Lab 10 - Language Modeling with Transformers (Guide)

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

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

```
Mounted at /content/drive
```

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

```
/content/drive/MyDrive/UCCD3074_Labs/UCCD3074_Lab10
```

```
import os
import math
import torch
import torch.nn as nn
from torch.nn import functional as F

import time
```

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

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

Preprocessing

Read the dataset into the string `raw_data`.

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

Create the vocabulary

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

```
vocab_size: 65
```

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

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

```
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`.

```
def get_batch(batch_size, block_size, device):
    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
```

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)`

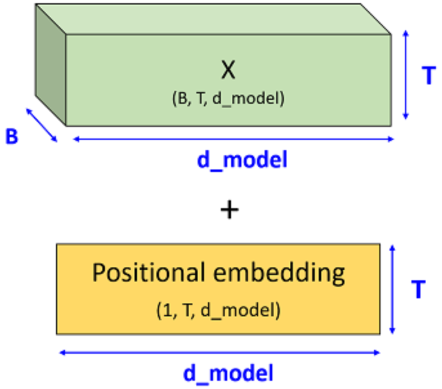
In the following, we create a new module class to implement `Position Embedding` in Section A. Then, we implment a language model with an encoder-only Transformer (`TransformerEncoder`) in Section B.

A. Implement the Positional Embedding module

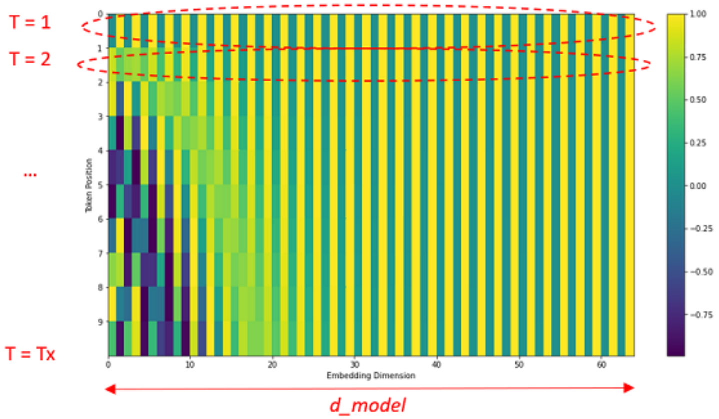
The following `PositionalEncoding` module injects some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed. To do so, we can use sine and cosine functions of different frequencies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

```
class PositionEmbedding(nn.Module):

    def __init__(self, d_model: int, batch_first: bool = False, max_len: int = 5000, dropout: float = 0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        self.batch_first = batch_first

        # Shape of position: (max_len, 1)
        position = torch.arange(max_len).unsqueeze(1)

        # Shape of div_term: 1d tensor of shape (d_model/2, )
        div_term1 = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        div_term2 = div_term1[:-1] if d_model % 2 == 1 else div_term1

        # shape of pe:  (max_len, 1, d_model)
```

```
        if self.batch_first:
            pe = torch.zeros(1, max_len, d_model)
            pe[0, :, 0::2] = torch.sin(position * div_term1)
            pe[0, :, 1::2] = torch.cos(position * div_term2)
            self.register_buffer('pe', pe)
        else:
            pe = torch.zeros(max_len, 1, d_model)
            pe[:, 0, 0::2] = torch.sin(position * div_term1)
            pe[:, 0, 1::2] = torch.cos(position * div_term2)
            self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        # Shape of self.pe: (T, 1, embedding_dim)
        # shape of x      : (T, B, embedding_dim)
        if self.batch_first:
            x = x + self.pe[:, :x.size(1), :]
        else:
            x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

Test the position embedding layer.

```
batch_size = 4
seq_len    = 8
d_model    = 512

x = torch.randn(batch_size, seq_len, d_model)
print('Shape of x:', x.shape)
```

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

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

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

```
pos_embedding.pe.shape
```

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

B. Implement the Language Model

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

1. `nn.Transformer` - a tranformer model. The module is constructed from `nn.TranformerEncoder` 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]

