MSSE 277B: Machine Learning Algorithms

Recurrent Neural Network, LSTM

Assigned Apr. 13 and Due Apr. 25

Student Name: Charis Liao

LSTM applied to SMILES string generation.

Using the SMILES string from the ANI dataset with upto 6 heavy atoms, build a LSTM generative model that can generate new smiles string with given initial character.

(a) (3pt) 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 [128…

```
# import modules
from pyanitools import anidataloader
import torch
import torch.nn as nn
import numpy as np
from sklearn.preprocessing import OneHotEncoder
```

In [148…

```
# Load data
data = anidataloader("../ANI-1_release/ani_gdb_s06.h5")
data_iter = data.__iter__()

# extract smiles strings
smile_strings = []
for mol in data_iter:
    sm = mol['smiles']
    sm = "".join(sm)
    sm = "^" + sm + "$"
    smile_strings.append(sm)


# print(len(smile_strings))
# print(len(smile_strings[0]))
# print(len(smile_strings[0][0]))
# print(smile_strings)
```

In [149…

```
# Get unique characters
unique_char = list(set("".join(smile_strings)))
print(unique_char)
print(len(unique_char))
```

```
['1', '2', 'c', 'o', 'H', '^', '$', 'n', ')', 'C', '=', '(', 'N', 'O', '[', '#', ']']
17
```

In [150…

```
vocab = {char: i for i, char in enumerate(unique_char)}
encoder = OneHotEncoder(categories=[unique_char], handle_unknown="ignore", sparse=False)
encoder.fit(np.array(unique_char).reshape(-1,1))

translate_smiles = []
smile_indicies = []
for s in smile_strings:
    smile_int = np.array(list(s)).reshape(-1,1)
    smile_indicies.append(smile_int)
    smile_translate = encoder.transform(smile_int)
    translate_smiles.append(smile_translate)

print(len(smile_indicies[0]))
```

```
69
```

(b) (7pt) 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?

In [165…

```
# Define the LSTM model
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super().__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.lstm = nn.LSTM(input_size, hidden_size, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        # Get an output the same size as unique characters


    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(out)
        return out, h

    def init_state(self, batch_size):
        return (torch.zeros(self.n_layers, batch_size, self.hidden_size),
                torch.zeros(self.n_layers, batch_size, self.hidden_size))
```

In [166…

```
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 characters (17)

       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))),dtype=torch.float) \
         for s in smiles]

        #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)

    num_batches = len(arr) // batchsize

    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(arr) % batchsize != 0:
#         num_dropped = len(arr) % batchsize
#         X_last = X[-num_dropped:]
#         y_last = y[-num_dropped:]
#         X = X[:-num_dropped]
#         y = y[:-num_dropped]
#         num_batches -= 1
#         yield X_last, y_last

#     if num_batches == 0:
#         raise ValueError("Batch size is larger than the number of data points.")
```

In [177…

```
# 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 batchsize of 1
            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(17) #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 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=['^'], 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)) #(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(['$']).reshape(-1, 1))
    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])
```

In [168…
```
gen = batches_gen(smile_indicies, 32, encoder)
print(gen)

# Get the first batch
X_batch_test, y_batch_test = next(gen)

# Print the shape of the batch data

print('X_batch shape:', X_batch_test.shape)
print('y_batch shape:', y_batch_test.shape)
```

```
<generator object batches_gen at 0x7fa7e42fd2e0>
X_batch shape: torch.Size([32, 73, 17])
y_batch shape: torch.Size([32, 73, 17])
```

In [169…
```
# Define the hyperparameters
input_size = 17
output_size = 17
hidden_size = 32
n_layers = 1
batch_size = 32
learning_rate = 0.001
num_epochs = 50
```

In [170…
```
# Define the model, loss function, and optimizer
lstm = LSTM(input_size, hidden_size, output_size, n_layers)
optimizer = torch.optim.Adam(lstm.parameters(), lr=learning_rate)   # optimize all cnn parameters
loss_func = nn.MSELoss()
print(lstm)
```

```
LSTM(
  (lstm): LSTM(17, 32, batch_first=True)
  (fc): Linear(in_features=32, out_features=17, bias=True)
)
```

In [172…
```
# Train the model
for epoch in range(num_epochs):
    for i, (x_batch, y_batch) in enumerate(batches_gen(smile_indicies, batch_size, encoder)):
        # Clear gradients
        optimizer.zero_grad()

        # Initialize hidden state
        h_state, c_state = lstm.init_state(batch_size)

        # Forward pass
        y_pred, h = lstm(x_batch, (h_state, c_state))
        h_state = h_state.detach()
        c_state = c_state.detach()
        loss = loss_func(y_pred, y_batch)

        # Backward pass
        loss.backward()
        optimizer.step()

        # Print the loss every 100 batches
        if epoch % 10 == 0 and i % 100 == 0:
            print(f'Epoch {epoch}, Batch {i}, Loss: {loss.item()}')
```

```
Epoch 0, Batch 0, Loss: 0.04570604860782623
Epoch 10, Batch 0, Loss: 0.009050055406987667
Epoch 20, Batch 0, Loss: 0.008065683767199516
Epoch 30, Batch 0, Loss: 0.007313530892133713
Epoch 40, Batch 0, Loss: 0.006817704066634178
```

In [33]:
```
len(translate_smiles[0][0])
```

Out[33]:
```
1
```

In [178…
```
sample(lstm, encoder, prime=['^'], top_k=3)
```

Out[178]:
```
'[H]OC([H])([H])C([H])(C([H])([H()[H1([[H][H]'
```

In [ ]: