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

Ni Zhan and John R. Kitchin

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 $y(x_i \mid \theta)$, and the theory of the delta method assumes that the data output is the sum of the model prediction and some Gaussian error.

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

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

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

Since we assumed ϵ_i was Gaussian,

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

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 $\hat{\theta}$

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

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

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

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

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

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

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

We estimate σ^2 as

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

Once obtaining standard errors for a prediction, we can construct confidence intervals. We use $t_{\frac{\alpha}{2}} \cdot \mathrm{se}(g(\hat{\theta}))$ 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

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

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)

```python
import autograd
import autograd.numpy as np
from autograd import hessian
import matplotlib.pyplot as plt
from autograd import grad
import autograd.numpy.random as npr
from scipy.stats.distributions import t
from scipy.optimize import minimize
from matplotlib.ticker import FormatStrFormatter

# lennard jones potential
def func(x, e, s):
    return 4 * e * (np.power(np.divide(s, x), 12) -
                    np.power(np.divide(s, x), 6))

etrue = 10
strue = 0.34
numpts = 23

#xfit is for plotting
xfit = np.arange(0.34, 0.49, 0.001)
xfit = np.expand_dims(xfit, axis=1)

# weightsparser to help roll and unroll weights and biases.
class WeightsParser(object):
    """A helper class to index into a parameter vector."""

    def __init__(self):
        self.idxs_and_shapes = {}
        self.N = 0

    def add_weights(self, name, shape):
```

```
        start = self.N
        self.N += np.prod(shape)
        self.idxs_and_shapes[name] = (slice(start, self.N), shape)

    def get(self, vect, name):
        idxs, shape = self.idxs_and_shapes[name]
        return np.reshape(vect[idxs], shape)

# params is a 1-d vector of weights and biases
# parser is object that makes it easy to unroll params into matrices of
# weights and biases.
def init_random_params(scale, layer_sizes, rs=None):
    if rs is None:
        rs = npr.RandomState(2)
    parser = WeightsParser()
    for i, shape in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
        parser.add_weights(('weights', i), shape)
        parser.add_weights(('biases', i), (1, shape[1]))
    return rs.randn(parser.N), parser

# nn predict by unrolling w parser.
def nn_predict(params, inputs, nonlinearity=np.tanh):
    cur_units = inputs
    for layer in range(len(layer_sizes) - 1):
        cur_W = parser.get(params, ('weights', layer))
        cur_B = parser.get(params, ('biases', layer))
        cur_units = np.dot(cur_units, cur_W) + cur_B
        if layer < len(layer_sizes) - 2:
            cur_units = nonlinearity(cur_units)
    return cur_units

#objective with regularization to be used with scipy minimize
def objectivel2(params, X, r, alpha=0):
    ypredict = nn_predict(params, X)
    errs = r - ypredict
    weights = params[idxs]
    return np.sum(errs**2) + alpha * np.linalg.norm(weights)

layer_sizes = [1, 4, 1]
_, parser = init_random_params(1, layer_sizes)

# get the index of the weights, because only regularizing weights.
idxs = []
for layer in range(len(layer_sizes) - 1):
    sliceidx, _ = parser.idxs_and_shapes[('weights', layer)]
    idxs += [np.r_[sliceidx]]
idxs = np.array(idxs).flatten()

#sum-squared-errors
def sse(params, X, r):
    ypredict = nn_predict(params, X)
    errs = r - ypredict
    return np.sum(errs**2)

#get inverse fisher information
def get_pcov(h):
    eigs0 = np.linalg.eigvalsh(h)[0]
    if (eigs0 <0):
        eps = max(1e-5, eigs0*-1.05)
    else:
        eps = 1e-5
    j = np.linalg.pinv(h + eps * np.identity(h.shape[0]))
    pcov1 = j * scaling
    u, v = np.linalg.eigh(pcov1)
    return v @ np.diag(np.maximum(u,0)) @ v.T

#get standard errors of prediction, confidence
def getpredse(x, params):
```

```
    gprime = autograd.elementwise_grad(nn_predict,0)(params, x)
    sesq = gprime @ pcov @ gprime
    return np.sqrt(sesq), np.sqrt(sesq + scaling)

#get standard errors for a dataset
def get_se_dataset(xfit, params):
    predses = []
    for i in xfit:
        predses += [getpredse(i, params)]
    return np.array(predses)

# to make plot
# data for panel 1.
numpts = 23
xa = np.linspace(0.35, 0.45, numpts)
np.random.seed(seed=0)

ya = func(xa, etrue, strue) + np.random.normal(scale=0.2, size=xa.shape)

Xa = np.expand_dims(xa, axis=1)
ra = np.expand_dims(ya, axis=1)

initial_guess, parser = init_random_params(1, layer_sizes)

sol = minimize(objectivel2, initial_guess, args=(Xa,ra,0.01) )
paramsa = sol.x

h = hessian(sse,0)(paramsa, Xa, ra)
numptsa = Xa.shape[0]
scaling = sse(paramsa, Xa, ra)/numptsa

pcov = get_pcov(h)

predsesa = get_se_dataset(xfit, paramsa)

#data for panel 2.
x1 = np.linspace(0.35, 0.365, 7)
x2 = np.linspace(0.415, 0.45, 9)
xb = np.concatenate((x1,x2))
yb = func(xb, etrue, strue) + np.random.normal(scale=0.2, size=xb.shape)

Xb = np.expand_dims(xb, axis=1)
rb = np.expand_dims(yb, axis=1)

initial_guess, _ = init_random_params(1, layer_sizes)

sol = minimize(objectivel2, initial_guess, args=(Xb,rb,0.005) )
paramsb = sol.x

h = hessian(sse,0)(paramsb, Xb, rb)
numptsb = Xb.shape[0]
scaling = sse(paramsb, Xb, rb)/numptsb

pcov = get_pcov(h)

predsesb = get_se_dataset(xfit, paramsb)


#make a plot.

plt.clf()
fig, ax = plt.subplots(ncols =1, nrows = 2, sharex=True, sharey=True)
fig.set_size_inches(3.25,5)
tvala = t.ppf(0.975, numptsa)
tvalb = t.ppf(0.975, numptsb)

ypreda = nn_predict(paramsa, xfit).flatten()
ypredb = nn_predict(paramsb, xfit).flatten()
```

```

#ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))

#ax[0].set_title(' ')
ax[0].plot(Xa, ra, 'bo')
ax[0].plot(xfit, ypreda)
ax[0].plot(xfit, func(xfit, etrue, strue))
ax[0].plot(xfit, ypreda + predsesa[:,0] * tvala, '--r')
ax[0].plot(xfit, ypreda - predsesa[:,0] * tvala, '--r')
#ax[0].set_xlabel('x')
ax[0].set_ylabel('y')

#ax[1].set_title(' ')
ax[1].plot(Xb, rb, 'bo')
ax[1].plot(xfit, ypredb)
ax[1].plot(xfit, func(xfit, etrue, strue))
ax[1].plot(xfit, ypredb + predsesb[:,0] * tvalb, '--r')
ax[1].plot(xfit, ypredb - predsesb[:,0] * tvalb, '--r')
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')
#ax[1].yaxis.set_major_formatter(FormatStrFormatter('%.0f'))

ax[0].legend(['Data', 'NN', 'f(x)', '95% confidence'])

plt.figtext(0.05, 0.90, 'A)')
plt.figtext(0.05, 0.48, 'B)')
plt.subplots_adjust(wspace=0)
plt.tight_layout()
plt.subplots_adjust(wspace=0)
for ext in ['png', 'eps']:
    plt.savefig(f'subplot-2panel-ou.{ext}', dpi=300)
print(f'''#+attr_org: :width 600
#+caption: Figure 2
[[./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".

