Parsimonious neural networks learn interpretable physical laws

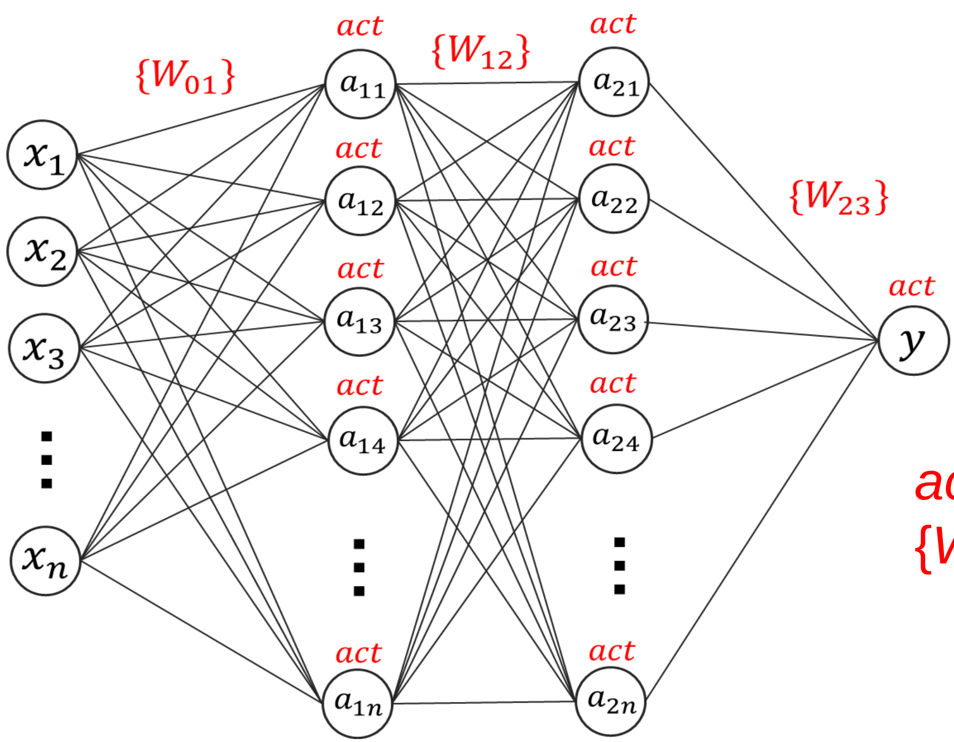
Saaketh Desai[1], Alejandro Strachan[2]

1. Centre for Integrated Nanotechnologies, Sandia National Laboratories

2. School of Materials Engineering, Purdue University

Encoding neural networks for genetic algorithms

Couple neural networks with genetic algorithms to balance interpretability and accuracy



Individual

[1,0,1,2,…, 2]

