Twitter Thread by abhishek

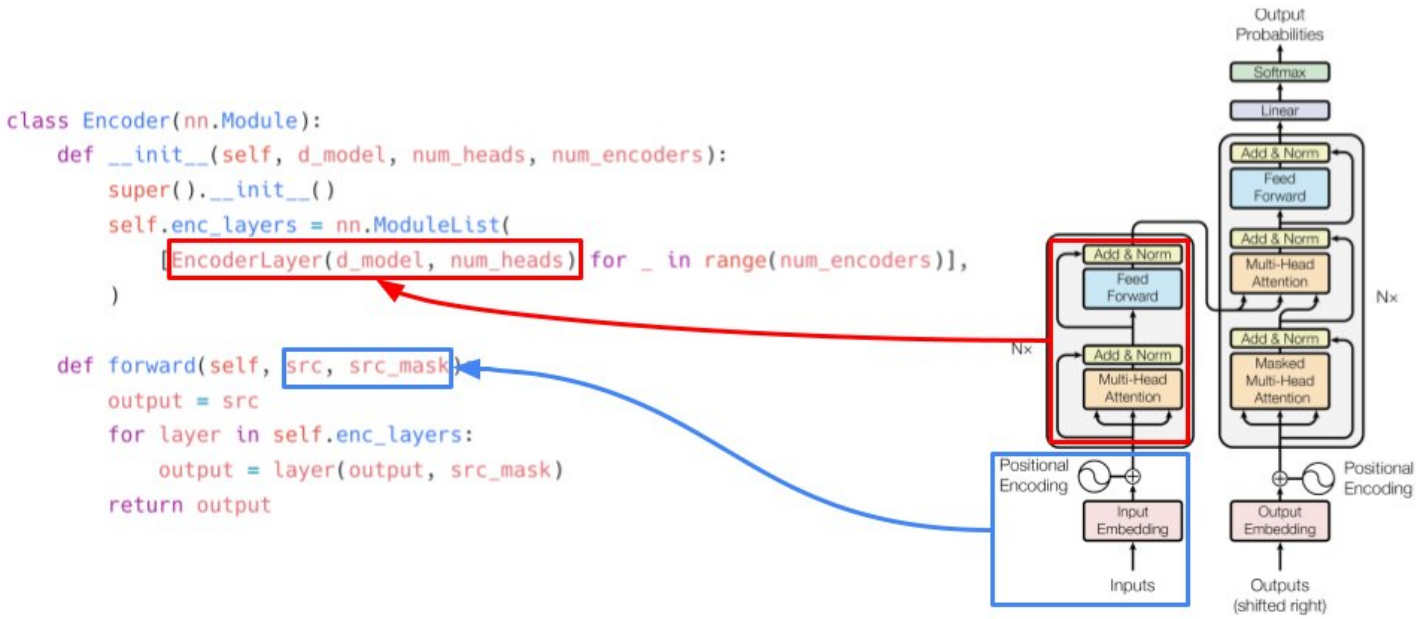
abhishek
@abhi1thakur

"Attention is all you need" implementation from scratch in PyTorch. A Twitter thread:

