Yet another explanation and implementation of the transformer model

This note contains some details of the transformer model that I found a little bit confusing when I first tried to implement it from scratch. It is not intended to be a complete explanation of the transformer model, since there is already a ton of useful material that one can find online. For some examples of implementing the transformer model for various tasks, see my GitHub repository: hbchen-one/Transformer-Models-from-Scratch.

1. Notation

- d_B : batch size
- d_S : sequence length
- d_E : embedding dimension
- h : numbers of attention head
- $d_k = d_E/h$: dimension of each attention head

For n-dimensional tensor X, I will denote its dimensions as follows

$$X : (d_1, d_2, \ldots . d_n), \tag{1}$$

where d_i is the size of the i-th dimension of X.

2 Encoder

In this section, I try to describe how the encoder part of the transformer model works, especially the multi-head self-attention layer, with some code for implementing it in PyTorch.

2.1 Padding and embedding

Let us have in mind the machine translation problem, where the training data will be sentence pairs in two different languages. We usually separate the training set into batches with a batch size of d_B. For the input to the encoder part of the transformer model, each batch will contain d_B source language sentences (assuming that they are tokenized). The lengths of the sentences in each batch may be different. So we find the length of the longest sentence, denote it as d_S, and pad the sentences with lengths smaller than d_S with zeros (or other padding tokens), such that all the sentences in each batch have the same length d_S. After this, the input tensor X_{input} has dimensions

$$X_{\text{input}} : (d_B, d_S). \tag{2}$$

For each word in the sentences, we map it to a d_E-dimensional embedding vector, and we also use a d_E-dimensional position embedding to encode the positional information of the words in each sentence. After the embedding layer, our embedded input $X^{\text{E}}_{\text{input}}$ will have dimension

$$X^{\text{E}}_{\text{input}} : (d_B, d_S, d_E). \tag{3}$$

2.2 Multi-head self-attention

$X^{\text{E}}_{\text{input}}$ will be the input to the multi-heads self-attention layer. The formula for doing attention is usually written as follows

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{4}$$

Let's see in detail how is formula is implemented.

From $X^{\text{E}}_{\text{input}}$, we first construct the query, key, and value tensors, which we denote as Q, K and V respectively. They are simply obtained by multiplying $X^{\text{E}}_{\text{input}}$ with tensors $W_{\text{query}},\ W_{\text{key}}, W_{\text{value}}$, which have dimensions:

$$W_{\text{query}}, W_{\text{key}}, W_{\text{value}} : (d_E, d_E = h \times d_k). \tag{5}$$

Here we are stacking the W matrices for different heads together along the second dimension. So the query, key, and values tensors Q, K, V have dimensions

$$
\begin{aligned}
Q &= X^{\mathrm{E}}_{\mathrm{input}} W_{\mathrm{query}} : (d_B, d_S, d_E) \times (d_E, d_E) = (d_B, d_S, d_E) \to (d_B, h, d_S, d_k), \\
K &= X^{\mathrm{E}}_{\mathrm{input}} W_{\mathrm{key}} \ : (d_B, d_S, d_E) \times (d_E, d_E) = (d_B, d_S, d_E) \to (d_B, h, d_S, d_k), \\
V &= X^{\mathrm{E}}_{\mathrm{input}} W_{\mathrm{value}} : (d_B, d_S, d_E) \times (d_E, d_E) = (d_B, d_S, d_E) \to (d_B, h, d_S, d_k).
\end{aligned} \tag{6}
$$

The dimensions on the left of the arrows are what we get after the matrix multiplication, we then reorganize the tensors such that they have the dimensions shown after the arrows above. This is done in PyTorch as follows

```
Q = torch.matmul(XEinput, Wq).view(d_B, -1, h, d_k).transpose(1,2)
K = torch.matmul(XEinput, Wk).view(d_B, -1, h, d_k).transpose(1,2)
V = torch.matmul(XEinput, Wv).view(d_B, -1, h, d_k).transpose(1,2)
```

where I have tried to keep the variable names as close as possible to the Latex notations. In actual implementation, one would register the matrix multiplication above as a linear layer, since we will need to do gradient descent on the matrix elements of the W matrices.

