For the final project we will develop a supervised learning ANN model applied to the ANI-1 data set. We will do a check-in once per week to see steady progress with appropriate entries of dates in the jupyter notebooks on what has been accomplished. I.e. this is not a project assignment that should be finished the night before. This will be part of the assessment. The finals project has the following expectations for assessment: Part I: An individual jupyter notebook should be maintained during the course of the project. There will be 3 progress check-ins, each 20% of the grade. You'll submit your notebook to Gradescope. (1) (April 14) Data preparation. Show that you're able to load the data, process them into model input format, and split the data into train, validation and test set with batching.

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

Grab Molecule from H5 file

In [2]:
```
from ANI1_release.ANI1_release.readers.lib.pyanitools import anidataloader
# data = anidataloader("../../ANI1_dataset/ANI-1_release/ani_gdb_s01.h5")
data = anidataloader("ANI1_release/ANI1_release/ani_gdb_s01.h5")
data_iter = data.__iter__()
```

In [5]:
```
mols = next(data_iter)
# Extract the data
P = mols['path']
X = mols['coordinates']
E = mols['energies']
S = mols['species']
sm = mols['smiles']

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

```
Path:    /gdb11_s01/gdb11_s01-2
  Smiles:      [H]O[H]
  Symbols:     ['O', 'H', 'H']
  Coordinates: (1800, 3, 3)
  Energies:    (1800,)
```

Function to calc AEV

In [6]:
```
import numpy as np


def calc_f_C(Rij, RC):
    f_C_value = 0.5 * np.cos(np.pi * Rij / RC) + 0.5
    indicator = ((Rij <= RC) & (Rij != 0)).astype(float) # Make f_C(0)=0 to
    return f_C_value * indicator

def radial_component(Rijs, eta, Rs, RC=5.2):
    # Rijs is a 1d array, all other parameters are scalars
    f_C_values = calc_f_C(Rijs, RC)
    individual_components = np.exp(-eta * (Rijs - Rs) ** 2) * f_C_values
    return np.sum(individual_components)

def angular_component(Rij_vectors, Rik_vectors, zeta, theta_s, eta, Rs, RC=3
    # Rij_vectors and Rik_vectors are 2d arrays with shape (n_atoms, 3), all
    # calculate theta_ijk values from vector operations
    dot_products = Rij_vectors.dot(Rik_vectors.T)
    Rij_norms = np.linalg.norm(Rij_vectors, axis=-1)
    Rik_norms = np.linalg.norm(Rik_vectors, axis=-1)
    norms = Rij_norms.reshape((-1, 1)).dot(Rik_norms.reshape((1, -1)))
    cos_values = np.clip(dot_products / (norms + 1e-8), -1, 1)
    theta_ijks = np.arccos(cos_values)
    theta_ijk_filter = (theta_ijks != 0).astype(float)
    mean_dists = (Rij_norms.reshape((-1, 1)) + Rik_norms.reshape((1, -1))) /
    f_C_values_Rij = calc_f_C(Rij_norms, RC)
    f_C_values_Rik = calc_f_C(Rik_norms, RC)
    f_C_values = f_C_values_Rij.reshape((-1, 1)).dot(f_C_values_Rik.reshape(
    individual_components = (1 + np.cos(theta_ijks - theta_s)) ** zeta * np.
    return 2 ** (1 - zeta) * np.sum(individual_components)

