The embedding module, nn.Embedding(V, D), is a pre-built module that allows for the mapping of a vocab-based token sequence into the embedding space.

The vocab-based token sequence is just a sequence of L integers, ranging from 0 to V-1, where V is the size of the vocabulary.

To be more precise, the input is a tensor of shape [L] with non-negative integers less than V.

The output is a tensor of shape [L, D], where each token in the vocabulary is embedded into a D-dimensional array of real numbers.

Just like other pre-built PyTorch modules, nn.Embedding supports batches. Therefore, if the input is a tensor of shape [B, L], it will return a tensor of shape [B, L, D].

input.shape = torch.Size([2, 5])

embedding V = 100, D = 8

output.shape = torch.Size([2, 5, 8])

Transformer

Iinput is a tensor with shape [B, Ls], and the output is a tensor with shape [B, Lt], where Ls and Lt are the lengths of the source and target sequences. Ls and Lt don't need to be the same, but both must be no larger than maxL, which is used in the positional encoding modules -- this is just so that we can pre-build the positional encoder once and for all in the constructor.

[B, Ls] --> [B, Ls, D] --> [B, Ls, D] --> ... --> [B, Ls, D]

Let's say we choose the embedding dimension to be 4, and there are 5 encoders on the stack (i.e. Nx = 5), the sequence for our example would be:

[2, 9] --> [2, 9, 4] --> [2, 9, 4] --> [2, 9, 4] --> [2, 9, 4] --> [2, 9, 4] --> [2, 9, 4]

source initial encoder #1 encoder #2 encoder #3 encoder #4 encoder #5

embedding ==========

final encoder output

Tensor C of shape [B, Ls, D]

Like the encoder, the decoder repeatedly "transforms" the tensor during each pass.

[B, Lt] --> [B, Lt, D] --> [B, Lt, D] --> ... --> [B, Lt, D]

Note that the final decoder output is a tensor of shape [B, Lt, D]. How does this turn into the next token?

Well, it turns out there are two more modules, a linear module and a softmax module, to turn the embeddings to actual probabilities:

[B, Lt, D] --> linear --> [B, Lt, Vt] --> softmax --> [B, Lt, Vt]

===========

probabilities

Transformer inference is done one-token at a time, so if you have 1000 tokens in the sentence, we will generate each one of the 1000 tokens one after the another.

Training, on the other hand, is done in a single pass! This is actually quite amazing.

A key innovation (which pre-dated transformer) is "teacher forcing". While we are still trying to predict the next token, once it is predicted, regardless of the prediction, we'll discard it and use the next correct token for the next prediction.

It may be instructive to think of the self-attention module as a kind of inner "enlightenment" within itself -- improving and enriching each individual based on the context of the entire group.

