

Supporting Information: Uncertainty Quantification in Machine Learning and Nonlinear Least Squares Regression Models

Ni Zhan and John R. Kitchen

November 12, 2021

1 Appendix for the paper

The delta method is based on regression, and gives a standard error of prediction by linearly approximating the model. We are doing a regression with data $\{x_i, y_i\}$. Our model predicts $y(x_i | \theta)$, and the theory of the delta method assumes that the data output is the sum of the model prediction and some Gaussian error.

$$y_i = y(x_i | \theta) + \epsilon_i$$

with $\epsilon_i \sim N(0, \sigma_i)$, y_i as data output, x_i as data input, and θ as model parameters.

The log likelihood of the data given the model, l_n , is

$$l_n = \log P(\{y_i\} | \theta)$$

Since we assumed ϵ_i was Gaussian,

$$l_n \propto -\frac{1}{2} \sum_i \left(\frac{y_i - y(x_i | \theta)}{\sigma_i} \right)^2$$

The above term includes the sum of squared errors which is common as the loss or regression objective function during training. In least squares regression, we minimize the sum squared errors to get the maximum likelihood estimate of parameters, $\hat{\theta}$.

The standard error of $\hat{\theta}$

$$\text{se}(\hat{\theta}) \approx \frac{1}{\sqrt{I_n(\theta)}}$$

where $I_n(\theta)$ is the Fisher information matrix defined as

$$I_n(\theta) = -\mathbb{E}_\theta \left[\frac{\partial^2 l_n(\{y_i\} | \theta)}{\partial \theta^2} \right]$$

The standard error of $\hat{\theta}$ is obtained from doing a Taylor's series expansion around $l_n''(\theta)$.¹ We are able to obtain this standard error by assuming $\hat{\theta}$ is centered and Gaussian around the true parameters θ .

In the Fisher information, note that l_n is the same log likelihood defined earlier, so the Fisher information is proportional to the Hessian of the loss with respect to model parameters, and thus can be readily obtained.

Now we will obtain the standard error of model prediction. Suppose for function $g(\hat{\theta})$, $g'(\hat{\theta})$ is nonzero, then

$$\text{se}(g(\hat{\theta})) \approx \sqrt{(g')^T I_n^{-1} g'}.$$

The standard error of $g(\hat{\theta})$ is obtained by doing a Taylor's series around $g(\theta)$ and using $\text{se}(\hat{\theta})$ obtained previously.¹

The standard error depends on the training data because the Fisher information depends on the training data. The standard error also depends on the model, its parameters, and the point we are predicting, because these determine g' .

In this work, we assume the error ϵ_i is independent of the data point x_i . This allows the simplification

$$l_n \propto -\frac{1}{2} \sum_i \left(\frac{y_i - y(x_i | \theta)}{\sigma_i} \right)^2 = -\frac{1}{2\sigma^2} \sum_i (y_i - y(x_i | \theta))^2$$

We estimate σ^2 as

$$\sigma^2 \approx \frac{1}{n} \sum_i^n (y_i - y(x_i | \theta))^2$$

Once obtaining standard errors for a prediction, we can construct confidence intervals. We use $t_{\frac{\alpha}{2}} \cdot \text{se}(g(\hat{\theta}))$ for $(1 - \alpha)\%$ confidence intervals. The confidence interval indicates confidence of fit. The prediction standard error has an additional term

$$\text{prediction se}(g(\hat{\theta})) = \sqrt{(g')^T I_n^{-1} g' + \sigma_r^2}$$

where σ_r^2 is residual variance and approximated by

$$\sigma_r^2 \approx \frac{1}{n} \sum_i^n (g_i - g(x_i | \theta))^2$$

A $(1 - \alpha)\%$ prediction interval is then $t_{\frac{\alpha}{2}} \cdot (\text{pred. se}(g(\hat{\theta})))$. The prediction interval represents how often a new point would fall in the interval.

2 One dimension input NN (Figure 2)

```

1 import autograd
2 import autograd.numpy as np
3 from autograd import hessian
4 import matplotlib.pyplot as plt
5 from autograd import grad
6 import autograd.numpy.random as npr
7 from scipy.stats.distributions import t
8 from scipy.optimize import minimize
9 from matplotlib.ticker import FormatStrFormatter
10
11 # lennard jones potential
12 def func(x, e, s):
13     return 4 * e * (np.power(np.divide(s, x), 12) -
14                    np.power(np.divide(s, x), 6))
15
16 etrue = 10
17 strue = 0.34
18 numpts = 23
19
20 #xfit is for plotting
21 xfit = np.arange(0.34, 0.49, 0.001)
22 xfit = np.expand_dims(xfit, axis=1)
23
24 # weightsparser to help roll and unroll weights and biases.
25 class WeightsParser(object):
26     """A helper class to index into a parameter vector."""
27
28     def __init__(self):
29         self.idx_and_shapes = {}
30         self.N = 0
31
32     def add_weights(self, name, shape):

```

```

33         start = self.N
34         self.N += np.prod(shape)
35         self.idxs_and_shapes[name] = (slice(start, self.N), shape)
36
37     def get(self, vect, name):
38         idxs, shape = self.idxs_and_shapes[name]
39         return np.reshape(vect[idxs], shape)
40
41     # params is a 1-d vector of weights and biases
42     # parser is object that makes it easy to unroll params into matrices of
43     # weights and biases.
44     def init_random_params(scale, layer_sizes, rs=None):
45         if rs is None:
46             rs = npr.RandomState(2)
47             parser = WeightsParser()
48             for i, shape in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
49                 parser.add_weights(('weights', i), shape)
50                 parser.add_weights(('biases', i), (1, shape[1]))
51             return rs.randn(parser.N), parser
52
53     # nn predict by unrolling w parser.
54     def nn_predict(params, inputs, nonlinearity=np.tanh):
55         cur_units = inputs
56         for layer in range(len(layer_sizes) - 1):
57             cur_W = parser.get(params, ('weights', layer))
58             cur_B = parser.get(params, ('biases', layer))
59             cur_units = np.dot(cur_units, cur_W) + cur_B
60             if layer < len(layer_sizes) - 2:
61                 cur_units = nonlinearity(cur_units)
62         return cur_units
63
64     #objective with regularization to be used with scipy minimize
65     def objective_l2(params, X, r, alpha=0):
66         ypredict = nn_predict(params, X)
67         errs = r - ypredict
68         weights = params[idxs]
69         return np.sum(errs**2) + alpha * np.linalg.norm(weights)
70
71     layer_sizes = [1, 4, 1]
72     _, parser = init_random_params(1, layer_sizes)
73
74     # get the index of the weights, because only regularizing weights.
75     idxs = []
76     for layer in range(len(layer_sizes) - 1):
77         sliceidx, _ = parser.idxs_and_shapes[('weights', layer)]
78         idxs += [np.r_[sliceidx]]
79     idxs = np.array(idxs).flatten()
80
81     #sum-squared-errors
82     def sse(params, X, r):
83         ypredict = nn_predict(params, X)
84         errs = r - ypredict
85         return np.sum(errs**2)
86
87     #get inverse fisher information
88     def get_pcov(h):
89         eigs0 = np.linalg.eigvalsh(h)[0]
90         if (eigs0 < 0):
91             eps = max(1e-5, eigs0*-1.05)
92         else:
93             eps = 1e-5
94         j = np.linalg.pinv(h + eps * np.identity(h.shape[0]))
95         pcov1 = j * scaling
96         u, v = np.linalg.eigh(pcov1)
97         return v @ np.diag(np.maximum(u, 0)) @ v.T
98
99     #get standard errors of prediction, confidence
100    def getpredse(x, params):

