UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

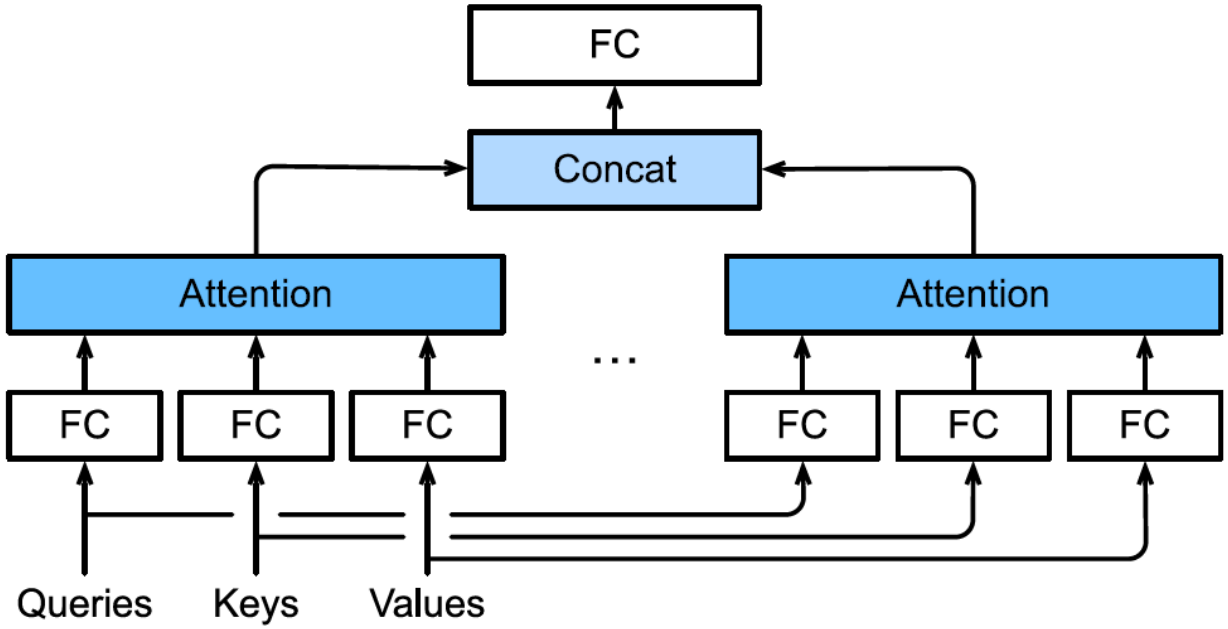
Introduction to ML
Lecture 13: Transformers Implementation

Hamed Tabkhi
Department of Electrical and Computer Engineering,
University of North Carolina Charlotte (UNCC)
htabkhiv@uncc.edu

Multi-Head Attention

- In practice, given the same set of queries, keys, and values we may want our model to combine knowledge from different behaviors of the same attention mechanism, such as capturing dependencies of various ranges (e.g., shorter-range vs. longer-range) within a sequence.
- Thus, it may be beneficial to allow our attention mechanism to jointly use different representation subspaces of queries, keys, and values.
- **To this end, instead of performing a single attention pooling, queries, keys, and values can be transformed with *h* independently learned linear projections/alignments.**
- Then these *h* projected queries, keys, and values are fed into attention pooling in parallel.
- In the end, *h* attention pooling outputs are concatenated and transformed with another learned linear projection to produce the final output.
- This design is called *multi-head attention*, where each of the *h* attention pooling outputs is a *head* (Vaswani *et al.*, 2017).
- Using fully connected layers to perform learnable linear transformations.

Multi-Head Attention



Multi-head attention, where multiple heads are concatenated then linearly transformed.

