Lab 09 - Character-level Language Model with LSTM

In this lab, your task is to build a character-level language model with LSTM layer.

Reference: Let's build GPT: from scratch, in code, spelled out (by Andrej Karpathy)

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
cd "/content/drive/MyDrive/UCCD3074_Labs/UCCD3074_Lab9"
```

```
/content/drive/MyDrive/UCCD3074_Labs/UCCD3074_Lab9
```

In [ ]:
```
import os
import torch
import torch.nn as nn
from torch.nn import functional as F

import time
```

In [ ]:
```
torch.manual_seed(1234)
device = 'cuda' if torch.cuda.is_available() else "cpu"
```

In [ ]:
```
if not os.path.exists('input.txt'):
    !wget 'https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt'
```

Load the dataset

Read the dataset into the string `raw_data`.

In [ ]:
```
with open('./input.txt', 'r', encoding='utf-8') as f:
    raw_data = f.read()
```

```
# print the length of the datasets
print('length of dataset in characterse:', len(raw_data))

# look at the first 1000 characters
print(raw_data[:100])
```

```
length of dataset in characterse: 1115394
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You
```

Create the vocabulary

Get the the vocabulary `vocab` from the raw data. `vocab` contains all unique characters in the raw data.

In [ ]:
```
vocab = sorted(list(set(raw_data)))
vocab_size=len(vocab)
print('vocab:', vocab)
print('vocab_size:', vocab_size)
```

```
vocab: ['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?', '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', 'Z', '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', 'z']
vocab_size: 65
```

Create the vocabulary mapping functions. `stoi` performs the mapping from the character token to its index, and `itos`, vice versa.

In [ ]:
```
stoi = {ch:i for i, ch in enumerate(vocab)}
itos = {i:ch for i, ch in enumerate(vocab)}
```

Create the function to encode and decode the text. `encode_text` encodes a string into its one-hot integer representation while `decode_text` decodes an integer representation back to its text.

In [ ]:
```
encode_text = lambda s : [stoi[c] for c in s]    # encode: take a string, output a list of integers
encoded = encode_text('Hello how do you do?')
print(encoded)
```

```
[20, 43, 50, 50, 53, 1, 46, 53, 61, 1, 42, 53, 1, 63, 53, 59, 1, 42, 53, 12]
```

In [ ]:
```
decode_text = lambda l : ''.join([itos[i] for i in l])
decoded = decode_text(encoded)
print(decoded)
```

```
Hello how do you do?
```

Now, we encode the raw data and then convert it into a 1-D integer tensor `data`.

In [ ]:
```
data = torch.tensor(encode_text(raw_data), dtype=torch.long)
print('Shape of data:', data.shape)
print('Type of data: ', data.dtype)
print('\nFirst 100 characters of data:\n', data[:100])
```

```
Shape of data: torch.Size([1115394])
Type of data:  torch.int64

First 100 characters of data:
 tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59])
```

To train the language model, the samples will be trained with a block of text with `block_size` characters. The function `get_batch` randomly sample a block of text as input `x`. The label `y` is the block of text shifted by 1 position of `x`.

In [ ]:
```
torch.manual_seed(1234)

def get_batch(batch_size, block_size):
    ix    = torch.randint(len(data) - block_size, (batch_size,))
    x     = torch.stack([data[i:i+block_size] for i in ix])
    y     = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

In [ ]:
```
batch_size=4
block_size=8
```

In [ ]:
```
x, y = get_batch(batch_size, block_size)
print(x.shape)
print(y.shape)
```

```
torch.Size([4, 8])
torch.Size([4, 8])
```

Note that when training on `(x, y)`, the model is learning multiple conditional probabilities `p(target|context)` simultaneously.

In [ ]:
```
x = data[:block_size]
y = data[1:block_size+1]

print('x:', x.tolist())
print('y:', y.tolist(), '\n')

for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f'when input is: {context}, the target is: {target}')
```

```
x: [18, 47, 56, 57, 58, 1, 15, 47]
y: [47, 56, 57, 58, 1, 15, 47, 58]

when input is: tensor([18]), the target is: 47
when input is: tensor([18, 47]), the target is: 56
when input is: tensor([18, 47, 56]), the target is: 57
when input is: tensor([18, 47, 56, 57]), the target is: 58
when input is: tensor([18, 47, 56, 57, 58]), the target is: 1
when input is: tensor([18, 47, 56, 57, 58,  1]), the target is: 15
when input is: tensor([18, 47, 56, 57, 58,  1, 15]), the target is: 47
when input is: tensor([18, 47, 56, 57, 58,  1, 15, 47]), the target is: 58
```