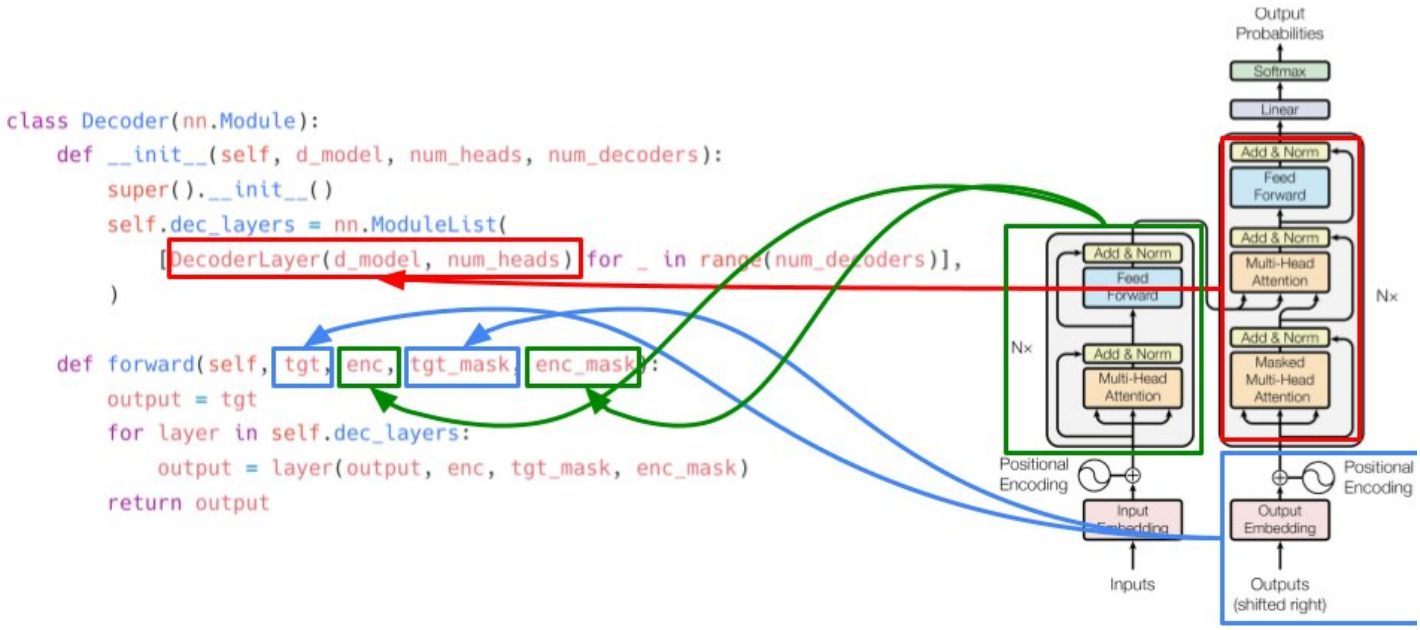
1/

There are two parts: encoder and decoder. Encoder takes source embeddings and source mask as inputs and decoder takes target embeddings and target mask. Decoder inputs are shifted right. What does shifted right mean? Keep reading the thread. 2/

The encoder is composed of N encoder layers. Let's implement this as a black box too. The output of one encoder goes as input to the next encoder and so on. The source mask remains the same till the end 3/

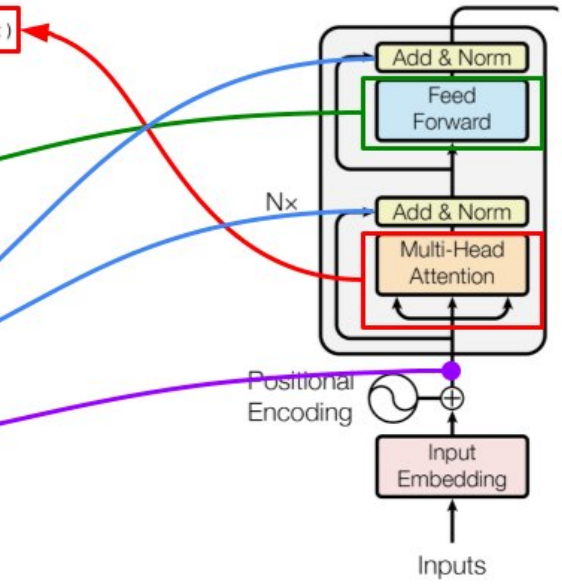


Similarly, we have the decoder composed of decoder layers. The decoder takes input from the last encoder layer and the target embeddings and target mask. enc_mask is the same as src_mask as explained previously 4/