act: {linear, squared, tanh, relu, …}

$\{W_{ij}\}$: {0, 1, ½, 2, …, trainable}

linear: 0
relu: 1
tanh: 2

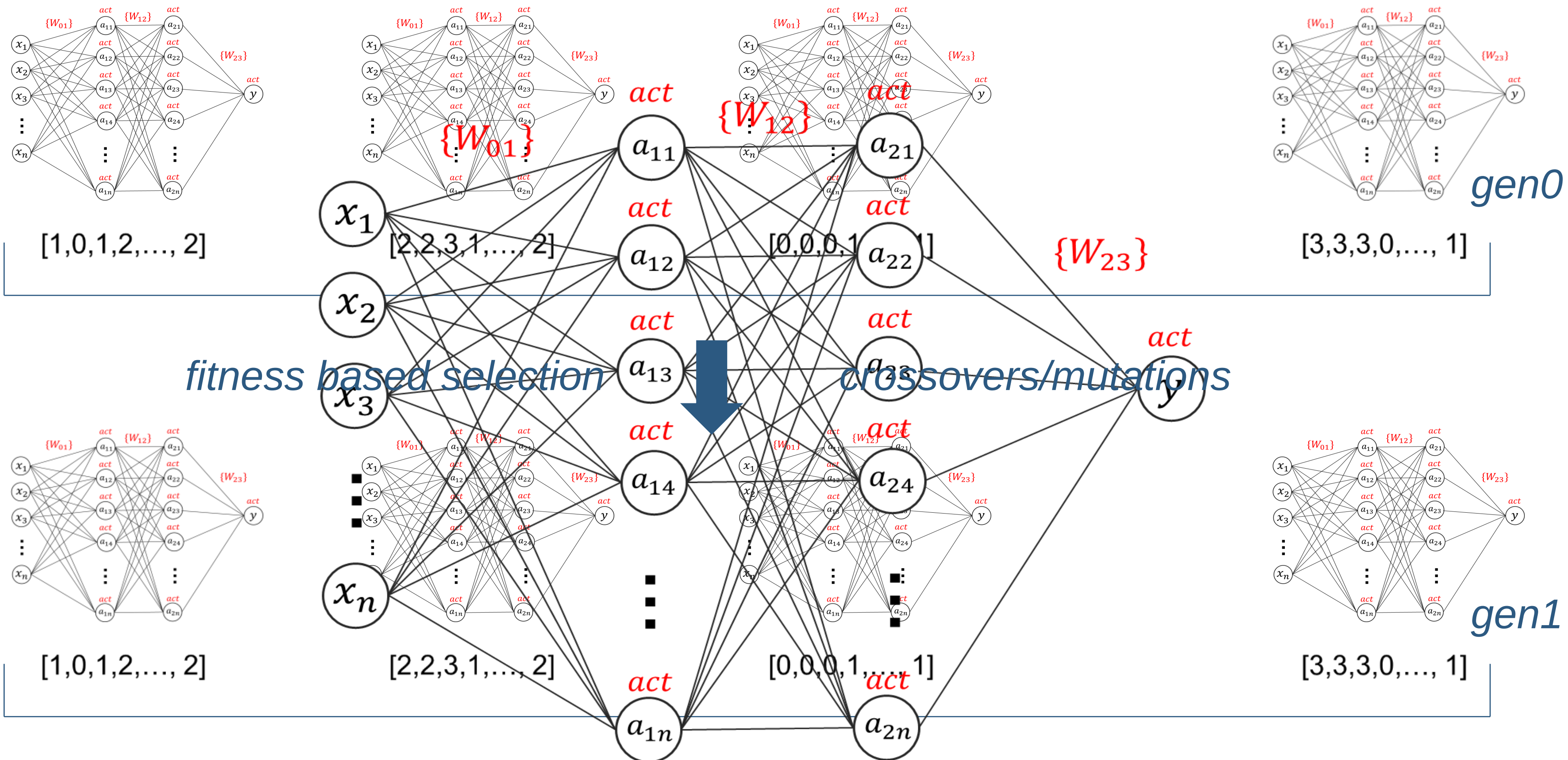
0: 0
1, ½ , 2: 1
trainable: 2

$$F = f_1(E_{test}) + p\left(\Sigma_{i=1}^{n_a} w_i^2 + \Sigma_{j=1}^{n_w} f_2(w_j)\right)$$

fitness — error on data — parsimony coefficient — simple activations — weight penalty

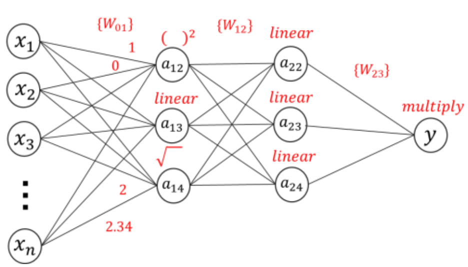
Keras

How to train a PNN?



DISTRIBUTED EVOLUTIONARY ALGORITHMS IN PYTHON

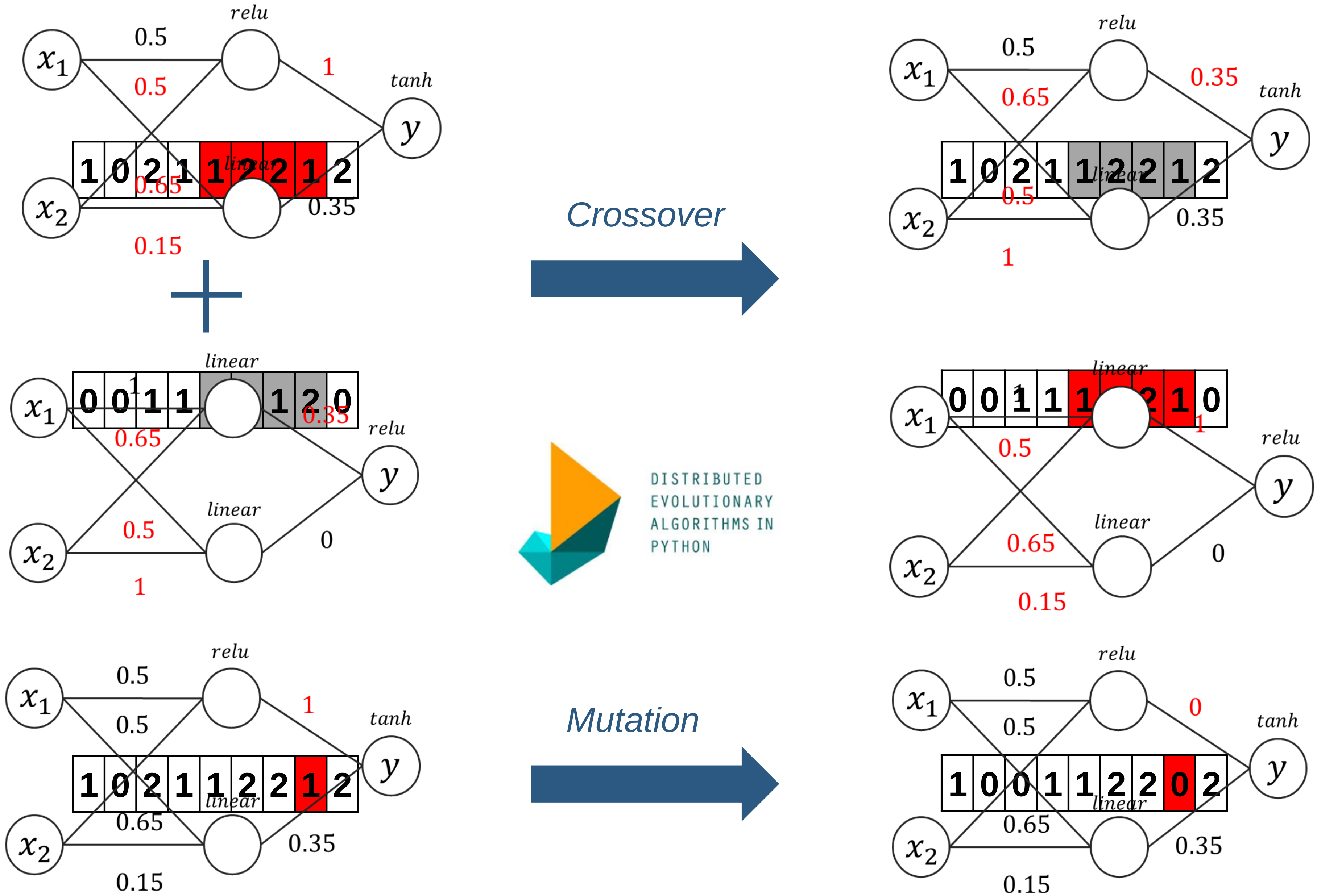
Fittest individual



Interpretable equation

genN

Genetic operations on neural networks



Parsimonious neural networks – melting point

Material	Volume	Density	Bulk modulus	Shear Modulus	Melting temp
BaZrO3	30.700078	5.983317	143.0	88.0	2813.15
Nd2O3	139.781930	6.395583	124.0	51.0	2543.15
BaO2	17.382291	5.391927	67.0	35.0	723.15

