
▾ Lab 10 - Language Modeling with Transformers (Guide)

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

```
1 from google.colab import drive
2 drive.mount('/content/drive')

Mounted at /content/drive

1 cd "/content/drive/MyDrive/UCCD3074_Labs/UCCD3074_Lab10"

/content/drive/MyDrive/UCCD3074_Labs/UCCD3074_Lab10

1 import os
2 import math
3 import torch
4 import torch.nn as nn
5 from torch.nn import functional as F
6
7 import time

1 torch.manual_seed(1234)
2 device = 'cuda' if torch.cuda.is_available() else "cpu"

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

▾ Preprocessing

Read the dataset into the string `raw_data`.

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

Create the vocabulary

```
1 vocab = sorted(list(set(raw_data)))
2 vocab_size=len(vocab)
3 print('vocab_size:', vocab_size)

vocab_size: 65

1 stoi = {ch:i for i, ch in enumerate(vocab)}
2 itos = {i:ch for i, ch in enumerate(vocab)}
```

```

1 encode_text = lambda s : [stoi[c] for c in s]    # encode: take a string, output a list of integers
2 decode_text = lambda l : ''.join([itos[i] for i in l])

1 data = torch.tensor(encode_text(raw_data), dtype=torch.long)

```

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`.

```

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

```

▼ Character-level language model with Transformer model

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

Layer	Configuration	Shape
Input	-	(B, T)
Token Embedding	num_embedding = vocab_size, embedding_dim = d_model	(B, T, d_model)
Position Embedding	d_model = d_model	(B, T, d_model)
TransformerEncoder	d_model (default: 512), nhead (default = 6), dim_feedforward, batch_first (default: False)	(B, T, d_model)
fc	in_features = d_model, out_features = vocab_size	(B, T, vocab_size)

In the following, we create a new module class to implement `Position Embedding` in Section A. Then, we implement a language model with an encoder-only Transformer (`TransformerEncoder`) in Section B.

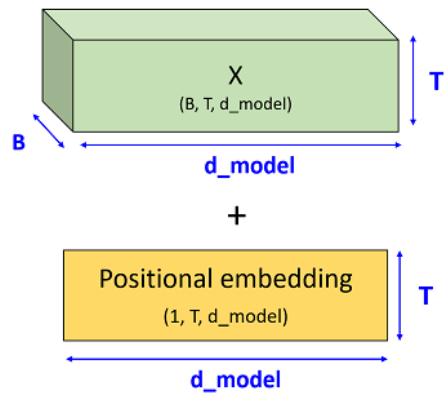
▼ A. Implement the Positional Embedding module

The following `PositionalEncoding` module injects some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed. To do so, we can use sine and cosine functions of different frequencies to embed the distance.

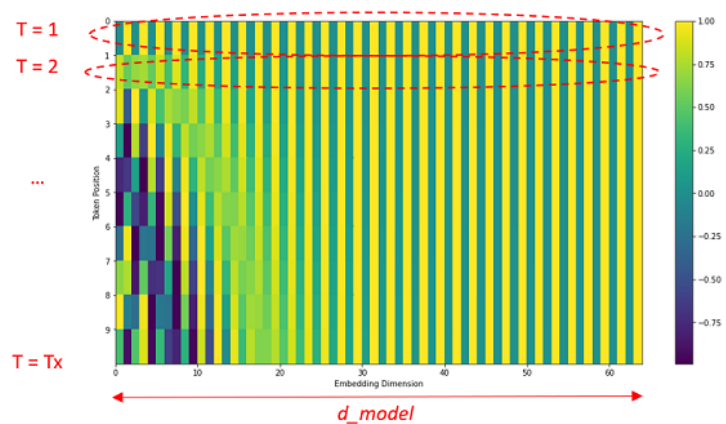
$$PE_{(t,2i)} = \sin(t/10000^{2i/d_{model}})$$

$$PE_{(t,2i+1)} = \cos(t/10000^{2i/d_{model}})$$

where t is the (time) index of the token in the input sequence and i is the index of the embedding feature representing the token



The following diagram shows how the position embedding looks like for different time steps.



Implement the Position Embedding

