Aryaman Pandya
Dec 19, 2023

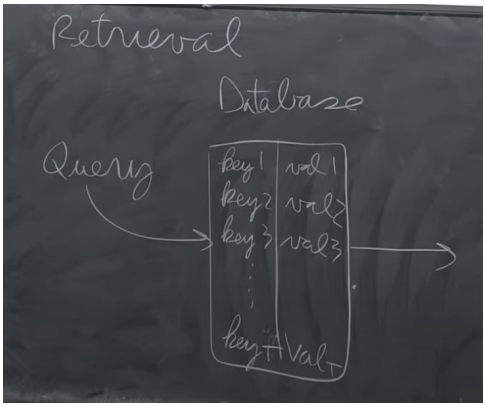
CS 680, Waterloo Intro ML, Lec 19 Attention and Transformer Networks

Attention (previously covered briefly, Recurrent Nets to Attention to Transformers) was starting to be used in multiple DL tasks.

- In CV, attention was used to highlight important parts of an image that contribute to a desired output
- In NLP, aligned machine translation (in prev doc), then language modeling with **transformers**

Transformer v/s RNN:
Transformers can facilitate long range dependencies, don't vanish or explode gradients, use fewer steps and the lack of recurrence facilitates parallel computation greater than that possible with RNNs

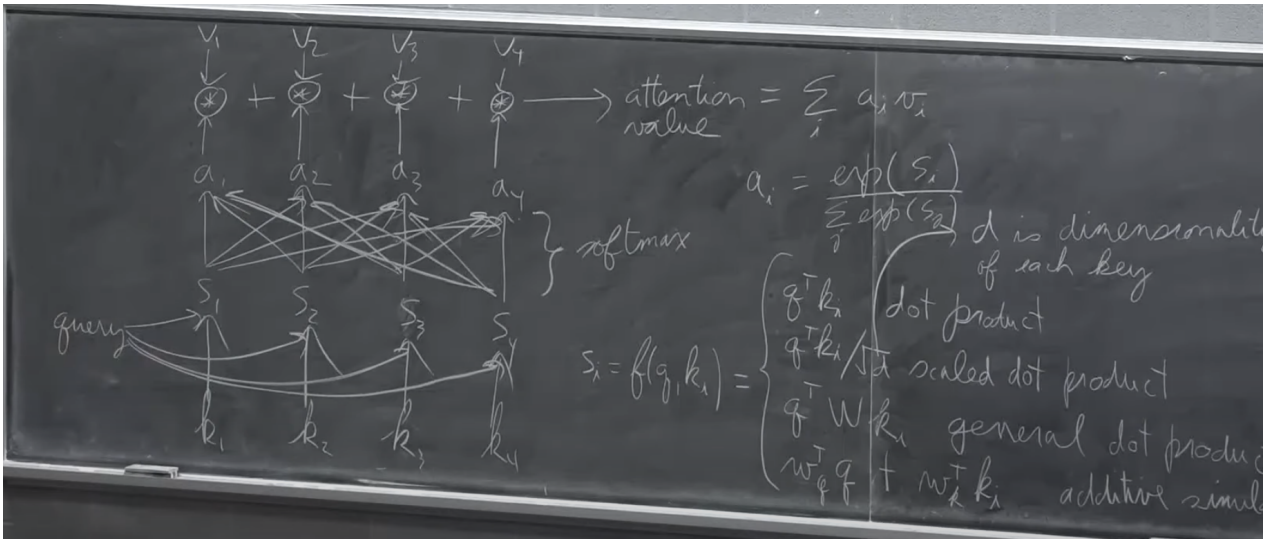
Transformer refined Attention mechanism

- Mimics the retrieval of a value for a query based on a key in a database
- Database intuition:
    - 

    - Given some query, we find which key it matches the most and then return its value
- Much like this intuition, we mathematically compute the similarity between key and query, and multiply this by the value. We then sum these to get our attention vector:
    - $attention(q, \boldsymbol{k}, \boldsymbol{v}) = \sum_i similarity(q, k_i) \times v_i$
- In the db case, we return a one hot version of this essentially (1 for the most similar and 0 for everything else, giving us the one most similar as our output) in the attention case we compute probabilities for all possible values -> we return a distribution