The next step is to do self-attention. This means that we compute the dot product of the query tensor Q and the key tensor K. This is done as follows. We transpose the last two dimensions of K, and use torch.matmul, after which we get a tensor S of dimensions (d_B, h, d_S, d_S):

$$
S = \frac{QK^T}{\sqrt{d_k}} : (d_B, h, d_S, d_k) \times (d_B, h, d_k, d_S) = (d_B, h, d_S, d_S). \tag{7}
$$

Where I also divide the result of the matrix multiplication by $\sqrt{d_k}$. In PyTorch, this is simply implemented by

```
S = torch.matmul(Q, K.transpose(-2,-1))/math.sqrt(self.d_k)
```

Next, we compute the softmax of S. Here, we want to mask out the padding tokens in the sequences so that they will not participate in the softmax computation. We simply assign a very large negative value to the elements of S that correspond to the padding tokens. The is done as follows:

```
# PAD is the integer corresponding to the padding token
# In the convention of this note, we mask out the locations where the
# matrix elements of the mask matrix are true.
mask = (Xinput == PAD).view(d_B, 1, 1, d_S)
S = S.masked_fill(mask, float('-inf'))
S = torch.nn.functional.softmax(S, -1)
```

And the softmax above is along the last dimension of S, after which the dimensions of S do not change.

The final step of self-attention is to compute the average of the value tensor V with weights given by S. This is simply a matrix multiplication:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_E}}\right)V : (d_B, h, d_S, d_S) \times (d_B, h, d_S, d_k) = (d_B, h, d_S, d_k) \to (d_B, d_S, d_E)$$

where after the matrix multiplication, we transpose the second and third dimension, and flatten the last two dimensions such that the output of an attention layer has the same dimensions as its input. In Pytorch, this is simply

```
x = torch.matmul(S, V).view(d_B, d_seq, d_E)
```

After the self-attension, one adds a position-wise fully connected (FC) layer (which is constructed such that it does not change the dimension of x above). By position-wise FC layer, we mean the same FC layer will act on the final embedding of each token independent, that is, it is acting on the last dimension of x above.

What I have described so far is basically all the components of an encoder block. And the encoder of the transformer is simply a stack of N encoder blocks. Of course, in the actual implementation, we also need to add layer-norms, dropouts, and residual connections in various places, but those are simple to understand.

One important thing to notice is that most of the operations are done separately for each token. The whole model structure is almost permutation invariant except for the position embedding layer, which is essential for text tasks.

3 Decoder

The decoder part of the transformer model is very similar to the encoder part. For the machine translation task that we have in mind, the input to the encoder will be the translated sentences. Similar to what we did for the input to the encoder, we first pad the (tokenized) translated sentences in each batch to have the same sequence length. We denote the padded sentences as Y_{input}. We then do a token embedding and a positional embedding, and compute the sum of them. Let's denote the result after the embedding layers as $Y^{\text{E}}_{\text{input}}$. Next, we apply a self-attention layer to $Y^{\text{E}}_{\text{input}}$, as in the encoder case.

3.1 Masking

However, when computing the softmax in self-attention, we not only need to mask out the padding tokens, but also the "future tokens". The reason is that in the decoder, we are trying to generate text (for example, generate the translation of the input text to the encoder), and we do not want the model to be able to look at the answer. So during training, when generating the i-th token in the translated sentence, the model is only allowed to look at the first $i-1$ tokens in the translated sentence. The code block below construct the padding mask from Y_{input} and also the "future mask" (also called subsequent mask). Since we are using the convention that the locations where the matrix elements of the mask matrix are true are masked out, we simply apply the torch.logical_or operation to the pad_mask and future_mask to obtain the full mask for the decoder.

```
pad_mask = (Yinput == PAD).view(d_B, 1, 1, d_S)
temp = torch.ones(d_S, d_S, requires_grad=False)
future_mask = torch.triu(temp, diagonal=1)!=0
decoder_mask = torch.logical_or(pad_mask, future_mask)
# Assumming that S is the result of the matrix multiplication of the
# key and query tensor, then we apply the mask as follows:
S = S.masked_fill(decoder_mask, float('-inf'))
```

3.2 Another attention

Another thing that the decoder is different from the encoder is that, after this self-attention layer, we have another attention layer, with the key and value tensors from the output of the encoder layer, and query being the output of the self-attention layer of the decoder. This enables the model to learn the information of the source text that it is trying to translate. We will not say more since this is very similar to other attention layers discussed in this note. Similar to the encoder case, we apply a position-wise fully connected layer after the two attention layers.

The two attention layers and the position-wise fully connected layer form the decoder block. The decoder is simply a stack of N decoder blocks followed by another position-wise fully connected layer at the end. The output dimension of this last layer should be equal to the vocabulary size of the target language, since the output will be the logits of the words in the target language vocabulary.