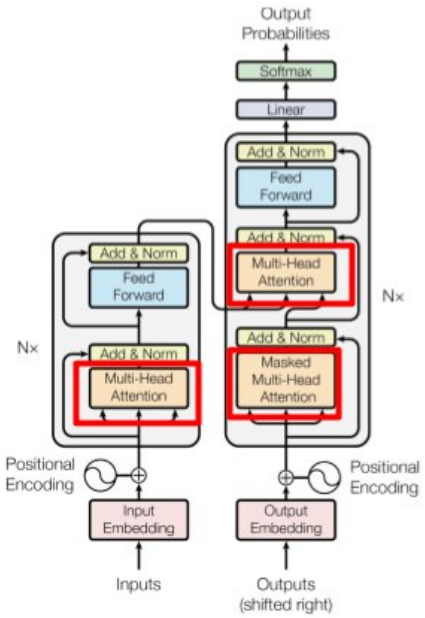
Let's take a look at the encoder layer. It consists of multi-headed attention, a feed forward network and two layer normalization layers. See forward(...) function to understand how skip-connection works. Its just adding original inputs to the outputs. 5/

```python
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff=2048, dropout=0.3):
        super().__init__()
        # attention
        self.attn = MultiHeadedAttention(d_model, num_heads, dropout=dropout)

        # ffn
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(inplace=True),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )

        # layer norm
        self.attn_norm = nn.LayerNorm(d_model)
        self.ffn_norm = nn.LayerNorm(d_model)

    def forward(self, src, src_mask):
        x = src
        x = x + self.attn(q=x, k=x, v=x, mask=src_mask)
        x = self.attn_norm(x)
        x = x + self.ffn(x)
        x = self.ffn_norm(x)
        return x
```



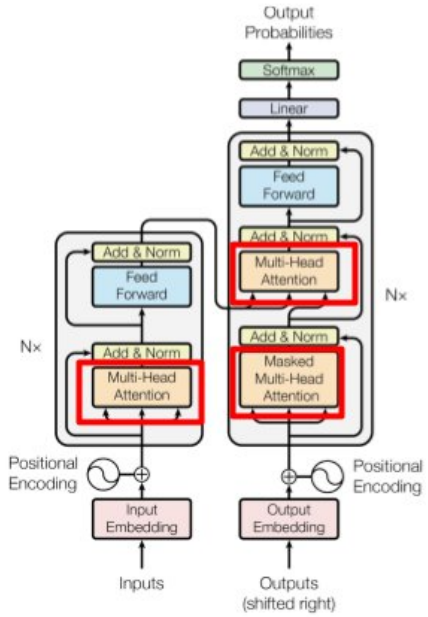
Now comes the fun part. Multi-head attention. We see it 3 times in the architecture. Multi-headed attention is nothing but many different self-attention layers. The outputs from these self-attentions are concatenated to form output the same shape as input. 6/

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout):
        super().__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.dropout = dropout
        self.attn_output_size = self.d_model // self.num_heads
        self.attentions = nn.ModuleList(
            [
              SelfAttention(d_model, self.attn_output_size)
              for _ in range(self.num_heads)
            ],
        )
        self.output = nn.Linear(self.d_model, self.d_model)

    def forward(self, q, k, v, mask):
        x = torch.cat(
          [
            layer(q, k, v, mask) for layer in self.attentions
          ], dim=-1
        )
        x = self.output(x)
        return x
```



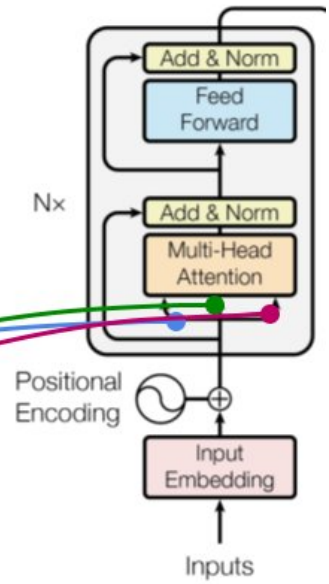
If the number of heads is 8 and d_model (embedding size) is 512, each self-attention will produce an output of size 64. These will be concatenated together to give the final output of size 64 x 8 = 512. This output is passed through a dense layer. 7/

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout):
        super().__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.dropout = dropout
        self.attn_output_size = self.d_model // self.num_heads
        self.attentions = nn.ModuleList(
            [
              SelfAttention(d_model, self.attn_output_size)
              for _ in range(self.num_heads)
            ],
        )
        self.output = nn.Linear(self.d_model, self.d_model)

    def forward(self, q, k, v, mask):
        x = torch.cat(
          [
            layer(q, k, v, mask) for layer in self.attentions
          ], dim=-1
        )
        x = self.output(x)
        return x
```