```

```

101     gprime = autograd.elementwise_grad(nn_predict,0)(params, x)
102     sesq = gprime @ pcov @ gprime
103     return np.sqrt(sesq), np.sqrt(sesq + scaling)
104
105 #get standard errors for a dataset
106 def get_se_dataset(xfit, params):
107     preds = []
108     for i in xfit:
109         preds += [getpredse(i, params)]
110     return np.array(preds)
111
112 # to make plot
113 # data for panel 1.
114 numpts = 23
115 xa = np.linspace(0.35, 0.45, numpts)
116 np.random.seed(seed=0)
117
118 ya = func(xa, etrue, strue) + np.random.normal(scale=0.2, size=xa.shape)
119
120 Xa = np.expand_dims(xa, axis=1)
121 ra = np.expand_dims(ya, axis=1)
122
123 initial_guess, parser = init_random_params(1, layer_sizes)
124
125 sol = minimize(objective12, initial_guess, args=(Xa,ra,0.01) )
126 paramsa = sol.x
127
128 h = hessian(sse,0)(paramsa, Xa, ra)
129 numptsa = Xa.shape[0]
130 scaling = sse(paramsa, Xa, ra)/numptsa
131
132 pcov = get_pcov(h)
133
134 predsesa = get_se_dataset(xfit, paramsa)
135
136 #data for panel 2.
137 x1 = np.linspace(0.35, 0.365, 7)
138 x2 = np.linspace(0.415, 0.45, 9)
139 xb = np.concatenate((x1,x2))
140 yb = func(xb, etrue, strue) + np.random.normal(scale=0.2, size=xb.shape)
141
142 Xb = np.expand_dims(xb, axis=1)
143 rb = np.expand_dims(yb, axis=1)
144
145 initial_guess, _ = init_random_params(1, layer_sizes)
146
147 sol = minimize(objective12, initial_guess, args=(Xb,rb,0.005) )
148 paramsb = sol.x
149
150 h = hessian(sse,0)(paramsb, Xb, rb)
151 numptsb = Xb.shape[0]
152 scaling = sse(paramsb, Xb, rb)/numptsb
153
154 pcov = get_pcov(h)
155
156 predseseb = get_se_dataset(xfit, paramsb)
157
158
159 #make a plot.
160
161 plt.clf()
162 fig, ax = plt.subplots(ncols=1, nrows=2, sharex=True, sharey=True)
163 fig.set_size_inches(3.25,5)
164 tvala = t.ppf(0.975, numptsa)
165 tvalb = t.ppf(0.975, numptsb)
166
167 ypreda = nn_predict(paramsa, xfit).flatten()
168 ypredb = nn_predict(paramsb, xfit).flatten()

```

```

169
170 #ax.yaxis.set_major_formatter(FormatStrFormatter('%0.2f'))
171
172 #ax[0].set_title(' ')
173 ax[0].plot(Xa, ra, 'bo')
174 ax[0].plot(xfit, ypreda)
175 ax[0].plot(xfit, func(xfit, etrue, strue))
176 ax[0].plot(xfit, ypreda + predsesa[:,0] * tvala, '--r')
177 ax[0].plot(xfit, ypreda - predsesa[:,0] * tvala, '--r')
178 #ax[0].set_xlabel('x')
179 ax[0].set_ylabel('y')
180
181 #ax[1].set_title(' ')
182 ax[1].plot(Xb, rb, 'bo')
183 ax[1].plot(xfit, ypredb)
184 ax[1].plot(xfit, func(xfit, etrue, strue))
185 ax[1].plot(xfit, ypredb + predsesb[:,0] * tvalb, '--r')
186 ax[1].plot(xfit, ypredb - predsesb[:,0] * tvalb, '--r')
187 ax[1].set_xlabel('x')
188 ax[1].set_ylabel('y')
189 #ax[1].yaxis.set_major_formatter(FormatStrFormatter('%0.0f'))
190
191 ax[0].legend(['Data', 'NN', 'f(x)', '95% confidence'])
192
193 plt.figtext(0.05, 0.90, 'A')
194 plt.figtext(0.05, 0.48, 'B')
195 plt.subplots_adjust(wspace=0)
196 plt.tight_layout()
197 plt.subplots_adjust(wspace=0)
198 for ext in ['png', 'eps']:
199     plt.savefig(f'subplot-2panel-ou.{ext}', dpi=300)
200 print(f'#+attr_org: :width 600')
201 #+caption: Figure 2
202 [./subplot-2panel-ou.png]'''

```

3 Training a SingleNN model

The database file used for the first potential contained configurations with 3.934 Å lattice constant. 

The following code uses singleNN code found here: <https://github.com/lmj1029123/SingleNN>, and mostly follows the github tutorial. The code splits the dataset, configures the singleNN, and trains the model. The code generates a directory folder "lattice39-2" with relevant files: splitted dataset files "final_train.sav", "final_val.sav", "test.sav"; model file "best_model".

```

1 import sys
2
3 sys.path.append("../SimpleNN")
4 sys.path.append("../")
5
6 import os
7 from ase.db import connect
8 import torch
9 from ContextManager import cd
10 from preprocess import train_test_split, train_val_split, get_scaling, CV
11 from preprocess import snn2sav
12 from NN import MultiLayerNet
13 from train import train, evaluate
14 from fp_calculator import set_sym, calculate_fp
15 import pickle
16

```

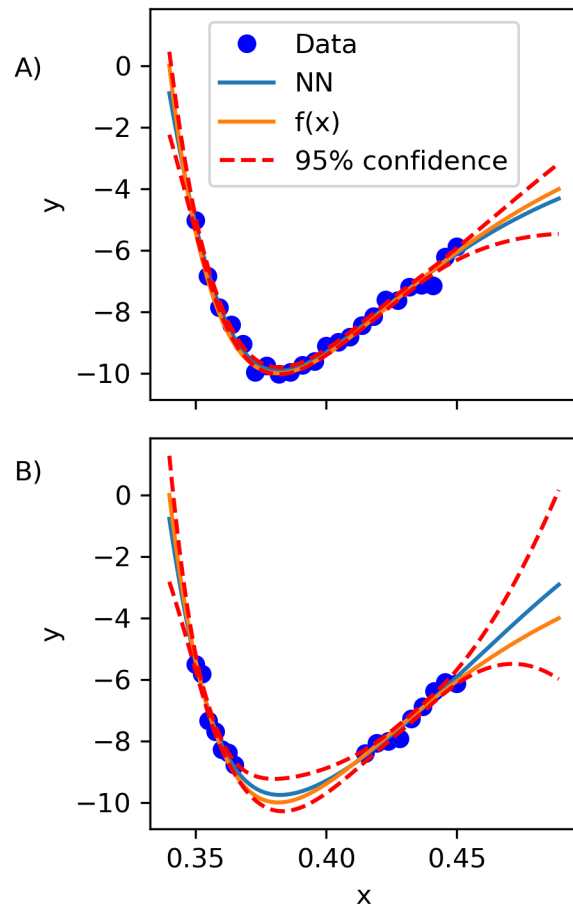


Figure 2

```

17  is_train = True
18  is_transfer = False
19  is_force = True
20
21  if is_train and is_transfer:
22      raise ValueError('train and transfer could not be true at the same time.')
23
24  #####
25  #Hyperparameters
26  #####
27  E_coeff = 100
28  if is_force:
29      F_coeff = 1
30  else:
31      F_coeff = 0
32
33  val_interval = 10
34  n_val_stop = 10
35  epoch = 3000
36
37  opt_method = 'lbfgs'
38
39
40  if opt_method == 'lbfgs':
41      history_size = 100
42      lr = 1
43      max_iter = 10
44      line_search_fn = 'strong_wolfe'
45
46
47  convergence = {'E_cov':0.0005, 'F_cov':0.005}
48
49  # min_max will scale fingerprints to (0,1)
50  fp_scale_method = 'min_max'
51  e_scale_method = 'min_max'
52
53
54  test_percent = 0.2
55  # Percentage from train+val
56  val_percent = 0.2
57
58  # Training model configuration
59  SEED = [2]
60  n_nodes = [11,11]
61  activations = [torch.nn.Sigmoid(), torch.nn.Sigmoid()]
62  lr = 1
63  hp = {'n_nodes': n_nodes, 'activations': activations, 'lr': lr}
64
65  #####
66  #Configuration
67  #####
68
69  if is_train:
70      # The Name of the training
71      Name = f'lattice39'
72      for seed in SEED:
73          if not os.path.exists(Name+f'-' +str(seed)):
74              os.makedirs(Name+f'-' +str(seed))
75
76      dbfile = f'data/lattice39.db'
77      db = connect(dbfile)
78
79      elements = ['Pd', 'Au']
80      nelem = len(elements)
81      # This is the energy of the metal in its ground state structure
82      # if you don't know the energy of the ground state structure,
83      # you can set it to None
84      element_energy = None

