In [1]:
```
from ANI1_release.ANI1_release.readers.lib import pyanitools as pya
from sklearn.model_selection import train_test_split


import torch
from torch import nn
import numpy as np
import torchani


from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import pandas as pd
```

customized aev function used to calculate aev of each molecule

In [2]:
```
def calc_aev(X,S):
    Rcr=5.2
    Rca=3.5
    EtaR=torch.tensor([16],dtype=torch.float)
    ShfR=torch.tensor([0.900000,1.168750,1.437500,1.706250,1.975000,2.243750
    EtaA=torch.tensor([8],dtype=torch.float)
    Zeta=torch.tensor([32],dtype=torch.float)
    ShfA=torch.tensor([0.900000,1.550000,2.200000,2.850000],dtype=torch.floa
    ShfZ=torch.tensor([0.19634954,0.58904862,0.9817477,1.3744468,1.7671459,2
    num_species=4
    X=torch.tensor(X,dtype=torch.float)
    mapping={"H":0, "C":1, "N":2, "O":3}


    elements = np.array([mapping[atom] for atom in S])
    elements2=torch.tensor(elements,dtype=torch.long).repeat(X.shape[0],1)

    aevs_X=torchani.AEVComputer(Rcr,Rca,EtaR,ShfR,EtaA,Zeta,ShfA,ShfZ,num_sp



    aev_calc=aevs_X((elements2,X))
    molecule_aev=aev_calc[1]
    molecule_aev_shape=aev_calc[1].shape


    return molecule_aev,molecule_aev_shape,S,elements
```

function to process files into list and concatenate all the h5 files

In [3]:
```
# Set the HDF5 file containing the data
hdf5file_1 = 'ANI1_release/ANI1_release/ani_gdb_s01.h5'
hdf5file_2 = 'ANI1_release/ANI1_release/ani_gdb_s02.h5'
hdf5file_3 = 'ANI1_release/ANI1_release/ani_gdb_s03.h5'

hdf5file_4 = 'ANI1_release/ANI1_release/ani_gdb_s04.h5'
hdf5file_5 = 'ANI1_release/ANI1_release/ani_gdb_s05.h5'
hdf5file_6 = 'ANI1_release/ANI1_release/ani_gdb_s06.h5'

hdf5file_7 = 'ANI1_release/ANI1_release/ani_gdb_s07.h5'
hdf5file_8 = 'ANI1_release/ANI1_release/ani_gdb_s08.h5'


# Construct the data loader class
adl_1 = pya.anidataloader(hdf5file_1)
adl_2 = pya.anidataloader(hdf5file_2)
adl_3 = pya.anidataloader(hdf5file_3)

adl_4 = pya.anidataloader(hdf5file_4)
adl_5 = pya.anidataloader(hdf5file_5)
adl_6 = pya.anidataloader(hdf5file_6)

adl_7 = pya.anidataloader(hdf5file_7)
adl_8 = pya.anidataloader(hdf5file_8)




def open_h5_list(adl):
    list_1=list()

    for data in adl:

        # Extract the data
        P = data['path']
        X = data['coordinates']
        E = data['energies']
        S = data['species']
        sm = data['smiles']

        # Print the data
        # print("Path:   ", P)
        # print("  Smiles:      ","".join(sm))
        # print("  Symbols:     ", S)
        # print("  Coordinates: ", X.shape)
        # print("  Energies:    ", E.shape, "\n")



        list_1.append(data)


    return list_1

adl=open_h5_list(adl_1)+open_h5_list(adl_2)+open_h5_list(adl_3)+open_h5_list
# +open_h5_list(adl_5)
```

```
# +open_h5_list(adl_5)
# +open_h5_list(adl_6)+open_h5_list(adl_7)+open_h5_list(adl_8)

count=len(adl)

print(count)
```

```
def adl_info_store(adl):
    list_coords=[]
    for i in range(count):
        list_coords.append(adl[i]['coordinates'])


    list_species=[]
    for i in range(count):
        list_species.append(adl[i]['species'])


    list_energies=[]
    for i in range(count):
        list_energies.append(adl[i]['energies'])


    list_energies_len=[]
    for i in range(count):
        list_energies_len.append(len(list_energies[i]))

    return list_coords,list_species,list_energies,list_energies_len
```

```
97
```

functions to filter data because want to batch 128 and 80 train :20 test split # data generator is not compatible with declared batch size below the actual inputs so this problem will be resolved by filtering data