Multi-Head Attention: Model

Before providing the implementation of multi-head attention, let's formalize this model mathematically. Given a query $\mathbf{q} \in \mathbb{R}^{d_q}$, a key $\mathbf{k} \in \mathbb{R}^{d_k}$, and a value $\mathbf{v} \in \mathbb{R}^{d_v}$, each attention head $\mathbf{h}_i$ $(i = 1, \ldots, h)$ is computed as

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}, \tag{11.5.1}$$

where learnable parameters $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ and $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$, and f is attention pooling, such as additive attention and scaled dot-product attention in Section 11.3. The multi-head attention output is another linear transformation via learnable parameters $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$ of the concatenation of h heads:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}.$$

Multi-Head Attention: Implementation

In our implementation, we choose the scaled dot-product attention for each head of the multi-head attention. To avoid significant growth of computational cost and parameterization cost, we set $p_q = p_k = p_v = p_o/h$. Note that h heads can be computed in parallel if we set the number of outputs of linear transformations for the query, key, and value to $p_q h = p_k h = p_v h = p_o$. In the following implementation, p_o is specified via the argument `num_hiddens`.

Self-Attention and Positional Encoding

- In deep learning, we often use CNNs or RNNs to encode sequences.
- Now with attention mechanisms in mind, imagine feeding a sequence of tokens into an attention mechanism such that at each step, each token has its own query, keys, and values.
- Here, when computing the value of a token's representation at the next layer, the token can attend (via its query vector) to each other token (matching based on their key vectors).
- Using the full set of querykey compatibility scores, we can compute, for each token, a representation by building the appropriate weighted sum over the other tokens. Because each token is attending to each other token
- Unlike the case where decoder steps attend to encoder steps), such architectures are typically described as *self-attention* models (Lin *et al.*, 2017, Vaswani *et al.*, 2017), and elsewhere described as *intra-attention* model (Cheng *et al.*, 2016, Parikh *et al.*, 2016, Paulus *et al.*, 2017).
- We will discuss sequence encoding using self-attention, including using additional information for the sequence order.

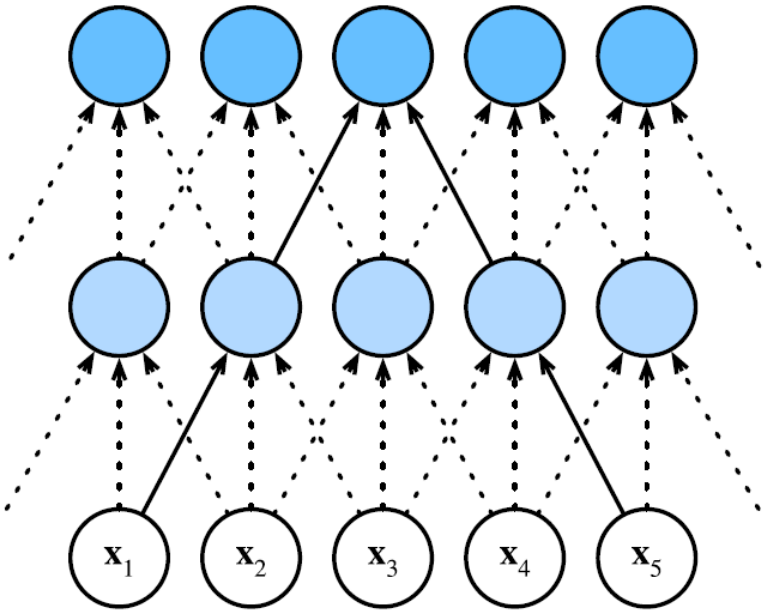
Self-Attention

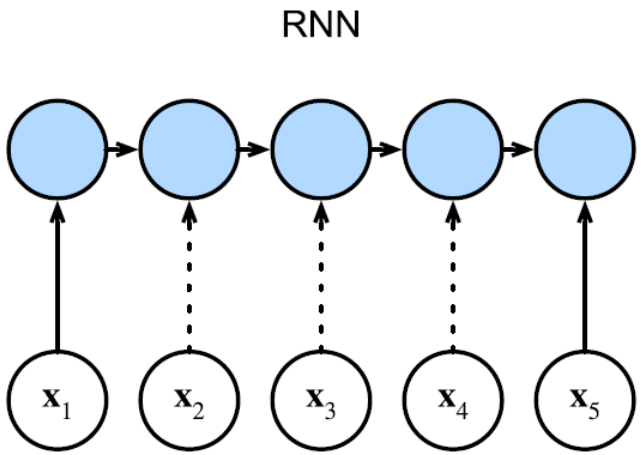
Given a sequence of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where any $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$, its self-attention outputs a sequence of the same length $\mathbf{y}_1, \ldots, \mathbf{y}_n$, where

