In [28]:
```
import pickle
import torch
import torch.nn as nn
import numpy as np

from sklearn.preprocessing import OneHotEncoder
import numpy as np
import pickle
import math
from torch import nn
import torch
from torch.optim import SGD, Adam
import torch.nn
from sklearn.model_selection import KFold
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

1 LSTM applied to SMILES string generation

In [5]:
```
smiles = pickle.load(open("./ani_smiles.pkl", "rb"))
```

In [7]:
```
def batches_gen(smiles, batchsize, encoder):
    '''
    Create a generator that returns batches of size (batch_size,seq_leng,nchars) from smiles,
    where seq_leng is the length of the longest smiles string and nchar is the length of one-hot encoded cl

    Arguments
    ---------
    smiles:
    python list(nsmiles,nchar) smiles array shape you want to make batches from
    batchsize:
    Batch size, the number of sequences per batch
    encoder:
    one hot encoder
```

```
    '''
    arr=[torch.tensor(np.array(encoder.transform(np.array(s).reshape(-1,1)).toarray()),dtype=torch.float) 
        #size (nsmiles,seq_length(variable),nchars)

    # The features
    X = [s[:-1,:] for s in arr]
    # The targets, shifted by one
    y = [s[1:,:] for s in arr]
    # pad sequence so that all smiles are the same length
    X = nn.utils.rnn.pad_sequence(X,batch_first=True)
    y = nn.utils.rnn.pad_sequence(y,batch_first=True)

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

    # drop last batch that is not the same size due to hidden state constraint
    if len(X) % batchsize != 0:
        X = X[:-batchsize]
        y = y[:-batchsize]

# define your one hot encoder
encoder = OneHotEncoder()
```

1a

Process the smiles strings from ANI dataset by adding a starting character at the beginning and an ending character at the end. Look over the dataset and define the vocabulary, use one hot encoding to encode your smiles strings.

In [34]:
```
# define vocabulary (range of possible letters)
vocab = set()
for smile in smiles:
    for char in smile:
        vocab.add(char)

# FUNCTION TO ADD START AND END CHARACTERS, ENDCODE SMILES STRINGS, AND CONVERT TO TENSORS
def process_smiles_strings(smiles):
    # Add start and end characters, [ ]
    smiles = ['SOS' + str(s) + 'EOS' for s in smiles]

    # define vocabulary (range of possible letters)
```

```
    vocab = set()
    for smile in smiles:
        for char in smile:
            vocab.add(char)

    # create char-to-index mapping
    char_to_idx = {char: i for i, char in enumerate(sorted(vocab))}

    # one-hot encode SMILES strings
    X = []
    # find length of longest string
    max_len = max(len(smile) for smile in smiles)
    for smile in smiles:
        encoded_smile = []
        for char in smile:
            encoded_smile.append(char_to_idx[char])
        # pad with zeros if shorter than max length
        encoded_smile.extend([0] * (max_len - len(encoded_smile)))
        X.append(encoded_smile)

    # convert X to torch tensor
    X = torch.tensor(X, dtype=torch.long)

    # shift targets by one (remember the RNN will predict the next character given a previous character)
    y = X.clone()
    y[:, :-1] = y[:, 1:]
    y[:, -1] = X[:, 0]

    return X
```

In [35]:

```
# PROCESS SMILE STRINGS WITH FUNCTION
encoded_smiles = process_smiles_strings(smiles)
# USE BATCH_GEN FUNCTION
batch_size = 590  # batch size of 590 will give 2 batches
batches = batches_gen(encoded_smiles, batch_size, encoder)
```

1b

Build a LSTM model with 1 recurrent layer. Starting with the starting character and grow a string character by character using model prediction until it reaches a ending character. Look at the string you grown, is it a valid SMILES string?

- use trainer function from past assignments, modify to be RNN with hidden state

In [44]:
```
class LSTM(nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()
        self.n_layers = 1
        self.n_hidden = 32