In [4]:
```
def filter_data(list1,list2):
    dictionary = dict(zip(list1, list2))

    newDict = dict()
    # Iterate over all the items in dictionary and filter items which has ev
    for (key, value) in dictionary.items():
        # Check if key is even then add pair to new dictionary
        if key >= 640:
            newDict[key] = value

    return list(newDict.keys()),list(newDict.values())


filt=filter_data(adl_info_store(adl)[3],adl_info_store(adl)[1])
print(len(filt[1]))
```

```
26
```

In [5]:
```
def filter_species(l1,l2):
    s=[]
    list_ = list(zip(l1, l2))
    for i,j in list_:
        if [i][0]>= 640:
            s.append(j)

    return s

filter_species=filter_species(adl_info_store(adl)[3],adl_info_store(adl)[1])
print(len(filter_species))
```

```
95
```

In [6]:
```
def filter_coords_eng(l1,l2):
    s=[]
    for i in l1:
        for k in l2:
            if i.shape[0]==k:
                s.append(i)

    return s

filter_coords=filter_coords_eng(adl_info_store(adl)[0],filt[0])
filter_eng=filter_coords_eng(adl_info_store(adl)[2],filt[0])
print(len(filter_eng))
print(len(filter_coords))
```

```
95
95
```

data generator that batches into a molecules'confirmations

In [7]:
```
def data_gen(X,y, batchsize):
    '''
    Generator for data
    '''
    if len(X)//batchsize == 0:
        return X[:batchsize],y[:batchsize]

    for i in range(len(X)//batchsize):
        yield X[i*batchsize:(i+1)*batchsize],y[i*batchsize:(i+1)*batchsize]
    i+=1
    yield X[i*batchsize:],y[i*batchsize:]
```

In [8]:
```
from functools import wraps
from time import time

def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))
        return result
    return wrap
```

trainer class takes X's that correspond to the molecules' coordinates and molecules actual energy, atom_types is a list of lists of atoms that make up that individual molecule. Loss is collected for each molecule then added iteratively.

In [9]:
```
class Trainer():
    def __init__(self, model, optimizer_type, learning_rate, epoch, batch_si
        """ The class for training the model
        model: nn.Module
            A pytorch model
        optimizer_type: 'adam' or 'sgd'
        learning_rate: float
        epoch: int
        batch_size: int
        input_transform: func
            transforming input. Can do reshape here
        """
        self.model = model
        if optimizer_type == "sgd":
            self.optimizer = SGD(model.parameters(), learning_rate,momentum=
        elif optimizer_type == "adam":
            self.optimizer = Adam(model.parameters(), learning_rate)

        self.epoch = epoch
        self.batch_size = batch_size
        self.input_transform = input_transform


    @timing
    def train(self, inputs, outputs, val_inputs, val_outputs,atom_types,draw
        """ train self.model with specified arguments
        inputs: np.array, The shape of input_transform(input) should be (nda
        outputs: np.array shape (ndata,)
        val_nputs: np.array, The shape of input_transform(val_input) should
        val_outputs: np.array shape (ndata,)
        early_stop: bool
        l2: bool
        silent: bool. Controls whether or not to print the train and val err
        """

        train_gen_coords_energies=[]
        for i in range(len(inputs)):
            #shuffle the data in each epoch
            idx =torch.randperm(val_inputs[i].shape[0])
            train_gen=data_gen(inputs[i][idx],outputs[i][idx],self.batch_siz
            train_gen_coords_energies.append(next(train_gen))


        train_gen_coords=[]
        for i in range(len(inputs)):
            train_gen_coords.append(train_gen_coords_energies[i][0])

        train_gen_energies=[]
        for i in range(len(inputs)):
            train_gen_energies.append(train_gen_coords_energies[i][1])

        list_species_coords=zip(train_gen_coords,atom_types)
        list_aevs=[calc_aev(X,S) for X,S in list_species_coords]
        print(len(list_aevs))




        losses = []
```

```
        losses = []
        val_losses = []
        test_losses=[]
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        lowest_test_loss = np.inf



        for n_epoch in tqdm(range(self.epoch), leave=False):
            self.model.train()
            epoch_loss = 0
            for i in range(len(list_aevs)):

                #hartree to kcal/mol

                batch_predictions = self.model(list_aevs[i][0],list_aevs[i][
                batch_predictions = torch.mul(batch_predictions, 627.5)
                batch_importance = len(train_gen_coords_energies[i][1]) / le
                loss = nn.MSELoss()(batch_predictions, torch.tensor(train_ge
                if l2:
                    l2_lambda = 1e-5
                    l2_norm = sum(p.pow(2.0).sum() for p in self.model.param
                    loss = loss + l2_lambda * l2_norm

                elif l1:
                    l1_lambda = 0.001
                    l1_norm= sum(p.abs().sum() for p in self.model.parameter
                    loss = loss + l1_lambda * l1_norm

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
            epoch_loss += loss.detach().cpu().item() * batch_importance

            val_loss = self.evaluate(val_inputs, val_outputs, atom_types)
            if n_epoch % 10 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f" % (n_epoch + 1, self.epoch,
                print("              Val_loss: %.3f " % (val_loss))

            losses.append(epoch_loss)
            val_losses.append(val_loss)



            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()

            if early_stop:
                if val_loss < lowest_test_loss:
                    lowest_test_loss = test_loss
                    weights = self.model.state_dict()
        if draw_curve:
            plt.figure()
            plt.plot(np.arange(self.epoch) + 1,losses,label='Training loss')
            plt.plot(np.arange(self.epoch) + 1,val_losses,label='Validation
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
```

```
            plt.ylabel('Loss')
            plt.legend()

        if early_stop:
            self.model.load_state_dict(weights)
        return {"losses": losses,  "val_losses": val_losses}

    def evaluate(self, inputs, outputs, atom_types):
        gen_coords_energies=[]
        for i in range(len(inputs)):
            idx =torch.randperm(inputs[i].shape[0])
            gen=data_gen(inputs[i][idx],outputs[i][idx],self.batch_size)
            gen_coords_energies.append(next(gen))

        gen_coords=[]
        for i in range(len(inputs)):
            gen_coords.append(gen_coords_energies[i][0])

        gen_energies=[]
        for i in range(len(inputs)):
            gen_energies.append(gen_coords_energies[i][1])

        self.model.eval()
        list_species_coords=zip(gen_coords,atom_types)
        list_aevs=[calc_aev(X,S) for X,S in list_species_coords]
        losses = 0
        for i in range(len(list_aevs)):
            batch_importance = len(gen_coords_energies[i][1]) / len(outputs)
            with torch.no_grad():
                batch_predictions = self.model(list_aevs[i][0],list_aevs[i][
                batch_predictions = torch.mul(batch_predictions, 627.5)
                loss = nn.MSELoss()(batch_predictions, torch.tensor(gen_ener

            losses += loss.detach().cpu().item() * batch_importance

        return losses
```