Create the character-level language model with LSTM

Network structure

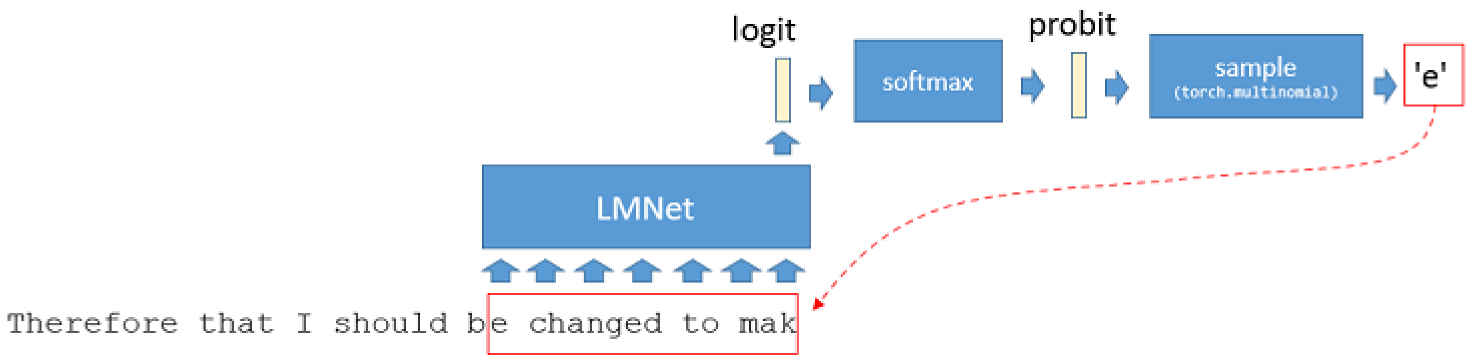
Now, let's build the character-level language model with LSTM. The network `LMNet` has the following layers:

Layer	Configuration	Shape
Input	-	`(B, T)`
Embedding	num_embedding = vocab_size, embedding_dim = n_embd	`(B, T, n_embd)`
LSTM	input_size = n_embd, hidden_size = n_embd num_layers = 1, batch_first = true	`(B, T, n_embd)`
fc	in_features = n_embd, out_features = vocab_size	`(B, T, vocab_size)`

Generating novel text

To generate novel text, we implement the method `generate`. Since the network is trained on a sequence of length `T` = `block_size`, when generating the text, we feed the most recent `block_size` characters into the network to generate the next character. Here are the steps:

1. Crop the most recent `block_size` characters in the generated text
2. The cropped text is fed to the generative model to generate the next character. The network output the `logit` value.
3. Convert the logit of the network to `probit` value by performing `softmax` operation.
4. Sample a character from the `probit` by using `torch.multinorm`
5. Append the sampled character to the end of the generated text.
6. Repeat steps 1-5 for `text_len` times



```
In [ ]: class LMNet(nn.Module):
    def __init__(self, vocab_size, n_embd):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size, n_embd)
        self.lstm            = nn.LSTM(input_size=n_embd, hidden_size=n_embd, num_layers=1, batch_first=True)
        self.fc              = nn.Linear(n_embd, vocab_size)

    def forward(self, x):                 # (B,T)
        x = self.token_embedding(x)   # (B,T,n_embd)
        x, _ = self.lstm(x)           # (B,T,n_embd)
        x = self.fc(x)                # (B,T,vocab_size)

        return x

    def generate(self, text_len, block_size):

        text = torch.zeros((1,1), dtype=torch.long).to(device)  # text token

        # repeat until the length of text = "text_len"
        for _ in range(text_len):

            # crop text to the last block-size tokens
            text_cond = text[:, -block_size:]

            # get the predictions
            yhat  = self(text_cond)  # logits: (B, T, C)

            # focus oly on the last time step
            yhat = yhat[:, -1, :] # becomes (B, C)

            # apply soft max to get probabilities
            probs = F.softmax(yhat, dim=-1)  # (B, C)

            # sample from distribution
            next_token = torch.multinomial(probs, num_samples=1)  # (B, 1)

            # append sampled index to the running sequence
            text = torch.cat((text, next_token), dim=1) # (B, T+1)

            # print the sample
            print(itos[next_token.item()], end='')
            time.sleep(0.1)
```

Create the model for testing

```
model = LMNet(vocab_size=len(vocab), n_embd=32).to(device)
```

```
x, y = get_batch(batch_size=4, block_size=8)
x, y = x.to(device), y.to(device)

yhat = model(x)

print(x.shape)
print(yhat.shape)
```

```
torch.Size([4, 8])
torch.Size([4, 8, 65])
```