```
import sys

sys.path.append("../SimpleNN")
sys.path.append("../")

import os
from ase.db import connect
import torch
from ContextManager import cd
from preprocess import train_test_split, train_val_split, get_scaling, CV
from preprocess import snn2sav
from NN import MultiLayerNet
from train import train, evaluate
from fp_calculator import set_sym, calculate_fp
import pickle

```

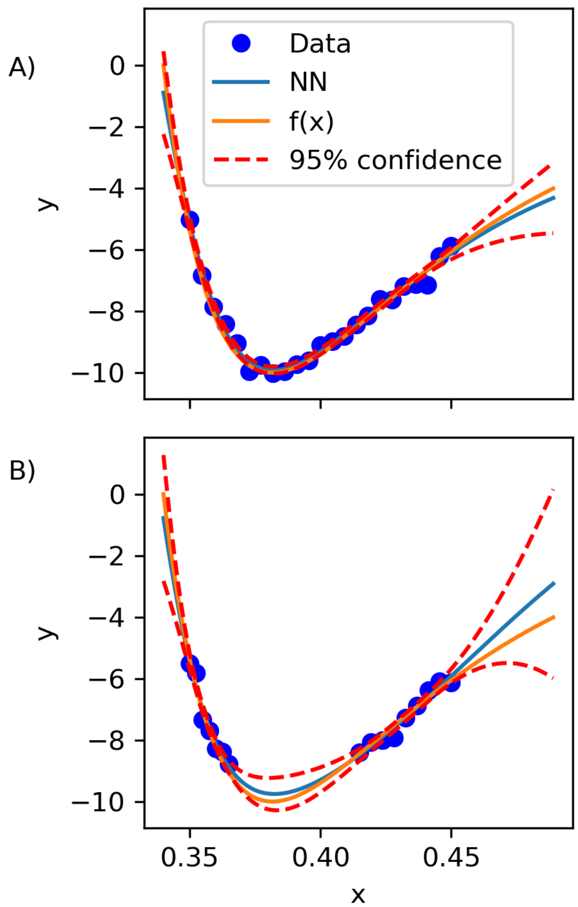


Figure 2

```python
is_train = True
is_transfer = False
is_force = True

if is_train and is_transfer:
    raise ValueError('train and transfer could not be true at the same time.')

##################################################################################
#Hyperparameters
##################################################################################
E_coeff = 100
if is_force:
    F_coeff = 1
else:
    F_coeff = 0

val_interval = 10
n_val_stop = 10
epoch = 3000

opt_method = 'lbfgs'


if opt_method == 'lbfgs':
    history_size = 100
    lr = 1
    max_iter = 10
    line_search_fn = 'strong_wolfe'


convergence = {'E_cov':0.0005,'F_cov':0.005}

# min_max will scale fingerprints to (0,1)
fp_scale_method = 'min_max'
e_scale_method = 'min_max'


test_percent = 0.2
# Pecentage from train+val
val_percent = 0.2

# Training model configuration
SEED = [2]
n_nodes = [11,11]
activations = [torch.nn.Sigmoid(), torch.nn.Sigmoid()]
lr = 1
hp = {'n_nodes': n_nodes, 'activations': activations, 'lr': lr}

##################################################################################
#Configuration
##################################################################################

if is_train:
    # The Name of the training
    Name = f'lattice39'
    for seed in SEED:
        if not os.path.exists(Name+f'-{seed}'):
            os.makedirs(Name+f'-{seed}')

    dbfile = f'data/lattice39.db'
    db = connect(dbfile)

    elements = ['Pd', 'Au']
    nelem = len(elements)
    # This is the energy of the metal in its ground state structure
    #if you don't know the energy of the ground state structure,
    # you can set it to None
    element_energy = None
```

```python
    # Allen electronegativity
    weights =[1.58, 1.92]


    Gs = [22]
    cutoff = 6.35
    g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
    g2_Rses = [0.0]


    sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]
    params_set = set_sym(elements, Gs, cutoff,
                         g2_etas=g2_etas, g2_Rses=g2_Rses,
                         weights=weights)

    N_sym = params_set[elements[0]]['num']

#################################################################################
#Training
#################################################################################

Name = f'lattice39'
if is_train:
    for seed in SEED:
        # This use the context manager to operate in the data directory
        with cd(Name+f'-{seed}'):
            pickle.dump(sym_params, open("sym_params.sav", "wb"))
            logfile = open('log.txt','w+')
            resultfile = open('result.txt','w+')

            if os.path.exists('test.sav'):
                logfile.write('Did not calculate symfunctions.\n')
            else:
                data_dict = snn2sav(db, Name, elements, params_set,
                                    element_energy=element_energy)
                train_dict = train_test_split(data_dict,1-test_percent,seed=seed)
                train_val_split(train_dict,1-val_percent,seed=seed)

            logfile.flush()

            train_dict = torch.load('final_train.sav')
            val_dict = torch.load('final_val.sav')
            test_dict = torch.load('test.sav')
            scaling = get_scaling(train_dict, fp_scale_method, e_scale_method)


            n_nodes = hp['n_nodes']
            activations = hp['activations']
            lr = hp['lr']
            model = MultiLayerNet(N_sym, n_nodes, activations, nelem, scaling=scaling)
            if opt_method == 'lbfgs':
                optimizer = torch.optim.LBFGS(model.parameters(), lr=lr,
                                              max_iter=max_iter, history_size=history_size,
                                              line_search_fn=line_search_fn)

            results = train(train_dict, val_dict,
                            model,
                            opt_method, optimizer,
                            E_coeff, F_coeff,
                            epoch, val_interval,
                            n_val_stop,
                            convergence, is_force,
                            logfile)
            [loss, E_MAE, F_MAE, v_loss, v_E_MAE, v_F_MAE] = results

            test_results = evaluate(test_dict, E_coeff, F_coeff, is_force)
            [test_loss, test_E_MAE, test_F_MAE] =test_results
            resultfile.write(f'Hyperparameter: n_nodes = {n_nodes}, activations = {activations}, lr = {lr}\n')
```