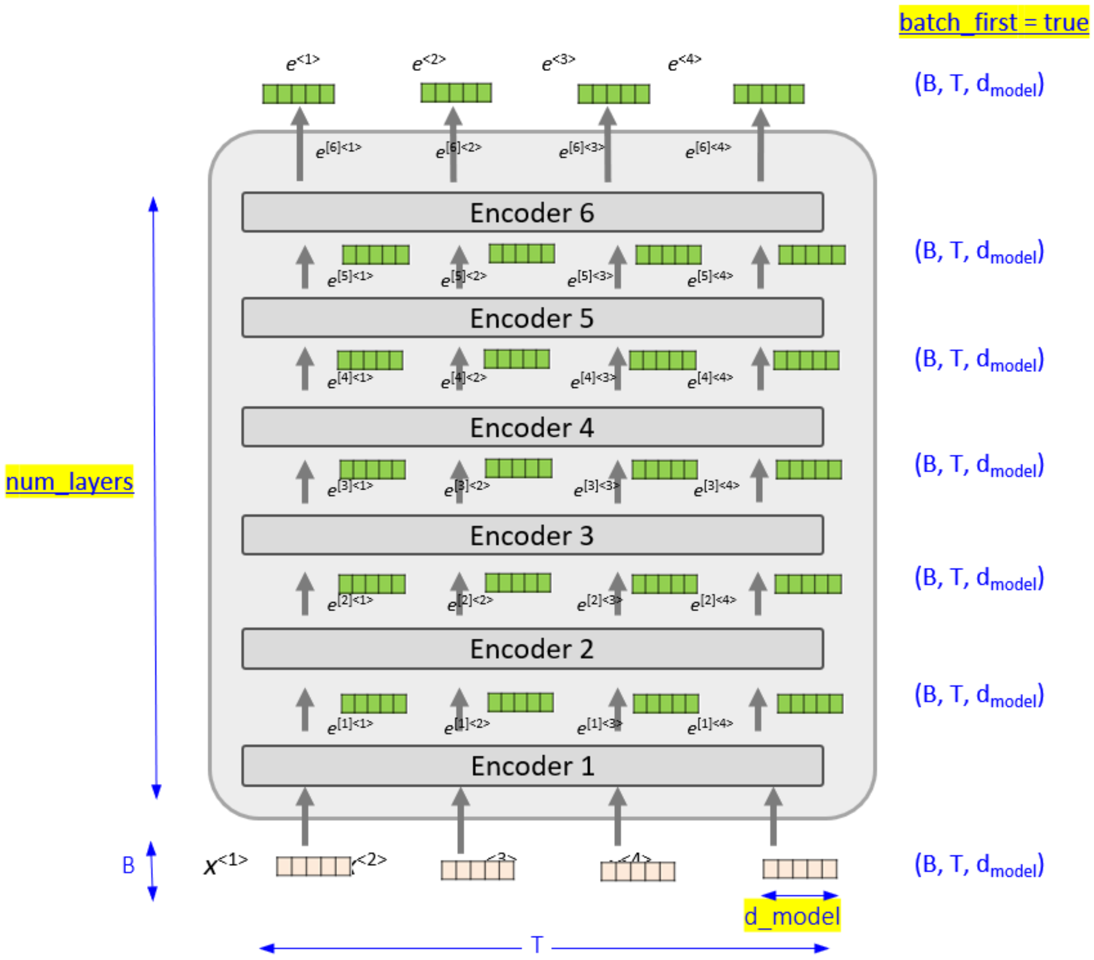
TransformerEncoder is a stack of N encoder layers. Users can build the BERT(https://arxiv.org/abs/1810.04805) model with corresponding parameters.

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.



The nn.TransformerEncoderLayer module

The `nn.TransformerEncoderLayer` implements an *encoder layer*.

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]