Can we predict the melting temp based on fundamental inputs?

fitting constant

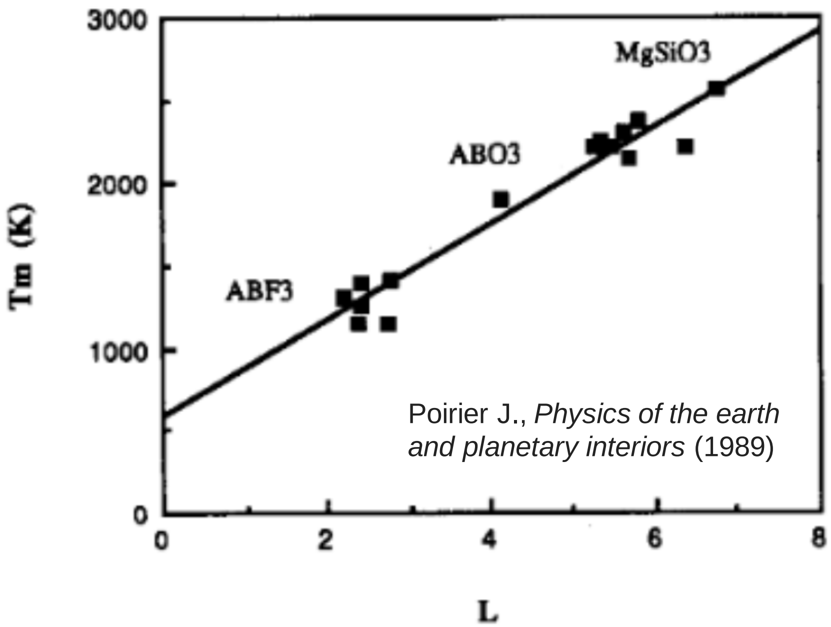
mean atomic mass

$$T_m^{lind} = \left(\frac{4\pi^2}{9h^2}\right) f^2\, a^2\, m\, T_D^2$$

interatomic spacing

Debye temperature

Lindemann law developed in 1910



Poirier J., *Physics of the earth and planetary interiors* (1989)

Can PNNs learn improved descriptions of melting from data?

Dimensional analysis on inputs

$$\theta_0 = \frac{\hbar v_m}{k_b a} \qquad \theta_1 = \frac{\hbar^2}{m a^2 k_b} \qquad \theta_2 = \frac{a^3 G}{k_b} \qquad \theta_3 = \frac{a^3 K}{k_b}$$

Temperature units

$$v_s = \sqrt{\frac{G}{\rho}} \qquad v_p = \sqrt{\frac{K + \frac{4}{3}G}{\rho}} \qquad v_m = \left[\frac{3}{\left(\frac{1}{v_p}\right)^3 + 2\left(\frac{1}{v_s}\right)^3}\right]^{\frac{1}{3}}$$

Dimensionless inputs

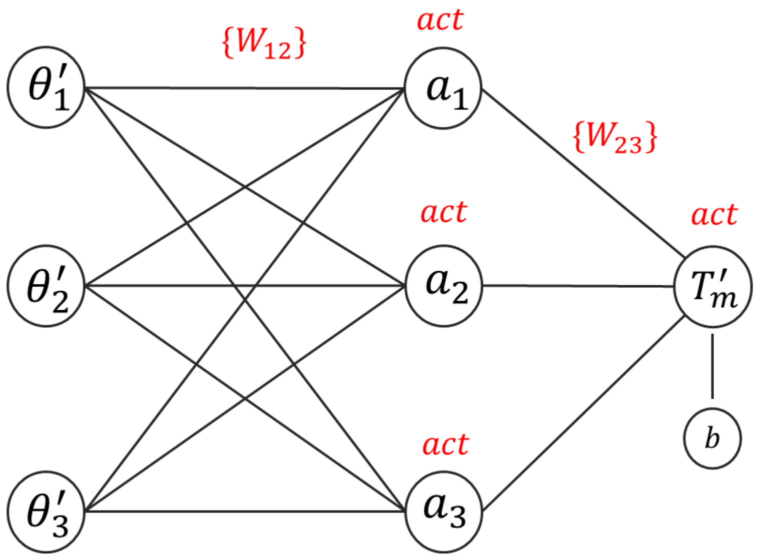
$$\theta_1' = \frac{\hbar}{m a v_m} = \frac{\theta_1}{\theta_0}$$

