Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 10

December 13th, 2022

5.5 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

5.5 Multi-head attention

- to this end, instead of performing a single attention pooling, queries, keys, and values can be transformed with h independently learned linear projections
- then, these h projected queries, keys, and values are fed into attention pooling in parallel
- in the end, the 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*
- using fully-connected layers to perform learnable linear transformations, Figure 6 describes multi-head attention

5.5 Multi-head attention

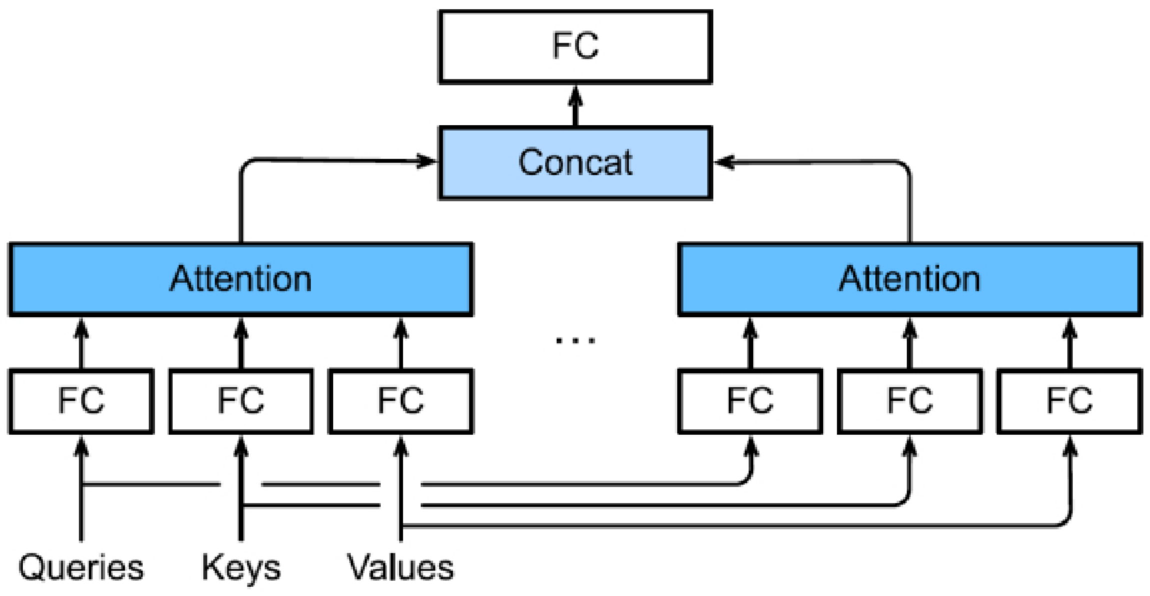

Figure 6: Multi-head attention, where multiple heads are concatenated, then linearly transformed.

5.5 Multi-head attention

- let us formalize this model mathematically; given a query $\boldsymbol{q} \in \mathbb{R}^{d_q}$, a key $\boldsymbol{k} \in \mathbb{R}^{d_k}$, and a value $\boldsymbol{v} \in \mathbb{R}^{d_v}$, each attention head $\boldsymbol{h}_i$ ($i = 1, \dots, h$) is computed as:

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

  where the parameters $\boldsymbol{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\boldsymbol{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$, and $\boldsymbol{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ are learnable, and f is attention pooling, such as additive attention or scaled dot-product attention, discussed in Section 5.3
- the multi-head attention output is another linear transformation via learnable parameters $\boldsymbol{W}_o \in \mathbb{R}^{p_o \times hp_v}$ of the concatenation of the h heads:

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

- based on this design, each head may attend to different parts of the input; more sophisticated functions than the simple weighted average can be expressed

5.6 Self-attention and positional encoding

- in deep learning, we often use CNNs or RNNs to encode a sequence
- now, with attention mechanisms, imagine that we feed a sequence of tokens into attention pooling, so that the same set of tokens act as queries, keys, and values
- specifically, each query attends to all the key-value pairs and generates one attention output
- since the queries, keys, and values come from the same place, this performs *self-attention*, which is also called *intra-attention*
- in this section, we will discuss sequence encoding using self-attention, including using additional information for the sequence order

5.6 Self-attention and positional encoding

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

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

  according to the definition of attention pooling f in (5)
- let us compare architectures for mapping a sequence of n tokens to another sequence of equal length, where each input or output token is represented by a d-dimensional vector
- specifically, we will consider CNNs, RNNs, and self-attention
- we will compare their computational complexity, sequential operations, and maximum path lengths
- note that sequential operations prevent parallel computation, while a shorter path between any combination of sequence positions makes it easier to learn long-range dependencies within the sequence