```

1 class PositionEmbedding(nn.Module):
2
3     def __init__(self, d_model: int, batch_first: bool = False, max_len: int = 5000, dropout: float = 0.1):
4         super().__init__()
5         self.dropout = nn.Dropout(p=dropout)
6         self.batch_first = batch_first
7
8         # Shape of position: (max_len, 1)
9         position = torch.arange(max_len).unsqueeze(1)
10
11         # Shape of div_term: 1d tensor of shape (d_model/2, )
12         div_term1 = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
13         div_term2 = div_term1[:-1] if d_model % 2 == 1 else div_term1
14
15         # shape of pe: (max_len, 1, d_model)

```

```

16     if self.batch_first:
17         pe = torch.zeros(1, max_len, d_model)
18         pe[0, :, 0::2] = torch.sin(position * div_term1)
19         pe[0, :, 1::2] = torch.cos(position * div_term2)
20         self.register_buffer('pe', pe)
21     else:
22         pe = torch.zeros(max_len, 1, d_model)
23         pe[:, 0, 0::2] = torch.sin(position * div_term1)
24         pe[:, 0, 1::2] = torch.cos(position * div_term2)
25         self.register_buffer('pe', pe)
26
27     def forward(self, x):
28         """
29         Arguments:
30             x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
31         """
32         # Shape of self.pe: (T, 1, embedding_dim)
33         # shape of x      : (T, B, embedding_dim)
34         if self.batch_first:
35             x = x + self.pe[:, :x.size(1), :]
36         else:
37             x = x + self.pe[:x.size(0)]
38         return self.dropout(x)

```

Test the position embedding layer.

```

1 batch_size = 4
2 seq_len    = 8
3 d_model    = 512
4
5 x = torch.randn(batch_size, seq_len, d_model)
6 print('Shape of x:', x.shape)

```

```
Shape of x: torch.Size([4, 8, 512])
```

```

1 pos_embedding = PositionEmbedding(d_model=d_model, batch_first=True)
2 xp = pos_embedding(x)
3 print('Shape of xp:', xp.shape)

```

```
Shape of xp: torch.Size([4, 8, 512])
```

```

1 pos_embedding.pe.shape

torch.Size([1, 5000, 512])

```

▼ B. Implement the Language Model

PyTorch offers the necessary [transformer layers](#) to build the transformer model, including:

1. [nn.Transformer](#) - a transformer model. The module is constructed from `nn.TransformerEncoder` and `nn.TransformerDecoder` below.
2. [nn.TransformerEncoder](#) - a stack of N encoder layers
3. [nn.TransformerDecoder](#) - a stack of N decoder layers
4. [nn.TransformerEncoderLayer](#) - made up of multi-head (self) attention and feedforward network

5. [nn.TransformerDecoderLayer](#) - made up of multi-head (self) attention, multi-head (encoder) attention and feedforward network

To implement a language model, we can use an Encoder-only Transformer. The network can be constructed with `nn.TransformerEncoder` and `nn.TransformerEncoderLayer` in two steps:

1. Create an encoder layer with `nn.TransformerEncoderLayer`.
 - `d_model` (int) – the number of expected features in the input (required).
 - `nhead` (int) – the number of heads in the multiheadattention models (required).
 - `dim_feedforward` (int) – the dimension of the feedforward network model (default=2048).
 - `batch_first` (bool) – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False (seq, batch, feature).
2. Create the encoder block with `nn.TransformerEncoder` by stacking multiple encoder layers
 - `encoder_layer` – an instance of the `TransformerEncoderLayer()` class (required).
 - `num_layers` – the number of sub-encoder-layers in the encoder (required).

The `nn.TransformerEncoder` module

The `nn.TransformerEncoder` implements a Transformer's Encoder block by stacking multiple encoder layers.

```
CLASS torch.nn.TransformerEncoder(encoder_layer, num_layers, norm=None,  
enable_nested_tensor=True, mask_check=True) [SOURCE]
```



`TransformerEncoder` is a stack of `N` encoder layers. Users can build the BERT(<https://arxiv.org/abs/1810.04805>) model with corresponding parameters.

Parameters:

- **`encoder_layer`** – an instance of the `TransformerEncoderLayer()` class (required).
- **`num_layers`** – the number of sub-encoder-layers in the encoder (required).
- **`norm`** – the layer normalization component (optional).
- **`enable_nested_tensor`** – if True, input will automatically convert to nested tensor (and convert back on output). This will improve the overall performance of `TransformerEncoder` when padding rate is high. Default: `True` (enabled).

A `nn.TransformerEncoder` object receives an encoder layer object, i.e., an instance of the `nn.TransformerEncoderLayer` class. Then, it creates and stacks `num_layers` copies of the encoder layer instance to form the Encoder-only Network.

The input to the and the output of all layers have the same dimensionality, i.e., (B, T, d_{model}) if `batch_first = True`. The `batch_first` setting is pre-configured in the encoder layer object.