$$\theta_2' = \frac{a^4 G}{\hbar v_m} = \frac{\theta_2}{\theta_0}$$

$$\theta_3' = \frac{a^4 K}{\hbar v_m} = \frac{\theta_3}{\theta_0}$$

act: {linear, multiply, squared, tanh, …}

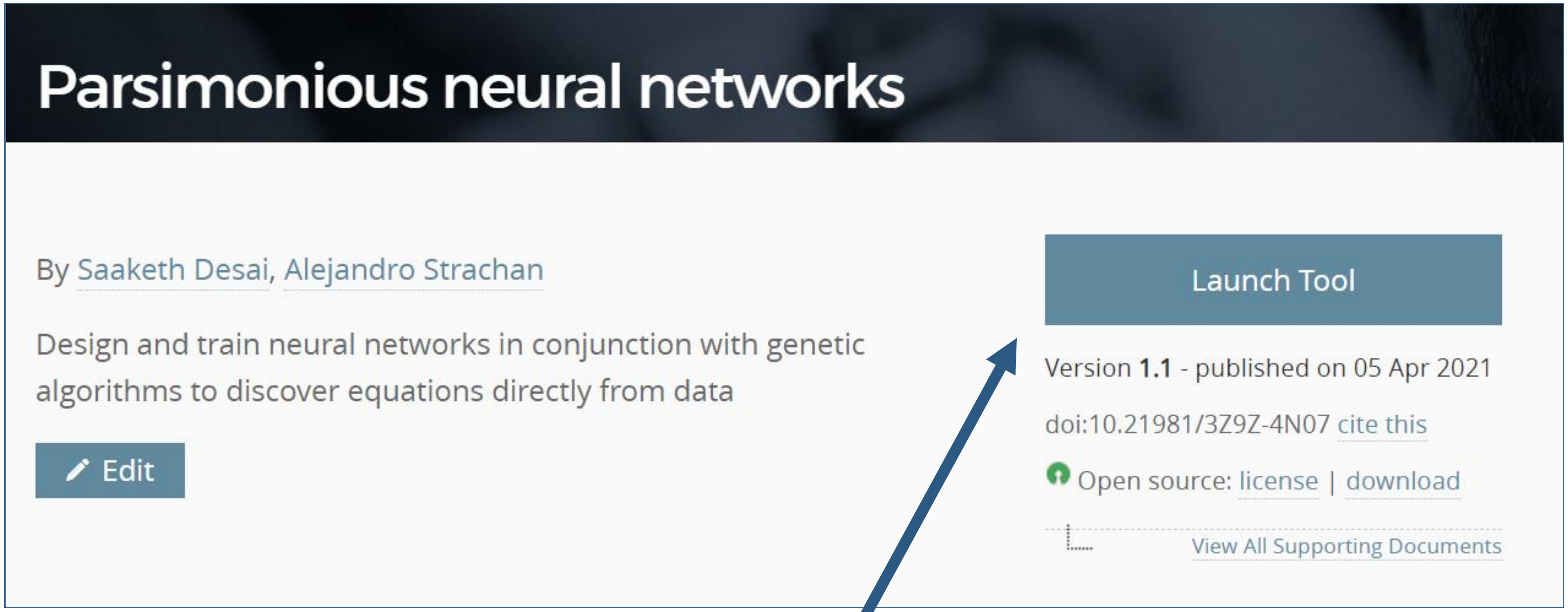
$\{W_{ij}\}$: {0, 1, …, trainable}



Launching the nanoHUB tool

Parsimonious neural networks

From your browser go to link: https://nanohub.org/tools/pnndemo/



Click on Launch Tool to begin

Landing page

Navigate to the 4th notebook to access the notebook we will be working on during the workshop

Discovering classical equations of motion using parsimonious neural networks

Saaketh Desai and *Alejandro Strachan* , School of Materials Engineering, Purdue University

These notebooks will demonstrate the use of neural networks and genetic algorithms to discover scientific equations, in this case a discretized version of the classic networks as parsimonious neural networks (PNNs) as they are designed not only to reproduce the training and testing datasets, but also learn for the simplest, most data.

- **Get started** Click on the links below to begin each tutorial.
- **Important** To exit individual tutorials and return to this page, use File -> Close and Halt. "Terminate Session" (top right) will kill your entire Jupyter session.

Designing a parsimonious neural network - non-linear potential:

- Discover the underlying equations for a particle in a non-linear external potential
- Combine the Keras and DEAP packages to drive neural network training with genetic algorithms

Evaluating a parsimonious neural network - non-linear potential:

- Evaluate a PNN model using the metrics defined while training
- Check for conservation of energy and time reversibility

Designing a parsimonious neural network - linear potential:

- Train and discover the Verlet integration scheme without using genetic algorithms for this simple case

Designing a parsimonious neural network - predict melting temperature:

- Discover melting laws directly from data

Evaluating a parsimonious neural network - predict melting temperature:

- Evaluate a PNN model to predict the melting temperature

Import libraries and read in data

```
import sys
import os
import random

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from matplotlib import pyplot as plt

import tensorflow as tf
import keras
from keras import backend as K
from keras import initializers
from keras.layers import Dense, Input, Activation, multiply
from keras.models import Sequential, Model, load_model
from keras.layers.merge import add, concatenate

from deap import base, creator, tools, algorithms
from multiprocessing import Pool
```

Import Keras layers to build custom neural networks

Import modules from the ‘deap’ package

Step 1: Read training and testing data

```
df = pd.read_csv("../data/Combined_data_v3.csv")
print (df.shape)
```

```
display(df)
```

Read in CSV data file from tool directory

Display data and compute equations

mp_id	pretty_formula	formula	density	K_VRH	G_VRH	volume_uc	volume_per_atom	volume_per_formula_unit	natoms_uc	natoms_formula_unit	mean_mass	Tm
mp-1019544	BaZrO3	{'Ba': 1.0, 'Zr': 1.0, 'O': 3.0}	5.983317	143.0	88.0	153.500391	30.700078	153.500391	5	5	81.516800	2813.15
mp-1045	Nd2O3	{'Nd': 2.0, 'O': 3.0}	6.395583	124.0	51.0	698.909648	139.781930	698.909648	5	5	80.120700	2543.15
mp-1105	BaO2	{'Ba': 1.0, 'O': 2.0}	5.391927	67.0	35.0	52.146872	17.382291	52.146872	3	3	76.663200	723.15
mp-1132	CdO	{'Cd': 1.0, 'O': 1.0}	7.791435	126.0	45.0	27.367291	13.683646	27.367291	2	2	64.205200	509.35
mp-1143	Al2O3	{'Al': 2.0, 'O': 3.0}	3.873498	232.0	147.0	87.420037	17.484007	87.420037	5	5	21.490469	2313.15
mp-1147	Ti3O5	{'Ti': 3.0, 'O': 5.0}	4.187248	171.0	76.0	177.344754	22.168094	177.344754	8	8	31.933200	2050.15
mp-12105	RbO2	{'Rb': 1.0, 'O': 2.0}	3.144136	21.0	8.0	62.038631	20.679544	62.038631	3	3	50.733600	685.15

$$\theta_0 = \frac{\hbar v_m}{k_b a} \qquad \theta_1 = \frac{\hbar^2}{m a^2 k_b} \qquad \theta_2 = \frac{a^3 G}{k_b} \qquad \theta_3 = \frac{a^3 K}{k_b}$$