```



```

85     # Allen electronegativity
86     weights = [1.58, 1.92]
87
88
89     Gs = [22]
90     cutoff = 6.35
91     g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
92     g2_Rses = [0.0]
93
94
95     sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]
96     params_set = set_sym(elements, Gs, cutoff,
97                          g2_etas=g2_etas, g2_Rses=g2_Rses,
98                          weights=weights)
99
100     N_sym = params_set[elements[0]]['num']
101
102     #####
103     #Training
104     #####
105
106     Name = f'lattice39'
107     if is_train:
108         for seed in SEED:
109             # This use the context manager to operate in the data directory
110             with cd(Name+f'--{seed}'):
111                 pickle.dump(sym_params, open("sym_params.sav", "wb"))
112                 logfile = open('log.txt', 'w+')
113                 resultfile = open('result.txt', 'w+')
114
115                 if os.path.exists('test.sav'):
116                     logfile.write('Did not calculate symfunctions.\n')
117                 else:
118                     data_dict = snn2sav(db, Name, elements, params_set,
119                                         element_energy=element_energy)
120                     train_dict = train_test_split(data_dict, 1-test_percent, seed=seed)
121                     train_val_split(train_dict, 1-val_percent, seed=seed)
122
123                 logfile.flush()
124
125                 train_dict = torch.load('final_train.sav')
126                 val_dict = torch.load('final_val.sav')
127                 test_dict = torch.load('test.sav')
128                 scaling = get_scaling(train_dict, fp_scale_method, e_scale_method)
129
130
131                 n_nodes = hp['n_nodes']
132                 activations = hp['activations']
133                 lr = hp['lr']
134                 model = MultiLayerNet(N_sym, n_nodes, activations, nelem, scaling=scaling)
135                 if opt_method == 'lbfgs':
136                     optimizer = torch.optim.LBFGS(model.parameters(), lr=lr,
137                                                    max_iter=max_iter, history_size=history_size,
138                                                    line_search_fn=line_search_fn)
139
140                 results = train(train_dict, val_dict,
141                                model,
142                                opt_method, optimizer,
143                                E_coeff, F_coeff,
144                                epoch, val_interval,
145                                n_val_stop,
146                                convergence, is_force,
147                                logfile)
148                 [loss, E_MAE, F_MAE, v_loss, v_E_MAE, v_F_MAE] = results
149
150                 test_results = evaluate(test_dict, E_coeff, F_coeff, is_force)
151                 [test_loss, test_E_MAE, test_F_MAE] = test_results
152                 resultfile.write(f'Hyperparameter: n_nodes = {n_nodes}, activations = {activations}, lr = {lr}\n')



```

```

153     resultfile.write(f'loss = {loss}, E_MAE = {E_MAE}, F_MAE = {F_MAE}.\n')
154     resultfile.write(f'v_loss = {v_loss}, v_E_MAE = {v_E_MAE}, v_F_MAE = {v_F_MAE}.\n')
155     resultfile.write(f'test_loss = {test_loss}, test_E_MAE = {test_E_MAE}, test_F_MAE = {test_F_MAE}.\n')
156
157
158     logfile.close()
159     resultfile.close()

```

4 Preprocessing the predict-4.0 and 4.1 datasets

The database files containing configurations with 4.034 Å lattice constant: , and configurations with 4.134 Å lattice constant: .

The following code splits the predict-4.0 and 4.1 datasets, generating directory folders "lattice40_pred-2" and "lattice41_pred-2" with relevant files: split dataset files "final_train.sav", "final_val.sav", "test.sav".

```

1  import sys
2
3  sys.path.append("../SimpleNN")
4  sys.path.append("../")
5
6  import os
7  from ase.db import connect
8  from ContextManager import cd
9  from preprocess import train_test_split, train_val_split, get_scaling, CV
10 from preprocess import snn2sav
11 from fp_calculator import set_sym, calculate_fp
12
13
14 # min_max will scale fingerprints to (0,1)
15 fp_scale_method = 'min_max'
16 e_scale_method = 'min_max'
17
18
19 test_percent = 0.2
20 # Percentage from train+val
21 val_percent = 0.2
22
23 # Training model configuration
24 SEED = [2]
25
26 #####
27 #Split Predict-4.0 dataset
28 #####
29
30
31 Name = f'lattice40_pred'
32
33 for seed in SEED:
34     if not os.path.exists(Name+f'-{seed}'):
35         os.makedirs(Name+f'-{seed}')
36
37 dbfile = 'data/lattice40.db'
38 db = connect(dbfile)
39
40 elements = ['Pd', 'Au']
41 nelem = len(elements)
42
43 element_energy = None
44 weights = [1.58, 1.92]
45
46 Gs = [22]

```

```

47 cutoff = 6.35
48 g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
49 g2_Rses = [0.0]
50
51
52 sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]
53 params_set = set_sym(elements, Gs, cutoff,
54                      g2_etas=g2_etas, g2_Rses=g2_Rses,
55                      weights=weights)
56 N_sym = params_set[elements[0]]['num']
57
58 with cd(Name+f'-{seed}'):
59     data_dict = snn2sav(db, Name, elements, params_set,
60                       element_energy=element_energy)
61
62     train_dict = train_test_split(data_dict,1-0.2,seed=seed)
63     train_val_split(train_dict,1-0.2,seed=seed)
64
65 #####
66 #Split Predict-4.1 dataset
67 #####
68
69 Name = f'lattice41_pred'
70
71 for seed in SEED:
72     if not os.path.exists(Name+f'-{seed}'):
73         os.makedirs(Name+f'-{seed}')
74
75 dbfile = 'data/lattice41.db'
76 db = connect(dbfile)
77
78 with cd(Name+f'-{seed}'):
79     data_dict = snn2sav(db, Name, elements, params_set,
80                       element_energy=element_energy)
81
82     train_dict = train_test_split(data_dict,1-0.2,seed=seed)
83     train_val_split(train_dict,1-0.2,seed=seed)

```

5 Uncertainty and plots for first model

The following code imports functions from the python file: .

```

1 import torch
2 from uncert import evaluate_uncert
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from scipy.stats.distributions import t
6 from Batch import batch_pad
7 from matplotlib.ticker import StrMethodFormatter
8
9 #get inverse fisher information
10 def get_pcov(h):
11     eigs0 = np.linalg.eigvalsh(h)[0]
12     if (eigs0 < 0):
13         eps = max(1e-5, eigs0*-1.05)
14     else:
15         eps = 1e-5
16     j = np.linalg.pinv(h + eps*np.identity(h.shape[0]))
17     pcov1 = j*alpha
18     u, v = np.linalg.eigh(pcov1)
19     return v @ np.diag(np.maximum(u,0)) @ v.T
20
21
22 def flatten_gprime(agrad):

