CHEM277B Homework 10

(A)

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.

```
import torch
import numpy as np
from pyanitools import anidataloader

data = anidataloader("ani_gdb_s06.h5")
data_iter = data.__iter__()
#mols = next(data_iter)
sm = []
for mol in data:
  # Extract the data
  sm.append(mol['smiles'])
sm_raw = sm
```

Add SOS and EOS characters

```
print(len(sm))
for idx in range(len(sm)):
  sm[idx].insert(0, 'SOS')
  sm[idx].append('EOS')
```

```
1406
```

```
print(sm[0])
print(type(sm[1][0]))
```

```
['SOS', '[', 'H', ']', 'N', '(', '[', 'H', ']', ')', 'C', '(', '[', 'H', ']',
<class 'str'>
```

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder()
s = ['C', 'O', 'H', 'N', '[', ']', '(', ')', '#', '=', '0', '1', '2', 'c','h', 'o',
```

```
s_np = np.array(s).reshape(-1, 1)
enc.fit(s_np)
```

```
▾ OneHotEncoder
OneHotEncoder()
```

```
import torch.nn as nn
import numpy as np

def batches_gen(smiles, batchsize, encoder):
    '''Create a generator that returns batches of size (batch_size,seq_leng,nchars)
    where seq_leng is the length of the longest smiles string and nchar is the leng

       Arguments
       ---------
       smiles: python list(nsmiles,nchar) smiles array shape you want to make batch
       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
        #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



#testing batches_gen
batches = batches_gen(smiles = sm_raw, batchsize = 1, encoder = enc)
```

(B)

Build an LSTM with one recurrent layer.

```
class LSTM(nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()
        self.n_layers = 1
        self.n_hidden = 64
        self.chars = ['C', 'O', 'H', 'N', '[', ']', '(', ')', '#', '=', '0', '1', '



        self.lstm = nn.LSTM(
            input_size = 19,
            hidden_size = self.n_hidden,
            num_layers = self.n_layers,
            batch_first = True, #(batch, time_step, input_size)


        )
        self.out = nn.Linear(64, 19)

    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_out, h_state = self.lstm(x, h_state)
        outs = self.out(r_out)
        #outs = nn.Softmax(dim=0)(outs)
        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
```

Train the LSTM Model

```
lstm = LSTM()
h_state, c_state = lstm.init_state(128)
print(lstm)

optimizer = torch.optim.Adam(lstm.parameters(), lr = 0.01)
loss_func = nn.CrossEntropyLoss()
```

```
LSTM(
  (lstm): LSTM(19, 64, batch_first=True)
  (out): Linear(in_features=64, out_features=19, bias=True)
)
```

```
for i in range(50):
    batches = batches_gen(smiles = sm, batchsize = 128, encoder = enc)
```

```
    for batch in batches:

      prediction, (h_state, c_state) = lstm(batch[0], (h_state, c_state))   # rnn c
    # !! next step is important !!
#    h_state = h_state.data       # repack the hidden state, break the connection
#     # you can also do
      h_state = h_state.detach()
      c_state = c_state.detach()

      loss = loss_func(prediction, batch[1])         # calculate loss
      optimizer.zero_grad()                        # clear gradients for this training s
      loss.backward()                              # backpropagation, compute gradients
      optimizer.step()

    if i % 5 == 0:
      print(f"Epoch {i} : Loss = {loss}")
```

```
Epoch 0 : Loss = 7.9578022956848145
Epoch 5 : Loss = 4.4139275550842285
Epoch 10 : Loss = 4.0715203285217285
Epoch 15 : Loss = 3.9366157054901123
Epoch 20 : Loss = 3.889498710632324
Epoch 25 : Loss = 3.778305768966675
Epoch 30 : Loss = 3.767561197280884
Epoch 35 : Loss = 3.720902681350708
Epoch 40 : Loss = 3.749427318572998
Epoch 45 : Loss = 3.7441751956939697
```

```
# 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_smil
            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
        p = nn.Softmax(dim=2)(p)

        # 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())
        # return the onehot encoded value of the predicted char and the hidden stat
        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)
    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(1,1,19)
        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(1,1,19)

    # 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(-1,19)
    chars = encoder.inverse_transform(chars).reshape(-1)
    return ''.join(chars[:-1])


#test the LSTM with the sample method

for i in range(20):
  print(sample(lstm, enc))
```

```
=NN=N1
N=C1
O[1o
O1
OC2([H])c1
```

```
N=C1
OC1=NN([H])N([H])N1
[H])N1o
[H])o1
OC#N
0h0)C12N(02)n2
[#])N1
=C1
OC1oc
OC1
=O
C#N
N2N(SOS[0h)[H]
[H])N#
N=NOC1
```

The output of the sample method is nonsensical strings that are not valid SMILES codes. There is an error somewhere in the code which is preventing the model from training further, although it does reduce the Cross Entropy Loss over 50 epochs, it does not find a good minimum and ceases to train past 50 epochs. One could consider two linear layers to increase the capacity of the LSTM model.