def calc_aev(atom_types, coords, i_index):
    # atom_types are np.array of ints
    relative_coordinates = coords - coords[i_index]
    nearby_atom_indicator = np.linalg.norm(relative_coordinates, axis=-1) <
    relative_coordinates = relative_coordinates[nearby_atom_indicator]
    atom_types = atom_types[nearby_atom_indicator]
    radial_aev = np.array([radial_component(np.linalg.norm(relative_coordina
                           for atom in [0, 1, 2, 3] for eta in [16] \
                           for Rs in [0.900000,1.168750,1.437500,1.706250,1.
                                   3.318750,3.587500,3.856250,4.125000,4.393
    angular_aev = np.array([angular_component(relative_coordinates[atom_type
                                              zeta, theta_s, eta, Rs) \
                            for atom_j in [0, 1, 2, 3] for atom_k in range(a
                            for theta_s in [0.19634954,0.58904862,0.9817477,
                            for eta in [8] for Rs in [0.900000,1.550000,2.20
#     print(len(radial_aev), len(angular_aev))
    return np.concatenate([radial_aev, angular_aev])
```

Function to convert coords to AEV

In [7]:
```
def conv_coords_aev(coords,species):
    mapping={"H":0, "C":1, "N":2, "O":3}
    elements= np.array([mapping[atom] for atom in species])
    old=[]
    for j in range(len(elements)):
        new=[]
        for i in range(len(coords)):
            new.append(calc_aev(elements, coords[i],j))
        old.append(new)
    return np.array(old)
```

In [9]:
```
water=conv_coords_aev(X,S)
water.shape
```

Out[9]:
```
(3, 1800, 384)
```

Train_test_split the AEV's and energies

In [41]:
```
train_X, train_y, val_X, val_y =train_test_split(water.reshape(1800,3,384),E

print(train_X.shape)
train_y.shape
val_X.shape
val_y.shape
```

Out[41]:
```
(1200, 3, 384)
(600,)
```

Trainer class

In [21]:
```
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
```

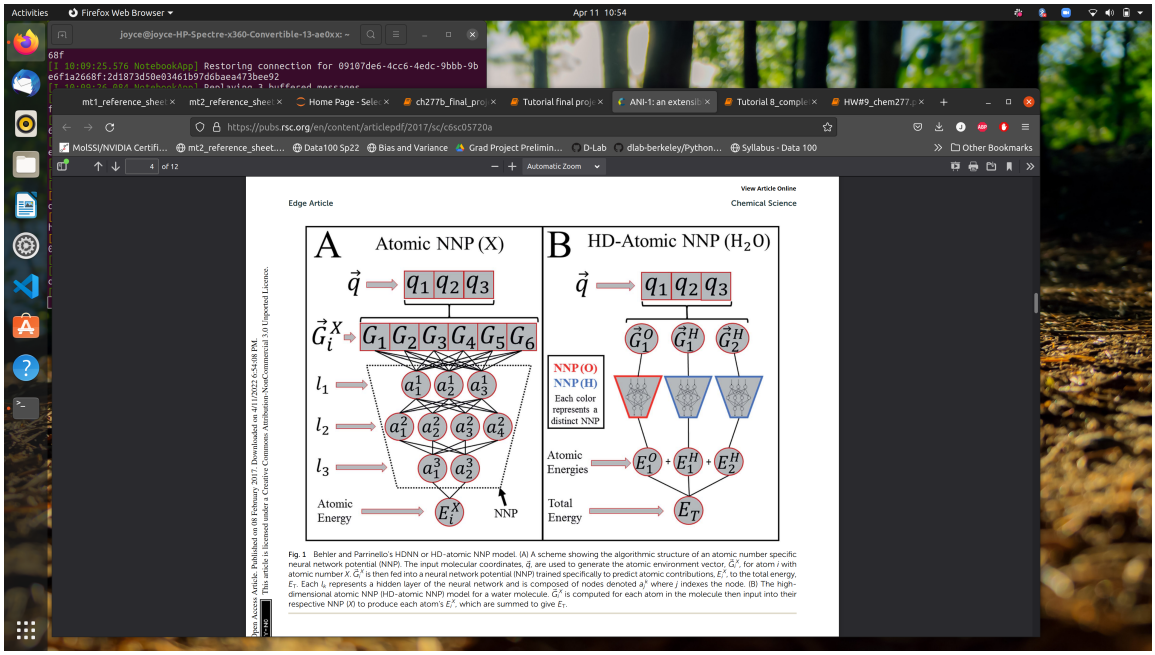
In [46]:
```
from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split
import torch.nn