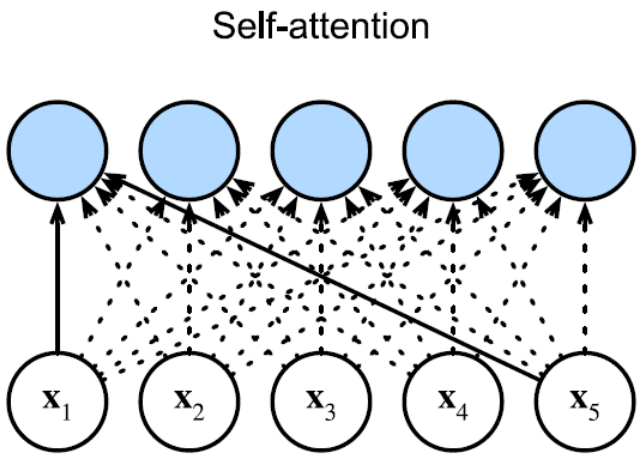
$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \tag{11.6.1}$$

according to the definition of attention pooling f in `eq_attn-pooling-def`. Using multi-head attention, the following code snippet computes the self-attention of a tensor with shape (batch size, number of time steps or sequence length in tokens, d). The output tensor has the same shape.

Comparing CNNs, RNNs, and Self-Attention







UNC CHARLOTTE

Comparing CNNs, RNNs, and Self-Attention

- Consider a convolutional layer whose kernel size is *k*. We will provide more details about sequence processing using CNNs in later chapters. For now, we only need to know that since the sequence length is *n*, the numbers of input and output channels are both *d*, the computational complexity of the convolutional layer is *O*(*knd*^*2*).

- When updating the hidden state of RNNs, multiplication of the *d* *d* weight matrix and the *d*-dimensional hidden state has a computational complexity of *O* (*d*^*2*). Since the sequence length is *n*, the computational complexity of the recurrent layer is *O*(*nd*^*2*).

- In self-attention, the queries, keys, and values are all *n* *d* matrices. Consider the scaled dot-product attention, where a *n* *d* matrix is multiplied by a *d* *n* matrix, then the output *n* *n* matrix is multiplied by a *n* *d* matrix. As a result, the self-attention has a *O*(*n*^*2* *d*) computational complexity.