Temperature units

$$\theta_1' = \frac{\hbar}{m a v_m} = \frac{\theta_1}{\theta_0} \qquad \theta_2' = \frac{a^4 G}{\hbar v_m} = \frac{\theta_2}{\theta_0} \qquad \theta_3' = \frac{a^4 K}{\hbar v_m} = \frac{\theta_3}{\theta_0}$$

<u>Dimensionless inputs</u>

Compute PNN inputs

```
h = 6.62607015*1e-34
k = 1.380649*1e-23
Na = 6.0221407*1e23
pi = np.pi
hbar = 1.054571817*1e-34

vs = np.sqrt(df['G_VRH']/df['density']) #from Zack
vp = np.sqrt((df['K_VRH'] + (4/3)*df['G_VRH'])/df['density']) #from Zack
vm = ( 3/( (1/vp)**3 + 2*(1/vs)**3 ) )**(1/3) #from JP Poirier paper

df['debye_temp'] = 10**13*(h/k)*(3/(4*pi*df['volume_per_atom']))**(1/3)*vm

df['a'] = (df['volume_per_atom'])**(1/3)

a = df['a']
m = df['mean_mass']
G = df['G_VRH']
K = df['K_VRH']
```

Compute Debye temp and effective sound speed

```
theta0 = (1.054571817/1.380649)*100*vm/a #hcross*vm/(k*a)
theta1 = (1.054571817**2*6.0221407/1.380649)*10*(1/(m*a**2)) #hcross**2/(m*a**2*k)
theta2 = (1/1.380649)*100*(a**3*G) #a**3*G/k
theta3 = (1/1.380649)*100*(a**3*K) #a**3*K/k

theta1_prime = theta1/theta0
theta2_prime = theta2/theta0
theta3_prime = theta3/theta0

ones = np.ones(len(theta1_prime))

Tm_prime = df['Tm']/theta0

inputs = np.array([theta1_prime, theta2_prime, theta3_prime, ones], dtype='float')
inputs = inputs.T
outputs = np.array(Tm_prime).reshape(-1, 1)

print (inputs.shape, outputs.shape)

train_inputs, test_inputs, train_outputs, test_outputs = train_test_split(inputs, outputs, test_size=0.2, random_state=0)
print (train_inputs.shape, train_outputs.shape)
print (test_inputs.shape, test_outputs.shape)
```

Compute theta(s)

Compute theta'(s)

Create input/output arrays

Split into train/test sets

Create a node in the PNN

Step 2: Create a generic model

```
def squared_act(x):
    return x*x

def inverse_act(x):
    return 1/x
```

```
def create_node(input1, input2, input3, name, trainable1, trainable2, trainable3, act, x, idx):
    base = name
    n1 = base + "1"
    n2 = base + "2"
    n3 = base + "3"
    an1 = Dense(1, activation = 'linear', use_bias = False, name=n1, trainable=trainable1) (input1)
    an2 = Dense(1, activation = 'linear', use_bias = False, name=n2, trainable=trainable2) (input2)
    an3 = Dense(1, activation = 'linear', use_bias = False, name=n3, trainable=trainable3) (input3)

    node_list = [an1, an2, an3]
    if (act == "multiply"):
        non_zero_list = []
        zero_list = []
        for i, j in enumerate(node_list):
            if (x[idx+i] == 1 or x[idx+i] == 2):
                non_zero_list.append(j)
            else:
                zero_list.append(j)
        if ( len(non_zero_list) == 0 ):
            non_zero_list = node_list
            an = multiply(non_zero_list)
        if ( len(non_zero_list) == 1 ):
            anx = non_zero_list[0]
            an = add([anx, zero_list[0], zero_list[1]])
        else:
            an = multiply(non_zero_list)
    else:
        an = add(node_list)
        if (act == "squared"):
            an = Activation(squared_act) (an)
        elif (act == "inverse"):
            an = Activation(inverse_act) (an)
        else:
            an = Activation(act) (an)
    return an
```

Each connection is a Dense layer with 1 input and 1 output

For a multiply activation, multiply non-zero nodes

Add each connection

Apply activation

Create a model using custom nodes

Step 2: Create a generic model

```
def create_model(x):
    #initializer = keras.initializers.RandomUniform(minval=-0.001, maxval=0.001, seed=0)
    bias_initial = keras.initializers.Zeros()

    trainable_list = []
    for i in range(nweight_terms):
        if (x[i+nact_terms] == 2):
            trainable_list.append(True)
        else:
            trainable_list.append(False)

    input1 = Input(shape=(1,))
    input2 = Input(shape=(1,))
    input3 = Input(shape=(1,))
    input4 = Input(shape=(1,))

    a1 = create_node(input1, input2, input3, "a1", trainable_list[0], trainable_list[1],
                    trainable_list[2], act_dict[x[0]], x, 0+nact_terms)
    a2 = create_node(input1, input2, input3, "a2", trainable_list[3], trainable_list[4],
                    trainable_list[5], act_dict[x[1]], x, 3+nact_terms)
    a3 = create_node(input1, input2, input3, "a3", trainable_list[6], trainable_list[7],
                    trainable_list[8], act_dict[x[2]], x, 6+nact_terms)
```

Create nodes a1, a2, a3 in the first hidden layer

```
an1 = Dense(1, activation = 'linear', use_bias = False, name='output1', trainable=trainable_list[9]) (a1)
an2 = Dense(1, activation = 'linear', use_bias = False, name='output2', trainable=trainable_list[10]) (a2)
an3 = Dense(1, activation = 'linear', use_bias = False, name='output3', trainable=trainable_list[11]) (a3)

an4 = Dense(1, activation = 'linear', use_bias = False, name='output4', trainable=trainable_list[12]) (input4)

act = act_dict[x[3]]
node_list = [an1, an2, an3, an4]
```