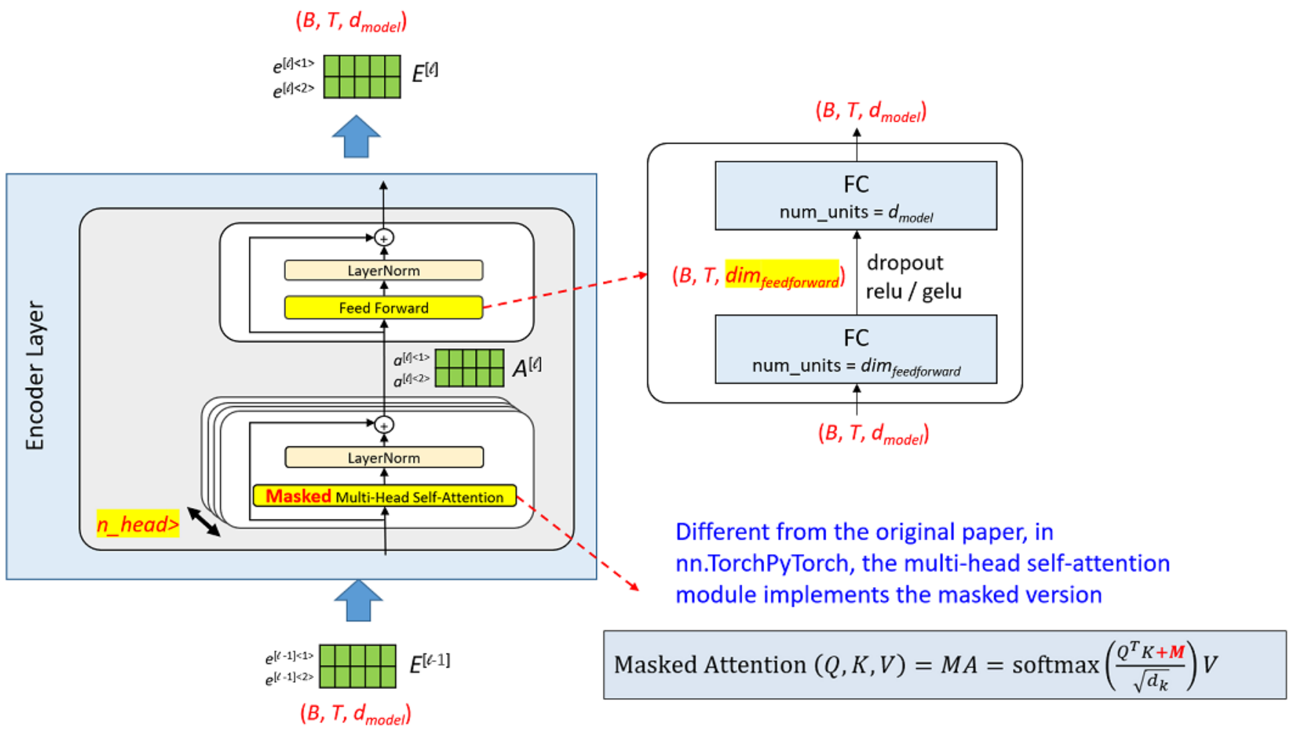
TransformerEncoderLayer is made up of self-attn and feedforward network. This standard encoder layer is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you 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:

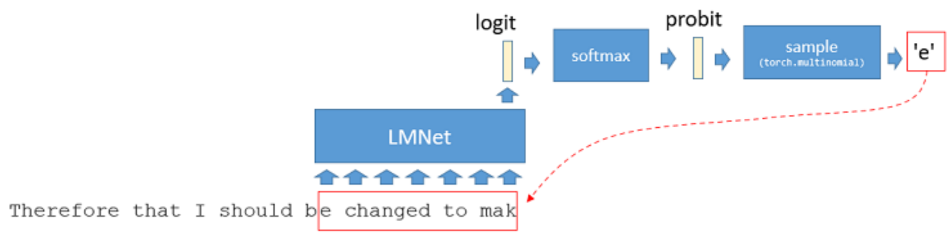
1. **Self-attention module**. Different from the origin transformer, PyTorch implementation replaces the *multi-head self-attention* layer with the **masked** version as shown below. This allows it the encoder be used for generative language modeling.
2. **Feed forward module**. This modules consists 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 = 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`](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

```
class LMNet(nn.Module):
    def __init__(self, vocab_size, d_model=512, nhead=8, dim_feedforward=2048, num_layers=6, batch_first=True, device="cuda"):
        super().__init__()

        # token embedding
        self.token_embedding = nn.Embedding(vocab_size, d_model)

        # positional embedding layer
        self.pos_embedding = PositionEmbedding(d_model=d_model, batch_first=batch_first)

        # create the encoder block
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead, dim_feedforward=dim_feedforward, batch_first=batch_first)
        self.encoder = nn.TransformerEncoder(encoder_layer=encoder_layer, num_layers=num_layers)

        # create the linear layer
        self.fc      = nn.Linear(in_features=d_model, out_features=vocab_size)

        # transfer to targeted device
        self = self.to(device)

    def forward(self, x, mask = None):

        # convert each token into its embedding vetor
        x = self.token_embedding(x)

        # add position embedding to x
        x = self.pos_embedding(x)

        # perform sequence model inference
        if mask is not None:
            x = self.encoder(x, mask)
        else:
            x = self.encoder(x)

        # generate logits
        x = self.fc(x)

        return x

    def generate(self, text_len, block_size):

