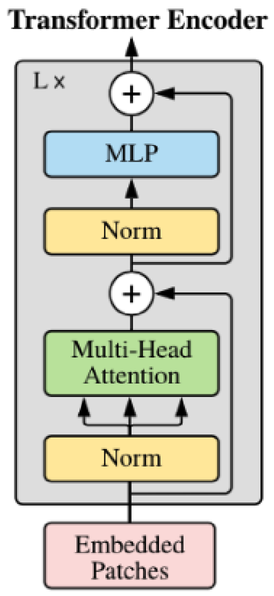
P1: Implement Encoder Block in Transformer (40%)



Transformer is the most popular architecture in many fields (computer vision, speech recognition, natural language processing, etc). The image above shows a typical transformer encoder.

In this problem, you will implement the same transformer encoder as shown in the image. Particularly you will implement multi-head attention and use provided functions to finish the missing parts in the encoder block.

Some things to note:

- Use LayerNorm for the Norm as shown in the figure above.
- FeedFoward is MLP in the figure.
- The gray rectangle illustrates the structure of EncoderBlock, and you only need to implement that.

```
import torch
import torch.nn as nn
from torch.nn import functional as F
import numpy as np

# hyperparameters
batch_size = 1
block_size = 32 # Maximum context length
device = 'cuda' if torch.cuda.is_available() else 'cpu'
n_embd = 2 # emgedding dimension
n_head = 2 # number of heads in multi-head attention
# ------------
torch.manual_seed(568)
```

```
<torch._C.Generator at 0x7c3bc43f9410>
```

```python
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size)
        self.query = nn.Linear(n_embd, head_size)
        self.value = nn.Linear(n_embd, head_size)

    def forward(self, x):
        # TODO: implement the single-head attention
        selfAttention = torch.bmm(self.query(x), self.key(x).transpose(-2, -1)) / (x.shape[2] ** 0.5)
        selfAttention = F.softmax(selfAttention, dim=-1)
        out = torch.bmm(selfAttention, self.value(x))
        return out

class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])

    def forward(self, x):
        # TODO: implement the multi-head attention
        out = torch.cat([head(x) for head in self.heads], dim=-1)
        return out

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
        )

    def forward(self, x):
        return self.net(x)

class EncoderBlock(nn.Module):
    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads
        super().__init__()
        self.sa = MultiHeadAttention(n_head, n_embd // n_head)# TODO: initialize multi-head self attention
        self.mlp = FeedFoward(n_embd) # TODO: create feed forward network
        self.norm1 = nn.LayerNorm(n_head) # TODO: create layer norm
        self.norm2 = nn.LayerNorm(n_embd) # TODO: create layer norm


    # initialize the weights with an identity matrix to make sure we get the same weight everytime
    # it is used just for reproducing the results
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.eye_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
            elif isinstance(m, nn.LayerNorm):
                nn.init.ones_(m.weight)
                nn.init.zeros_(m.bias)

    def forward(self, x):
        # TODO: reference the provided figure of transformer encoder to implement the forward function
        out = x + self.sa(self.norm1(x))
        out = out + self.mlp(self.norm2(out))
        return out
```

Initialize the encoder block.

```
encoder_block = EncoderBlock(n_embd, n_head).to(device)
encoder_block._initialize_weights()
```

Now, download data_p1.pt from Canvas and upload to your working directory.

```
data_dict_loaded = torch.load('data_p1.pt')
ans = data_dict_loaded['ans']
data = data_dict_loaded['data']
```