- Similarity function options:
    - Dot product q dot k_i
        - Scaled dot product
    - General dot product
        - q^TW k_i -> use the weight matrix W to project the dot product into a new space (transforming our query to be in the same space as our key)
    - Additive similarity
        - w_q^Tq + w_k^Tk_i
- If it isn't straightforward to compute the similarity, the **weighted methods above are used where the system learns the similarity weights**
- These **similarities are then used as inputs to compute our attention weights**
    - This is done via a softmax, **where a_i is the softmax(s)**, just like in the older version of attention in the previous lecture/document
- We then **multiply the attention weights with the corresponding values** (from key-values, to reduce ambiguity) and then **sum this to get our attention value**

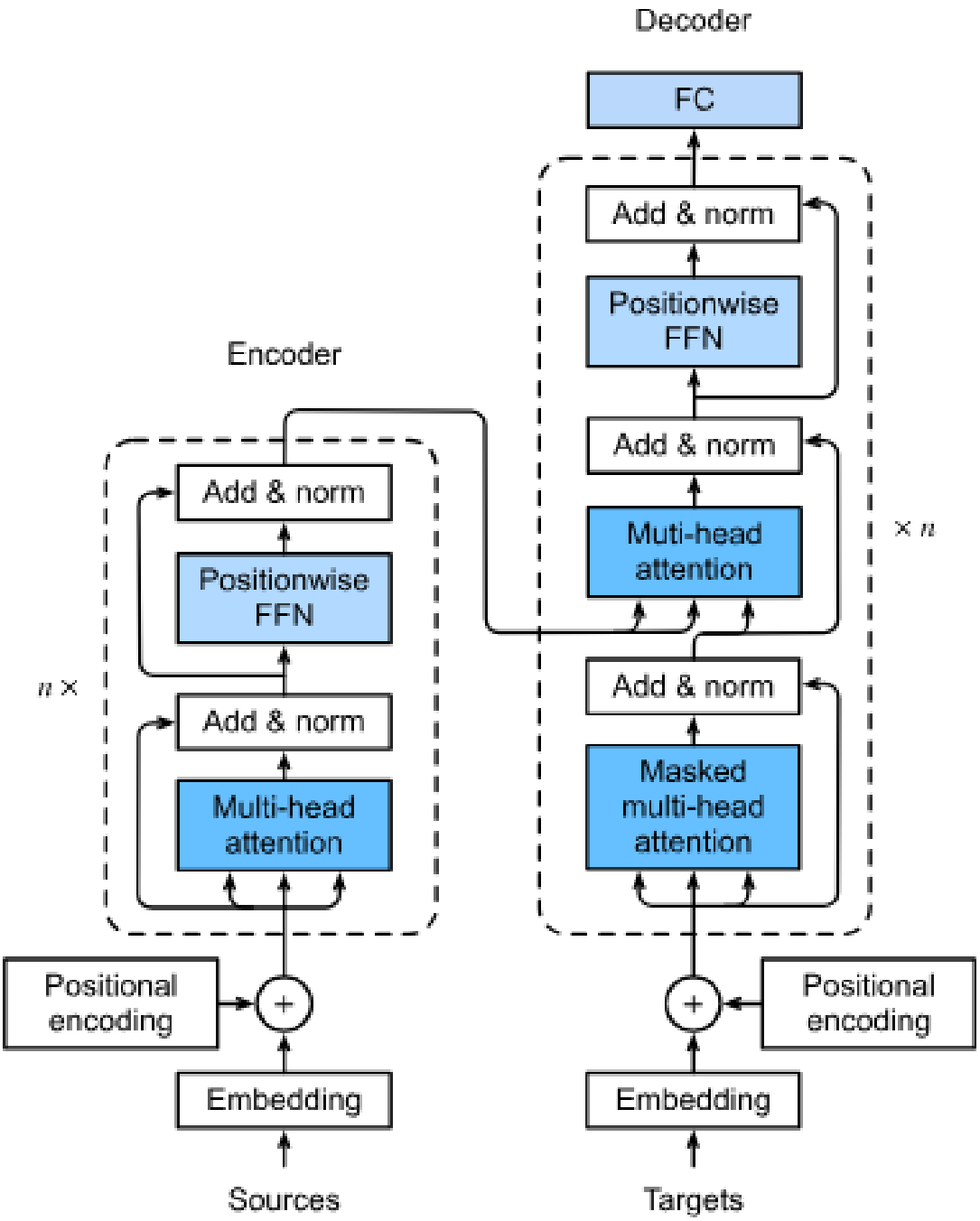
All of the above notes are summarized by the following image:



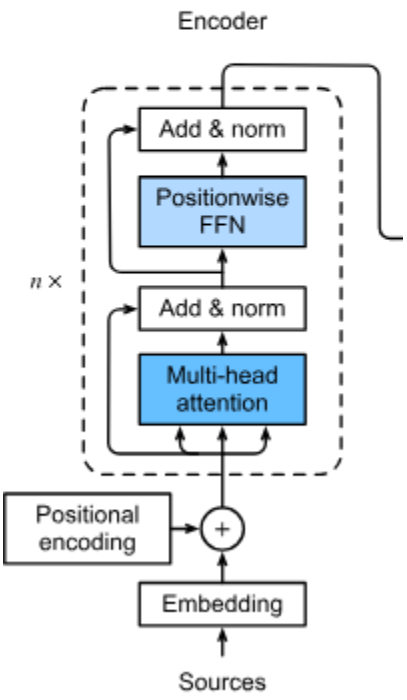
k_is are vectors, s_is are scalars, a_is are scalars, v_is are vectors

Transformer Networks

Vaswani et al (2017) Attention is all you need
Encoder-decoder based on attention (no recurrence)



Encoder



Input embedding is the first step, followed by addition of positional encoding.

- Embedding converts to vector space.
- Positional encoding ensures we can differentiate between different instances of the same word (where they occur) in the sentence

Then, we run the input through a multi headed attention layer that computes the attention between every position and every other position

- Treat each word as a query (vector coming out of positional encoding) and find keys that correspond to other words in the sentence and take a linear combination of the corresponding value. Here, the values are going to be the same as the keys, **add it to the attention value to give us a better embedding of the input,** then normalize

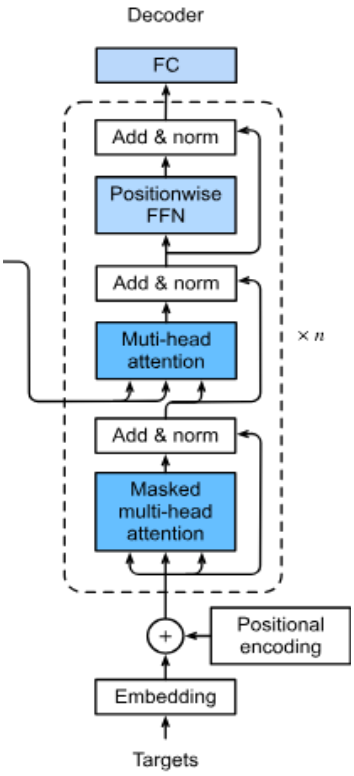
Norm in the Add+norm block is a layer normalization

This is then followed by a feed forward network (Position wise MLP) with a residual connection to an add and norm block

This summarizes the encoder architecture. The encoder block is repeated N times, where we compute attention with regards to words, pairs of words, pairs of pairs of words,... Nx. **This way we can understand different order relationships between the input tokens**

The output of the encoder can be thought of as a large embedding containing information about the words and the words they attended to to build an encoding of the input sequence

Decoder



One of the big differences in the decoder when compared to the encoder is there exists some linear and softmax output layers that allow for us to compute some output probabilities. These could be classifications, etc.

Looking into the decoder:

The decoder block also repeats N times - >gradually build up combinations and get better embeddings