3.3 Training and generating text

In the discussion above, we denote the input to the decoder as Y_{input}, which is a batch of sentences. To be more precise, in actual training, we input $Y_{\text{input}}[:, :-1]$ to the decoder, and use $Y_{\text{input}}[:, 1:]$ as the training target. In this way, with the 'future mask', each input sentence just needs to go through the decoder once, and the i-th output will be compared with the $i+1$-th token in the sentence to compute the loss. That is, we use the first i tokens in the input sentence to generate the $i+1$ token.

After training, when using the model to do translation or generating text, the initial input to the decoder will usually just be a beginning of sentence token (BOS), and we use the model to generate the next token auto-regressively. That is, after generating the first i tokens, we will use those as input to the model to generate the $i+1$ token. Therefore, if we want to generate n tokens, we will need to run the model n times.

4. Implementation of the transformer model

I have described the transformer model in the main text of this note, but I omitted some details. My purpose was to try to emphasize some confusing points when implementing the model. For actual implementation of various transformer models on various tasks, see my GitHub repository: [hbchen-one/Transformer-Models-from-Scratch](). The notations in the notebooks there will be a little different from this note, but should be understandable. The transformer model itself is not that complicated. It only takes less than 150 lines of code to implement (including the multi-head self-attention module), which I reproduce here for the convenience of the readers.

Multi-head attention

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_embed, dropout=0.0):
        super(MultiHeadedAttention, self).__init__()
        assert d_embed % h == 0 # check the h number
        self.d_k = d_embed//h
        self.d_embed = d_embed
        self.h = h
        self.WQ = nn.Linear(d_embed, d_embed)
        self.WK = nn.Linear(d_embed, d_embed)
        self.WV = nn.Linear(d_embed, d_embed)
```

```
        self.linear = nn.Linear(d_embed, d_embed)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x_query, x_key, x_value, mask=None):
        nbatch = x_query.size(0) # get batch size
        # 1) Linear projections to get the multi-head query, key and value
tensors
        # x_query, x_key, x_value dimension: nbatch * seq_len * d_embed
        # LHS query, key, value dimensions: nbatch * h * seq_len * d_k
        query = self.WQ(x_query).view(nbatch, -1, self.h,
self.d_k).transpose(1,2)
        key   = self.WK(x_key).view(nbatch, -1, self.h,
self.d_k).transpose(1,2)
        value = self.WV(x_value).view(nbatch, -1, self.h,
self.d_k).transpose(1,2)
        # 2) Attention
        # scores has dimensions: nbatch * h * seq_len * seq_len
        scores = torch.matmul(query, key.transpose(-2,
-1))/math.sqrt(self.d_k)
        # 3) Mask out padding tokens and future tokens
        if mask is not None:
            scores = scores.masked_fill(mask, float('-inf'))
        # p_atten dimensions: nbatch * h * seq_len * seq_len
        p_atten = torch.nn.functional.softmax(scores, dim=-1)
        p_atten = self.dropout(p_atten)
        # x dimensions: nbatch * h * seq_len * d_k
        x = torch.matmul(p_atten, value)
        # x now has dimensions:nbtach * seq_len * d_embed
        x = x.transpose(1, 2).contiguous().view(nbatch, -1, self.d_embed)
        return self.linear(x) # final linear layer
```

Residual connection

```
class ResidualConnection(nn.Module):
  '''residual connection: x + dropout(sublayer(layernorm(x))) '''
  def __init__(self, dim, dropout):
      super().__init__()
      self.drop = nn.Dropout(dropout)
      self.norm = nn.LayerNorm(dim)

  def forward(self, x, sublayer):
      return x + self.drop(sublayer(self.norm(x)))
```

Encoder and encoder blocks

```
# I simply let the model learn the positional embeddings in this notebook,
since this
# almost produces identital results as using sin/cosin functions embeddings,
as claimed
# in the original transformer paper. Note also that in the original paper,
they multiplied
# the token embeddings by a factor of sqrt(d_embed), which I do not do here.

