, VaspSubmitted):
37             ready = False
38
39 if ready:
40     # Take all energies in reference to the last
41     nrg = np.array(nrg) - nrg[-1]
42
43     fig = plt.figure(figsize=(6, 4))
44     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.set_ylabel('Calculation time (min)', color='r')
58     ax2.set_xlim(encut.min(), encut.max())
59     ax2.tick_params(axis='y', colors='r')
60     ax2.set_ylim(0, 160)
61
62     ax1.set_xlabel('Energy cutoff (eV)')
63     plt.tight_layout()
64     plt.savefig('./images/conv-encut.png')

```

1.3 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 3 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 *
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 = [], []

```

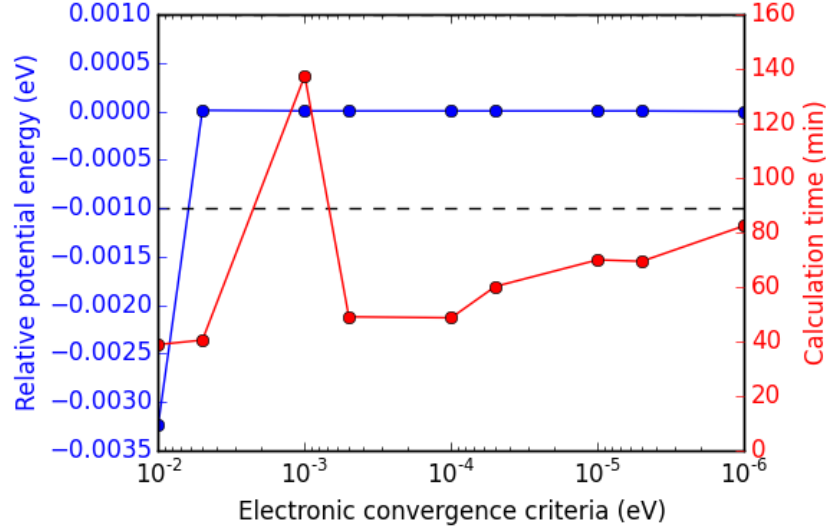


Figure 3: Electronic convergence criteria (ediff) convergence metrics for a single atom unit cell of fcc Pd.

```

22 ready = True
23 for k in ediff:
24
25     with jasp('DFT/structure=fcc/convergence=ediff/ediff={0:1.6f}'.format(k),
26              xc='PBE',
27              kpts=(16, 16, 16), # Choose a relatively large value
28              encut=400,
29              ediff=k,
30              ibrion=-1, # Perform a single-point calculation
31              atoms=atoms) as calc:
32
33         try:
34             atoms = calc.get_atoms()
35             nrg += [atoms.get_potential_energy()]
36             t += [calc.get_elapsed_time() / 60.0]
37         except (VaspQueued, VaspSubmitted):
38             ready = False
39
40 if ready:
41     # Take all energies in reference to the last
42     nrg = np.array(nrg) - nrg[-1]
43
44     fig = plt.figure(figsize=(6, 4))
45     ax1 = fig.add_subplot(111)
46     ax1.semilogx(ediff, nrg, 'bo-')
47
48     tol = 0.001
49     ax1.plot([ediff.min(), ediff.max()], [tol, tol], 'k--')


```

```

49     ax1.plot([ediff.min(), ediff.max()], [-tol, -tol], 'k--')
50
51     ax1.set_xlim(ediff.min(), ediff.max())
52     ax1.set_ylabel('Relative potential energy (eV)', color='b')
53     ax1.tick_params(axis='y', colors='b')
54     ax1.invert_xaxis()
55
56     ax2 = ax1.twinx()
57
58     ax2.semilogx(ediff, t, 'ro-')
59     ax2.set_ylabel('Calculation time (min)', color='r')
60     ax2.set_xlim(ediff.min(), ediff.max())
61     ax2.tick_params(axis='y', colors='r')
62     ax2.invert_xaxis()
63     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')