neural network that takes atom type iteratively than predicts an energy for each molecule

In [10]:
```
class ANI(nn.Module):
    def __init__(self):
        super().__init__()
        self.sub_nets = nn.ModuleDict({"C": ANI_sub(nn.Sequential(nn.Linear(
                                        "H": ANI_sub(nn.Sequential(nn.Linear(
                                        "N": ANI_sub(nn.Sequential(nn.Linear(
                                        "O": ANI_sub(nn.Sequential(nn.Linear(

    def forward(self, aevs, atom_types):
        num_conf, num_atoms, aev_vec=aevs.shape
        atomic_energies=torch.empty((num_conf, 0),dtype=torch.float)
        aev_shape=aevs.reshape(num_atoms, num_conf, aev_vec)
        for i in range(len(atom_types)):
            atom=self.sub_nets[atom_types[i]](aev_shape[i])

            atomic_energies=torch.cat((atomic_energies,atom),1,)

        total_energies = torch.sum(atomic_energies,dim=-1,dtype=torch.float,

        return total_energies

class ANI_sub(nn.Module):
    def __init__(self, architecture):
        super().__init__()
        self.layers= architecture

    def forward(self, aev):
        atomic_energy = self.layers(aev)
        return atomic_energy
```

train_test_split for tests, validation, and train tests for each molecule that is appended to each respective lists

In [11]:
```
train_X=[]
train_y=[]
test_X=[]
test_y=[]
val_X=[]
val_y=[]
for i in range(len(filter_coords)):
    train_Xs, test_Xs, train_ys, test_ys = train_test_split(filter_coords[i]
#     train_Xs, val_Xs, train_ys, val_ys = train_test_split(train_Xs,train_y

    train_X.append(train_Xs)
    train_y.append(train_ys)
    test_X.append(test_Xs)
    test_y.append(test_ys)
#     val_X.append(val_Xs)
#     val_y.append(val_ys)



def count_min(train):
    s=[]
    for i in train:
        if i.shape[0] > 640:
            s.append(i.shape[0])

    return min(s)


# print(count_min(train_X))
# print(count_min(test_X))
# print(count_min(val_X))
```

trainer class running to measure loss using MSE and summing loss for each atom b/c padding results in losing integrated of molecules / proper aev cannot be calculated to its respective energy

hyperparameter tuning w/ learning rates b/c batchsize cannot be more than 128 due to the filter setting above

There is a tradeoff for bigger and smaller batch size which have their own disadvantage, making it a hyperparameter to tune in some sense. Theory says that, bigger the batch size, lesser is the noise in the gradients and so better is the gradient estimate. This allows the model to take a better step towards a minim