Forward pass the data to the encoder block and collect the reuslts into a list.

```
results = []
for datum in data:
    results.append(encoder_block(datum.to(device)).detach().cpu())
```

```
ans
```

```
              [-1.7702,  2.3148],
              [-0.9737,  1.1272],
              [ 1.6078,  0.1909],
              [ 1.6731, -0.3200],
              [ 3.1005,  0.5838],
              [-1.2745,  1.0775],
              [ 2.7161, -0.6739],
              [ 2.8139,  0.6781],
              [ 0.6114, -0.4760],
              [-1.7369,  0.6209],
              [-1.5547,  1.4092],
              [ 0.6270, -0.4385],
              [ 3.5181, -0.6285],
              [-2.1928,  1.6743],
              [ 3.6704, -0.4864],
              [-1.0429,  1.2313],
              [-1.6441,  0.2750],
              [ 1.7162,  0.3108],
              [-1.1280,  1.5796],
              [ 0.2795,  1.3356],
              [-0.0457,  1.4466],
              [ 3.4281,  0.3123],
              [ 2.9692,  0.7618],
              [ 3.3695, -1.2879],
              [ 3.2384,  0.6489],
              [ 3.3479, -0.6007],
              [ 0.5302,  2.4495]]]),
      tensor([[[-1.0259,  0.6152],
              [-2.2288, -0.3617],
              [-0.9368,  0.4891],
              [ 1.9080, -0.0684],
              [-1.5225,  0.8566],
              [-0.9295,  1.3268],
              [-1.3441,  0.9111],
              [-1.5945, -0.4248],
              [ 1.1780, -3.0424],
              [-0.5267,  0.8443],
              [-0.7441,  0.6056],
              [ 1.6322, -0.4910],
              [-1.2116,  0.3364],
              [ 1.9748, -0.4159],
              [ 1.2068, -0.3376],
              [-1.7814,  0.6242],
              [-2.9207,  0.3983],
              [ 2.1858, -0.2490],
              [ 2.4302,  0.8357],
              [-1.2421,  0.2006],
              [-1.1448, -0.9206],
              [ 2.0967,  0.0365],
              [ 2.2502,  1.0912],
              [ 1.8053, -0.3889],
              [-0.1921,  1.3671],
```

```
1 results
```

```
               [-0.3857,  2.3982],
               [ 1.6878,  0.1076],
               [ 1.3279, -1.0041],
               [-0.8110,  0.3060]]]),
     tensor([[[ 2.6773,  1.5188],
               [-0.2199,  0.9685],
               [-1.7405,  1.7295],
               [-1.3190,  0.6758],
               [ 2.2541, -1.1568],
               [ 2.7457,  1.0191],
               [ 1.6217,  0.1682],
               [ 3.2371, -0.6552],
               [-1.3366,  1.7308],
               [ 3.1672,  0.7356],
               [-2.2987, -0.1141],
               [-1.1696,  1.5834],
               [-1.5938,  0.0408],
               [ 2.4058, -0.0688],
               [ 2.3611,  0.3585],
               [-1.9571, -0.3031],
               [-1.9340,  0.5315],
               [-1.7469,  0.6587],
               [ 1.7582, -1.0192],
               [-1.7132,  0.6404],
               [-2.1351,  0.6079],
               [ 0.2586,  3.3612],
               [ 2.8285,  1.5993],
               [-1.0132,  0.0518],
               [-1.5383,  0.3088],
               [-1.5515,  1.9327],
               [ 1.5559, -0.1682],
               [-2.2796,  0.9164],
               [ 1.5258, -0.2152],
               [ 1.8087, -0.9843],
               [-1.5651,  0.9921],
               [-1.7276,  0.3684]]]),
     tensor([[[ 1.2099,  0.1582],
               [ 3.2397,  0.0430],
               [-2.2824,  1.4001],
               [ 2.7031, -0.1075],
               [ 2.8890,  0.7328],
               [-1.0576,  1.4579],
               [-2.2220,  0.6479],
               [ 2.2413,  1.1112],
               [-0.5619,  0.4465],
               [-1.3867,  1.8416],
               [-0.1727,  0.9043],
               [ 0.2314,  1.3074],
               [ 1.7920,  0.4774],
               [ 0.5770, -1.2372],
               [-0.3655,  0.7035],
               [ 0.3585, -1.1778],
               [-2.3048,  1.1799],
               [-0.2198,  0.9215],
               [ 2.1537, -1.7752],
               [ 1.5642,  0.0756],
               [ 0.9455, -1.2763],
               [-2.2637,  0.0240],
```

Make sure your model produce the same result for every test case.

```
## Note, I changed the relative tolerance due to the fact that the values only ever go upto the 0.0001 place anyway.
## going further gives not use.

for i in range(len(data)):
    assert torch.allclose(results[i], ans[i], rtol=0.0001), f"Your implementation is wrong for the {i}-th example."
```

```
print('Pass')
```

```
Pass
```

Save your model to a file named **enc_block.pth** and include **enc_block.pth** in your submission. We will test your encoder block with different test cases for you to get full credits.

```
PATH = './enc_block.pth'
torch.save(encoder_block.state_dict(), PATH)
```

In the decoder of transformer models, masking in self-attention is crucial for two primary reasons:

- Preventing Future Information Leakage: In tasks like language translation or text generation, the future tokens (words that come later in the sequence) should not influence the prediction of the current token. Masking ensures that while predicting a particular word, the model only has access to previously generated words and not to any future words. This is essential for the model to learn a proper language structure and generate coherent and contextually appropriate text.
- Maintaining Autoregressive Property: Transformers, especially in tasks like text generation, operate in an autoregressive manner, meaning they generate one part of the sequence at a time. By masking future tokens in the self-attention mechanism, the model respects this autoregressive nature, ensuring that each step of generation or translation is based solely on already known or generated information.

Please see https://arxiv.org/abs/1706.03762 for more details on Transformer.