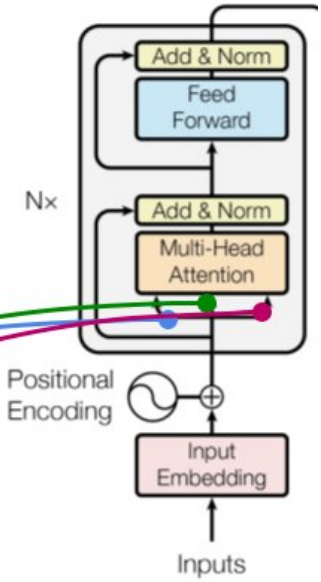
self-attention in simple words is attention on the same sequence. I like to define it as a layer that tells you which token loves another token in the same sequence. for self-attention, the input is passed through 3 linear layers: query, key, value. 8/

```python
class SelfAttention(nn.Module):
    def __init__(self, d_model, output_size, dropout=0.3):
        super().__init__()
        self.query = nn.Linear(d_model, output_size)
        self.key = nn.Linear(d_model, output_size)
        self.value = nn.Linear(d_model, output_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, q, k, v, mask=None):
        bs = q.shape[0]
        tgt_len = q.shape[1]
        seq_len = k.shape[1]
        query = self.query(q)
        key = self.key(k)
        value = self.value(v)

        dim_k = key.size(-1)
        scores = torch.bmm(query, key.transpose(1, 2)) / np.sqrt(dim_k)
```



In the forward function, we apply the formula for self-attention. softmax(Q.K´/ dim(k))V. torch.bmm does matrix multiplication of batches. dim(k) is the sqrt of k. Please note: q, k, v (inputs) are the same in the case of self-attention. 9/

```python
class SelfAttention(nn.Module):
    def __init__(self, d_model, output_size, dropout=0.3):
        super().__init__()
        self.query = nn.Linear(d_model, output_size)
        self.key = nn.Linear(d_model, output_size)
        self.value = nn.Linear(d_model, output_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, q, k, v, mask=None):
        bs = q.shape[0]
        tgt_len = q.shape[1]
        seq_len = k.shape[1]
        query = self.query(q)
        key = self.key(k)
        value = self.value(v)

        dim_k = key.size(-1)
        scores = torch.bmm(query, key.transpose(1, 2)) / np.sqrt(dim_k)
```



Let's look at the forward function and the formula for self-attention (scaled). Ignoring the mask part, everything is pretty easy to implement. 10/

```python
def forward(self, q, k, v, mask=None):
    bs = q.shape[0]
    tgt_len = q.shape[1]
    seq_len = k.shape[1]
    query = self.query(q)
    key = self.key(k)
    value = self.value(v)

    dim_k = key.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / np.sqrt(dim_k)

    if mask is not None:
        expanded_mask = mask[:, None, :].expand(bs, tgt_len, seq_len)
        scores = scores.masked_fill(expanded_mask == 0, -float("Inf"))

    weights = F.softmax(scores, dim=-1)
    outputs = torch.bmm(weights, value)
    return outputs
```

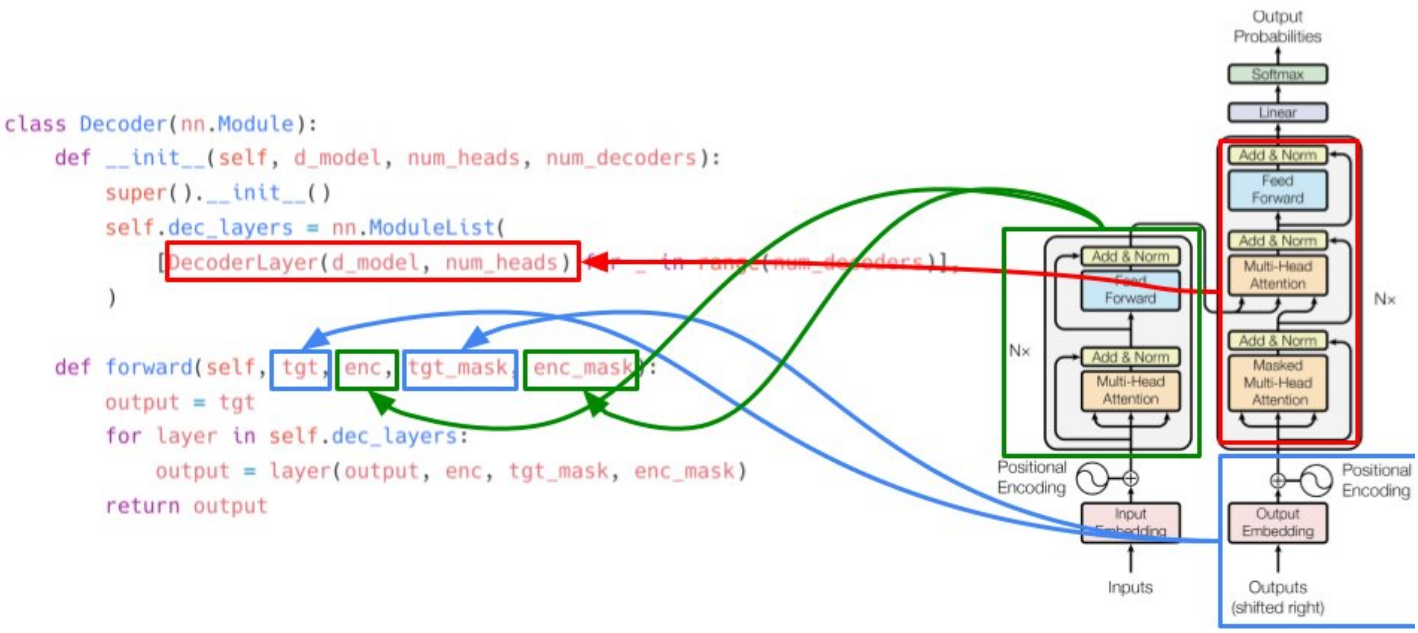
$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The mask just tells where not to look (e.g. padding tokens) 11/

