CharacterLevelLSTM

November 30, 2023

0.0.1 Character-Level Language Model

This notebook contains a generative model working at the level of characters.

```
[ ]: import numpy as np
     from numpy.random import randint,rand,seed,normal,permutation,choice

     import string
     import math

     import matplotlib.pyplot as plt
     from copy import deepcopy
     from tqdm import tqdm

     import torch
     from torch import nn, optim
     import torch.nn.functional as F
     from torch.utils.data import random_split,Dataset,DataLoader



     # from torchsummary import summary                     # must install using pip␣
      ↪install torchsummary
```

Load a text file. We chose a poem, to see how it did with line breaks.

```
[ ]: with open("concatenatedJavaFiles.txt", "r") as text_file:
         text = text_file.read()

     text[:100]
```

```
[ ]: 'public class Graph {\n    \n    private static boolean isEdge(Vertex u, Vertex
     v, boolean [][] E) {\n  '
```

No normalization will be performed, however, we will run out of RAM if we attempt to use the entire poem as data. We have chosen here to use 10K characters, out of a total of

```
[ ]: print(f"Text is {len(text)} characters long.")
```

```
size = 10000

text = text[:size]
```

```
Text is 52033 characters long.
```

Next we figure out how many distinct characters there are in the text; this will be what is generated at each step of the generation.

[ ]:
```
chars_in_text = sorted(list(set(text)))

num_chars = len(chars_in_text)

print(f'There are {num_chars} characters in the text.')


print(f'Character set: {chars_in_text}.')
```

```
There are 71 characters in the text.
Character set: ['\n', ' ', '!', '"', '(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', ';', '<', '=', '>', 'A', 'B', 'C',
'D', 'E', 'F', 'G', 'H', 'I', 'M', 'O', 'P', 'Q', 'R', 'S', 'T', 'V', '[', ']',
'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', '{', '}'].
```

[ ]:
```
# Create functions mapping characters to integers and back

def char2int(c):
    return chars_in_text.index(c)

def int2char(i):
    return chars_in_text[i]
```

As we're going to predict the next character in the sequence at each time step, we'll have to divide each sentence into

- Input data
  - The last input character should be excluded as it does not need to be fed into the model
- Target/Ground Truth Label
  - One time-step ahead of the Input data as this will be the "correct answer" for the model at each time step corresponding to the input data

The sample length is a critical parameter which tells us how much of the source data to ingest at each training step. You might want to play around with this as one of the hyperparameters.

[ ]:
```
sample_len = 100

# Creating lists that will hold our input and target sample sequences
```

```python
input_seq_chars = []
target_seq_chars = []

for k in range(len(text)-sample_len+1):

    # Remove last character for input sequence
    input_seq_chars.append(text[k:k+sample_len-1])

    # Remove firsts character for target sequence
    target_seq_chars.append(text[k+1:k+sample_len])

for i in range(5):
    print(f'Input sequence:\n{input_seq_chars[i]}')
    print(f'Target sequence:\n{target_seq_chars[i]}')
    print()
```

```
Input sequence:
public class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {

Target sequence:
ublic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {


Input sequence:
ublic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {

Target sequence:
blic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {


Input sequence:
blic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {

Target sequence:
lic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {
```

```
Input sequence:
lic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {

Target sequence:
ic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {


Input sequence:
ic class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {

Target sequence:
c class Graph {

    private static boolean isEdge(Vertex u, Vertex v, boolean [][] E) {
```