```
            resultfile.write(f'loss = {loss}, E_MAE = {E_MAE}, F_MAE = {F_MAE}.\n')
            resultfile.write(f'v_loss = {v_loss}, v_E_MAE = {v_E_MAE}, v_F_MAE = {v_F_MAE}.\n')
            resultfile.write(f'test_loss = {test_loss}, test_E_MAE = {test_E_MAE}, test_F_MAE = {test_F_MAE}.\n')


            logfile.close()
            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".

```
import sys

sys.path.append("../SimpleNN")
sys.path.append("../")

import os
from ase.db import connect
from ContextManager import cd
from preprocess import train_test_split, train_val_split, get_scaling, CV
from preprocess import snn2sav
from fp_calculator import set_sym, calculate_fp


# min_max will scale fingerprints to (0,1)
fp_scale_method = 'min_max'
e_scale_method = 'min_max'


test_percent = 0.2
# Pecentage from train+val
val_percent = 0.2

# Training model configuration
SEED = [2]

#################################################################################
#Split Predict-4.0 dataset
#################################################################################


Name = f'lattice40_pred'

for seed in SEED:
    if not os.path.exists(Name+f'-{seed}'):
        os.makedirs(Name+f'-{seed}')

dbfile = 'data/lattice40.db'
db = connect(dbfile)

elements = ['Pd', 'Au']
nelem = len(elements)

element_energy = None
weights =[1.58, 1.92]

Gs = [22]
```

```
cutoff = 6.35
g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
g2_Rses = [0.0]


sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]
params_set = set_sym(elements, Gs, cutoff,
                         g2_etas=g2_etas, g2_Rses=g2_Rses,
                          weights=weights)
N_sym = params_set[elements[0]]['num']

with cd(Name+f'-{seed}'):
    data_dict = snn2sav(db, Name, elements, params_set,
                                    element_energy=element_energy)

    train_dict = train_test_split(data_dict,1-0.2,seed=seed)
    train_val_split(train_dict,1-0.2,seed=seed)

################################################################################
#Split Predict-4.1 dataset
################################################################################

Name = f'lattice41_pred'

for seed in SEED:
    if not os.path.exists(Name+f'-{seed}'):
        os.makedirs(Name+f'-{seed}')

dbfile = 'data/lattice41.db'
db = connect(dbfile)

with cd(Name+f'-{seed}'):
    data_dict = snn2sav(db, Name, elements, params_set,
                                    element_energy=element_energy)

    train_dict = train_test_split(data_dict,1-0.2,seed=seed)
    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: .

```
import torch
from uncert import evaluate_uncert
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats.distributions import t
from Batch import batch_pad
from matplotlib.ticker import StrMethodFormatter

#get inverse fisher information
def get_pcov(h):
    eigs0 = np.linalg.eigvalsh(h)[0]
    if (eigs0 <0):
        eps = max(1e-5, eigs0*-1.05)
    else:
        eps = 1e-5
    j = np.linalg.pinv(h + eps*np.identity(h.shape[0]))
    pcov1 = j*alpha
    u, v = np.linalg.eigh(pcov1)
    return v @ np.diag(np.maximum(u,0)) @ v.T


def flatten_gprime(agrad):
```

```
    cnt = 0
    for g in agrad:
        g_vector = g.contiguous().view(-1) if cnt ==0 else torch.cat([g_vector, g.contiguous().view(-1)])
        cnt = 1
    return g_vector

#get uncertainties for a dataset
def get_uncerts(name, data_dict):
    model = torch.load(name)
    scaling = model.scaling
    gmin = scaling['gmin']
    gmax = scaling['gmax']
    emin = scaling['emin']
    emax = scaling['emax']

    ids = np.array(list(data_dict.keys()))
    batch_info = batch_pad(data_dict,ids)
    b_fp = batch_info['b_fp']

    b_e_mask = batch_info['b_e_mask']
    b_fp.requires_grad = True
    sb_fp = (b_fp - gmin) / (gmax - gmin)

    N_atoms = batch_info['N_atoms'].view(-1)
    b_e = batch_info['b_e'].view(-1)
    b_f = batch_info['b_f']

    Atomic_Es = model(sb_fp)
    E_predict = torch.sum(Atomic_Es * b_e_mask, dim = [1,2])
    E_predict = E_predict/N_atoms
    E_predict = E_predict * (emax - emin) + emin

    uncerts = []
    for i, ei in enumerate(E_predict):
        gprime = torch.autograd.grad(ei, model.parameters(), create_graph=True, retain_graph=True)
        gprime = flatten_gprime(gprime).detach().numpy()
        se = gprime @ pcov @ gprime
        uncerts += [(np.sqrt(se), np.sqrt(se + rmse.item()**2), np.linalg.norm(gprime))]
    uncerts = np.array(uncerts)
    return uncerts


Name = 'lattice39-2'

#load datasets
train_dict = torch.load(f'{Name}/final_train.sav')
val_dict = torch.load(f'{Name}/final_val.sav')
test_dict = torch.load(f'{Name}/test.sav')

#get NN predictions, RMSE, hessian
pred_e, actual_e, rmse, h = evaluate_uncert(f'{Name}/best_model',train_dict, True)
h = h.detach().numpy()
pred_e_val, actual_e_val, rmse_val = evaluate_uncert(f'{Name}/best_model',val_dict, False)
pred_e_test, actual_e_test, rmse_test = evaluate_uncert(f'{Name}/best_model',test_dict, False)


ndata = pred_e.shape[0]
alpha = rmse.item()**2
pcov = get_pcov(h)

#get uncertainties
uncerts_val = get_uncerts(f'{Name}/best_model',val_dict)
uncerts_train = get_uncerts(f'{Name}/best_model',train_dict)
uncerts_test = get_uncerts(f'{Name}/best_model',test_dict)

################################################################################
#Parity Plot
################################################################################
```

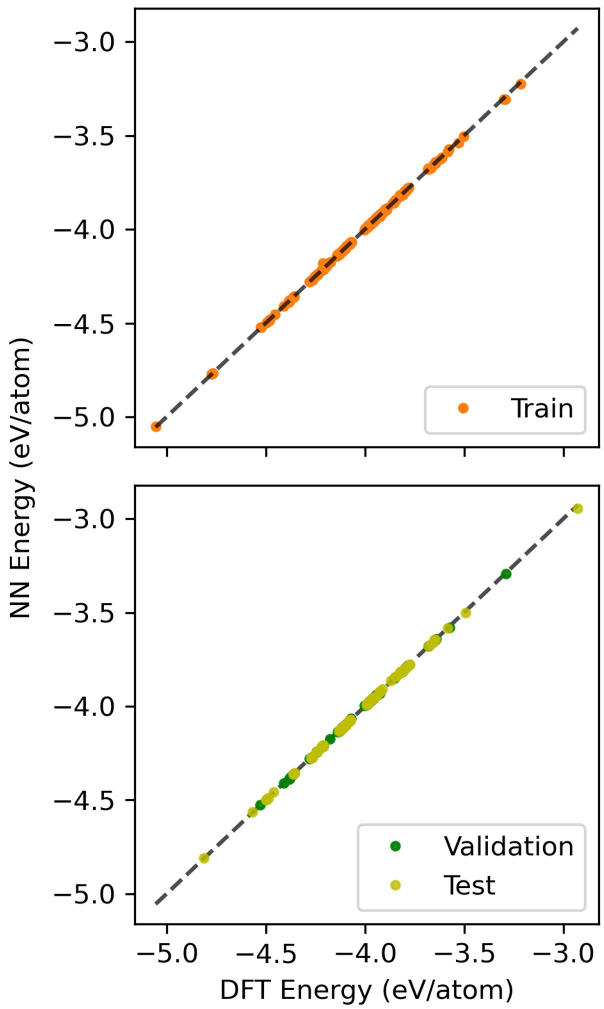
```python