```

```
        # set to evaluation mode
        model.eval()

        text = torch.zeros((1,1), dtype=torch.long).to(device)  # text token (B, T) where B is fixed to 1, T = 1 initially

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

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

            # get the predictions
            seq_len = text_cond.shape[-1]
            mask = create_mask(seq_len, device)

            # disable gradient computation
            with torch.no_grad():

                # predict next token
                yhat  = self(text_cond, mask)  # 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)

```

Settings to train the model

```
d_model=256
nhead=8
dim_feedforward=1024
num_layers=6

vocab_size = len(vocab)
batch_first = True
```

Test the model without any masking

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

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

```

yhat = model(x)

print('Shape of x:    ', x.shape)
print('Shape of yhat: ', yhat.shape)
```

```
Shape of x:     torch.Size([4, 8])
Shape of yhat:  torch.Size([4, 8, 65])
```

Masking the input to generate auto-regressive output.

Our model is auto-regressive, i.e., it should only use past and current tokens $\{1, \ldots, t\}$and ignore future tokens $\{t+1, \ldots, T_x\}$ 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:

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

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:

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

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.

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

    # set the diagonal and lower left part of mask to 0 (use `torch.triu` with `diagonal` set to 1 )
    mask = mask.triu(diagonal=1)

    # transfer to device
    mask = mask.to(device)
    return mask
```

```
mask = create_mask(Tx=5, device=device)
print(mask)
```

```
tensor([[0., -inf, -inf, -inf, -inf],
        [0., 0., -inf, -inf, -inf],
        [0., 0., 0., -inf, -inf],
```

```
        [0., 0., 0., 0., -inf],
        [0., 0., 0., 0., 0.]], device='cuda:0')
```

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

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

```
Shape of yhat: torch.Size([4, 8, 65])
```

Train the model

Settings for training

```
max_iters     = 5000
batch_size    = 128
block_size    = 250
lr            = 5e-4
show_interval = 200
```

Create the model

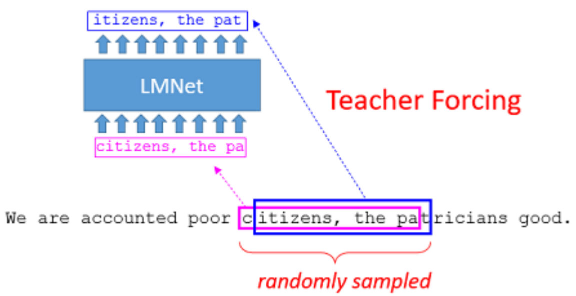
```
model = LMNet(vocab_size, d_model, nhead, dim_feedforward, num_layers).to(device)
```

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.

```
# create the mask for the btach data
mask = create_mask(block_size, device)

# set to traiing mode
model.train()

# train until convergence
for steps in range(max_iters):

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

    # forward propagation
    yhat = model(x, mask)

    # 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
    optimizer.zero_grad()

    # print the training loss
    if steps % show_interval == 0 or (steps+1)%max_iters == 0:
        print(f"Iter {steps}: train loss {loss:.4f}")

torch.save(model.state_dict(), 'model.pth')
```

```
Iter 0: train loss 4.4800
Iter 200: train loss 2.2296
Iter 400: train loss 1.8350
Iter 600: train loss 1.6568
Iter 800: train loss 1.5745
Iter 1000: train loss 1.4966
Iter 1200: train loss 1.4358
Iter 1400: train loss 1.4206
Iter 1600: train loss 1.3741
Iter 1800: train loss 1.3184
Iter 2000: train loss 1.3292
Iter 2200: train loss 1.3059
Iter 2400: train loss 1.2867
Iter 2600: train loss 1.2640
Iter 2800: train loss 1.2709
Iter 3000: train loss 1.2262
Iter 3200: train loss 1.2376
Iter 3400: train loss 1.2040
Iter 3600: train loss 1.1868
Iter 3800: train loss 1.1770
```

Generate text

Load the saved model

```
new_model = LMNet(vocab_size, d_model, nhead, dim_feedforward, num_layers, batch_first)
new_model.load_state_dict(torch.load('model.pth'))
```

```
<All keys matched successfully>
```

Generate a shakespeare-like text