```

2 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 4 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).

```

1  from ase.lattice.cubic import FaceCenteredCubic
2  import numpy as np
3  from jasp import *
4  import matplotlib.pyplot as plt
5  import os
6  from ase.utils.eos import EquationOfState
7  JASPRC['queue.walltime'] = '24:00:00'
8
9  # Functions produced by Jacob Boes for work in computational catalysis
10 # These are freely available at: https://github.com/jboes/jbtools.git
11 import jbtools.gilgamesh as jb
12
13 if os.path.exists('data.db'):
14     os.unlink('data.db')
15
16 # Fraction of equilibrium lattice constant to be calculated
17 factor = np.append(np.linspace(0.85, 1.2, 71),
18                    np.linspace(1.23, 2.07, 29))

```

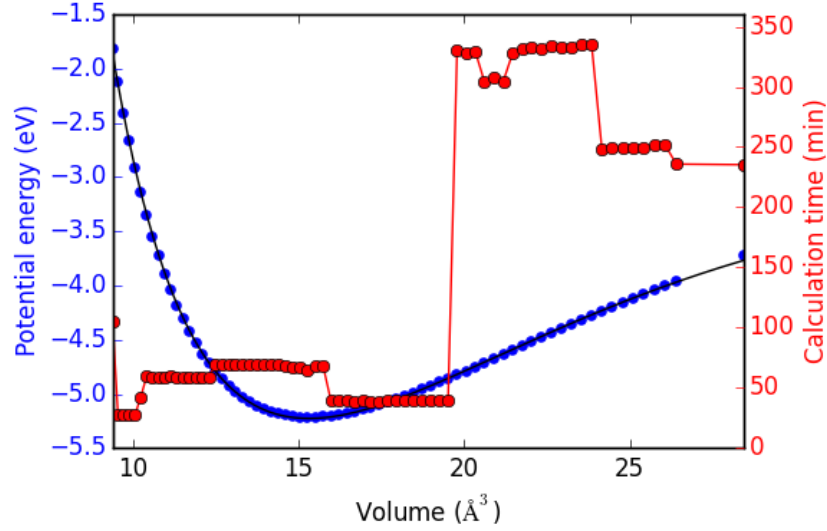


Figure 4: Electronic convergence criteria (ediff) convergence metrics for a single atom unit cell of fcc Pd.

```

19
20 nrg, vol, t = [], [], []
21 ready = True
22 for x in factor:
23
24     atoms = FaceCenteredCubic('Pd',
25                               directions=[[0, 1, 1],
26                                           [1, 0, 1],
27                                           [1, 1, 0]],
28                               latticeconstant=3.939)
29
30     delta = np.array([[x, 0., 0.],
31                       [0., x, 0.],
32                       [0., 0., x]])
33     atoms.set_cell(np.dot(atoms.get_cell(), delta),
34                    scale_atoms=True)
35
36     with jasp('DFT/structure=fcc/convergence=None/factor={0:1.3f}'.format(x),
37              xc='PBE',
38              kpts=(14, 14, 14), # Choose an appropriate value
39              encut=400,
40              ibrion=-1,
41              atoms=atoms) as calc:
42         try:
43             atoms = calc.get_atoms()
44             nrg += [atoms.get_potential_energy()]
45             vol += [atoms.get_volume()]

```



```

46         t += [calc.get_elapsed_time() / 60.0]
47
48         # Here we collect the data to an ASE database
49         # for easy future manipulation
50         jb.makedb(calc, dbname='~/research/amp/data.db')
51     except (VaspQueued, VaspSubmitted):
52         ready = False
53
54 if ready:
55
56     # We will use only the energies \pm 15 $AA^{3}$ about
57     # the minimum energy for the figure.
58     min_nrg = vol[nrg.index(min(nrg))]
59     ind = (np.array(vol) > min_nrg - 15) & (np.array(vol) < min_nrg + 15)
60     vol = np.array(vol)[ind]
61     nrg = np.array(nrg)[ind]
62     t = np.array(t)[ind]
63
64     # Fit the data to SJEOS
65     eos = EquationOfState(vol, nrg)
66     v0, e0, B, fit = eos.fit()
67
68     x = np.linspace(vol.min(), vol.max(), 250)
69
70     fig = plt.figure(figsize=(6, 4))
71     ax1 = fig.add_subplot(111)
72     ax1.scatter(vol, nrg, color='b')
73     ax1.plot(x, fit(x**-(1.0 / 3)), 'k-')
74
75     ax1.set_xlim(vol.min(), vol.max())
76     ax1.set_ylabel('Potential energy (eV)', color='b')
77     ax1.tick_params(axis='y', colors='b')
78
79     ax2 = ax1.twinx()
80
81     ax2.plot(vol, t, 'ro-')
82     ax2.set_ylabel('Calculation time (min)', color='r')
83     ax2.set_xlim(vol.min(), vol.max())
84     ax2.tick_params(axis='y', colors='r')
85     ax2.set_ylim(0, 360)
86
87     ax1.set_xlabel('Volume ($AA^{3}$)')
88     plt.tight_layout()
89     plt.savefig('./images/eos.png')