Setup connections for output layer

Create a model using custom nodes

Step 2: Create a generic model

```
def create_model(x):
    #initializer = keras.initializers.RandomUniform(minval=-0.001, maxval=0.001, seed=0)
    bias_initial = keras.initializers.Zeros()

    trainable_list = []
    for i in range(nweight_terms):
        if (x[i+nact_terms] == 2):
            trainable_list.append(True)
        else:
            trainable_list.append(False)

    input1 = Input(shape=(1,))
    input2 = Input(shape=(1,))
    input3 = Input(shape=(1,))
    input4 = Input(shape=(1,))

    a1 = create_node(input1, input2, input3, "a1", trainable_list[0], trainable_list[1],
                     trainable_list[2], act_dict[x[0]], x, 0+nact_terms)
    a2 = create_node(input1, input2, input3, "a2", trainable_list[3], trainable_list[4],
                     trainable_list[5], act_dict[x[1]], x, 3+nact_terms)
    a3 = create_node(input1, input2, input3, "a3", trainable_list[6], trainable_list[7],
                     trainable_list[8], act_dict[x[2]], x, 6+nact_terms)
```

Create nodes a1, a2, a3 in the first hidden layer

```
an1 = Dense(1, activation = 'linear', use_bias = False, name='output1', trainable=trainable_list[9]) (a1)
an2 = Dense(1, activation = 'linear', use_bias = False, name='output2', trainable=trainable_list[10]) (a2)
an3 = Dense(1, activation = 'linear', use_bias = False, name='output3', trainable=trainable_list[11]) (a3)

an4 = Dense(1, activation = 'linear', use_bias = False, name='output4', trainable=trainable_list[12]) (input4)

act = act_dict[x[3]]
node_list = [an1, an2, an3, an4]
```

Setup connections for output layer

Create a model using custom nodes

```
if (act == "multiply"):
    non_zero_list = []
    zero_list = []
    for i, j in enumerate(node_list):
        if (x[9+i] == 1 or x[9+i] == 2):
            non_zero_list.append(j)
        else:
            zero_list.append(j)
    if ( len(non_zero_list) == 0 ):
        non_zero_list = node_list
        an = multiply(non_zero_list)
    elif ( len(non_zero_list) == 1 ):
        anx = non_zero_list[0]
        an = add([anx, zero_list[0], zero_list[1], zero_list[2]])
    else:
        an = multiply(non_zero_list)
else:
    an = add(node_list)
    if (act == "squared"):
        an = Activation(squared_act) (an)
    elif (act == "inverse"):
        an = Activation(inverse_act) (an)
    else:
        an = Activation(act) (an)
output = an
```

Add/multiply connections and apply activation functions to get output neuron

```
model = Model(inputs=[input1, input2, input3, input4], outputs=[output])
optimizer = tf.train.AdamOptimizer(learning_rate=1e-3)
model.compile(loss='mse', optimizer=optimizer)
```

Define model with 3 inputs, 1 bias, and 1 output

Setup model training

```
for i in range(len(layer_list)):
    model.layers[layer_list[i]].set_weights( [ np.array( [[ weight_dict[x[nact_terms+i]] ]] ) ] )
    #model.layers[layer_list[i]].set_weights( [ np.array( [[ weights_list[i] ]] ) ] )

#model.summary()
```

Set some model weights

```
losses = []
class PrintEpNum(keras.callbacks.Callback): # This is a function for the Epoch Counter
    def on_epoch_end(self, epoch, logs):
        sys.stdout.flush()
        sys.stdout.write("Current Epoch: " + str(epoch+1) + ' Loss: ' + str(logs.get('loss')) + '
        losses.append(logs.get('loss'))

def train(model, train_inputs, train_outputs, verbose=False):
    mae_es= keras.callbacks.EarlyStopping(monitor='val_loss', patience=1000,
                                          min_delta=1e-5, verbose=1, mode='auto', restore_best_weights=True)

    terminate = keras.callbacks.TerminateOnNaN()

    EPOCHS = 100 # Number of EPOCHS
    history = model.fit([train_inputs[:,0], train_inputs[:,1], train_inputs[:,2], train_inputs[:,3]], train_outputs[:,0],
                        epochs=EPOCHS,
                        shuffle=False, batch_size=len(train_inputs), verbose = False, callbacks=[mae_es, terminate],
                        validation_split=0.2)

    if verbose:
        plt.figure()
        plt.xlabel('Epoch')
        plt.ylabel('Mean Sq Error')
        plt.plot(history.epoch, np.array(history.history['loss']),label='Training loss')
        plt.legend()
        plt.show()
    return history
```

EarlyStopping criterion to prevent overfitting

Model.fit(…) performs training

Define objective function

```
def objective_function(individual):
    new_model, trainable = create_model(individual)
    #print ("Trainable: ", trainable)
    valid_flag = True
    stringlist = []
    new_model.summary(print_fn=lambda x: stringlist.append(x))
    for string in stringlist:
        if ("Trainable params" in string):
            ntrainable = int(string[-1])

    if (ntrainable > 0):
        train(new_model, train_inputs, train_outputs, verbose=False)

    mse_train = new_model.evaluate([train_inputs[:, 0], train_inputs[:, 1], train_inputs[:, 2], train_inputs[:, 3]],
                                   train_outputs, verbose=0)
    mse_test = new_model.evaluate([test_inputs[:, 0], test_inputs[:, 1], test_inputs[:, 2], test_inputs[:, 3]],
                                  test_outputs, verbose=0)

    if (np.isnan(mse_train) or np.isnan(mse_test) or np.isinf(mse_train) or np.isinf(mse_test)):
        valid_flag = False

    weights = new_model.get_weights()
    weight_list = []
    for weight in weights:
        weight_list.append(weight[0][0])
    weight_list = np.array(weight_list)
```

Determine # of trainable weights

Train and evaluate model

Collect final weights of model

```
#handle nan weights
if (np.isnan(weight_list).any()):
    valid_flag = False

if (valid_flag):
    print (weight_list)
else:
    mse_test = 1e50

actfunc_term = [i**2 for i in individual[:nact_terms]]
weights = individual[nact_terms:]
weight_term = 0
for j in range(nweight_terms):
    weight_term += f3(weights[j])

mse_test_term = np.log10(mse_test)

p = 0.1
obj = mse_test_term + p*(np.sum(actfunc_term) + weight_term)
print ("Individual: ", individual, flush=True)
print ("Objective function: ", mse_test, np.sum(actfunc_term), weight_term, obj, flush=True)

keras.backend.clear_session()
tf.reset_default_graph()
return (obj,)
```