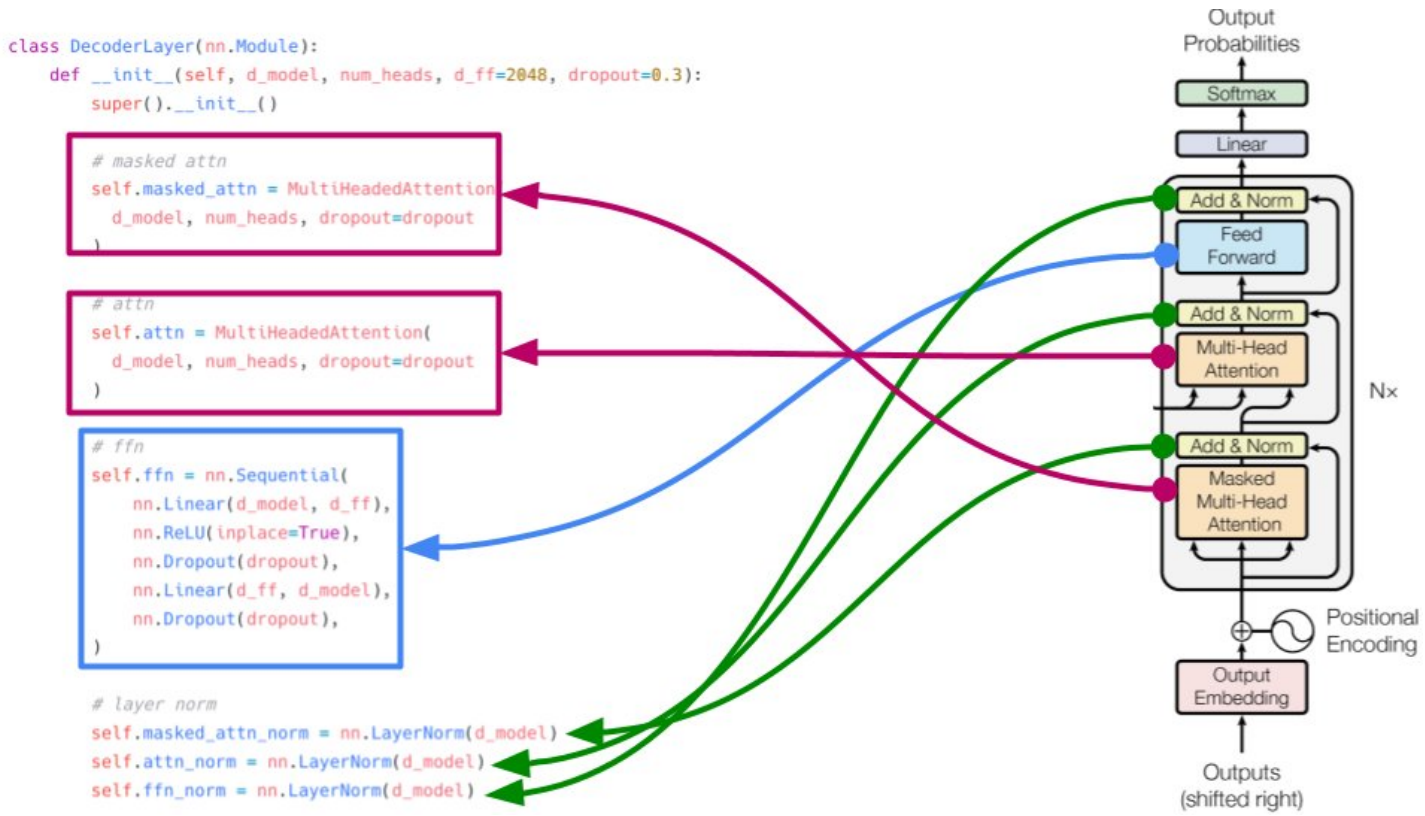
```
if mask is not None:
    expanded_mask = mask[:, None, :].expand(bs, tgt_len, seq_len)
    scores = scores.masked_fill(expanded_mask == 0, -float("Inf"))
```

- mask tells the model where not to look
- padding = 0, tokens = 1
- mask: batch size x sequence length
- expanded mask: batch size x sequence length x sequence length
- don’t use mask = 0 for scores

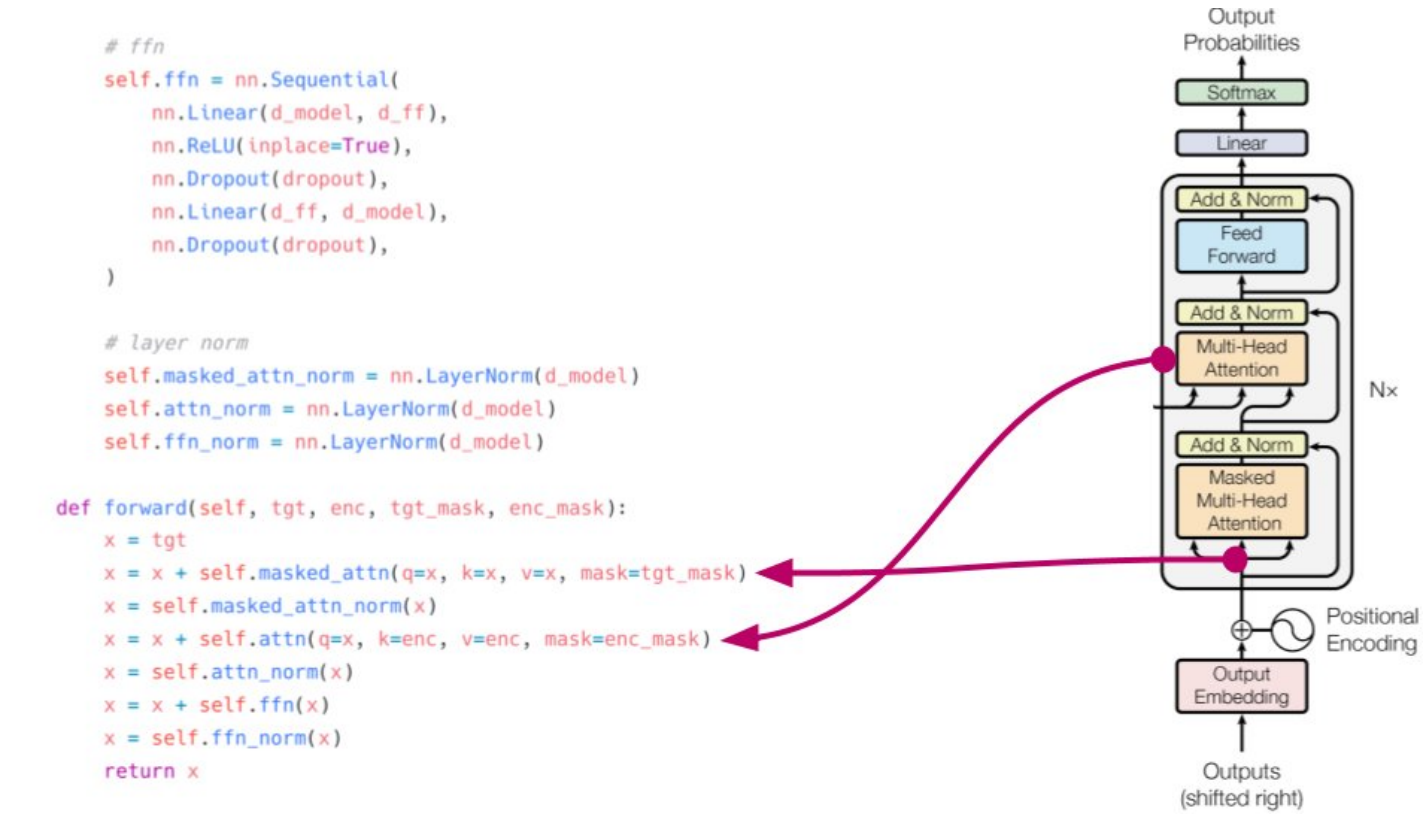
Let's take a look at decoder now. The implementation is similar to that of the encoder except for the fact that each decoder also takes the final encoder's output as input. 12/



The decoder layer consists of two different types of attention. the masked version has an extra mask in addition to padding mask. We will come to that. The normal multi-head attention takes key and value from final encoder output. key and value here are same. 13/



Query comes from output of masked multi-head attention (after layernorm). Checkout the forward function and things are very easy to understand :) 14/



