Some details of the transformer model

This short 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.

1. Notation

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

For the dimensions of a tensor X, I will denote it as follows

$$X : (d_1, d_2, \ldots . d_n) ,$$

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's 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 contains d_B 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 length 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

dimension

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

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

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

2.2 Multi-head self-attention

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

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

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_{input} with tensors W_{query}, $W_{\text{key}}, W_{\text{value}}$, which have dimensions:

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

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

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

The dimensions on the left of the arrows are what we get after the matrix multiplication, but 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 notation. 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 take 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):

$$S \equiv QK^T : (d_B, h, d_S, d_k) \times (d_B, h, d_k, d_S) = (d_B, h, d_S, d_S) \, .$$

In Pytorch, this is simply implemented by

```
S = torch.matmul(Q, K.transpose(-2,-1))
```

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
mask = (Xinput == PAD).view(d_B, 1, 1, d_S)
S = S.masked_fill(mask, float('-inf'))
S = torch.nn.functional.softmax(S, -1)/math.sqrt(self.d_E)
```

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:

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

where after the matrix multiplication, we transpose the second and third dimension, and flatten the last two dimension 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, -1, h*d_k)
```

After the self-attension, one adds a fully connected layer (which does not change the dimension of x above). And that is basically it for the encoder of the transformer model.

To be continued...