```

```

23     cnt = 0
24     for g in agrad:
25         g_vector = g.contiguous().view(-1) if cnt ==0 else torch.cat([g_vector, g.contiguous().view(-1)])
26         cnt = 1
27     return g_vector
28
29 #get uncertainties for a dataset
30 def get_uncerts(name, data_dict):
31     model = torch.load(name)
32     scaling = model.scaling
33     gmin = scaling['gmin']
34     gmax = scaling['gmax']
35     emin = scaling['emin']
36     emax = scaling['emax']
37
38     ids = np.array(list(data_dict.keys()))
39     batch_info = batch_pad(data_dict,ids)
40     b_fp = batch_info['b_fp']
41
42     b_e_mask = batch_info['b_e_mask']
43     b_fp.requires_grad = True
44     sb_fp = (b_fp - gmin) / (gmax - gmin)
45
46     N_atoms = batch_info['N_atoms'].view(-1)
47     b_e = batch_info['b_e'].view(-1)
48     b_f = batch_info['b_f']
49
50     Atomic_Es = model(sb_fp)
51     E_predict = torch.sum(Atomic_Es * b_e_mask, dim = [1,2])
52     E_predict = E_predict/N_atoms
53     E_predict = E_predict * (emax - emin) + emin
54
55     uncerts = []
56     for i, ei in enumerate(E_predict):
57         gprime = torch.autograd.grad(ei, model.parameters(), create_graph=True, retain_graph=True)
58         gprime = flatten_gprime(gprime).detach().numpy()
59         se = gprime @ pcov @ gprime
60         uncerts += [(np.sqrt(se), np.sqrt(se + rmse.item()*2), np.linalg.norm(gprime))]
61     uncerts = np.array(uncerts)
62     return uncerts
63
64
65 Name = 'lattice39-2'
66
67 #load datasets
68 train_dict = torch.load(f'{Name}/final_train.sav')
69 val_dict = torch.load(f'{Name}/final_val.sav')
70 test_dict = torch.load(f'{Name}/test.sav')
71
72 #get NN predictions, RMSE, hessian
73 pred_e, actual_e, rmse, h = evaluate_uncert(f'{Name}/best_model',train_dict, True)
74 h = h.detach().numpy()
75 pred_e_val, actual_e_val, rmse_val = evaluate_uncert(f'{Name}/best_model',val_dict, False)
76 pred_e_test, actual_e_test, rmse_test = evaluate_uncert(f'{Name}/best_model',test_dict, False)
77
78
79 ndata = pred_e.shape[0]
80 alpha = rmse.item()*2
81 pcov = get_pcov(h)
82
83 #get uncertainties
84 uncerts_val = get_uncerts(f'{Name}/best_model',val_dict)
85 uncerts_train = get_uncerts(f'{Name}/best_model',train_dict)
86 uncerts_test = get_uncerts(f'{Name}/best_model',test_dict)
87
88 #####
89 #Parity Plot
90 #####

```

```

91
92 plt.clf()
93 plt.rcParams.update({'font.size': 10})
94 fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)
95
96 fig.set_size_inches(3.25, 5.5)
97
98 eline = np.linspace(np.min(np.concatenate((actual_e, actual_e_test))),
99                     np.max(np.concatenate((actual_e, actual_e_test))), 10)
100
101 #ax[0].set_title(' ')
102 ax[0].plot(actual_e, pred_e, '.', color='tab:orange', alpha=1, label='Train')
103 #ax[0].set_xlabel(' ')
104 ax[0].legend(loc='lower right')
105 ax[0].plot(eline, eline, 'k--', alpha=0.7)
106
107 ax[1].plot(eline, eline, 'k--', alpha=0.7)
108 ax[1].plot(actual_e_val, pred_e_val, '.', color='g', alpha=0.9, label='Validation')
109 ax[1].plot(actual_e_test, pred_e_test, '.', color='y', alpha=0.8, label='Test')
110 ax[1].legend(loc='lower right')
111
112 plt.figtext(0.01, 0.4, "NN Energy (eV/atom)", rotation='vertical', size=10)
113 ax[1].set_xlabel('DFT Energy (eV/atom)')
114 plt.tight_layout()
115 plt.subplots_adjust(left=0.21)
116 for ext in ['png', 'eps', 'pdf']:
117     plt.savefig(f'subplotparityslides-energy-only.{ext}', dpi=300)
118 print(f'"+attr_org: :width 600
119 #+caption: Figure 3
120 [./subplotparityslides-energy-only.png]]')
121
122 #####
123 # Distribution of uncertainties
124 #####
125
126 plt.clf()
127 plt.figure(figsize=(3.25, 3))
128 plt.hist(uncerts_train[:,0], label='Train', density=True, alpha=0.5, color='tab:orange')
129 plt.hist(uncerts_val[:,0], label='Validation', density=True, alpha=0.5, color='g')
130 plt.hist(uncerts_test[:,0], label='Test', density=True, alpha=0.5, color='y')
131 plt.legend()
132 plt.xlabel('Standard Error Confidence (eV/atom)')
133 plt.ylabel('Density')
134 plt.locator_params(axis='x', nbins=7)
135 plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.3f}'))
136 plt.tight_layout()
137 for ext in ['png', 'eps', 'pdf']:
138     plt.savefig(f'hist-uncerts-pot1.{ext}', dpi=300)
139 print(f'"+attr_org: :width 600
140 #+caption: Figure 4
141 [./hist-uncerts-pot1.png]]')
142
143
144 #####
145 # Parity plot with 95% prediction interval
146 #####
147
148 plt.clf()
149 plt.figure(figsize=(3.25, 4.0))
150 tval = t.ppf(0.975, ndata)
151 plt.errorbar(actual_e_test, pred_e_test, yerr=tval*uncerts_test[:,1], fmt='y_',
152             ecolor='m', label='Test, 95% prediction')
153
154 plt.xlabel('DFT Energy (eV/atom)')
155 plt.ylabel('NN Energy (eV/atom)')
156 plt.plot([np.min(actual_e_test), np.max(actual_e_test)],
157         [np.min(actual_e_test),
158          np.max(actual_e_test)], 'k--', alpha=0.7, linewidth=0.3)

```

```

159 plt.legend()
160 plt.tight_layout()
161 for ext in ['png', 'eps', 'pdf']:
162     plt.savefig(f'parity-errorbar-test-pot1-prediction.{ext}', dpi=300)
163     print(f'#+attr_org: :width 600')
164     #+caption: Figure 5
165     [./parity-errorbar-test-pot1-prediction.png]]')
166
167 #####
168 #Inference on predict-4.0 and 4.1 dataset
169 #####
170
171 data_dict = torch.load(f'lattice40_pred-2/test.sav')
172 pred_e_40p, actual_e_40p, rmse_40p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
173 uncersts_40p = get_uncerts(f'{Name}/best_model', data_dict)
174
175 data_dict = torch.load(f'lattice41_pred-2/test.sav')
176 pred_e_41p, actual_e_41p, rmse_41p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
177 uncersts_41p = get_uncerts(f'{Name}/best_model', data_dict)
178
179 #make plot
180
181 plt.clf()
182 plt.rc('legend', fontsize=10)
183 fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)
184 fig.set_size_inches(3.25, 4.5)
185 #ax[0].set_title(' ')
186 ax[0].errorbar(actual_e_40p, pred_e_40p, yerr = tval * uncersts_40p[:,1], color='tab:pink',
187               fmt = '_', ecolor='r', label='Predict 4.0, \n95% prediction')
188 #ax[0].set_xlabel(' ')
189 #ax[0].set_ylabel('NN Energy (eV/atom)')
190 ax[0].legend(loc='upper left')
191 eline = np.linspace(np.min(np.concatenate((actual_e_40p, actual_e_41p))),
192                    np.max(np.concatenate((actual_e_40p, actual_e_41p))), 10)
193 ax[0].plot(eline, eline, 'k--', alpha=0.8, linewidth=0.5)
194
195 ax[1].errorbar(actual_e_41p, pred_e_41p, yerr = tval * uncersts_41p[:,1],
196               fmt = 'b_', ecolor='c', label='Predict 4.1, \n95% prediction')
197 ax[1].legend()
198 ax[1].plot(eline, eline, 'k--', alpha=0.7, linewidth=0.5)
199 ax[1].set_xlabel("DFT Energy (eV/atom)")
200 plt.figtext(0.01, 0.4, "NN Energy (eV/atom)", rotation='vertical', size=10)
201
202 plt.tight_layout()
203 plt.subplots_adjust(left=0.21)
204 for ext in ['png', 'eps', 'pdf']:
205     plt.savefig(f'subplot-parity-40-41-pot-prediction.{ext}', dpi=300)
206     print(f'#+attr_org: :width 600')
207     #+caption: Figure 6
208     [./subplot-parity-40-41-pot-prediction.png]]\n')
209
210 #####
211 # Uncertainty vs True Error Scatterplot
212 #####
213
214
215 def scatter_hist(x, y, ax, ax_histx, ax_histy, label, color=None):
216     # no labels
217     ax_histx.tick_params(axis="x", labelbottom=False)
218     ax_histy.tick_params(axis="y", labelleft=False)
219
220     # the scatter plot:
221     ax.scatter(x, y, alpha=0.5, label=label, color=color)
222
223     # now determine nice limits by hand:
224     binwidth = 0.0001
225     xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
226     lim = (int(xymax / binwidth) + 1) * binwidth