def create_chunks(complete_list, chunk_size=None, num_chunks=None):
    '''
    Cut a list into multiple chunks, each having chunk_size (the last chunk
    '''
    chunks = []
    if num_chunks is None:
        num_chunks = math.ceil(len(complete_list) / chunk_size)
    elif chunk_size is None:
        chunk_size = math.ceil(len(complete_list) / num_chunks)
    for i in range(num_chunks):
        chunks.append(complete_list[i * chunk_size: (i + 1) * chunk_size])
    return chunks

class Trainer():
    def __init__(self, learning_rate, epoch, batch_size):
        """ 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.epoch = epoch
        self.batch_size = batch_size


    @timing
    def train(self, inputs, outputs, val_inputs, val_outputs,early_stop=Fals
        """ 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

        @return
        a dictionary of arrays with train and val losses and accuracies
        """
        ### convert data to tensor of correct shape and type here ###
        inputs = torch.tensor(inputs, dtype=torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)


        for n_epoch in tqdm(range(self.epoch), leave=False):
            batch_indices = list(range(inputs.shape[0]))
```

```
            batch_indices = list(range(inputs.shape[2]))
            random.shuffle(batch_indices)
            batch_indices = create_chunks(batch_indices, chunk_size=self.bat
            epoch_loss = 0
            epoch_acc = 0

            for batch in batch_indices:
                batch_importance = len(batch) / len(outputs)
                batch_input = inputs[batch]
                batch_output = outputs[batch]

                ### Compute epoch_loss and epoch_acc

            if n_epoch % 10 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f - Acc: %.3f" % (n_epoch + 1,



        return batch_input.shape, batch_output.shape

    def evaluate(self, inputs, outputs, print_acc=True):
        """ evaluate model on provided input and output
        inputs: np.array, The shape of input_transform(input) should be (nda
        outputs: np.array shape (ndata,)
        print_acc: bool

        @return
        losses: float
        acc: float
        """
        inputs = torch.tensor(inputs, dtype=torch.float)
        outputs = torch.tensor(outputs, dtype=torch.int64)

        batch_indices = list(range(inputs.shape[0]))
        batch_indices = create_chunks(batch_indices, chunk_size=self.batch_s

        for batch in batch_indices:
            batch_importance = len(batch) / len(outputs)
            batch_input = inputs[batch]
            batch_output = outputs[batch]



        return batch_output.shape,batch_input.shape
```

Batch w/epochs

In [48]:
```
trainer = Trainer(1e-3, 50, 128)
log=trainer.train(train_X, train_y, val_X, val_y)
```

```
Epoch 1/50 - Loss: 0.000 - Acc: 0.000
Epoch 11/50 - Loss: 0.000 - Acc: 0.000
Epoch 21/50 - Loss: 0.000 - Acc: 0.000
Epoch 31/50 - Loss: 0.000 - Acc: 0.000
Epoch 41/50 - Loss: 0.000 - Acc: 0.000
func:'train'  took: 0.1440 sec
```



In [ ]:
```
from torch import nn
import torch
from torchsummary import summary

class ANI(nn.Module):
    def __init__(self):
        super().__init__()
        self.sub_nets = nn.ModuleDict({"C": ANI_sub([architecture]), "H": AN

    def forward(self, aevs, atom_types):
        atomic_energies = nn.Softmax(dim=-1)(self.sub_nets[1](atom_types))
        total_energies = torch.sum(atomic_energies,dim=-1)
        return total_energies

class ANI_sub(nn.Module):
    def __init__(self, architecture):
        super().__init__()
        self.fc = nn.ModuleList([nn.Linear(331,50),nn.Linear(50,10)])
        self.activation = nn.Sigmoid()

    def forward(self, aev):

        aev = nn.Flatten()(self.activation(self.conv[1](x)))
        aev = self.activation(self.fc[0](x))
        aev = nn.Softmax(dim=-1)(self.fc[1](x))
        atomic_energy = torch.sum(aev,dim=-1)
        return atomic_energy
```