Train the model

```
max_iters     = 5000
batch_size    = 128
block_size    = 256
lr            = 3e-4
max_iters     = 10000
show_interval = 500
n_embd        = 256
```

Create the model

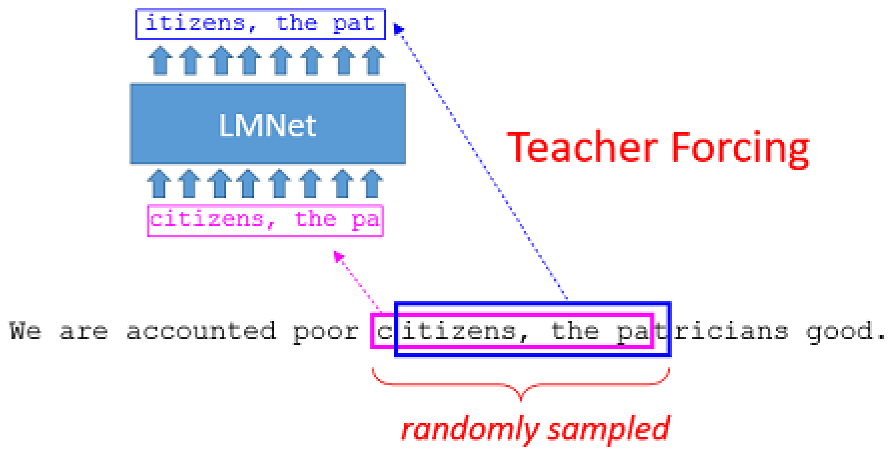
```
model = LMNet(vocab_size=len(vocab), n_embd=n_embd).to(device)
```

Create the optimizer

```
optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
```

Train the model

To train the model, we use **teacher forcing** where the predicted output is simply the 1-shifted sequence of the input sequence. We shall train the network with sentence sequence of length `block_size`. Since the network is trained on sequences of length `block_size`, during inference, the generative model should use input sequence of similar length to get good results.



During training, the network is based on the many-to-many architecture. However, during inference (Figure at generating novel text), the network is based on many-to-one architecture.

In [ ]:
```
for steps in range(max_iters):

    # sample a batch of data
    x, y = get_batch(batch_size, block_size)

    # forward propagation
    yhat = model(x)

    # compute loss
    B, T, C = yhat.shape
    yhat = yhat.view(B*T, C)
    y = y.view(B*T)
    loss = F.cross_entropy(yhat, y)

    # backpropagation
    loss.backward()
    optimizer.step()

    # reset the optimizer
```

```python
    optimizer.zero_grad()

    # print the training loss
    if steps % show_interval == 0:
        print(f"Iter {steps}: train loss {loss:.4f}")
```

```
Iter 0: train loss 4.1738
Iter 500: train loss 1.8172
Iter 1000: train loss 1.6056
Iter 1500: train loss 1.4905
Iter 2000: train loss 1.4199
Iter 2500: train loss 1.3855
Iter 3000: train loss 1.3280
Iter 3500: train loss 1.3207
Iter 4000: train loss 1.3075
Iter 4500: train loss 1.2774
Iter 5000: train loss 1.2612
Iter 5500: train loss 1.2214
Iter 6000: train loss 1.2395
Iter 6500: train loss 1.2026
Iter 7000: train loss 1.1993
Iter 7500: train loss 1.1891
Iter 8000: train loss 1.1983
Iter 8500: train loss 1.1793
Iter 9000: train loss 1.1513
Iter 9500: train loss 1.1721
```

Generate text

In [ ]:
```python
model.generate(text_len=1000, block_size=block_size)
```

I would have not hear.

VOLUMNIA:
O, speak with!

FRIAR LAURENCE:
That rules are us, it.

AUTOLYCUS:
Now the arms of course and pierch and there;
One for him and the grace or Petant.
Come, dear, sir, he borne his tender peeds,
Which bright but some pardon my cozning one
Shall forse where'st never cousins; wed patien.

QUEEN MARGARET:
Even wingest you aburth, doing me that last,
Will to me to happy have I within.

ANGELO:
He, sign lay instruct you.
Alack when he sweet in this learn'st, gring
His pale prirold in this world behord of
Your revenge and leave makery aught whose news,
Like dially not followed us.

ANGELO:
Call John. Hark you you agan.

BENVOLIO:
She shall be speak; for best dead with your lovick
Hath writting me no with want for me.

QUEEN MARGARET:
For my forceed take her. John in good, 'tis door.

DUCHESS OF YORK:
Which hath sing me foolicries?

LEONTES:
Prother:
Let's have I thank you.

MENENIUS:

I will play the book, and upon thee at his country
You shall be my father was