```

```

227
228     #bins = np.arange(0, lim + binwidth, binwidth)
229     ax_histx.hist(x, alpha=0.5, color=color, density=True)
230     ax_histy.hist(y, orientation='horizontal', alpha=0.5, color=color, density=True)
231
232     fig = plt.figure(figsize=(3.25, 4.))
233     # Add a gridspec with two rows and two columns and a ratio of 2 to 7 between
234     # the size of the marginal axes and the main axes in both directions.
235     # Also adjust the subplot parameters for a square plot.
236     gs = fig.add_gridspec(2, 2, width_ratios=(7, 2), height_ratios=(2, 7),
237                          left=0.11, right=0.98, bottom=0.07, top=0.97, wspace=0.05,
238                          hspace=0.05)
239
240     ax = fig.add_subplot(gs[1, 0])
241     ax_histx = fig.add_subplot(gs[0, 0], sharex=ax)
242     ax_histy = fig.add_subplot(gs[1, 1], sharey=ax)
243
244     # use the previously defined function
245
246     scatter_hist(np.absolute(actual_e_test-pred_e_test), uncerts_test[:,0],
247                  ax, ax_histx, ax_histy, 'Test', 'y')
248
249     scatter_hist(np.absolute(actual_e_40p-pred_e_40p), uncerts_40p[:,0],
250                  ax, ax_histx, ax_histy, 'Predict 4.0', 'tab:pink')
251
252     scatter_hist(np.absolute(pred_e_41p-actual_e_41p), uncerts_41p[:,0],
253                  ax, ax_histx, ax_histy, 'Predict 4.1')
254
255
256     ax.set_xlabel('Absolute Error Energy (eV/atom)')
257     ax.set_ylabel('Standard Error Confidence (eV/atom)')
258     ax.legend()
259     for ext in ['png', 'eps', 'pdf']:
260         plt.savefig(f'uncert-v-error-w-hist-pot1-origw-test.{ext}', dpi=300, bbox_inches='tight')
261     print(f'"+attr_org: :width 600
262     #+caption: Figure 8
263     [./uncert-v-error-w-hist-pot1-origw-test.png]]')

```

6 Fingerprints

```

1  import torch
2  from uncert import get_fps
3  import matplotlib.pyplot as plt
4
5  Name = 'lattice39-2'
6  train_dict = torch.load(f'{Name}/final_train.sav')
7  fp_train, e_mask_train = get_fps(f'{Name}/best_model', train_dict)
8
9  data_dict = torch.load(f'lattice40_pred-2/test.sav')
10 fp_40, e_mask_40 = get_fps(f'{Name}/best_model', data_dict)
11
12 data_dict = torch.load(f'lattice41_pred-2/test.sav')
13 fp_41, e_mask_41 = get_fps(f'{Name}/best_model', data_dict)
14
15 plt.rcParams.update({'font.size': 10})
16 plt.figure(figsize=(3.25, 4.))
17 for i in range(2):
18     for j in range(4):
19         plt.clf()
20         plt.hist(fp_train[e_mask_train[:, :, i]==1][:, j], alpha=0.5, density=True, label='Train', color='y')
21         plt.hist(fp_40[e_mask_40[:, :, i]==1][:, j], alpha=0.5, density=True, label='Predict 4.0', color='tab:pink')
22
23         plt.hist(fp_41[e_mask_41[:, :, i]==1][:, j], alpha=0.5, density=True, label='Predict 4.1')
24         plt.xlabel('Fingerprint Value')
25         plt.ylabel('Density')
26         plt.legend()