plt.clf()
plt.rcParams.update({'font.size': 10})
fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)

fig.set_size_inches(3.25, 5.5)

eline = np.linspace(np.min(np.concatenate((actual_e, actual_e_test))),
                    np.max(np.concatenate((actual_e, actual_e_test))), 10)

#ax[0].set_title(' ')
ax[0].plot(actual_e, pred_e, '.',color='tab:orange', alpha=1, label='Train')
#ax[0].set_xlabel(' ')
ax[0].legend(loc='lower right')
ax[0].plot(eline, eline,'k--',alpha=0.7)

ax[1].plot(eline, eline,'k--',alpha=0.7)
ax[1].plot(actual_e_val, pred_e_val, '.',color='g', alpha=0.9, label='Validation')
ax[1].plot(actual_e_test, pred_e_test, '.',color='y', alpha=0.8, label='Test')
ax[1].legend(loc='lower right')

plt.figtext( 0.01, 0.4, "NN Energy (eV/atom)", rotation='vertical', size=10)
ax[1].set_xlabel('DFT Energy (eV/atom)')
plt.tight_layout()
plt.subplots_adjust(left=0.21)
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'subplotparityslides-energy-only.{ext}', dpi=300)
print('''#+attr_org: :width 600
#+caption: Figure 3
[[./subplotparityslides-energy-only.png]]''')

################################################################################
# Distribution of uncertainties
################################################################################

plt.clf()
plt.figure(figsize=(3.25, 3))
plt.hist(uncerts_train[:,0], label='Train', density=True, alpha=0.5, color='tab:orange')
plt.hist(uncerts_val[:,0], label='Validation', density='True', alpha=0.5, color='g')
plt.hist(uncerts_test[:,0], label='Test', density='True', alpha=0.5, color='y')
plt.legend()
plt.xlabel('Standard Error Confidence (eV/atom)')
plt.ylabel('Density')
plt.locator_params(axis='x', nbins=7)
plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.3f}'))
plt.tight_layout()
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'hist-uncerts-pot1.{ext}', dpi=300)
print('''#+attr_org: :width 600
#+caption: Figure 4
[[./hist-uncerts-pot1.png]]''')


################################################################################
# Parity plot with 95% prediction interval
################################################################################

plt.clf()
plt.figure(figsize=(3.25, 4.0))
tval = t.ppf(0.975, ndata)
plt.errorbar(actual_e_test, pred_e_test, yerr=tval*uncerts_test[:,1], fmt='y_',
             ecolor='m', label='Test, 95% prediction')

plt.xlabel('DFT Energy (eV/atom)')
plt.ylabel('NN Energy (eV/atom)')
plt.plot([np.min(actual_e_test), np.max(actual_e_test)],
         [np.min(actual_e_test),
          np.max(actual_e_test)],'k--', alpha=0.7, linewidth=0.3)
```

```python
plt.legend()
plt.tight_layout()
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'parity-errorbar-test-pot1-prediction.{ext}', dpi=300)
print('''#+attr_org: :width 600
#+caption: Figure 5
[[./parity-errorbar-test-pot1-prediction.png]]''')

##################################################################################
#Inference on predict-4.0 and 4.1 dataset
##################################################################################

data_dict = torch.load(f'lattice40_pred-2/test.sav')
pred_e_40p, actual_e_40p, rmse_40p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
uncerts_40p = get_uncerts(f'{Name}/best_model',data_dict)

data_dict = torch.load(f'lattice41_pred-2/test.sav')
pred_e_41p, actual_e_41p, rmse_41p = evaluate_uncert(f'{Name}/best_model', data_dict, False)
uncerts_41p = get_uncerts(f'{Name}/best_model',data_dict)

#make plot

plt.clf()
plt.rc('legend', fontsize=10)
fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)
fig.set_size_inches(3.25, 4.5)
#ax[0].set_title(' ')
ax[0].errorbar(actual_e_40p, pred_e_40p, yerr = tval * uncerts_40p[:,1], color='tab:pink',
               fmt = '_', ecolor='r', label='Predict 4.0, \n95% prediction')
#ax[0].set_xlabel(' ')
#ax[0].set_ylabel('NN Energy (eV/atom)')
ax[0].legend(loc='upper left')
eline = np.linspace(np.min(np.concatenate((actual_e_40p, actual_e_41p))),
                    np.max(np.concatenate((actual_e_40p, actual_e_41p))), 10)
ax[0].plot(eline, eline,'k--', alpha=0.8, linewidth=0.5)

ax[1].errorbar(actual_e_41p, pred_e_41p, yerr =  tval * uncerts_41p[:,1],
               fmt = 'b_', ecolor='c', label='Predict 4.1, \n95% prediction')
ax[1].legend()
ax[1].plot(eline, eline,'k--', alpha=0.7, linewidth=0.5)
ax[1].set_xlabel("DFT Energy (eV/atom)")
plt.figtext( 0.01, 0.4, "NN Energy (eV/atom)", rotation='vertical', size=10)

plt.tight_layout()
plt.subplots_adjust(left=0.21)
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'subplot-parity-40-41-pot-prediction.{ext}', dpi=300)
print('''#+attr_org: :width 600
#+caption: Figure 6
[[./subplot-parity-40-41-pot-prediction.png]]\n''')

##################################################################################
# Uncertainty vs True Error Scatterplot
##################################################################################


def scatter_hist(x, y, ax, ax_histx, ax_histy, label, color=None):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the scatter plot:
    ax.scatter(x, y, alpha=0.5, label=label, color=color)

    # now determine nice limits by hand:
    binwidth = 0.0001
    xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
    lim = (int(xymax / binwidth) + 1) * binwidth
```

```
    #bins = np.arange(0, lim + binwidth, binwidth)
    ax_histx.hist(x, alpha=0.5, color=color, density=True)
    ax_histy.hist(y, orientation='horizontal', alpha=0.5, color=color, density=True)

fig = plt.figure(figsize=(3.25, 4.))
# Add a gridspec with two rows and two columns and a ratio of 2 to 7 between
# the size of the marginal axes and the main axes in both directions.
# Also adjust the subplot parameters for a square plot.
gs = fig.add_gridspec(2, 2,  width_ratios=(7, 2), height_ratios=(2, 7),
                      left=0.11, right=0.98, bottom=0.07, top=0.97, wspace=0.05,
                      hspace=0.05)

ax = fig.add_subplot(gs[1, 0])
ax_histx = fig.add_subplot(gs[0, 0], sharex=ax)
ax_histy = fig.add_subplot(gs[1, 1], sharey=ax)