Positional Encoding

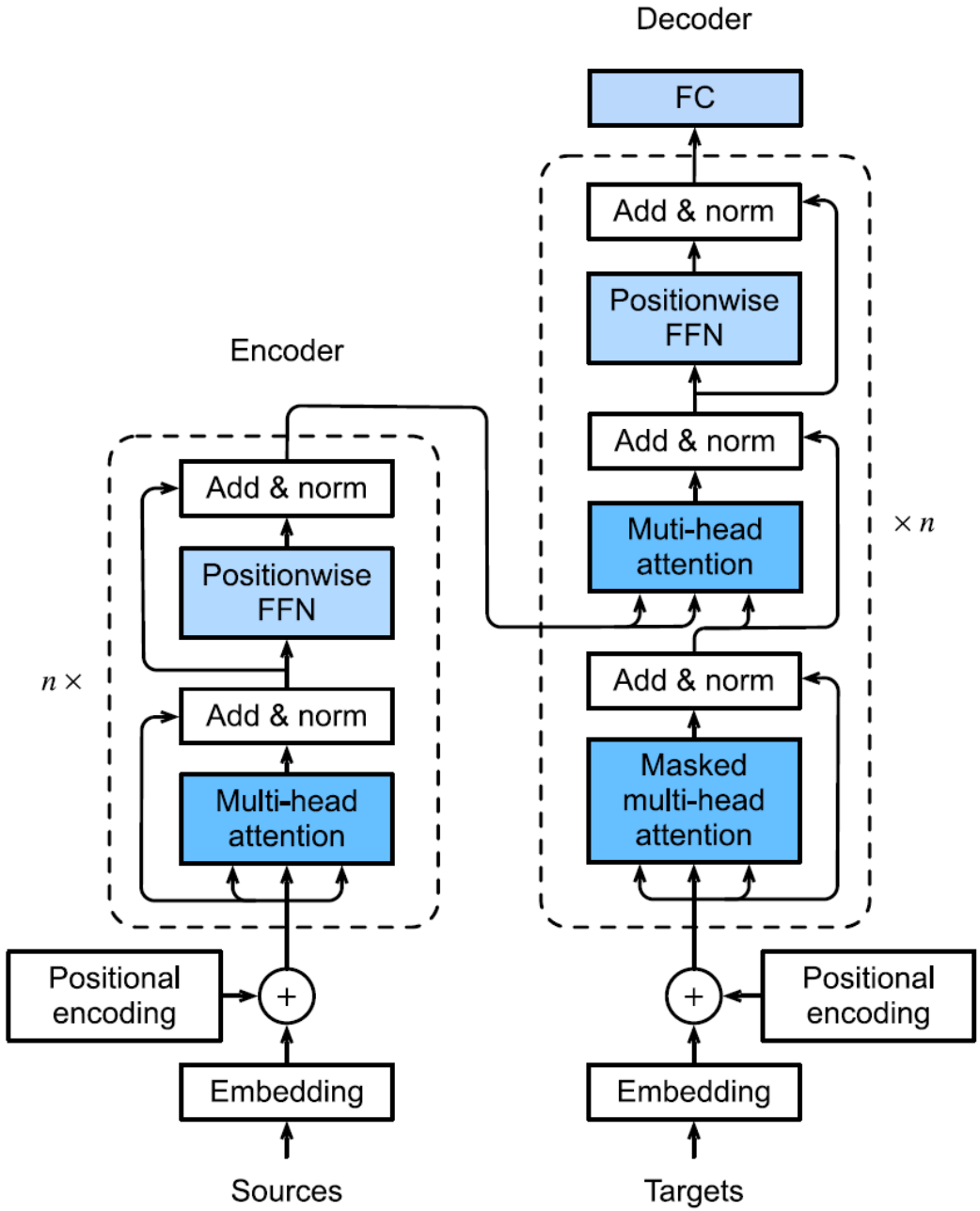
- Unlike RNNs, which recurrently process tokens of a sequence one by one, self-attention ditches sequential operations in favor of parallel computation.
- Note, however, that self-attention by itself does not preserve the order of the sequence. What do we do if it really matters that the model knows in which order the input sequence arrived?
- The dominant approach for preserving information about the order of tokens is to represent this to the model as an additional input associated with each token.
- These inputs are called *positional encodings*. and they can either be learned or fixed a priori.
- We now describe a simple scheme for fixed positional encodings based on sine and cosine functions (Vaswani *et al.*, 2017).

Positional Encoding

Suppose that the input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains the d-dimensional embeddings for n tokens of a sequence. The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$ of the same shape, whose element on the i^{th} row and the $(2j)^{\text{th}}$ or the $(2j+1)^{\text{th}}$ column is

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{11.6.2}$$

Transformer Architecture



UNC CHARLOTTE

Transformer Architecture

- On a high level, the Transformer encoder is a stack of multiple identical layers, where each layer has two sublayers (either is denoted as sublayer).
- The first is a multi-head self-attention pooling and the second is a positionwise feed-forward network.
- Specifically, in the decoder self-attention, queries, keys, and values are all from the outputs of the previous decoder layer. Inspired by the ResNet design, a residual connection is employed around both sublayers.
- Transformer encoder outputs a *d*-dimensional vector representation for each position of the input sequence.

Transformer Architecture

- The Transformer decoder is also a stack of multiple identical layers with residual connections and layer normalizations.
- Besides the two sublayers described in the encoder, the decoder inserts a third sublayer, known as the encoder-decoder attention, between these two.
- Encoder-decoder attention, queries are from the outputs of the previous decoder layer, and the keys and values are from the Transformer encoder outputs.
- In the decoder self-attention, queries, keys, and values are all from the outputs of the previous decoder layer. However, each position in the decoder is allowed to only attend to all positions in the decoder up to that position.
- This *masked* attention preserves the auto-regressive property, ensuring that the prediction only depends on those output tokens that have been generated.