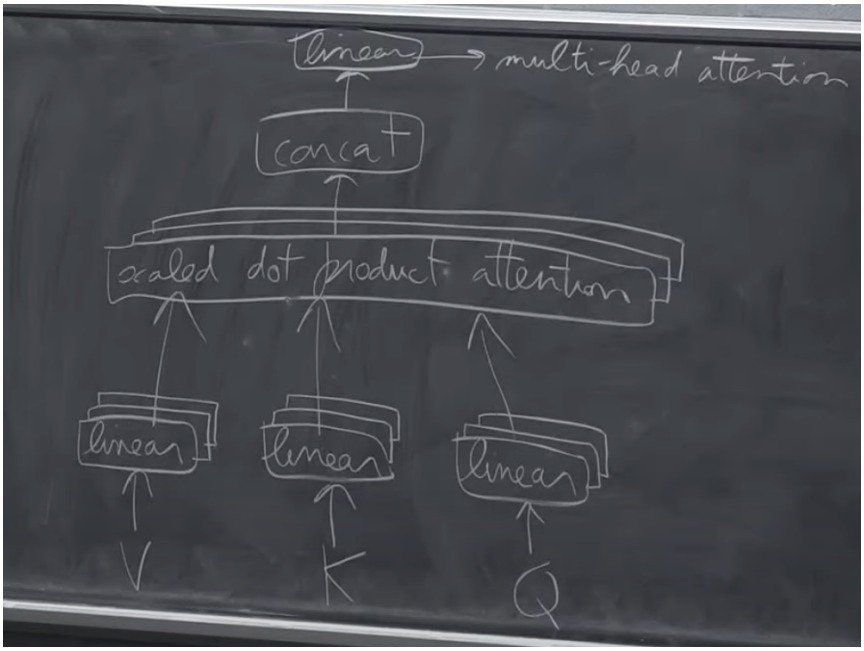
Another big difference is that we have two multi headed attention blocks. One computes the attention values for output words with output words and one computes the attention values for output words with input words

When we run the self attention layer in the decoder, we have to be careful that we only attend to previous words since the current word is only generated **after** the previous words
That's why this is called "Masked Multi-head Attention" since we **mask** the future words

Multihead attention

Compute multiple attentions per query with different weights

1. Feed V, K, Q into some linear layer
2. Compute a scaled dot product attention
3. Concatenate outputs
4. Feed through another linear layer
5. Output is a multihead attention



It's called a multihead attention because we compute multiple different attention weights

Consider the output of the linear layers to be a projection of the Q/K/V. Now, we use multiple layers stacked there, to create multiple projections

We also stack the dot product attention like this to get 3 different values.

This idea is kind of like multiple filters in CNNs, each learning some different features. Here, we learn different projections (feature representations)

The concat layer then concatenates the different scaled dot product attentions. We use all of them to compute the output multihead attention.

MaskedAttention:

$$maskedAttention(Q,K,V) = softmax\left(\frac{Q^T K + M}{\sqrt{d_k}}\right)V$$

where M is a mask matrix of 0's and $-\infty$'s

The need/idea is to mask tokens that are in the future. We only want to look at past tokens since they are the only ones that influence the present token. The exp of -infinity would drive the probability for that key/value to zero

The mask is inside the softmax so we don't need to re-compute the probability distribution after the fact

Back to the Transformer as a whole

The first word produced by the output softmax (token with highest probability) can then be fed into the decoder network as input (bottom of the decoder block where it says Outputs right shifted) – **AT TEST TIME.** During training, we do not have any recurrence like this. More below:

Note - look into teacher forcing

- Idea is when you train the network you have both the input and output sentences. So we feed in output from this. We don't create any recurrence.

We don't have recurrence at train time due to teacher forcing, but we do during test time

Other Layers

Layer Normalization:

- On top of every multi head attention and feed forward layer
- Helps reduce the number of steps needed by SGD to optimize parameters

We do this by:

- Normalizing values in each layer to have 0 mean and 1 variance
  - For each hidden unit h_i compute $h_i \leftarrow \frac{g}{\sigma}(h_i - \mu)$

    where g is a variable, $\mu = \frac{1}{H}\sum_{i=1}^{H} h_i$ and $\sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(h_i - \mu)^2}$

This reduces "covariate shift" ie. gradient dependencies between each layer and therefore fewer training iterations are needed

Positional Embedding

Applied right after input/output embedding. The embedding will convert to vector space, then the positional embedding embeds information about relative position within the sentence (or stream of inputs, if it is not a NLP task)

- Attention doesn't take into account position - positional embedding is needed to introduce this
- Not clear that this is the best way to do this, but it works – "engineering hack"

The authors, Vaswani et. al. Chose to **add** the positional embedding instead of concatenate it. The professor thinks that it might make more sense to concatenate so that we don't potentially lose information from the word embeddings.

Comparison of different methods

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

GPT and GPT-2

Decoder only transformer- no reason to separate input and output sequences, therefore there's no real need for the encoder

Zero-shot learning - no task training, only unsupervised language modeling

BERT (Bidirectional encoder representations from transformers

Decoder transformer that predicts a missing word based on surrounding words by computing probability

- Mask missing wordwith masked multihead attention
- Not zero shot results, pre training + fine tuning instead