Now we can convert our input and target sequences to sequences of integers instead of characters by mapping them using the functions we created above. This will allow us to one-hot-encode our input sequence later.

```
[ ]: input_seq = []
     target_seq = []

     for i in range(len(input_seq_chars)):
         input_seq.append( [char2int(ch) for ch in input_seq_chars[i]])
         target_seq.append([char2int(ch) for ch in target_seq_chars[i]])

     print(input_seq[0])
```

```
[59, 64, 45, 55, 52, 46, 1, 46, 55, 44, 62, 62, 1, 31, 61, 44, 59, 51, 1, 69, 0,
1, 1, 1, 1, 0, 1, 1, 1, 1, 59, 61, 52, 65, 44, 63, 48, 1, 62, 63, 44, 63, 52,
46, 1, 45, 58, 58, 55, 48, 44, 57, 1, 52, 62, 29, 47, 50, 48, 4, 41, 48, 61, 63,
48, 67, 1, 64, 8, 1, 41, 48, 61, 63, 48, 67, 1, 65, 8, 1, 45, 58, 58, 55, 48,
44, 57, 1, 42, 43, 42, 43, 1, 29, 5, 1, 69, 0, 1]
```

```
[ ]: # convert an integer into a one-hot encoding of the given size (= number of␣
     ↪characters)
     def int2OneHot(X,size):
```

```
    def int2OneHot1(x,size=10):
        tmp = np.zeros(size)
        tmp[int(x)] = 1.0
        return tmp

    return np.array([ int2OneHot1(x, size) for x in X ]).astype('double')

int2OneHot( np.array([ 2,3,1,2,3,4 ]),10)
```

```
[ ]: array([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
            [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
            [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
            [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
            [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
            [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]])
```

```
[ ]: # do the same thing, but for a list/array of integers

def seq2OneHot(seq,size):
    return np.array([ int2OneHot(x, size) for x in seq ])

seq2OneHot( np.array([[ 2,3,1,2,3,4 ],[ 2,3,1,2,3,4 ],[ 2,3,1,2,3,4 ]]),10)
```

```
[ ]: array([[[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]],

            [[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]],

            [[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]])
```

```
[ ]: # Convert our input sequences to one-hot form

input_seq = seq2OneHot(input_seq,size=num_chars)
```

```
input_seq.shape
```

```
[ ]: (9901, 99, 71)
```

```
[ ]: # Convert our target sequences to one-hot form

     target_seq = seq2OneHot(target_seq,size=num_chars)
     target_seq.shape
```

```
[ ]: (9901, 99, 71)
```

Since we're done with all the data pre-processing, we can now move the data from numpy arrays to tensors.

```
[ ]: input_seq = torch.Tensor(input_seq).type(torch.DoubleTensor)
     target_seq = torch.Tensor(target_seq).type(torch.DoubleTensor)
```

Now we will build a data loader to manage the batching.

```
[ ]: class Basic_Dataset(Dataset):

         def __init__(self, X,Y):
             self.X = X
             self.Y = Y

         def __len__(self):
             return len(self.X)

         # return a pair x,y at the index idx in the data set
         def __getitem__(self, idx):
             return self.X[idx], self.Y[idx]

     ds = Basic_Dataset(input_seq,target_seq)

     ds.__len__()
```

```
[ ]: 9901
```

Batch size is a hyperparameter that will mostly determine how efficiently you can process the data on a GPU.

```
[ ]: # hyperparameters
     hidden_dims = [128, 256, 512] # Different sizes for the hidden dimension
     n_layers_options = [1, 2, 3] # Number of LSTM layers
     dropout_options = [0.0, 0.2, 0.5] # Dropout rates
     learning_rates = [0.001, 0.0005, 0.0001] # Learning rates
     epochs_options = [10, 20, 30] # Number of epochs
     batch_sizes = [64, 128, 256]
```

```
# batch_size = 128

# data_loader = DataLoader(ds, batch_size=batch_size, shuffle=True)
```

Check if a GPU is available and use it if it is.

```
[ ]: # torch.cuda.is_available() checks and returns a Boolean True if a GPU is␣
      ↪available, else it'll return False
     is_cuda = torch.cuda.is_available()

     # If we have a GPU available, we'll set our device to GPU. We'll use this␣
      ↪device variable later in our code.
     if is_cuda:
         device = torch.device("cuda")
         print("GPU is available")
     else:
         device = torch.device("cpu")
         print("GPU not available, CPU used")
```

```
GPU not available, CPU used
```