```
class torch.nn.TransformerEncoderLayer(d_model, nhead, dim_feedforward=2048, dropout=0.1,
activation=<function relu>, layer_norm_eps=1e-05, batch_first=False,
norm_first=False, device=None, dtype=None) [SOURCE]
```

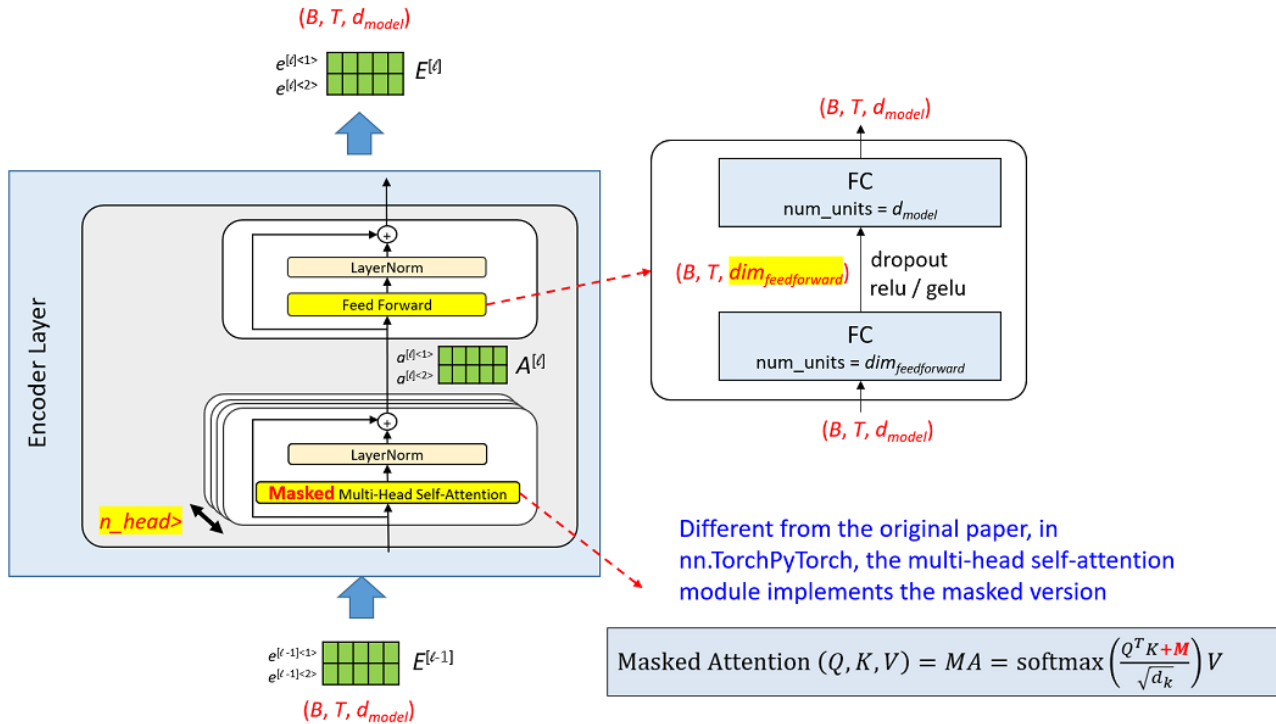
TransformerEncoderLayer is made up of self-attn and feedforward network. This standard encoder layer is based on the paper “Attention Is All You Need”. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000-6010. Users may modify or implement in a different way during application.

Parameters:

- **d_model** (*int*) – the number of expected features in the input (required).
- **nhead** (*int*) – the number of heads in the multiheadattention models (required).
- **dim_feedforward** (*int*) – the dimension of the feedforward network model (default=2048).
- **dropout** (*float*) – the dropout value (default=0.1).
- **activation** (*Union[str, Callable[[Tensor], Tensor]]*) – the activation function of the intermediate layer, can be a string (“relu” or “gelu”) or a unary callable. Default: relu
- **layer_norm_eps** (*float*) – the eps value in layer normalization components (default=1e-5).
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False` (seq, batch, feature).
- **norm_first** (*bool*) – If `True`, layer norm is done prior to attention and feedforward operations, respectively. Otherwise it’s done after. Default: `False` (after).

The encoder layer are shown as follows. It contains a total of `n_head` heads. Each head consists of two modules:

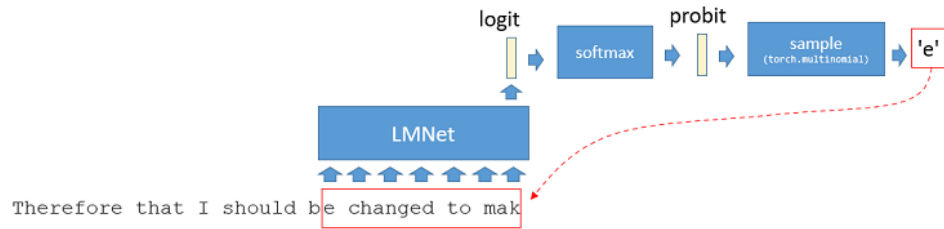
1. **Self-attention module.** Different from the origin transformer, PyTorch implementation replaces the *multi-head self-attention* layer with the **masked** version as shown below. This allows it the encoder be used for generative language modeling.
2. **Feed forward module.** This modules consists of two linear layer. The output feature length of the 1st linear layer is set to `dim_feedforward` while the 2nd linear layer is set to `d_model`.



Generating novel text

To generate novel text, we implement the method `generate`. Since the network is trained on a sequence of length $T = \text{block_size}$, 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



▼ Implementing the Language Model