# use the previously defined function

scatter_hist(np.absolute(actual_e_test-pred_e_test),  uncerts_test[:,0],
             ax, ax_histx, ax_histy, 'Test', 'y')

scatter_hist(np.absolute(actual_e_40p-pred_e_40p), uncerts_40p[:,0],
             ax, ax_histx, ax_histy, 'Predict 4.0', 'tab:pink')

scatter_hist(np.absolute(pred_e_41p-actual_e_41p), uncerts_41p[:,0],
             ax, ax_histx, ax_histy, 'Predict 4.1')


ax.set_xlabel('Absolute Error Energy (eV/atom)')
ax.set_ylabel('Standard Error Confidence (eV/atom)')
ax.legend()
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'uncert-v-error-w-hist-pot1-origw-test.{ext}', dpi=300, bbox_inches='tight')
print('''#+attr_org: :width 600
#+caption: Figure 8
[[./uncert-v-error-w-hist-pot1-origw-test.png]]''')
```

6 Fingerprints

```
import torch
from uncert import get_fps
import matplotlib.pyplot as plt

Name = 'lattice39-2'
train_dict = torch.load(f'{Name}/final_train.sav')
fp_train, e_mask_train = get_fps(f'{Name}/best_model', train_dict)

data_dict = torch.load(f'lattice40_pred-2/test.sav')
fp_40, e_mask_40 = get_fps(f'{Name}/best_model', data_dict)

data_dict = torch.load(f'lattice41_pred-2/test.sav')
fp_41, e_mask_41 = get_fps(f'{Name}/best_model', data_dict)

plt.rcParams.update({'font.size': 10})
plt.figure(figsize=(3.25, 4.))
for i in range(2):
    for j in range(4):
        plt.clf()
        plt.hist(fp_train[e_mask_train[:,:,i]==1][:,j],alpha=0.5, density=True,label='Train', color='y')
        plt.hist(fp_40[e_mask_40[:,:,i]==1][:,j],alpha=0.5, density=True,label='Predict 4.0', color='tab:pink')

        plt.hist(fp_41[e_mask_41[:,:,i]==1][:,j],alpha=0.5, density=True,label='Predict 4.1')
        plt.xlabel('Fingerprint Value')
        plt.ylabel('Density')
        plt.legend()
```



NN Energy (eV/atom)
DFT Energy (eV/atom)
Train
Validation
Test
−3.0
−3.5
−4.0
−4.5
−5.0

Figure 3

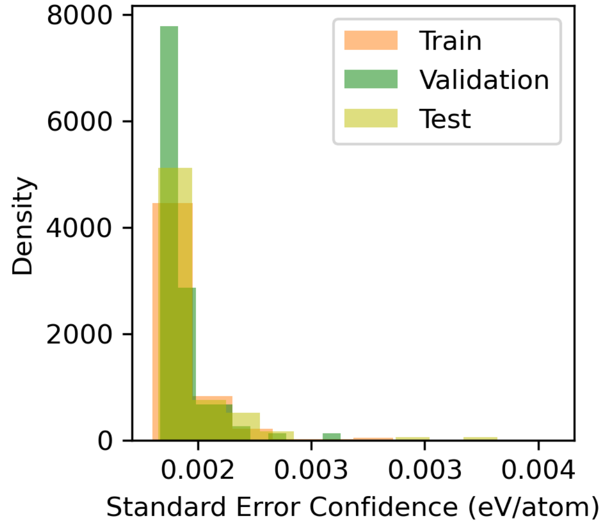


Figure 4

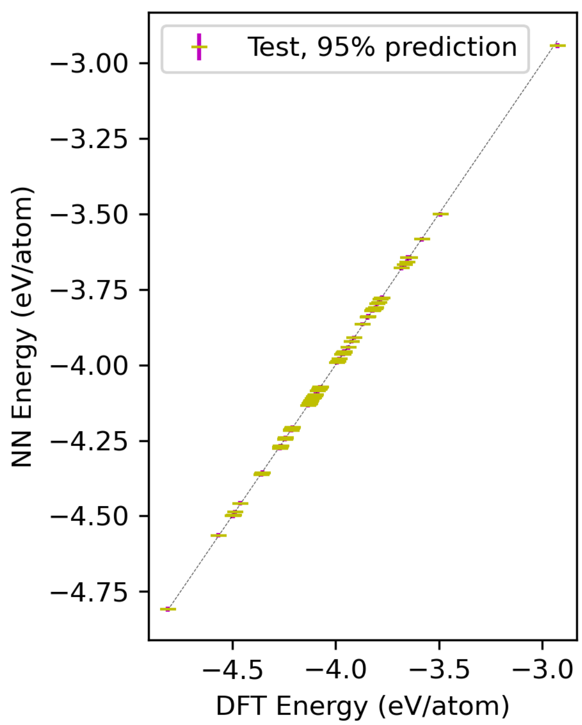


Figure 5

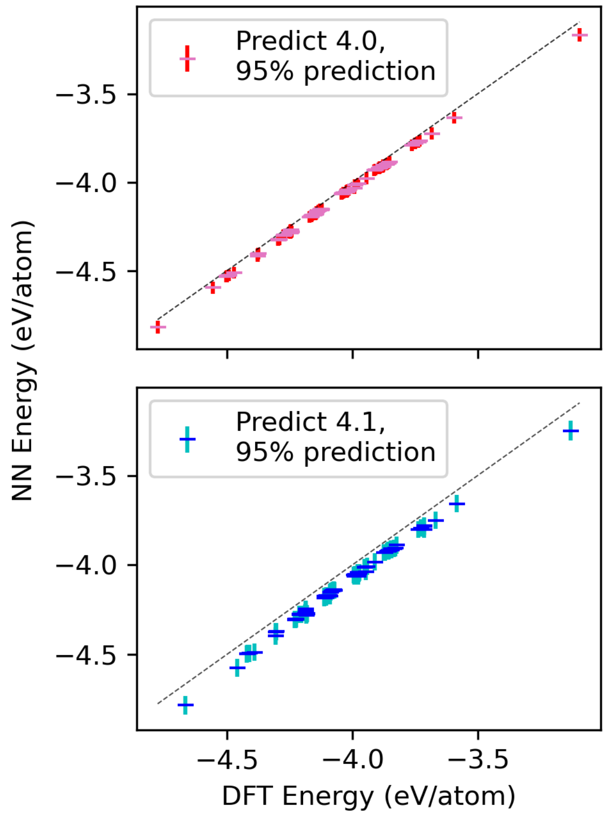


Figure 6

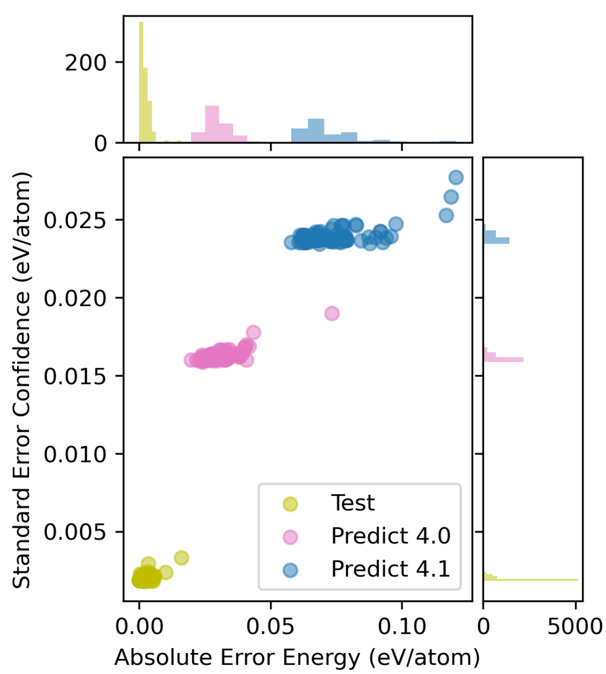


Figure 8