The model will use an LSTM layer and a single linear layer to produce a softmax of the next character. Various hyperparameters can be chosen to modify this model. A messy detail is that two vectors, h0 and c0, have to be created for the hidden state in the LSTM layer (these correspond to the two connections shown in lecture for an LSTM neuron to send to itself in the next time step).

```
[ ]: from os import device_encoding
     class Model(nn.Module):
         def __init__(self, input_size, output_size, hidden_dim, n_layers,dropout):
             super(Model, self).__init__()

             # Defining some parameters
             self.hidden_dim = hidden_dim
             self.n_layers = n_layers

             #Defining the layers
             self.lstm = nn.LSTM(input_size, hidden_dim,␣
      ↪n_layers,dropout=dropout,batch_first=True)
             # Fully connected layer
             self.fc1 = nn.Linear(hidden_dim, output_size)

         def forward(self, x):

             hidden_state_size = x.size(0)

             x = x.to(torch.double)
```

```
        h0 = torch.zeros(self.n_layers,hidden_state_size,self.hidden_dim).
↪double().to(device)
        c0 = torch.zeros(self.n_layers,hidden_state_size,self.hidden_dim).
↪double().to(device)

        self.lstm = self.lstm.double()

        self.fc1 = self.fc1.double()

        # Passing in the input and hidden state into the model and obtaining␣
↪outputs
        out, (hx,cx) = self.lstm(x, (h0,c0))

        # Reshaping the outputs such that it can be fit into the fully␣
↪connected layer
        out = out.contiguous().view(-1, self.hidden_dim)
        out = self.fc1(out)

        return out
```