```

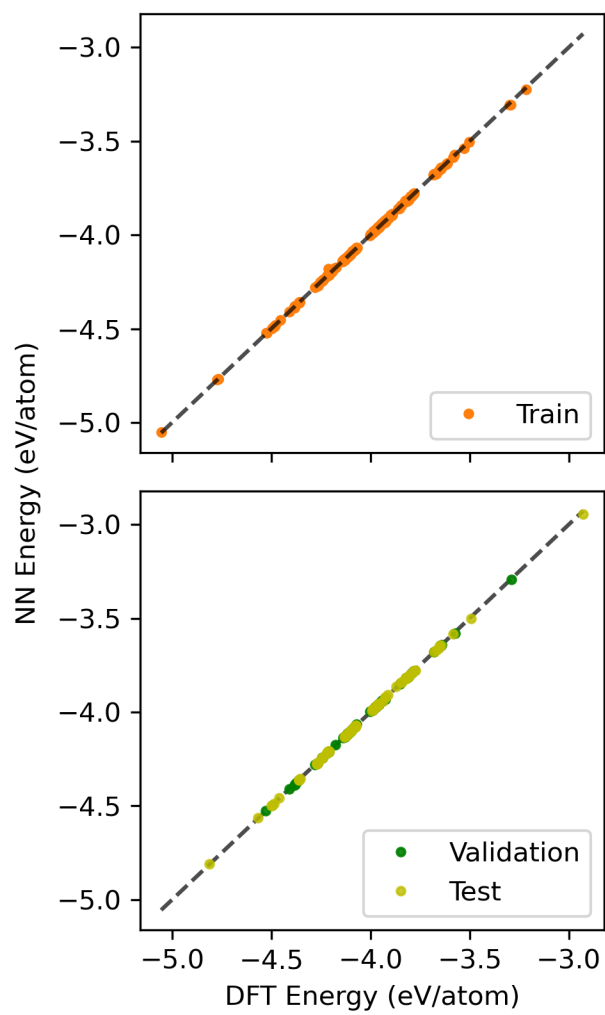


Figure 3

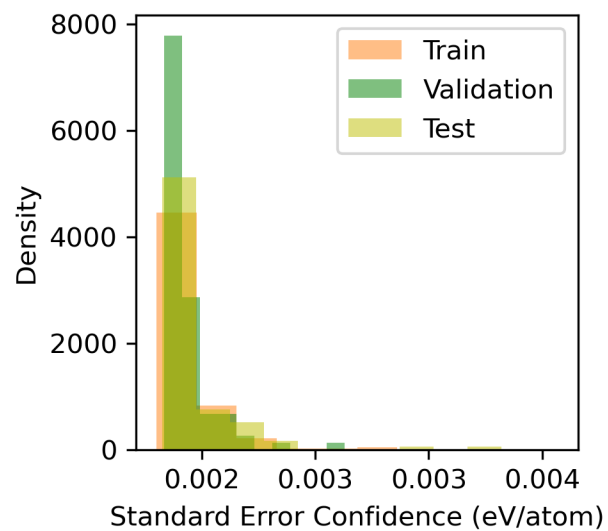


Figure 4

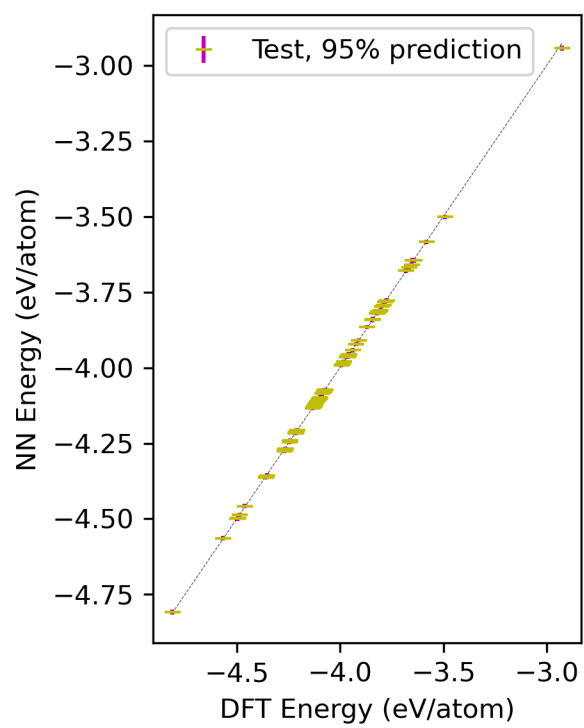


Figure 5

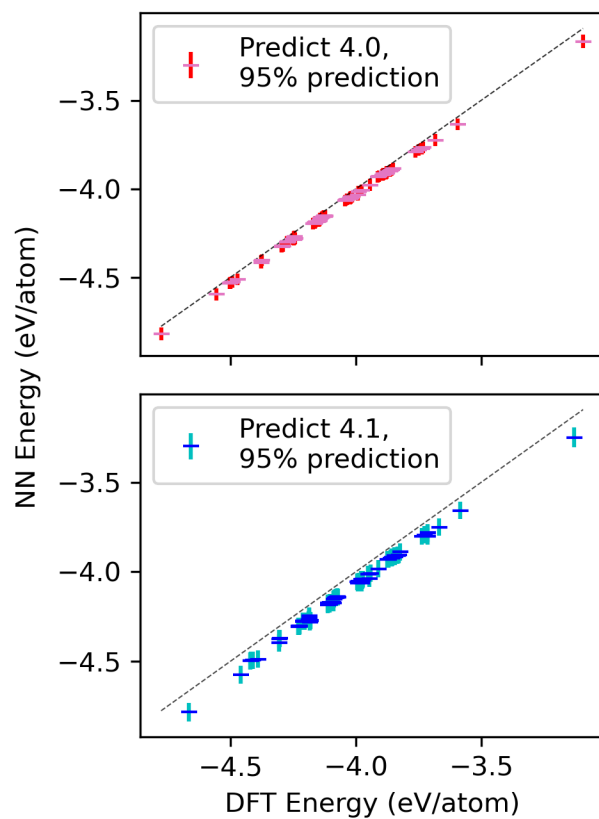


Figure 6

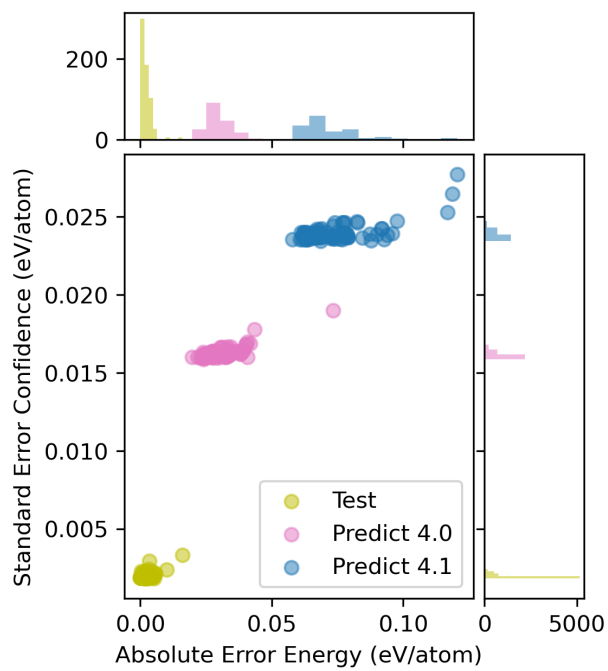


Figure 8

```

27     plt.tight_layout()
28     for ext in ['png', 'eps', 'pdf']:
29         plt.savefig(f'fps-hist-el{i}-fp{j}-{ext}', dpi=300)
30
31 print(f'''+attr_org: :width 600
32 #+caption: Figure 7
33 [[./fps-hist-el0-fp0.png]]''')

```

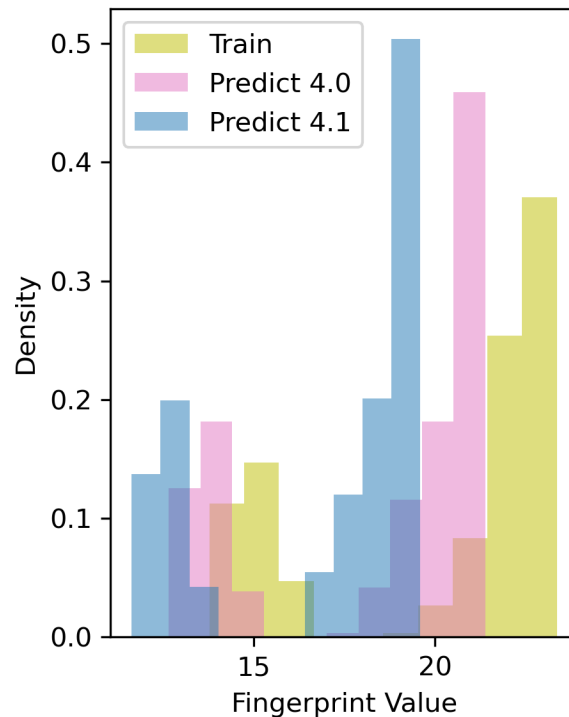


Figure 7

7 Model retraining

The following code concatenates the original training-data with training portion of the predict-4.0 and 4.1 datasets. The code trains the potential and generates a directory folder "lattice39-40-41-2" with relevant files: concatenated dataset files "final_train.sav", "final_val.sav", "test.sav"; model file "best_model".

```

1  import sys
2
3  sys.path.append("../SimpleNN")
4  sys.path.append("../")
5
6  import os
7  from ase.db import connect
8  import torch
9  from ContextManager import cd
10 from preprocess import train_test_split, train_val_split, get_scaling, CV
11 from preprocess import snn2sav
12 from NN import MultiLayerNet
13 from train import train, evaluate

```

```

14 from fp_calculator import set_sym, calculate_fp
15 import pickle
16
17 is_train = True
18 is_transfer = False
19 is_force = True
20
21 if is_train and is_transfer:
22     raise ValueError('train and transfer could not be true at the same time.')
23
24 #####
25 #Hyperparameters
26 #####
27 E_coeff = 100
28 if is_force:
29     F_coeff = 1
30 else:
31     F_coeff = 0
32
33 val_interval = 10
34 n_val_stop = 10
35 epoch = 3000
36
37 opt_method = 'lbfgs'
38
39
40 if opt_method == 'lbfgs':
41     history_size = 100
42     lr = 1
43     max_iter = 10
44     line_search_fn = 'strong_wolfe'
45
46
47 convergence = {'E_cov':0.0005, 'F_cov':0.005}
48
49 # min_max will scale fingerprints to (0,1)
50 fp_scale_method = 'min_max'
51 e_scale_method = 'min_max'
52
53
54 test_percent = 0.2
55 # Percentage from train+val
56 val_percent = 0.2
57
58 # Training model configuration
59 SEED = [2]
60 n_nodes = [11,11]
61 activations = [torch.nn.Sigmoid(), torch.nn.Sigmoid()]
62
63 lr = 1
64 hp = {'n_nodes': n_nodes, 'activations': activations, 'lr': lr}
65
66 #####
67 #Configuration
68 #####
69
70 elements = ['Pd', 'Au']
71 nelem = len(elements)
72
73 element_energy = None
74 weights = [1.58, 1.92]
75
76 Gs = [22]
77 cutoff = 6.35
78 g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
79 g2_Rses = [0.0]
80
81 sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]