```
        plt.tight_layout()
        for ext in ['png', 'eps', 'pdf']:
            plt.savefig(f'fps-hist-el{i}-fp{j}.{ext}', dpi=300)

print(f'''#+attr_org: :width 600
#+caption: Figure 7
[[./fps-hist-el0-fp0.png]]''')
```

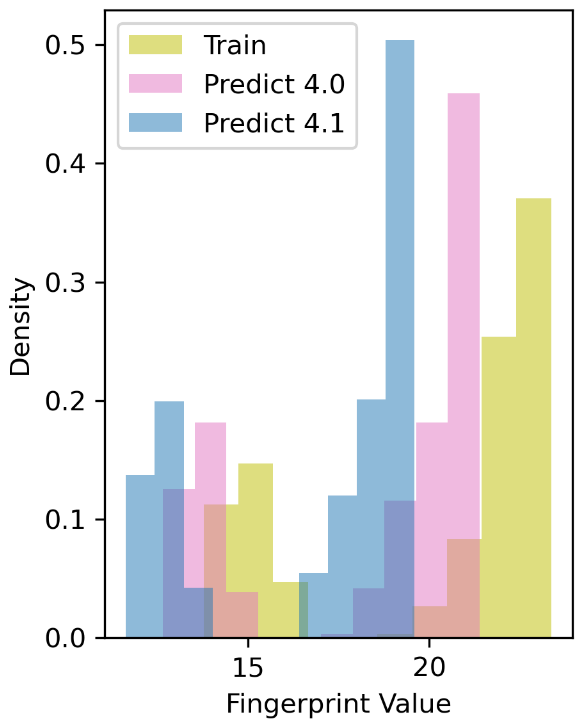


Figure 7

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".

```
import sys

sys.path.append("../SimpleNN")
sys.path.append("../")

import os
from ase.db import connect
import torch
from ContextManager import cd
from preprocess import train_test_split, train_val_split, get_scaling, CV
from preprocess import snn2sav
from NN import MultiLayerNet
from train import train, evaluate
```

```python
from fp_calculator import set_sym, calculate_fp
import pickle

is_train = True
is_transfer = False
is_force = True

if is_train and is_transfer:
    raise ValueError('train and transfer could not be true at the same time.')

##################################################################################
#Hyperparameters
##################################################################################
E_coeff = 100
if is_force:
    F_coeff = 1
else:
    F_coeff = 0

val_interval = 10
n_val_stop = 10
epoch = 3000

opt_method = 'lbfgs'


if opt_method == 'lbfgs':
    history_size = 100
    lr = 1
    max_iter = 10
    line_search_fn = 'strong_wolfe'


convergence = {'E_cov':0.0005,'F_cov':0.005}

# min_max will scale fingerprints to (0,1)
fp_scale_method = 'min_max'
e_scale_method = 'min_max'


test_percent = 0.2
# Pecentage from train+val
val_percent = 0.2

# Training model configuration
SEED = [2]
n_nodes = [11,11]
activations = [torch.nn.Sigmoid(), torch.nn.Sigmoid()]

lr = 1
hp = {'n_nodes': n_nodes, 'activations': activations, 'lr': lr}

##################################################################################
#Configuration
##################################################################################

elements = ['Pd', 'Au']
nelem = len(elements)

element_energy = None
weights =[1.58, 1.92]

Gs = [22]
cutoff = 6.35
g2_etas = [0.00, 0.10713, 0.285686, 0.892769]
g2_Rses = [0.0]

sym_params = [Gs, cutoff, g2_etas, g2_Rses, elements, weights, element_energy]
```

```
params_set = set_sym(elements, Gs, cutoff,
                         g2_etas=g2_etas, g2_Rses=g2_Rses,
                          weights=weights)
N_sym = params_set[elements[0]]['num']

#################################################################################
#Training
#################################################################################

Name = 'lattice39-40-41'

if is_train:
    for seed in SEED:
        # This use the context manager to operate in the data directory

        if not os.path.exists(Name+f'-{seed}'):
            os.makedirs(Name+f'-{seed}')

        with cd(Name+f'-{seed}'):
            pickle.dump(sym_params, open("sym_params.sav", "wb"))
            logfile = open('log.txt','w+')
            resultfile = open('result.txt','w+')

            if os.path.exists('test.sav'):
                logfile.write('Did not calculate symfunctions.\n')
            else:
                #this part is to concatenate the train-data subsets together.
                train_dict1 = torch.load('../lattice39-2/final_train.sav')
                train_dict2 = torch.load('../lattice40_pred-2/final_train.sav')
                train_dict3 = torch.load('../lattice41_pred-2/final_train.sav')
                train_dict = dict(train_dict1)
                new_dict = {k+1000: v for k, v in train_dict2.items()}
                train_dict.update(new_dict)
                new_dict = {k+2000: v for k, v in train_dict3.items()}
                train_dict.update(new_dict)

                val_dict1 = torch.load('../lattice39-2/final_val.sav')
                val_dict2 = torch.load('../lattice40_pred-2/final_val.sav')
                val_dict3 = torch.load('../lattice41_pred-2/final_val.sav')
                val_dict = dict(val_dict1)
                new_dict = {k+1000: v for k, v in val_dict2.items()}
                val_dict.update(new_dict)
                new_dict = {k+2000: v for k, v in val_dict3.items()}
                val_dict.update(new_dict)


                test_dict1 = torch.load('../lattice39-2/test.sav')
                test_dict2 = torch.load('../lattice40_pred-2/test.sav')
                test_dict3 = torch.load('../lattice41_pred-2/test.sav')
                test_dict = dict(test_dict1)
                new_dict = {k+1000: v for k, v in test_dict2.items()}
                test_dict.update(new_dict)
                new_dict = {k+2000: v for k, v in test_dict3.items()}
                test_dict.update(new_dict)



                torch.save(train_dict,'final_train.sav')
                torch.save(val_dict, 'final_val.sav')
                torch.save(test_dict, 'test.sav')

            scaling = get_scaling(train_dict, fp_scale_method, e_scale_method)

            n_nodes = hp['n_nodes']
            activations = hp['activations']
            lr = hp['lr']
            #model = torch.load('../lattice39-2/best_model')
            model = MultiLayerNet(N_sym, n_nodes, activations, nelem, scaling=scaling)
```

```
            if opt_method == 'lbfgs':
                optimizer = torch.optim.LBFGS(model.parameters(), lr=lr,
                                              max_iter=max_iter, history_size=history_size,
                                              line_search_fn=line_search_fn)

            results = train(train_dict, val_dict,
                            model,
                            opt_method, optimizer,
                            E_coeff, F_coeff,
                            epoch, val_interval,
                            n_val_stop,
                            convergence, is_force,
                            logfile)
            [loss, E_MAE, F_MAE, v_loss, v_E_MAE, v_F_MAE] = results

            test_results = evaluate(test_dict, E_coeff, F_coeff, is_force)
            [test_loss, test_E_MAE, test_F_MAE] =test_results
            resultfile.write(f'Hyperparameter: n_nodes = {n_nodes}, activations = {activations}, lr = {lr}\n')
            resultfile.write(f'loss = {loss}, E_MAE = {E_MAE}, F_MAE = {F_MAE}.\n')
            resultfile.write(f'v_loss = {v_loss}, v_E_MAE = {v_E_MAE}, v_F_MAE = {v_F_MAE}.\n')
            resultfile.write(f'test_loss = {test_loss}, test_E_MAE = {test_E_MAE}, test_F_MAE = {test_F_MAE}.\n')


            logfile.close()
            resultfile.close()
```

8 Uncertainty for retrained model

```
import torch
from uncert import evaluate_uncert
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats.distributions import t
from Batch import batch_pad

#get inverse fisher information
def get_pcov(h):
    eigs0 = np.linalg.eigvalsh(h)[0]
    if (eigs0 <0):
        eps = max(1e-5, eigs0*-1.05)
    else:
        eps = 1e-5
    j = np.linalg.pinv(h + eps*np.identity(h.shape[0]))
    pcov1 = j*alpha
    u, v = np.linalg.eigh(pcov1)
    return v @ np.diag(np.maximum(u,0)) @ v.T


def flatten_gprime(agrad):
    cnt = 0
    for g in agrad:
        g_vector = g.contiguous().view(-1) if cnt ==0 else torch.cat([g_vector, g.contiguous().view(-1)])
        cnt = 1
    return g_vector

#get uncertainties for a dataset
def get_uncerts(name, data_dict):
    model = torch.load(name)
    scaling = model.scaling
    gmin = scaling['gmin']
    gmax = scaling['gmax']
    emin = scaling['emin']
    emax = scaling['emax']

    ids = np.array(list(data_dict.keys()))
    batch_info = batch_pad(data_dict,ids)
```

```
    b_fp = batch_info['b_fp']

    b_e_mask = batch_info['b_e_mask']
    b_fp.requires_grad = True
    sb_fp = (b_fp - gmin) / (gmax - gmin)

    N_atoms = batch_info['N_atoms'].view(-1)
    b_e = batch_info['b_e'].view(-1)
    b_f = batch_info['b_f']

    Atomic_Es = model(sb_fp)
    E_predict = torch.sum(Atomic_Es * b_e_mask, dim = [1,2])
    E_predict = E_predict/N_atoms
    E_predict = E_predict * (emax - emin) + emin

    uncerts = []
    for i, ei in enumerate(E_predict):
        gprime = torch.autograd.grad(ei, model.parameters(), create_graph=True, retain_graph=True)
        gprime = flatten_gprime(gprime).detach().numpy()
        se = gprime @ pcov @ gprime
        uncerts += [(np.sqrt(se), np.sqrt(se + rmse.item()**2), np.linalg.norm(gprime))]
    uncerts = np.array(uncerts)
    return uncerts

Name = 'lattice39-40-41-2'

train_dict = torch.load(f'{Name}/final_train.sav')
val_dict = torch.load(f'{Name}/final_val.sav')
test_dict = torch.load(f'{Name}/test.sav')

pred_e, actual_e, rmse, h = evaluate_uncert(f'{Name}/best_model',train_dict, True)
h = h.detach().numpy()
pred_e_val, actual_e_val, rmse_val = evaluate_uncert(f'{Name}/best_model',val_dict, False)
pred_e_test, actual_e_test, rmse_test = evaluate_uncert(f'{Name}/best_model',test_dict, False)

ndata = pred_e.shape[0]
alpha = rmse.item()**2
pcov = get_pcov(h)

uncerts_val = get_uncerts(f'{Name}/best_model',val_dict)
uncerts_train = get_uncerts(f'{Name}/best_model',train_dict)
uncerts_test = get_uncerts(f'{Name}/best_model',test_dict)


##################################################################################
#Parity plot after retraining
##################################################################################

data_dict = torch.load(f'lattice40_pred-2/test.sav')
pred_e_40p, actual_e_40p, rmse_40p = evaluate_uncert(f'{Name}/best_model',data_dict, False)
uncerts_40p = get_uncerts(f'{Name}/best_model',data_dict)

data_dict = torch.load(f'lattice41_pred-2/test.sav')
pred_e_41p, actual_e_41p, rmse_41p = evaluate_uncert(f'{Name}/best_model',data_dict, False)
uncerts_41p = get_uncerts(f'{Name}/best_model',data_dict)

tval = t.ppf(0.975, ndata)
plt.clf()

plt.rcParams.update({'font.size': 10})
fig, ax = plt.subplots(ncols=1, nrows=2, sharex='col', sharey=False)
fig.set_size_inches(3.25, 6.5)
#ax[0].set_title(' ')
ax[0].errorbar(actual_e, pred_e, yerr = tval * uncerts_train[:,1], fmt = 'y_', ecolor='m',
               label='Train, \n95% prediction')
#ax[0].set_xlabel(' ')
#ax[0].set_ylabel('NN Energy (eV/atom)')
ax[0].legend()
```