[ ]:
```
# hyperparameters
hidden_dims = [128, 256, 512]
n_layers_options = [1, 2, 3]
dropout_options = [0.0, 0.2, 0.5]
learning_rates = [0.001, 0.0005, 0.0001]
epochs_options = [10, 20, 30]
batch_sizes = [64, 128, 256]

best_loss = float('inf')
best_model_info = ""

for hidden_dim in hidden_dims:
    for n_layers in n_layers_options:
        for dropout in dropout_options:
            for lr in learning_rates:
                for num_epochs in epochs_options:
                    for batch_size in batch_sizes:
                        # data Loader for current batch size
                        data_loader = DataLoader(ds, batch_size=batch_size,␣
↪shuffle=True)

                        # instantiate the model with current set of␣
↪hyperparameters
                        model = Model(input_size=num_chars,␣
↪output_size=num_chars, hidden_dim=hidden_dim, n_layers=n_layers,␣
↪dropout=dropout)
```

```
                        model = model.double().to(device)

                        # define Loss and Optimizer with current learning rate
                        optimizer = torch.optim.Adam(model.parameters(), lr=lr)
                        loss_fn = nn.CrossEntropyLoss()

                        # training Loop
                        losses = []
                        model.train()
                        for epoch in tqdm(range(num_epochs)):
                            for input_seq_batch, target_seq_batch in␣
 ↪data_loader:
                                input_seq_batch = input_seq_batch.to(device)
                                target_seq_batch = target_seq_batch.to(device)
                                optimizer.zero_grad()
                                target_seq_hat = model(input_seq_batch)
                                loss = loss_fn(target_seq_hat, target_seq_batch.
 ↪view(-1, num_chars))
                                loss.backward()
                                optimizer.step()

                            losses.append(loss.item())

                        # check if current model is the best
                        avg_loss = sum(losses) / len(losses)
                        if avg_loss < best_loss:
                            best_loss = avg_loss
                            best_model_info =␣
 ↪f'hidden{hidden_dim}_layers{n_layers}_dropout{dropout}_lr{lr}_epochs{num_epochs}_batch{batc
                            # Save the model
                            torch.save(model.state_dict(),␣
 ↪f'best_model_{best_model_info}.pth')

                        plt.figure()
                        plt.title(f'Loss - Hidden:{hidden_dim}, Layers:
 ↪{n_layers}, Dropout:{dropout}, LR:{lr}, Epochs:{num_epochs}, Batch:
 ↪{batch_size}')
                        plt.plot(losses)
                        plt.savefig(f'loss_{best_model_info}.png')
                        plt.close()

print(f"The best model is with {best_model_info} having a loss of {best_loss}")
```

```
100%|      | 10/10 [02:25<00:00, 14.57s/it]
100%|      | 10/10 [01:59<00:00, 11.96s/it]
100%|      | 10/10 [01:40<00:00, 10.01s/it]
100%|      | 20/20 [05:01<00:00, 15.07s/it]
```

```
100%|     | 20/20 [04:02<00:00, 12.11s/it]
100%|     | 20/20 [03:21<00:00, 10.07s/it]
100%|     | 30/30 [07:24<00:00, 14.83s/it]
100%|     | 30/30 [06:07<00:00, 12.25s/it]
100%|     | 30/30 [05:06<00:00, 10.21s/it]
100%|     | 10/10 [02:30<00:00, 15.04s/it]
100%|     | 10/10 [01:58<00:00, 11.87s/it]
100%|     | 10/10 [01:38<00:00,  9.86s/it]
100%|     | 20/20 [05:01<00:00, 15.05s/it]
100%|     | 20/20 [03:58<00:00, 11.90s/it]
100%|     | 20/20 [03:20<00:00, 10.01s/it]
100%|     | 30/30 [07:24<00:00, 14.81s/it]
100%|     | 30/30 [06:00<00:00, 12.00s/it]
100%|     | 30/30 [05:03<00:00, 10.11s/it]
100%|     | 10/10 [02:29<00:00, 14.96s/it]
100%|     | 10/10 [01:56<00:00, 11.69s/it]
100%|     | 10/10 [01:40<00:00, 10.02s/it]
100%|     | 20/20 [05:01<00:00, 15.05s/it]
100%|     | 20/20 [03:56<00:00, 11.84s/it]
100%|     | 20/20 [03:19<00:00,  9.97s/it]
100%|     | 30/30 [07:24<00:00, 14.83s/it]
100%|     | 30/30 [05:56<00:00, 11.89s/it]
100%|     | 30/30 [04:56<00:00,  9.87s/it]
/Users/yfsong/Library/Python/3.9/lib/python/site-
packages/torch/nn/modules/rnn.py:82: UserWarning: dropout option adds dropout
after all but last recurrent layer, so non-zero dropout expects num_layers
greater than 1, but got dropout=0.2 and num_layers=1
  warnings.warn("dropout option adds dropout after all but last "
 10%|         | 1/10 [00:28<04:18, 28.69s/it]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/Users/yfsong/Desktop/CS 505/homework/CharacterLevelLSTM.ipynb Cell 28 line 3

     <a href='vscode-notebook-cell:/Users/yfsong/Desktop/CS%20505/homework/
 ↪CharacterLevelLSTM.ipynb#X41sZmlsZQ%3D%3D?line=36'>37</a>     target_seq_hat =␣
 ↪model(input_seq_batch)
     <a href='vscode-notebook-cell:/Users/yfsong/Desktop/CS%20505/homework/
 ↪CharacterLevelLSTM.ipynb#X41sZmlsZQ%3D%3D?line=37'>38</a>     loss =␣
 ↪loss_fn(target_seq_hat, target_seq_batch.view(-1, num_chars))
---> <a href='vscode-notebook-cell:/Users/yfsong/Desktop/CS%20505/homework/
 ↪CharacterLevelLSTM.ipynb#X41sZmlsZQ%3D%3D?line=38'>39</a>     loss.backward()
     <a href='vscode-notebook-cell:/Users/yfsong/Desktop/CS%20505/homework/
 ↪CharacterLevelLSTM.ipynb#X41sZmlsZQ%3D%3D?line=39'>40</a>     optimizer.step(
     <a href='vscode-notebook-cell:/Users/yfsong/Desktop/CS%20505/homework/
 ↪CharacterLevelLSTM.ipynb#X41sZmlsZQ%3D%3D?line=41'>42</a> losses.append(loss.
 ↪item())
```

```
File ~/Library/Python/3.9/lib/python/site-packages/torch/_tensor.py:492, in␣
 ↪Tensor.backward(self, gradient, retain_graph, create_graph, inputs)
    482 if has_torch_function_unary(self):
    483     return handle_torch_function(
    484         Tensor.backward,
    485         (self,),
   (…)
    490         inputs=inputs,
    491     )
--> 492 torch.autograd.backward(
    493     self, gradient, retain_graph, create_graph, inputs=inputs
    494 )

File ~/Library/Python/3.9/lib/python/site-packages/torch/autograd/__init__.py:
 ↪251, in backward(tensors, grad_tensors, retain_graph, create_graph,␣
 ↪grad_variables, inputs)
    246     retain_graph = create_graph
    248 # The reason we repeat the same comment below is that
    249 # some Python versions print out the first line of a multi-line functio
    250 # calls in the traceback and some print out the last line
--> 251 Variable._execution_engine.run_backward(  # Calls into the C++ engine t␣
 ↪run the backward pass
    252     tensors,
    253     grad_tensors_,
    254     retain_graph,
    255     create_graph,
    256     inputs,
    257     allow_unreachable=True,
    258     accumulate_grad=True,
    259 )

KeyboardInterrupt:
```