```

3 Creating a 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 our neural network, we need to separate about 10% of our data into a validation set. This will be useful later, when deter-

mining 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.

```

1 from ase.db import connect
2 import os
3 import random
4 import numpy as np
5
6 db = connect('data.db')
7
8 n = db.count()
9 n_train = int(round(n * 0.9))
10
11 n_ids = np.array(range(n)) + 1
12
13 # This will sudo-randomly select 10% of the calculations
14 # Which is useful for reproducing our results.
15 random.seed(256)
16 train_samples = random.sample(n_ids, n_train)
17 valid_samples = set(n_ids) - set(train_samples)
18
19 db.update(list(train_samples), train_set=True)
20 db.update(list(valid_samples), train_set=False)
21
22 db0 = connect('train.db')
23
24 for d in db.select(['train_set=True']):
25     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).

3.1 training neural networks

For all of our 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 choose 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.

At the moment, I run these locally:

```

1  from amp import Amp
2  from amp.descriptor import *
3  from amp.regression import *
4  import os
5
6  n = 2
7  i = 0
8
9  label = 'l2n{0}i{1}'.format(n, i)
10 wd = os.path.join(os.getcwd(), 'networks/' + label)
11
12 if not os.path.exists(wd):
13     os.makedirs(wd)
14
15 calc = Amp(label="./networks/{0}/".format(label),
16            descriptor=Behler(cutoff=6.0),
17            regression=NeuralNetwork(hiddenlayers=(2, 2)))
18
19 calc.train("./train.db", # The training data
20           cores=2,
21           global_search=None, # not found the simulated annealing feature useful
22           extend_variables=False) # feature does not work properly and will crash

```

Once I figure out how to compile the necessary AMP files on Gilgamesh, we can submit these to the queue:

```

1  import os
2  import subprocess
3
4  home = os.getcwd()
5
6  # we will try two iterations of 2, 3, and 4 nodes for 2 hidden layers.
7  for n, i in [[2, 0], [2, 1],
8             [3, 0], [3, 1],
9             [4, 0], [4, 1]]:
10
11     label = 'l2n{0}i{1}'.format(n, i)
12     wd = os.path.join(home, 'networks/' + label)
13
14     if not os.path.exists(wd):
15         os.makedirs(wd)
16     else:

```

```

17         pass
18     os.chdir(wd)
19
20     run_amp = """#!/usr/bin/env python
21 from amp import Amp
22 from amp.descriptor import *
23 from amp.regression import *
24
25 calc = Amp(label="./networks/{0}/",
26             descriptor=Behler(cutoff=6.0),
27             regression=NeuralNetwork(hiddenlayers=(2, {1})))
28
29 calc.train("./train.db", # The training data
30            cores=4,
31            global_search=None, # not found the simulated annealing feature useful
32            extend_variables=False) # feature does not work properly and will crash
33 """ .format(label, n)
34
35     cmd = """#!/bin/bash
36 #PBS -N {0}
37 #PBS -l nodes=1:ppn=4
38 #PBS -l walltime=24:00:00
39 #PBS -l mem=3GB
40 #PBS -joe
41 cd $PBS_O_WORKDIR
42 ./submit.py
43 #end""" .format(wd)
44
45     with open('submit.py', 'w') as f:
46         f.write(run_amp)
47     os.chmod('submit.py', 0777)
48
49     with open('submit.sh', 'w') as f:
50         f.write(cmd)
51
52     subprocess.call(['qsub', 'submit.sh'])
53     os.unlink('submit.sh')

```

```

1305130.gilgamesh.cheme.cmu.edu
1305131.gilgamesh.cheme.cmu.edu
1305132.gilgamesh.cheme.cmu.edu
1305133.gilgamesh.cheme.cmu.edu
1305134.gilgamesh.cheme.cmu.edu
1305135.gilgamesh.cheme.cmu.edu

```