class Encoder(nn.Module):
    '''Encoder = token embedding + positional embedding -> a stack of N
EncoderBlock -> layer norm'''
    def __init__(self, config):
        super().__init__()
        self.d_embed = config.d_embed
        self.tok_embed = nn.Embedding(config.encoder_vocab_size,
config.d_embed)
        self.pos_embed = nn.Parameter(torch.zeros(1, config.max_seq_len,
config.d_embed))
        self.encoder_blocks = nn.ModuleList([EncoderBlock(config) for _ in
range(config.N_encoder)])
        self.dropout = nn.Dropout(config.dropout)
        self.norm = nn.LayerNorm(config.d_embed)

    def forward(self, input, mask=None):
        x = self.tok_embed(input)
        x_pos = self.pos_embed[:, :x.size(1), :]
        x = self.dropout(x + x_pos)
        for layer in self.encoder_blocks:
            x = layer(x, mask)
        return self.norm(x)
```

```
class EncoderBlock(nn.Module):
    '''EncoderBlock: self-attention -> position-wise fully connected feed-
forward layer'''
    def __init__(self, config):
        super(EncoderBlock, self).__init__()
        self.atten = MultiHeadedAttention(config.h, config.d_embed,
config.dropout)
        self.feed_forward = nn.Sequential(
            nn.Linear(config.d_embed, config.d_ff),
            nn.ReLU(),
            nn.Dropout(config.dropout),
            nn.Linear(config.d_ff, config.d_embed)
        )
        self.residual1 = ResidualConnection(config.d_embed, config.dropout)
        self.residual2 = ResidualConnection(config.d_embed, config.dropout)

    def forward(self, x, mask=None):
        # self-attention
        x = self.residual1(x, lambda x: self.atten(x, x, x, mask=mask))
        # position-wise fully connected feed-forward layer
        return self.residual2(x, self.feed_forward)
```

Decoder and decoder blocks

```
class Decoder(nn.Module):
    '''Decoder = token embedding + positional embedding -> a stack of N
DecoderBlock -> fully-connected layer'''
    def __init__(self, config):
        super().__init__()
        self.d_embed = config.d_embed
        self.tok_embed = nn.Embedding(config.decoder_vocab_size,
config.d_embed)
        self.pos_embed = nn.Parameter(torch.zeros(1, config.max_seq_len,
config.d_embed))
        self.dropout = nn.Dropout(config.dropout)
        self.decoder_blocks = nn.ModuleList([DecoderBlock(config) for _ in
range(config.N_decoder)])
        self.norm = nn.LayerNorm(config.d_embed)
        self.linear = nn.Linear(config.d_embed, config.decoder_vocab_size)
```

```
    def future_mask(self, seq_len):
        '''mask out tokens at future positions'''
        mask = (torch.triu(torch.ones(seq_len, seq_len, requires_grad=False),
diagonal=1)!=0).to(DEVICE)
        return mask.view(1, 1, seq_len, seq_len)

    def forward(self, memory, src_mask, trg, trg_pad_mask):
        seq_len = trg.size(1)
        trg_mask = torch.logical_or(trg_pad_mask, self.future_mask(seq_len))
        x = self.tok_embed(trg) + self.pos_embed[:, :trg.size(1), :]
        x = self.dropout(x)
        for layer in self.decoder_blocks:
            x = layer(memory, src_mask, x, trg_mask)
        x = self.norm(x)
        logits = self.linear(x)
        return logits
```

```
class DecoderBlock(nn.Module):
    ''' EncoderBlock: self-attention -> position-wise feed-forward (fully
connected) layer'''
    def __init__(self, config):
        super().__init__()
        self.atten1 = MultiHeadedAttention(config.h, config.d_embed)
        self.atten2 = MultiHeadedAttention(config.h, config.d_embed)
        self.feed_forward = nn.Sequential(
            nn.Linear(config.d_embed, config.d_ff),
            nn.ReLU(),
            nn.Dropout(config.dropout),
            nn.Linear(config.d_ff, config.d_embed)
        )
        self.residuals = nn.ModuleList(
          [ResidualConnection(config.d_embed, config.dropout)
                                    for i in range(3)])

    def forward(self, memory, src_mask, decoder_layer_input, trg_mask):
        x = memory
        y = decoder_layer_input
        y = self.residuals[0](y, lambda y: self.atten1(y, y, y,
mask=trg_mask))
```

```
        # keys and values are from the encoder output
        y = self.residuals[1](y, lambda y: self.atten2(y, x, x,
mask=src_mask))
        return self.residuals[2](y, self.feed_forward)
```

Transformer!

```
class Transformer(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, src_mask, trg, trg_pad_mask):
        return self.decoder(self.encoder(src, src_mask), src_mask, trg,
trg_pad_mask)
```

References

1. the Attention Is All You Need paper [arXiv:1706.03762]
2. [The Annotated Transformer by Alexander Rush]