[ ]:
```
best_model_info
```

[ ]:
```
'hidden128_layers1_dropout0.0_lr0.001_epochs30_batch64'
```

The temperature of a softmax function will determine the relative strength of different probabilities: - As temperature approaches 0, distribution approaches a one-hot with 1 for the max - As temperature increases, it approaches a uniform distribution

Generally we want to emphasize the higher probabilities, so we choose a reasonably low temperature.

[ ]:
```python
def softmax_with_temperature(vec, temperature):
    sum_exp = sum(math.exp(x/temperature) for x in vec)
    return [math.exp(x/temperature)/sum_exp for x in vec]

print("Example of softmax with temperature.")
```

```
dist = [0.1, 0.3, 0.6]
print('distribution:',dist)
print(softmax_with_temperature(dist,0.01))
print(softmax_with_temperature(dist,0.1))
print(softmax_with_temperature(dist,0.2))
print(softmax_with_temperature(dist,0.3))
print(softmax_with_temperature(dist,1))
print(softmax_with_temperature(dist,10))
```

```
Example of softmax with temperature.
distribution: [0.1, 0.3, 0.6]
[1.9287498479637375e-22, 9.3576229688393e-14, 0.9999999999999064]
[0.006377460922442302, 0.04712341652466416, 0.9464991225528936]
[0.06289001324586753, 0.1709527801977903, 0.7661572065563421]
[0.12132647558421489, 0.23631170657656433, 0.6423618178392208]
[0.2583896517379799, 0.3155978333128144, 0.4260125149492058]
[0.3255767455856355, 0.3321538321280155, 0.3422694222863489]
```

Choose a temperature and predict the next character, given a prompt of arbitrary length.

```
[ ]: temperature = 0.3

def predict(model, ch):

    # only look at last sample_len - 1 characters

    ch = ch[-(sample_len - 1):]

    # One-hot encoding our input to fit into the model
    ch = np.array([char2int(c) for c in ch])
    ch = np.array([int2OneHot(ch, num_chars)])
    ch = torch.from_numpy(ch).to(device)

    out = model(ch)

    # take the probability distribution of the last character in the sequence␣
 ↪produced by the model
    prob = softmax_with_temperature(out[-1],temperature)

    # Choosing a character based on the probability distribution, with␣
 ↪temperature
    char_ind = choice(list(range(num_chars)), p=prob)

    return int2char(char_ind)

predict(model,"for(int i = 0; i < V.length; ++i) {")
```

```
[ ]: ' '
```

Now take a prompt and iterate the previous prediction a specified number of times.

Prompt is generally taken to be a long sequence randomly selected from the text. You can also try a sequence of words similar to those in the text, but not an exact sequence. It does not have to be the exact length of the data sequences. However, very short prompts tend not to work as well.

```
[ ]: def sample(model, out_len, start):
         model.eval() # eval mode
         # First off, run through the starting characters
         chars = [ch for ch in start]
         size = out_len - len(chars)
         # Now pass in the previous characters and get a new one
         for ii in range(size):
             char = predict(model, chars)
             chars.append(char)

         return ''.join(chars)
```

Now we will run our model, but with the parameters we have chosen, and 10 epochs, you can see that it is getting some idea of words and lines, but it doesn't look like an English poem!

Run this for another 100 epochs, and observe that at that point, the network will have simply memorized the poem!

```
[ ]: '''
     results produced by the model that simply memorizes the poem
     '''
     print(sample(model, 1000, "for(int i = 0; i < V.length; ++i) {"))
```

```
for(int i = 0; i < V.length; ++i) {
             if(!V[j].visited)
```

```
[ ]: '''
     results produced by the best model
     '''
     print(sample(model, 1000, "for(int i = 0; i < V.length; ++i) {"))
```

```
for(int i = 0; i < V.length; ++i) {
```