Question: What should we modify if we want to implement masked attention head in transformer's decoder block? Please implement the masked attention by modifying your previous class Head.

```
class MaskedHead(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size)
        self.query = nn.Linear(n_embd, head_size)
        self.value = nn.Linear(n_embd, head_size)
        self.register_buffer('mask', torch.tril(torch.ones(block_size, block_size)))

    def forward(self, x):
        # TODO: implement the single-head attention
        _, maskShape, size = x.shape
        selfAttention = torch.bmm(self.query(x), self.key(x).transpose(-2, -1)) / (size ** 0.5)
        selfAttention = selfAttention.masked_fill(self.mask[:maskShape, :maskShape] == 0, float('-inf'))
        selfAttention = F.softmax(selfAttention, dim=-1)
        out = torch.bmm(selfAttention, self.value(x))
        return out
```

```
encoder_block = EncoderBlock(n_embd, n_head).to(device)
encoder_block._initialize_weights()

data_dict_loaded = torch.load('data_p1.pt')
ans = data_dict_loaded['ans']
data = data_dict_loaded['data']

results = []
for datum in data:
    results.append(encoder_block(datum.to(device)).detach().cpu())
```

```
results
```

```
                 [-0.2199,  0.9685],
                 [-1.7405,  1.7295],
                 [-1.3190,  0.6758],
                 [ 2.2541, -1.1568],
                 [ 2.7457,  1.0191],
                 [ 1.6217,  0.1682],
                 [ 3.2371, -0.6552],
                 [-1.3366,  1.7308],
                 [ 3.1672,  0.7356],
                 [-2.2987, -0.1141],
                 [-1.1696,  1.5834],
                 [-1.5938,  0.0408],
                 [ 2.4058, -0.0688],
                 [ 2.3611,  0.3585],
                 [-1.9571, -0.3031],
                 [-1.9340,  0.5315],
                 [-1.7469,  0.6587],
                 [ 1.7582, -1.0192],
                 [-1.7132,  0.6404],
                 [-2.1351,  0.6079],
                 [ 0.2586,  3.3612],
                 [ 2.8285,  1.5993],
                 [-1.0132,  0.0518],
                 [-1.5383,  0.3088],
                 [-1.5515,  1.9327],
                 [ 1.5559, -0.1682],
                 [-2.2796,  0.9164],
                 [ 1.5258, -0.2152],
                 [ 1.8087, -0.9843],
                 [-1.5651,  0.9921],
                 [-1.7276,  0.3684]]]),
        tensor([[[ 1.2099,  0.1582],
                 [ 3.2397,  0.0430],
                 [-2.2824,  1.4001],
                 [ 2.7031, -0.1075],
                 [ 2.8890,  0.7328],
                 [-1.0576,  1.4579],
                 [-2.2220,  0.6479],
                 [ 2.2413,  1.1112],
                 [-0.5619,  0.4465],
                 [-1.3867,  1.8416],
                 [-0.1727,  0.9043],
                 [ 0.2314,  1.3074],
                 [ 1.7920,  0.4774],
                 [ 0.5770, -1.2372],
                 [-0.3655,  0.7035],
                 [ 0.3585, -1.1778],
                 [-2.3048,  1.1799],
                 [-0.2198,  0.9215],
                 [ 2.1537, -1.7752],
                 [ 1.5642,  0.0756],
                 [ 0.9455, -1.2763],
                 [-2.2637,  0.0240],
```

```
1 ans
```

```
                [ 1.8053, -0.3889],
                [-0.1921,  1.3671],
                [ 2.1094,  0.7266]]]),
       tensor([[[ 0.8151, -0.9453],
                [ 2.5775, -1.7437],
                [-1.6758,  0.1372],
                [-1.5354,  0.0322],
                [-0.8102,  1.0794],
                [ 1.6140,  0.6365],
                [ 2.0891,  1.0063],
                [ 2.2872,  0.7041],
                [ 4.4790,  0.8228],
                [ 2.6819,  0.9185],
                [-0.6281,  1.5366],
                [ 0.0072,  1.1369],
                [-1.3940,  0.2573],
                [ 2.3398,  0.7757],
                [ 2.8084, -0.2453],
                [ 1.9982,  0.5587],
                [-1.8077,  1.1651],
                [ 3.3140,  1.9080],
                [-1.3583,  0.5102],
                [ 2.5500,  0.8867],
                [ 2.6635,  1.0557],
                [-1.8199,  1.7828],
                [ 3.2988,  1.5809],
                [-0.5473,  1.7100],
                [-1.6121,  1.0381],
                [-1.6970,  1.2286],
                [ 1.8611,  0.6345],
                [ 1.4477,  0.2314],
                [-0.3857,  2.3982],
                [ 1.6878,  0.1076],
                [ 1.3279, -1.0041],
                [-0.8110,  0.3060]]]),
       tensor([[[ 2.6773,  1.5188],
                [-0.2199,  0.9685],
```

```
1
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.