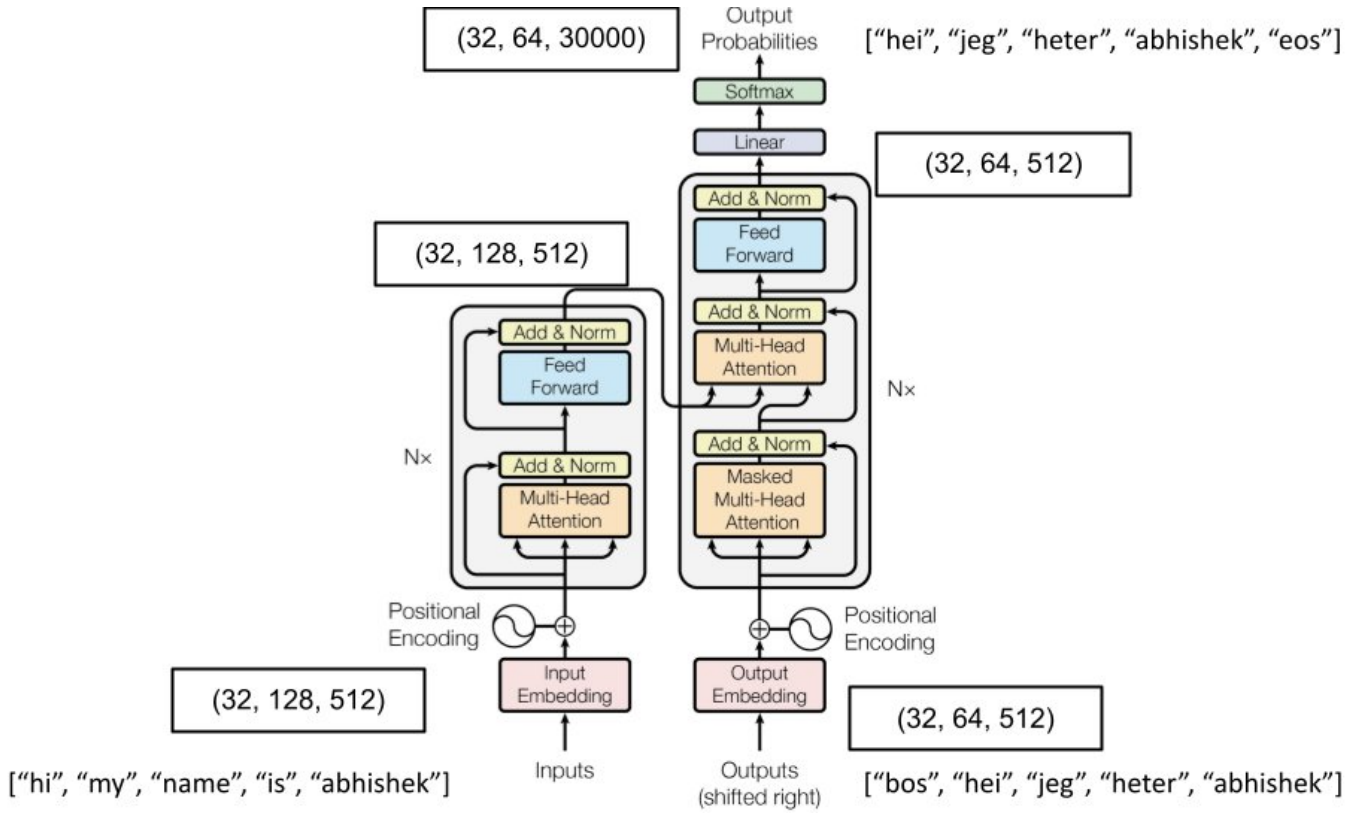
Now we come to the special mask for targets, aka subsequent mask. The subsequent mask just tells the decoder not to look at tokens in the future. This is used in addition to the padding mask and is used only for training part. 15/

```python
if mask is not None:
    expanded_mask = mask[:, None, :].expand(bs, tgt_len, seq_len)
    subsequent_mask = 1 - torch.triu(
        torch.ones((tgt_len, tgt_len), device=mask.device, dtype=torch.uint8), diagonal=1
    )
    subsequent_mask = subsequent_mask[None, :, :].expand(bs, tgt_len, tgt_len)
    scores = scores.masked_fill(expanded_mask == 0, -float("Inf"))
    scores = scores.masked_fill(subsequent_mask == 0, -float("Inf"))
```

	T-1	T-2	T-3	T-4	T-5	T-6	T-7	T-8
T-1	1	0	0	0	0	0	0	0
T-2	1	1	0	0	0	0	0	0
T-3	1	1	1	0	0	0	0	0
T-4	1	1	1	1	0	0	0	0
T-5	1	1	1	1	1	0	0	0
T-6	1	1	1	1	1	1	0	0
T-7	1	1	1	1	1	1	1	0
T-8	1	1	1	1	1	1	1	1

Now we have all the building blocks except positional encoding. Positional encoding tells the model an idea about where the tokens are located relative to each other. To implement positional encoding, we can simply use an embedding layer! 16/

And this is how inputs and outputs will look like. Here, batch size = 32, len of input seq = 128, len of output seq = 64. We add a linear + softmax to decoder output. This gives us a token prediction for each position (a classification problem) 17/



I hope you liked this thread. If there are any mistakes in my implementation, please let me know and I can fix them :) 18/