Model is overfitting and I am trying to add more hidden layers to resolve this. This is indicated in a low training loss versus validation loss b/c hyperparamter tuning is not working from regularization techniques to batch size change

NERSC GPU Node cannot be reserved on time so I cannot increase the number of molecules to train on to resolve a possible overfitting issue but will investigate further

train test split once to only get validation set bcuz test/validation loss are practically similar don't want redudancies and waste data for 1 additional splits

implement dropout and regularization to reduce model complexity to reduce the issues with overfitting

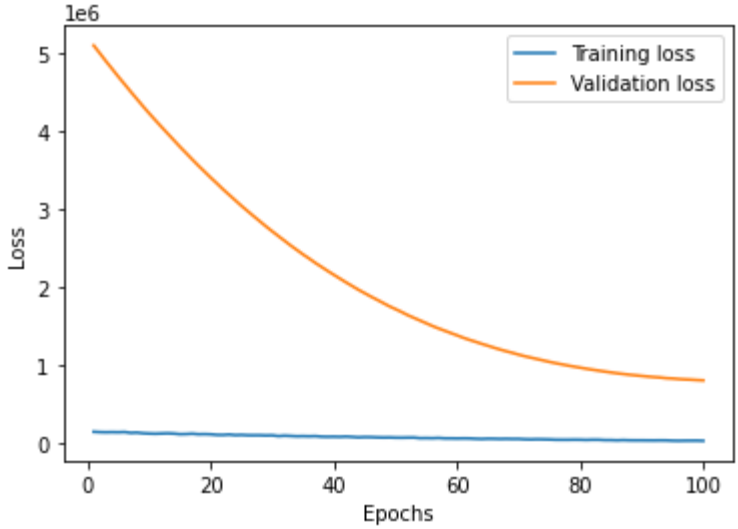
work on model complexity and see if batch size, learning rate, and epochs are not making a huge different due to the model overfitting

In [12]:
```
model=ANI()
trainer = Trainer(model, 'adam', 1e-5, 100, 128)
```

In [13]:
```
log=trainer.train(train_X, train_y, test_X, test_y,filter_species,draw_curve
```

```
/home/joyce/miniconda3/lib/python3.9/site-packages/torchani/aev.py:236: Use
rWarning: __floordiv__ is deprecated, and its behavior will change in a fut
ure version of pytorch. It currently rounds toward 0 (like the 'trunc' func
tion NOT 'floor'). This results in incorrect rounding for negative values.
To keep the current behavior, use torch.div(a, b, rounding_mode='trunc'), o
r for actual floor division, use torch.div(a, b, rounding_mode='floor').
  pair_sizes = counts * (counts - 1) // 2
95
  1%|▌                                         | 1/100 [00:03<06:28,  3.93
s/it]
Epoch 1/100 - Loss: 155370.632
              Val_loss: 5093494.813
 11%|████▍                                     | 11/100 [00:41<05:28,  3.69
s/it]
Epoch 11/100 - Loss: 130665.253
               Val_loss: 4139440.520
 21%|████████▎                                 | 21/100 [01:18<04:45,  3.61
s/it]
Epoch 21/100 - Loss: 115867.811
               Val_loss: 3330845.793
 31%|████████████▏                             | 31/100 [01:53<04:05,  3.55
s/it]
Epoch 31/100 - Loss: 100381.095
               Val_loss: 2656560.126
 41%|████████████████                          | 41/100 [02:28<03:23,  3.44
s/it]
Epoch 41/100 - Loss: 90668.632
               Val_loss: 2113257.967
 51%|███████████████████▉                      | 51/100 [03:02<02:48,  3.45
s/it]
Epoch 51/100 - Loss: 79797.347
               Val_loss: 1686496.698
 61%|███████████████████████▊                  | 61/100 [03:38<02:21,  3.64
s/it]
Epoch 61/100 - Loss: 70742.258
               Val_loss: 1358267.177
 71%|███████████████████████████▋              | 71/100 [04:13<01:39,  3.43
s/it]
Epoch 71/100 - Loss: 59769.126
               Val_loss: 1121407.938
 81%|███████████████████████████████▌          | 81/100 [04:47<01:05,  3.43
s/it]
Epoch 81/100 - Loss: 51453.253
               Val_loss: 961446.183
 91%|███████████████████████████████████▍      | 91/100 [05:22<00:32,  3.59
s/it]
Epoch 91/100 - Loss: 44970.774
               Val_loss: 861216.148

func:'train'  took: 357.9713 sec
```


1e6
Training loss
Validation loss
Loss
Epochs
0
1
2
3
4
5
0
20
40
60
80
100