```

```

82  params_set = set_sym(elements, Gs, cutoff,
83                        g2_etas=g2_etas, g2_Rses=g2_Rses,
84                        weights=weights)
85  N_sym = params_set[elements[0]]['num']
86
87  #####
88  #Training
89  #####
90
91  Name = 'lattice39-40-41'
92
93  if is_train:
94      for seed in SEED:
95          # This use the context manager to operate in the data directory
96
97          if not os.path.exists(Name+f'-{seed}'):
98              os.makedirs(Name+f'-{seed}')
99
100         with cd(Name+f'-{seed}'):
101             pickle.dump(sym_params, open("sym_params.sav", "wb"))
102             logfile = open('log.txt', 'w+')
103             resultfile = open('result.txt', 'w+')
104
105             if os.path.exists('test.sav'):
106                 logfile.write('Did not calculate symfunctions.\n')
107             else:
108                 #this part is to concatenate the train-data subsets together.
109                 train_dict1 = torch.load('../lattice39-2/final_train.sav')
110                 train_dict2 = torch.load('../lattice40_pred-2/final_train.sav')
111                 train_dict3 = torch.load('../lattice41_pred-2/final_train.sav')
112                 train_dict = dict(train_dict1)
113                 new_dict = {k+1000: v for k, v in train_dict2.items()}
114                 train_dict.update(new_dict)
115                 new_dict = {k+2000: v for k, v in train_dict3.items()}
116                 train_dict.update(new_dict)
117
118                 val_dict1 = torch.load('../lattice39-2/final_val.sav')
119                 val_dict2 = torch.load('../lattice40_pred-2/final_val.sav')
120                 val_dict3 = torch.load('../lattice41_pred-2/final_val.sav')
121                 val_dict = dict(val_dict1)
122                 new_dict = {k+1000: v for k, v in val_dict2.items()}
123                 val_dict.update(new_dict)
124                 new_dict = {k+2000: v for k, v in val_dict3.items()}
125                 val_dict.update(new_dict)
126
127
128                 test_dict1 = torch.load('../lattice39-2/test.sav')
129                 test_dict2 = torch.load('../lattice40_pred-2/test.sav')
130                 test_dict3 = torch.load('../lattice41_pred-2/test.sav')
131                 test_dict = dict(test_dict1)
132                 new_dict = {k+1000: v for k, v in test_dict2.items()}
133                 test_dict.update(new_dict)
134                 new_dict = {k+2000: v for k, v in test_dict3.items()}
135                 test_dict.update(new_dict)
136
137
138
139                 torch.save(train_dict, 'final_train.sav')
140                 torch.save(val_dict, 'final_val.sav')
141                 torch.save(test_dict, 'test.sav')
142
143         scaling = get_scaling(train_dict, fp_scale_method, e_scale_method)
144
145         n_nodes = hp['n_nodes']
146         activations = hp['activations']
147         lr = hp['lr']
148         #model = torch.load('../lattice39-2/best_model')
149         model = MultiLayerNet(N_sym, n_nodes, activations, nelelem, scaling=scaling)

```

```

150         if opt_method == 'lbfgs':
151             optimizer = torch.optim.LBFGS(model.parameters(), lr=lr,
152                                           max_iter=max_iter, history_size=history_size,
153                                           line_search_fn=line_search_fn)
154
155         results = train(train_dict, val_dict,
156                       model,
157                       opt_method, optimizer,
158                       E_coeff, F_coeff,
159                       epoch, val_interval,
160                       n_val_stop,
161                       convergence, is_force,
162                       logfile)
163         [loss, E_MAE, F_MAE, v_loss, v_E_MAE, v_F_MAE] = results
164
165         test_results = evaluate(test_dict, E_coeff, F_coeff, is_force)
166         [test_loss, test_E_MAE, test_F_MAE] = test_results
167         resultfile.write(f'Hyperparameter: n_nodes = {n_nodes}, activations = {activations}, lr = {lr}\n')
168         resultfile.write(f'loss = {loss}, E_MAE = {E_MAE}, F_MAE = {F_MAE}.\n')
169         resultfile.write(f'v_loss = {v_loss}, v_E_MAE = {v_E_MAE}, v_F_MAE = {v_F_MAE}.\n')
170         resultfile.write(f'test_loss = {test_loss}, test_E_MAE = {test_E_MAE}, test_F_MAE = {test_F_MAE}.\n')
171
172         logfile.close()
173         resultfile.close()
174

```

8 Uncertainty for retrained model

```

1  import torch
2  from uncert import evaluate_uncert
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.stats.distributions import t
6  from Batch import batch_pad
7
8  #get inverse fisher information
9  def get_pcov(h):
10     eigs0 = np.linalg.eigvalsh(h)[0]
11     if (eigs0 < 0):
12         eps = max(1e-5, eigs0*-1.05)
13     else:
14         eps = 1e-5
15     j = np.linalg.pinv(h + eps*np.identity(h.shape[0]))
16     pcov1 = j*alpha
17     u, v = np.linalg.eigh(pcov1)
18     return v @ np.diag(np.maximum(u,0)) @ v.T
19
20
21 def flatten_gprime(agrad):
22     cnt = 0
23     for g in agrad:
24         g_vector = g.contiguous().view(-1) if cnt ==0 else torch.cat([g_vector, g.contiguous().view(-1)])
25         cnt = 1
26     return g_vector
27
28 #get uncertainties for a dataset
29 def get_uncerts(name, data_dict):
30     model = torch.load(name)
31     scaling = model.scaling
32     gmin = scaling['gmin']
33     gmax = scaling['gmax']
34     emin = scaling['emin']
35     emax = scaling['emax']
36
37     ids = np.array(list(data_dict.keys()))
38     batch_info = batch_pad(data_dict,ids)

```

```

39     b_fp = batch_info['b_fp']
40
41     b_e_mask = batch_info['b_e_mask']
42     b_fp.requires_grad = True
43     sb_fp = (b_fp - gmin) / (gmax - gmin)
44
45     N_atoms = batch_info['N_atoms'].view(-1)
46     b_e = batch_info['b_e'].view(-1)
47     b_f = batch_info['b_f']
48
49     Atomic_Es = model(sb_fp)
50     E_predict = torch.sum(Atomic_Es * b_e_mask, dim = [1,2])
51     E_predict = E_predict/N_atoms
52     E_predict = E_predict * (emax - emin) + emin
53
54     uncerts = []
55     for i, ei in enumerate(E_predict):
56         gprime = torch.autograd.grad(ei, model.parameters(), create_graph=True, retain_graph=True)
57         gprime = flatten_gprime(gprime).detach().numpy()
58         se = gprime @ pcov @ gprime
59         uncerts += [(np.sqrt(se), np.sqrt(se + rmse.item()*2), np.linalg.norm(gprime))]
60     uncerts = np.array(uncerts)
61     return uncerts
62
63 Name = 'lattice39-40-41-2'
64
65 train_dict = torch.load(f'{Name}/final_train.sav')
66 val_dict = torch.load(f'{Name}/final_val.sav')
67 test_dict = torch.load(f'{Name}/test.sav')
68
69 pred_e, actual_e, rmse, h = evaluate_uncert(f'{Name}/best_model', train_dict, True)
70 h = h.detach().numpy()
71 pred_e_val, actual_e_val, rmse_val = evaluate_uncert(f'{Name}/best_model', val_dict, False)
72 pred_e_test, actual_e_test, rmse_test = evaluate_uncert(f'{Name}/best_model', test_dict, False)
73
74 ndata = pred_e.shape[0]
75 alpha = rmse.item()*2
76 pcov = get_pcov(h)
77
78 uncerts_val = get_uncerts(f'{Name}/best_model', val_dict)
79 uncerts_train = get_uncerts(f'{Name}/best_model', train_dict)
80 uncerts_test = get_uncerts(f'{Name}/best_model', test_dict)
81
82
83 #####
84 #Parity plot after retraining
85 #####
86
87 data_dict = torch.load(f'lattice40_pred-2/test.sav')
88 pred_e_40p, actual_e_40p, rmse_40p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
89 uncerts_40p = get_uncerts(f'{Name}/best_model', data_dict)
90
91 data_dict = torch.load(f'lattice41_pred-2/test.sav')
92 pred_e_41p, actual_e_41p, rmse_41p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
93 uncerts_41p = get_uncerts(f'{Name}/best_model', data_dict)
94
95 tval = t.ppf(0.975, ndata)
96 plt.clf()
97
98 plt.rcParams.update({'font.size': 10})
99 fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)
100 fig.set_size_inches(3.25, 6.5)
101 #ax[0].set_title(' ')
102 ax[0].errorbar(actual_e, pred_e, yerr = tval * uncerts_train[:,1], fmt = 'y_', ecolor='m',
103               label='Train, \n95% prediction')
104 #ax[0].set_xlabel(' ')
105 #ax[0].set_ylabel('NN Energy (eV/atom)')
106 ax[0].legend()