```

1 class LMNet(nn.Module):
2     def __init__(self, vocab_size, d_model=512, nhead=8, dim_feedforward=2048, num_layers=6, batch_first=True, device="cuda"):
3         super().__init__()
4
5         # token embedding
6         self.token_embedding = nn.Embedding(vocab_size, d_model)
7
8         # positional embedding layer
9         self.pos_embedding = PositionEmbedding(d_model=d_model, batch_first=batch_first)
10
11        # create the encoder block
12        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead, dim_feedforward=dim_feedforward, batch_first=batch_first)
13        self.encoder = nn.TransformerEncoder(encoder_layer=encoder_layer, num_layers=num_layers)
14
15        # create the linear layer
16        self.fc = nn.Linear(in_features=d_model, out_features=vocab_size)
17
18        # transfer to targeted device
19        self = self.to(device)
20
21    def forward(self, x, mask = None):
22
23        # convert each token into its embedding vector
24        x = self.token_embedding(x)
25
26        # add position embedding to x
27        x = self.pos_embedding(x)
28
29        # perform sequence model inference
30        if mask is not None:
31            x = self.encoder(x, mask)
32        else:
33            x = self.encoder(x)
34
35        # generate logits
36        x = self.fc(x)
37
38        return x
39
40    def generate(self, text_len, block_size):
41

```

```

42     # set to evaluation mode
43     model.eval()
44
45     text = torch.zeros((1,1), dtype=torch.long).to(device) # text token (B, T) where B is fixed to 1, T = 1 initially
46
47     # repeat until the length of text = "text_len"
48     for _ in range(text_len): # (B, T)
49
50         # crop text to the last block-size tokens
51         text_cond = text[:, -block_size:] # (B, T)
52
53         # get the predictions
54         seq_len = text_cond.shape[-1]
55         mask = create_mask(seq_len, device)
56
57         # disable gradient computation
58         with torch.no_grad():
59
60             # predict next token
61             yhat = self(text_cond, mask) # logits: (B, T, C)
62
63             # focus oly on the last time step
64             yhat = yhat[:, -1, :] # becomes (B, C)
65
66             # apply soft max to get probabilities
67             probs = F.softmax(yhat, dim=-1) # (B, C)
68
69             # sample from distribution
70             next_token = torch.multinomial(probs, num_samples=1) # (B, 1)
71
72             # append sampled index to the running sequence
73             text = torch.cat((text, next_token), dim=1) # (B, T+1)
74
75         # print the sample
76         print(itos[next_token.item()], end='')
77         time.sleep(0.1)
78

```