Add MSE term, activation function term and weight score term of get obj func

Setup genetic algorithm

```python
################### DEAP #####################
#create fitness class and individual class
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
#pool = Pool(1)
#toolbox.register("attr_int", random.randint, 0, 3)

def custom_initRepeat(container, func, max1, max2, n):
    func_list = []
    for i in range(n):
        if (i < nact_terms):
            func_list.append(func(0, max1))
        else:
            func_list.append(func(0, max2))
    return container(func_list[i] for i in range(n))

#gen = initRepeat(list, random.randint, 3, 7, 4)
toolbox.register("create_individual", custom_initRepeat, creator.Individual, random.randint,
                 max1=4, max2=2, n=nact_terms+nweight_terms)
toolbox.register("population", tools.initRepeat, list, toolbox.create_individual)

def custom_mutation(individual, max1, max2, indpb):
    size = len(individual)
    for i in range(size):
        if random.random() < indpb:
            if (i < nact_terms):
                individual[i] = random.randint(0, max1)
            else:
                individual[i] = random.randint(0, max2)
    return individual,
```

Determine # of trainable weights

Define custom repeat func to design tailormade individuals

Define custom mutation

```python
cxpb = 0.5
mutpb = 0.3
ngens = 5


toolbox.register("mate", tools.cxTwoPoint)
#toolbox.register("mutate", tools.mutUniformInt, low=0, up=3, indpb=mutpb)
toolbox.register("mutate", custom_mutation, max1=4, max2=2, indpb=mutpb)
toolbox.register("select", tools.selTournament, tournsize=10)
toolbox.register("evaluate", objective_function)

random.seed(100000)
population = toolbox.population(n=10)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
pop, logbook = algorithms.eaSimple(population, toolbox, cxpb, mutpb, ngens, stats=stats, halloffame=hof, verbose=True)
```

crossover

selection

Use simple evolutionary algorithm

Interpreting an individual

Step 3: Express network weights as interpretable equations

```
individual = [0, 2, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 2, 2, 0, 2]
weights_list =  "0.0000000e+00 0.0000000e+00 2.2341371e-02 1.0000000e+00 0.0000000e+00 \
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 2.2341387e-02 \
 7.9583116e-03 0.0000000e+00 1.1903398e+01"

weights_list = [float(x) for x in weights_list.split()]
```

Collect act, weights and biases from individual

```
name_list = []
for i in range(len(new_model.layers)):
    name = new_model.layers[i].name
    if ( ("activation" in name) or ("input" in name) or ("add" in name) or ("multiply" in name) ):
        continue
    else:
        name_list.append(name)
print (name_list)
```

Get list of layer names in model

```
act_nodes = {"a1": act_dict[individual[0]], "a2": act_dict[individual[1]], "a3": act_dict[individual[2]],
             "output": act_dict[individual[3]]}

weight_dict = {}
for i, weight in enumerate(weight_list):
    name = name_list[i]
    weight_dict[name] = weight
```

Set activation for each neuron

Interpreting an individual

```python
from sympy import *

theta1_prime, theta2_prime, theta3_prime = symbols('theta1_prime theta2_prime theta3_prime')

def return_value(i1, i2, i3, act, name):
    n1 = name + "1"
    n2 = name + "2"
    n3 = name + "3"
    if (act == 'linear'):
        value = i1*weight_dict[n1] + i2*weight_dict[n2] + i3*weight_dict[n3]
    elif (act == 'squared'):
        value = ( i1*weight_dict[n1] + i2*weight_dict[n2] + i3*weight_dict[n3] )**2
    elif (act == 'multiply'):
        value = i1*weight_dict[n1] * i2*weight_dict[n2] * i3*weight_dict[n3]
    elif (act == 'inverse'):
        value = 1/( i1*weight_dict[n1] + i2*weight_dict[n2] + i3*weight_dict[n3] )
    elif (act == 'tanh'):
        value = tanh( i1*weight_dict[n1] + i2*weight_dict[n2] + i3*weight_dict[n3] )
    return value
```

Define symbolic variables

Symbolically evaluate each node

```python
a1 = return_value(theta1_prime, theta2_prime, theta3_prime, act_nodes["a1"], "a1")
a2 = return_value(theta1_prime, theta2_prime, theta3_prime, act_nodes["a2"], "a2")
a3 = return_value(theta1_prime, theta2_prime, theta3_prime, act_nodes["a3"], "a3")

name = "output"

n1 = name + "1"; n2 = name + "2"; n3 = name + "3"; n4 = name + "4"
act = act_nodes["output"]
if (act == 'linear'):
    value = a1*weight_dict[n1] + a2*weight_dict[n2] + a3*weight_dict[n3] + 1*weight_dict[n4]
elif (act == 'squared'):
    value = ( a1*weight_dict[n1] + a2*weight_dict[n2] + a3*weight_dict[n3] + 1*weight_dict[n4] )**2
elif (act == 'multiply'):
    value = a1*weight_dict[n1] * a2*weight_dict[n2] * a3*weight_dict[n3] * 1*weight_dict[n4]
elif (act == 'inverse'):
    value = 1/( a1*weight_dict[n1] + a2*weight_dict[n2] + a3*weight_dict[n3] + 1*weight_dict[n4] )
elif (act == 'tanh'):
    value = tanh( a1*weight_dict[n1] + a2*weight_dict[n2] + a3*weight_dict[n3] + 1*weight_dict[n4] )

output = value

print (output)
```