```
eline = np.linspace(np.min(actual_e), np.max(actual_e_40p), 10)
ax[0].plot(eline, eline,'k--', alpha=0.7, linewidth=0.3)

ax[1].errorbar(actual_e_40p, pred_e_40p, yerr = tval * uncerts_40p[:,1], color='tab:pink',
               fmt = '_', ecolor='b', label='Predict 4.0, 4.1,\n95% prediction')
ax[1].errorbar(actual_e_41p, pred_e_41p, yerr =  tval * uncerts_41p[:,1], color='tab:pink',
               fmt = '_', ecolor='b', label='')
ax[1].legend(loc='upper left')
ax[1].plot(eline, eline,'k--', alpha=0.7, linewidth=0.3)
ax[1].set_xlabel('DFT Energy (eV/atom)')
plt.figtext( 0.01, 0.42, "NN Energy (eV/atom)", rotation='vertical', size=10)
plt.tight_layout()
plt.subplots_adjust(left=0.23)
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'subplot-parity-40-41-pot2-pred-v2.{ext}', dpi=300)
print('''#+attr_org: :width 600
#+caption: Figure 9
[[./subplot-parity-40-41-pot2-pred-v2.png]]
''')

##################################################################################
#Uncertainty vs True Error Scatterplot
##################################################################################

def scatter_hist(x, y, ax, ax_histx, ax_histy, label, color=None):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the scatter plot:
    ax.scatter(x, y, alpha=0.5, label=label, color=color)

    # now determine nice limits by hand:
    binwidth = 0.0001
    xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
    lim = (int(xymax/binwidth)+1)*binwidth

    #bins = np.arange(0, lim + binwidth, binwidth)
    ax_histx.hist(x,  alpha=0.5, color=color, density=True)
    ax_histy.hist(y,  orientation='horizontal', alpha=0.5, color=color, density=True)

fig = plt.figure(figsize=(3.25, 4.0))
# Add a gridspec with two rows and two columns and a ratio of 2 to 7 between
# the size of the marginal axes and the main axes in both directions.
# Also adjust the subplot parameters for a square plot.
gs = fig.add_gridspec(2, 2,  width_ratios=(7, 2), height_ratios=(2, 7),
                      left=0.11, right=0.98, bottom=0.07, top=0.97, wspace=0.05, hspace=0.05)

ax = fig.add_subplot(gs[1, 0])
ax_histx = fig.add_subplot(gs[0, 0], sharex=ax)
ax_histy = fig.add_subplot(gs[1, 1], sharey=ax)

# use the previously defined function
scatter_hist(np.absolute(actual_e-pred_e),  uncerts_train[:,0], ax, ax_histx, ax_histy,
             'Train', 'y')
scatter_hist(np.absolute(actual_e_40p-pred_e_40p), uncerts_40p[:,0], ax, ax_histx, ax_histy,
             'Predict 4.0', 'tab:pink')
scatter_hist(np.absolute(pred_e_41p-actual_e_41p), uncerts_41p[:,0], ax, ax_histx, ax_histy,
             'Predict 4.1')

ax.set_xlabel('Absolute Error Energy (eV/atom)')
ax.set_ylabel('Standard Error Confidence (eV/atom)')
ax.legend()
for ext in ['png', 'eps', 'pdf']:
    plt.savefig(f'uncert-v-error-w-hist-ret40-41-orig.{ext}', dpi=300, bbox_inches='tight')
print('''
#+attr_org: :width 600
#+caption: Figure 10
```

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

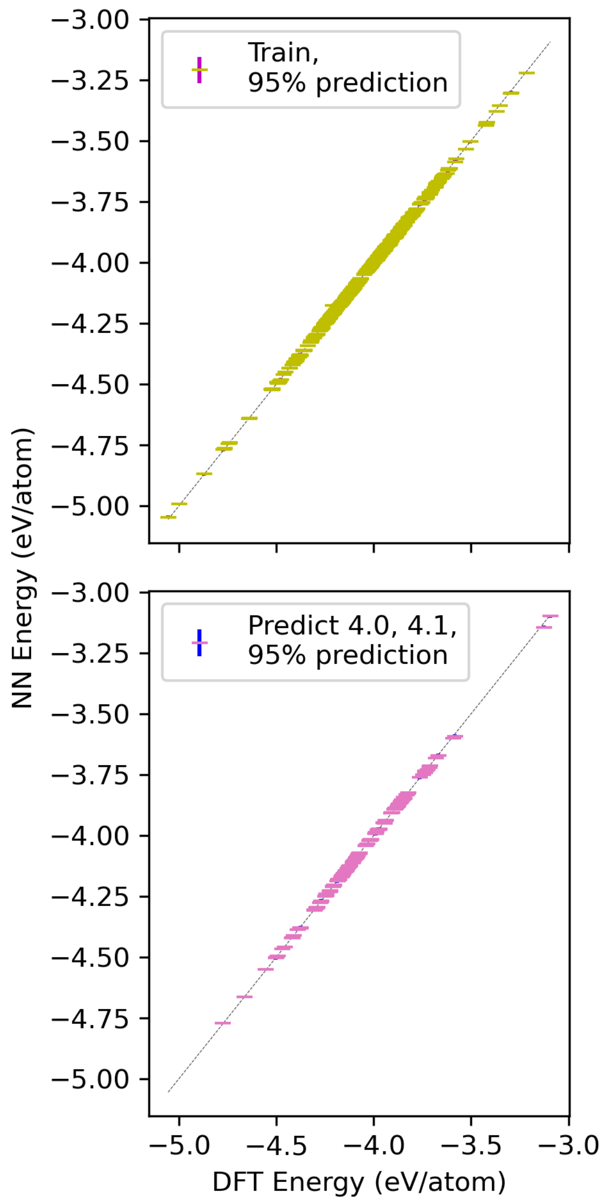


Figure 9

References

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

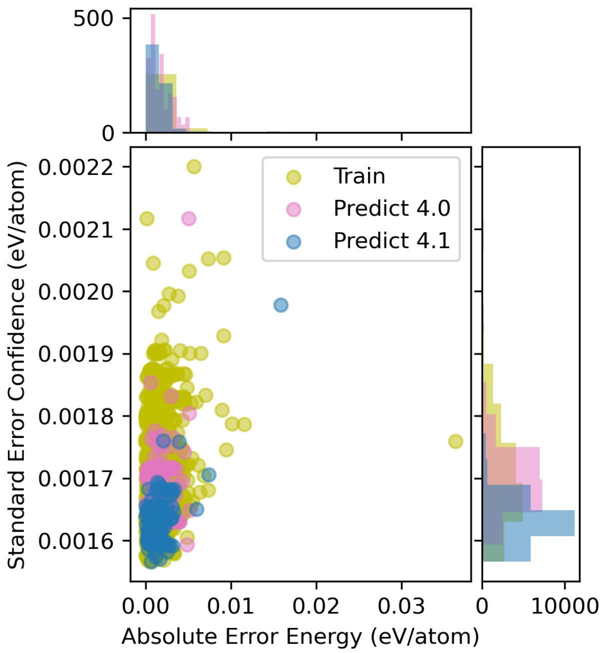

500
0
0.0022
0.0021
0.0020
0.0019
0.0018
0.0017
0.0016
Standard Error Confidence (eV/atom)
Train
Predict 4.0
Predict 4.1
0.00
0.01
0.02
0.03
Absolute Error Energy (eV/atom)
0
10000

Figure 10