```

1 d_model=512          # 256
2 nhead=8              # 4
3 dim_feedforward=2048 # 512
4 num_layers=6
5 vocab_size = len(vocab)
6 batch_first = True

1 batch_size=4
2 block_size=8

```

Test the model without any masking

```

1 model = LMNet(vocab_size, d_model, nhead, dim_feedforward, num_layers, batch_first)

```

```

-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-19-a82e82469110> in <cell line: 1>()
----> 1 model = LMNet(vocab_size, d_model, nhead, dim_feedforward, num_layers,
batch_first)

----- 5 frames -----
/usr/local/lib/python3.10/dist-packages/torch/cuda/__init__.py in _lazy_init()
    245     if 'CUDA_MODULE_LOADING' not in os.environ:
    246         os.environ['CUDA_MODULE_LOADING'] = 'LAZY'
--> 247     torch._C._cuda_init()
    248     # Some of the queued calls may reentrantly call _lazy_init();
    249     # we need to just return without initializing in that case.

RuntimeError: Found no NVIDIA driver on your system. Please check that you have an NVIDIA
1 x, y = get_batch(batch_size, block_size, device)
2 x, y = x.to(device), y.to(device)
3
4 yhat = model(x)
5
6 print('Shape of x: ', x.shape)
7 print('Shape of yhat: ', yhat.shape)

```

▼ Masking the input to generate auto-regressive output.

Our model is auto-regressive, i.e., it should only use past and current tokens $\{1, \dots, t\}$ and ignore future tokens $\{t+1, \dots, T_x\}$ when predicting current token. Since our input is a token sequence, we need to enforce the output at all timestep follow this requirement even though we are only interested in the last output token. To do this, we can use a *mask* M to disable future tokens:

$$\text{MA}(Q, K, V) = \text{softmax}\left(\frac{Q^T K + M}{\sqrt{d_k}}\right)V$$

The mask M is an upper triangular matrix of size $T_x \times T_x$ where the items at the upper left of M has values of $-\infty$.

For example, when the sequence length $T_x = 5$, the value of M is given by:

$$\begin{matrix} t = 1 \\ t = 2 \\ t = 3 \\ t = 4 \\ t = 5 \end{matrix} \begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty & -\infty \\ 0 & 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each row $M[t, :]$ represents the mask used to generate $\hat{y}^{<t>}$, i.e., the output at timestep t where columns (time step) with a value of 0 will be considered while those with a value of $-\infty$ discarded. For example, for $t = 2$, only tokens from time step 1, 2 and 3 (their values are 0) will be considered. Tokens from timestep 4 and 5 will be discarded.

Create the function to create mask for input with sequence length T_x .

```

1 def create_mask(Tx, device):
2     # create a tensor of shape (Tx, Tx) with value -inf (use `torch.full`)
3     mask = torch.full((5, 5), -torch.inf)

```

```

4
5     # set the diagonal and lower left part of mask to 0 (use `torch.triu` with `diagonal` set to 1 )
6     mask = mask.triu(diagonal=1)
7
8     # transfer to device
9     mask = mask.to(device)
10    return mask

```

```

1 mask = create_mask(block_size, device)
2 print(mask)

```

Modify the `forward` function of `LMNet` to pass the generated `mask` to the encoder object.

```

1 yhat = model(x, mask)
2 print('Shape of yhat:', yhat.shape)

```

▼ Train the model

```

1 max_iters    = 5000
2 batch_size   = 128
3 block_size   = 256
4 lr           = 5e-4
5 show_interval = 200

```

Create the model

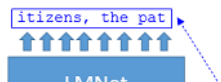
```
1 model = LMNet(vocab_size, d_model, nhead, dim_feedforward, num_layers).to(device)
```

Create the optimizer

```
1 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.

```

1 # create the mask for the batch data
2 mask = create_mask(block_size, device)
3
4 # set to training mode
5 model.train()
6
7 # train until convergence
8 for steps in range(max_iters):
9
10     # sample a batch of data
11     x, y = get_batch(batch_size, block_size, device)
12
13     # forward propagation
14     yhat = model(x, mask)
15
16     # compute loss
17     B, T, C = yhat.shape

```