```

```

107 eline = np.linspace(np.min(actual_e), np.max(actual_e_40p), 10)
108 ax[0].plot(eline, eline, 'k--', alpha=0.7, linewidth=0.3)
109
110 ax[1].errorbar(actual_e_40p, pred_e_40p, yerr = tval * uncerts_40p[:,1], color='tab:pink',
111               fmt = '_', ecolor='b', label='Predict 4.0, 4.1, \n95% prediction')
112 ax[1].errorbar(actual_e_41p, pred_e_41p, yerr = tval * uncerts_41p[:,1], color='tab:pink',
113               fmt = '_', ecolor='b', label='')
114 ax[1].legend(loc='upper left')
115 ax[1].plot(eline, eline, 'k--', alpha=0.7, linewidth=0.3)
116 ax[1].set_xlabel('DFT Energy (eV/atom)')
117 plt.figtext( 0.01, 0.42, "NN Energy (eV/atom)", rotation='vertical', size=10)
118 plt.tight_layout()
119 plt.subplots_adjust(left=0.23)
120 for ext in ['png', 'eps', 'pdf']:
121     plt.savefig(f'subplot-parity-40-41-pot2-pred-v2.{ext}', dpi=300)
122 print('''#+attr_org: :width 600
123 #+caption: Figure 9
124 [./subplot-parity-40-41-pot2-pred-v2.png]
125 ''')
126
127 #####
128 #Uncertainty vs True Error Scatterplot
129 #####
130
131 def scatter_hist(x, y, ax, ax_histx, ax_histy, label, color=None):
132     # no labels
133     ax_histx.tick_params(axis="x", labelbottom=False)
134     ax_histy.tick_params(axis="y", labelleft=False)
135
136     # the scatter plot:
137     ax.scatter(x, y, alpha=0.5, label=label, color=color)
138
139     # now determine nice limits by hand:
140     binwidth = 0.0001
141     xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
142     lim = (int(xymax/binwidth)+1)*binwidth
143
144     #bins = np.arange(0, lim + binwidth, binwidth)
145     ax_histx.hist(x, alpha=0.5, color=color, density=True)
146     ax_histy.hist(y, orientation='horizontal', alpha=0.5, color=color, density=True)
147
148 fig = plt.figure(figsize=(3.25, 4.0))
149 # Add a gridspec with two rows and two columns and a ratio of 2 to 7 between
150 # the size of the marginal axes and the main axes in both directions.
151 # Also adjust the subplot parameters for a square plot.
152 gs = fig.add_gridspec(2, 2, width_ratios=(7, 2), height_ratios=(2, 7),
153                       left=0.11, right=0.98, bottom=0.07, top=0.97, wspace=0.05, hspace=0.05)
154
155 ax = fig.add_subplot(gs[1, 0])
156 ax_histx = fig.add_subplot(gs[0, 0], sharex=ax)
157 ax_histy = fig.add_subplot(gs[1, 1], sharey=ax)
158
159 # use the previously defined function
160 scatter_hist(np.absolute(actual_e-pred_e), uncerts_train[:,0], ax, ax_histx, ax_histy,
161             'Train', 'y')
162 scatter_hist(np.absolute(actual_e_40p-pred_e_40p), uncerts_40p[:,0], ax, ax_histx, ax_histy,
163             'Predict 4.0', 'tab:pink')
164 scatter_hist(np.absolute(pred_e_41p-actual_e_41p), uncerts_41p[:,0], ax, ax_histx, ax_histy,
165             'Predict 4.1')
166
167 ax.set_xlabel('Absolute Error Energy (eV/atom)')
168 ax.set_ylabel('Standard Error Confidence (eV/atom)')
169 ax.legend()
170 for ext in ['png', 'eps', 'pdf']:
171     plt.savefig(f'uncert-v-error-w-hist-ret40-41-orig.{ext}', dpi=300, bbox_inches='tight')
172 print('''
173 #+attr_org: :width 600
174 #+caption: Figure 10

```


175 [[./uncert-v-error-w-hist-ret40-41-orig.png]]
176 ''')

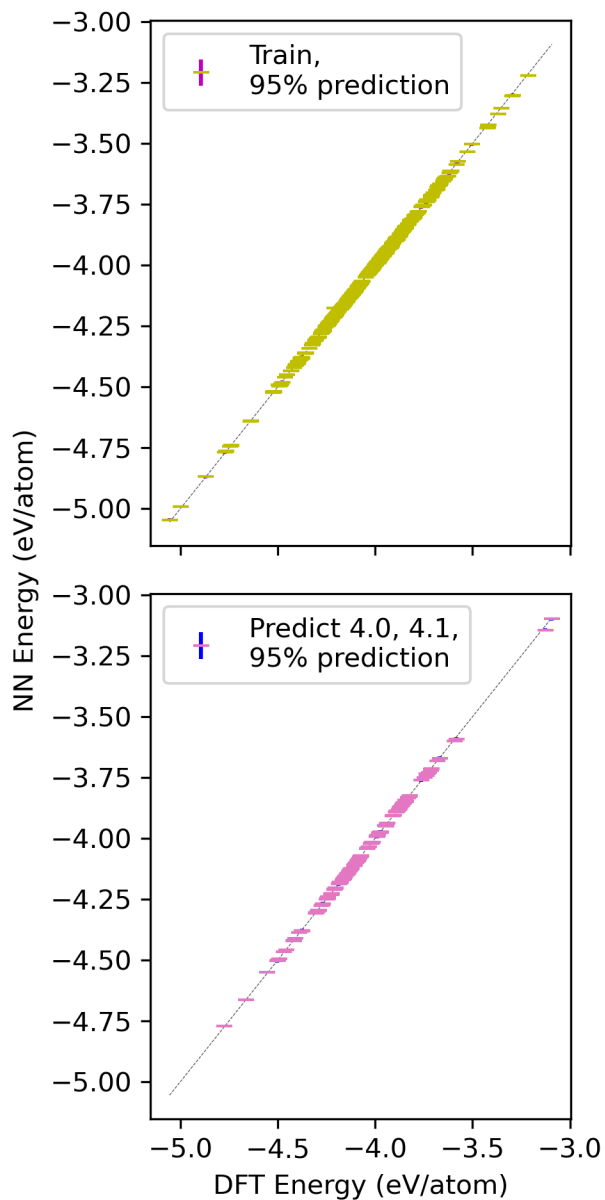


Figure 9

References

- [1] Larry Wasserman. *All of Statistics*. Springer Texts in Statistics. Springer New York, 2004.

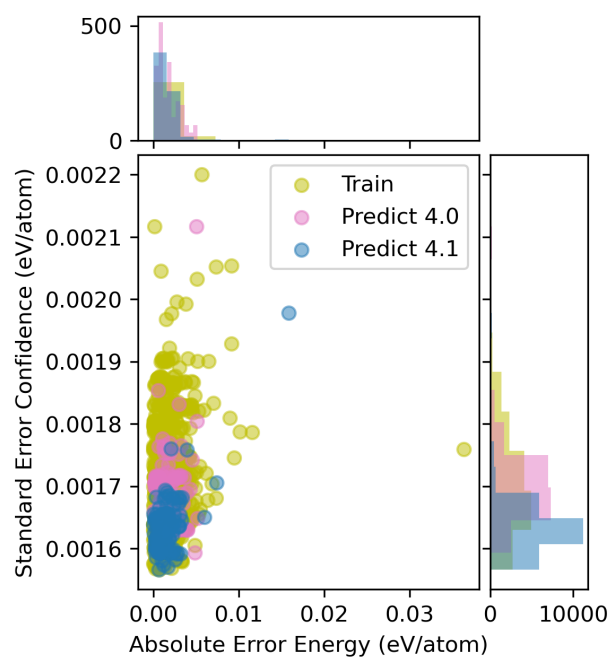


Figure 10