        self.lstm = nn.LSTM(
            input_size= len(vocab),
            hidden_size= 15,     # rnn hidden unit
            num_layers=1,       # number of rnn layer
            batch_first=True,   # input & output will has batch size as 1s dimension.
                                # e.g. (batch, time_step, input_size)
        )
        self.out = nn.Linear(32, 1)

    def forward(self, x, h_state):
        # x (batch, time_step, input_size)
        # h_state (n_layers, batch, hidden_size)
        # r_out (batch, time_step, hidden_size)
        r_output, h_state = self.lstm(x, h_state)
        outs = self.out(r_output[:, -1, :])  # take only output of last step
        return outs, h_state

    def init_state(self, batchsize):
        return (torch.zeros(self.n_layers, batchsize, self.n_hidden), #hidden state
                torch.zeros(self.n_layers, batchsize, self.n_hidden)) #cell state
```

In [ ]:
```
# initialize model
model = LSTM()
# set to evaluation mode
model.eval()
# define criterion
criterion = nn.MSELoss()

hidden_state = model.init_state(1)
start_char = 'SOS'
generated_smiles = start_char
optimizer = torch.optim.Adam(model.parameters())
hidden_state = model.init_state(1)
```

```
for i in range(16):
    input_tensor = torch.tensor(encoded_smiles[0][i]).unsqueeze(0).unsqueeze(0).float()
    output_tensor = torch.tensor(encoded_smiles[0][i + 1]).unsqueeze(0).unsqueeze(0).float()

    hidden_state = (hidden_state[0].detach(), hidden_state[1].detach())

    prediction, hidden_state = model(input_tensor, hidden_state)

    loss = loss_func(prediction, output_tensor)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

In [ ]:

```
# Defining a method to generate the next character
def predict(net, inputs, h, top_k=None):
        '''
        Given a onehot encoded character, predict the next character.
            Returns the predicted onehot encoded character and the hidden state.
        Arguments:
            net: the lstm model
            inputs: input to the lstm model. shape (batch, time_step/length_of_smiles, input_size) with ba
            h: hidden state (h,c)
            top_k: int. sample from top k possible characters

        '''
        # detach hidden state from history
        h = tuple([each.data for each in h])
        # get the output of the model
        out, h = net(inputs, h)
        # get the character probabilities
        p = out.data

        # get top characters
        if top_k is None:
            top_ch = np.arange(len(net.chars)) #index to choose from
        else:
            p, top_ch = p.topk(top_k)
            top_ch = top_ch.numpy().squeeze()
        # select the likely next character with some element of randomness
        p = p.numpy().squeeze()
        char = np.random.choice(top_ch, p=p/p.sum())
```

```python
        # return the onehot encoded value of the predicted char and the hidden state
        output = np.zeros(inputs.detach().numpy().shape)
        output[:,:,char] = 1
        output = torch.tensor(output,dtype=torch.float)
        return output, h

# Declaring a method to generate new text
def sample(net, encoder, prime=['SOS'], top_k=None):
    """generate a smiles string starting from prime. I use 'SOS' (start of string) and 'EOS'(end of string
    You may need to change this based on your starting and ending character.

    """
    net.eval() # eval mode
    # get initial hidden state with batchsize 1
    h = net.init_state(1)
    # First off, run through the prime characters
    chars=[]
    for ch in prime:
        ch = encoder.transform(np.array([ch]).reshape(-1, 1)).toarray() #(1,17)
        ch = torch.tensor(ch,dtype=torch.float).reshape(1,1,17)
        char, h = predict(net, ch, h, top_k=top_k)
    chars.append(char)
    end  = encoder.transform(np.array(['EOS']).reshape(-1, 1)).toarray()
    end = torch.tensor(end,dtype=torch.float).reshape(1,1,17)

    # Now pass in the previous character and get a new one
    while not torch.all(end.eq(chars[-1])):
        char, h = predict(net, chars[-1], h, top_k=top_k)
        chars.append(char)
    chars =[c.detach().numpy() for c in chars]
    chars = np.array(chars).reshape(-1,17)
    chars = encoder.inverse_transform(chars).reshape(-1)
    return ''.join(chars[:-1])
```