Collect expressions for node a1, a2, a3

Symbolically evaluate output

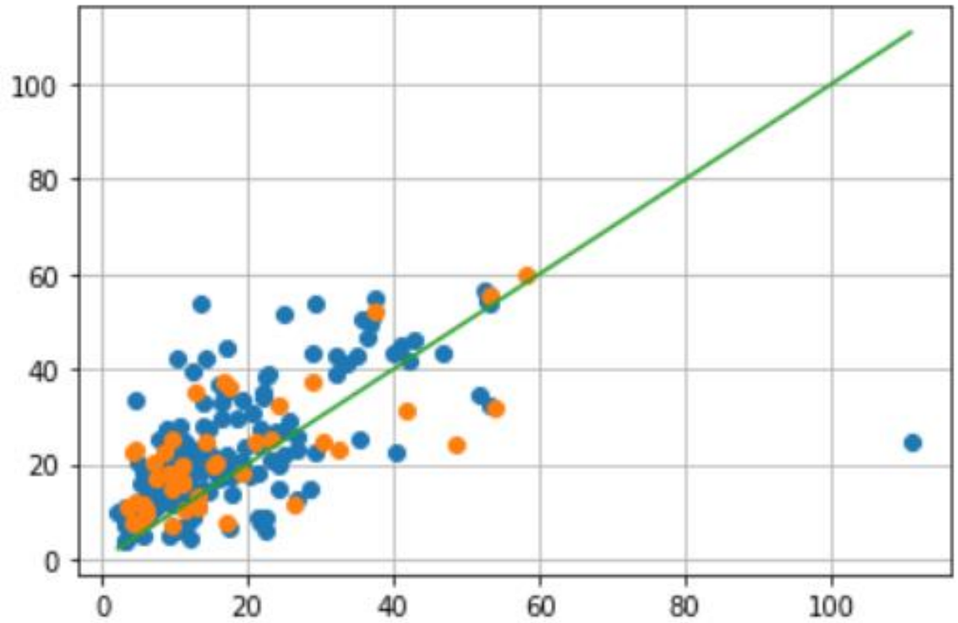
Evaluating an individual

Step 4: Evaluate model on dataset and evaluate objective function

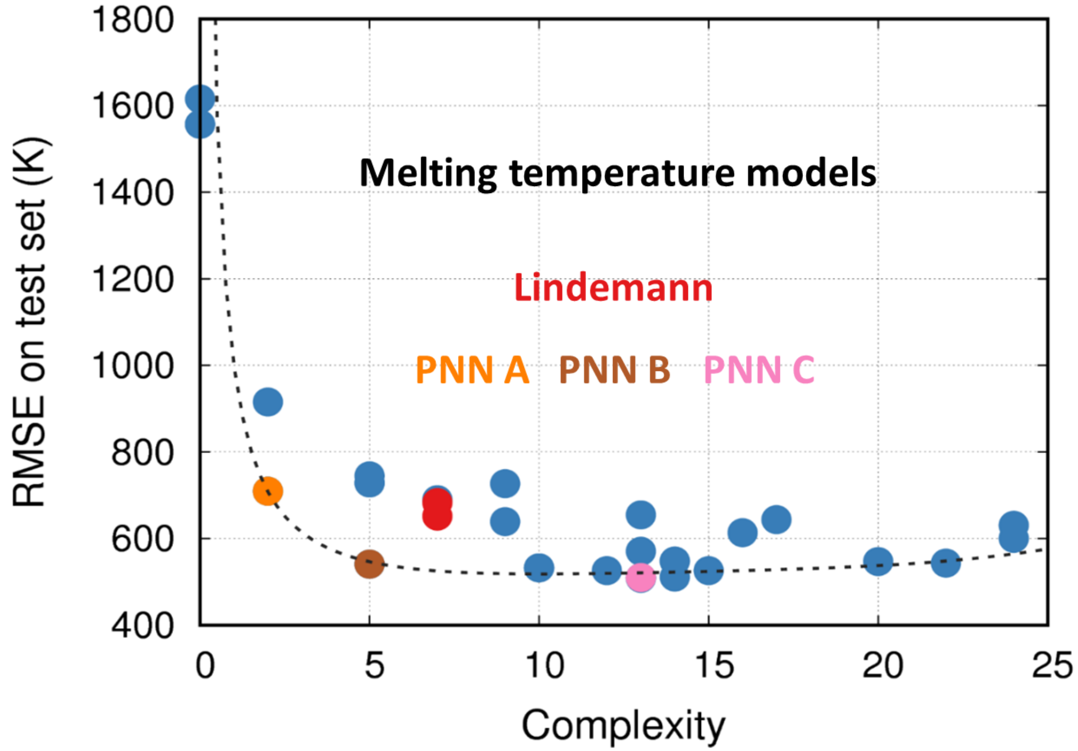
```
mse_train = new_model.evaluate([train_inputs[:, 0], train_inputs[:, 1], train_inputs[:, 2], train_inputs[:, 3]],
                               train_outputs, verbose=0)
mse_test = new_model.evaluate([test_inputs[:, 0], test_inputs[:, 1], test_inputs[:, 2], test_inputs[:, 3]],
                              test_outputs, verbose=0)
```

Evaluate model on train/test sets

```
test_preds = new_model.predict([test_inputs[:, 0], test_inputs[:, 1], test_inputs[:, 2], test_inputs[:, 3]])
train_preds = new_model.predict([train_inputs[:, 0], train_inputs[:, 1], train_inputs[:, 2], train_inputs[:, 3]])
```



Discovering melting point laws



<u>Melting temperature models</u>

$$T_m^{PNN\,A} = 21.8671\,\theta_0$$

$$T_m^{lind} = \frac{k_b}{9\hbar^2} f^2 a^2 m T_D^2 = C\,\frac{\theta_0^2}{\theta_1}$$

$$T_m^{PNN\,B} = 17.553\,\theta_0 + 0.00198\,\theta_2$$

$$T_m^{PNN\,C} = 11.903\,\theta_0 + 0.0005\,\theta_3 + 0.008\,\frac{\theta_0^2}{\theta_1}$$

Parsimonious neural networks learn non-linear interpretable laws