Positionwise Feed-Forward Networks

- The positionwise feed-forward network transforms the representation at all the sequence positions using the same MLP. This is why we call it *positionwise*.

- In the implementation below, the input X with shape (batch size, number of time steps or sequence length in tokens, number of hidden units or feature dimension) will be transformed by a two-layer MLP into an output tensor of shape (batch size, number of time steps, ffn_num_outputs).

Residual Connection and Layer Normalization

- Despite its pervasive applications in computer vision, batch normalization is usually empirically less effective than layer normalization in natural language processing tasks, whose inputs are often variablelength sequences.
- We can implement the AddNorm class using a residual connection followed by layer
- normalization.
- Dropout is also applied for regularization.
- The residual connection requires that the two inputs are of the same shape so that the output tensor also has the same shape after the addition operation.

Encoder

- With all the essential components to assemble the Transformer encoder, let's start by implementing a single layer within the encoder.
- The following TransformerEncoderBlock class contains two sublayers: multi-head self-attention and positionwise feed-forward networks, where a residual connection followed by layer normalization is employed around both sublayers.
- As we can see, any layer in the Transformer encoder does not change the shape of its input.

- Transformer encoder implementation, we stack num_blks instances of the above TransformerEncoderBlock classes.
- Since we use the fixed positional encoding whose values are always between -1 and 1, we multiply values of the learnable input embeddings by the square root of the embedding dimension to rescale before summing up the input embedding and the positional encoding.

Decoder

- The Transformer decoder is composed of multiple identical layers.
- Each layer is implemented in the following TransformerDecoderBlock class, which contains three sublayers: decoder self-attention, encoder-decoder attention, and positionwise feed-forward networks.
- These sublayers employ a residual connection around them followed by layer normalization.
- As we described earlier in this section, in the masked multi-head decoder self-attention (the first sublayer), queries, keys, and values all come from the outputs of the previous decoder layer.
- When training sequence-to-sequence models, tokens at all the positions (time steps) of the output sequence are known.
- During prediction the output sequence is generated token by token; thus, at any decoder time step only the generated tokens can be used in the decoder self-attention.
- To preserve auto-regression in the decoder, its masked self-attention specifies dec_valid_lens so that any query only attends to all positions in the decoder up to the query position.

Decoder

- To facilitate scaled dot-product operations in the encoder-decoder attention and addition operations in the residual connections, the feature dimension (num_hiddens) of the decoder is the same as that of the encoder.

- Now we construct the entire Transformer decoder composed of num_blks instances of TransformerDecoderBlock.
- In the end, a fully connected layer computes the prediction for all the vocab_size possible output tokens.
- Both of the decoder self-attention weights and the encoder-decoder attention weights are stored for later visualization.

Training

- Let's instantiate an encoder-decoder model by following the Transformer architecture.
- Here, we specify that both the Transformer encoder and the Transformer decoder have 2 layers using 4-head attention.
- Similar to before, we train the Transformer model for sequence to sequence learning on the English-French machine translation dataset.

Evaluation

- Let's visualize the Transformer attention weights when translating the last English sentence into French.
- The shape of the encoder self-attention weights is (number of encoder layers, number of attention heads, num_steps or number of queries, num_steps or number of keyvalue pairs).
- In the encoder self-attention, both queries and keys come from the same input sequence.
- Since padding tokens do not carry meaning, with specified valid length of the input sequence, no query attends to positions of padding tokens.
- In the following, two layers of multi-head attention weights are presented row by row.
- Each head independently attends based on a separate representation subspaces of queries, keys, and values.

Key Takehomes

- The Transformer is an instance of the encoder-decoder architecture, though either the encoder or the decoder can be used individually in practice.
- In the Transformer architecture, multi-head self-attention is used for representing the input sequence and the output sequence, though the decoder has to preserve the auto-regressive property via a masked version.
- Both the residual connections and the layer normalization in the Transformer are important for training a very deep model.
- The positionwise feed-forward network in the Transformer model transforms the representation at all the sequence positions using the same MLP.
- Although the Transformer architecture was originally proposed for sequence-to-sequence learning, as we will discover later in the book, either the Transformer encoder or the Transformer decoder is often individually used for different deep learning tasks.