5.6 Self-attention and positional encoding

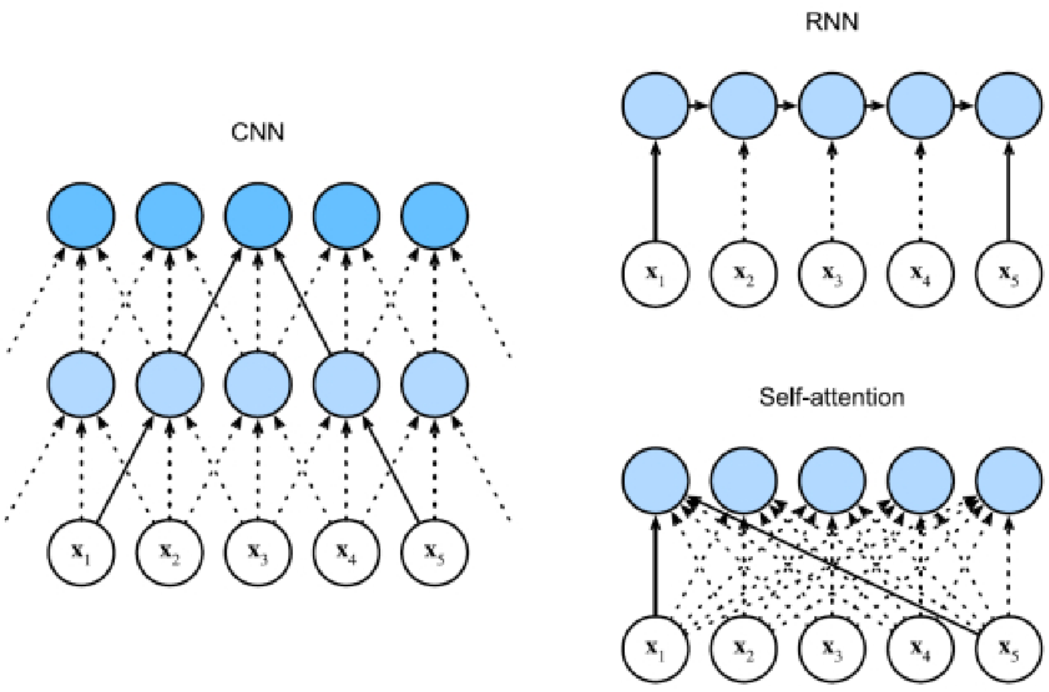

Figure 7: Comparing CNN (padding tokens are omitted), RNN, and self-attention architectures.

5.6 Self-attention and positional encoding

- consider a convolutional layer whose kernel size is k
- since the sequence length is n, the numbers of input and output channels are both d, the computational complexity of the convolutional layer is $\mathcal{O}(knd^2)$
- as Figure 7 shows, CNNs are hierarchical, so there are $\mathcal{O}(1)$ sequential operations and the maximum path length is $\mathcal{O}(n/k)$
- for example, $\boldsymbol{x}_1$ and $\boldsymbol{x}_5$ are within the receptive field of a two-layer CNN with kernel size 3 in Figure 7

5.6 Self-attention and positional encoding

- when updating the hidden state of RNNs, multiplication of the $d \times d$ weight matrix and the d-dimensional hidden state has a computational complexity of $\mathcal{O}(d^2)$
- since the sequence length is n, the computational complexity of the recurrent layer is $\mathcal{O}(nd^2)$
- according to Figure 7, there are $\mathcal{O}(n)$ sequential operations that cannot be parallelized and the maximum path length is also $\mathcal{O}(n)$

5.6 Self-attention and positional encoding

- in self-attention, the queries, keys, and values are all $n \times d$ matrices
- consider the scaled dot-product attention in (8), where a $n \times d$ matrix is multiplied by a $d \times n$ matrix, then the output $n \times n$ matrix is multiplied by a $n \times d$ matrix
- as a result, the self-attention has a $\mathcal{O}(n^2 d)$ computational complexity
- as we can see in Figure 7, each token is directly connected to any other token via self-attention
- therefore, computation can be parallel with $\mathcal{O}(1)$ sequential operations and the maximum path length is also $\mathcal{O}(1)$

5.6 Self-attention and positional encoding

- all in all, both CNNs and self-attention enjoy parallel computation, and self-attention has the shortest maximum path length
- however, the quadratic computational complexity with respect to the sequence length makes self-attention prohibitively slow for very long sequences
- unlike RNNs that recurrently process tokens of a sequence one by one, self-attention ditches sequential operations in favor of parallel computation
- to use the sequence order information, we can inject absolute or relative positional information by adding *positional encoding* to the input representations
- positional encodings can be either learned or fixed; in the following, we describe a fixed positional encoding based on sine and cosine functions

5.6 Self-attention and positional encoding

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

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{9}$$

5.6 Self-attention and positional encoding

- in the positional embedding matrix ***P***, rows correspond to positions within a sequence and columns represent different positional encoding dimensions
- in the example in Figure 8, we can see that the 6th and the 7th columns of the positional embedding matrix have a higher frequency than the 8th and the 9th columns
- the offset between the 6th and the 7th (same for the 8th and the 9th) columns is due to the alternation of sine and cosine functions

5.6 Self-attention and positional encoding

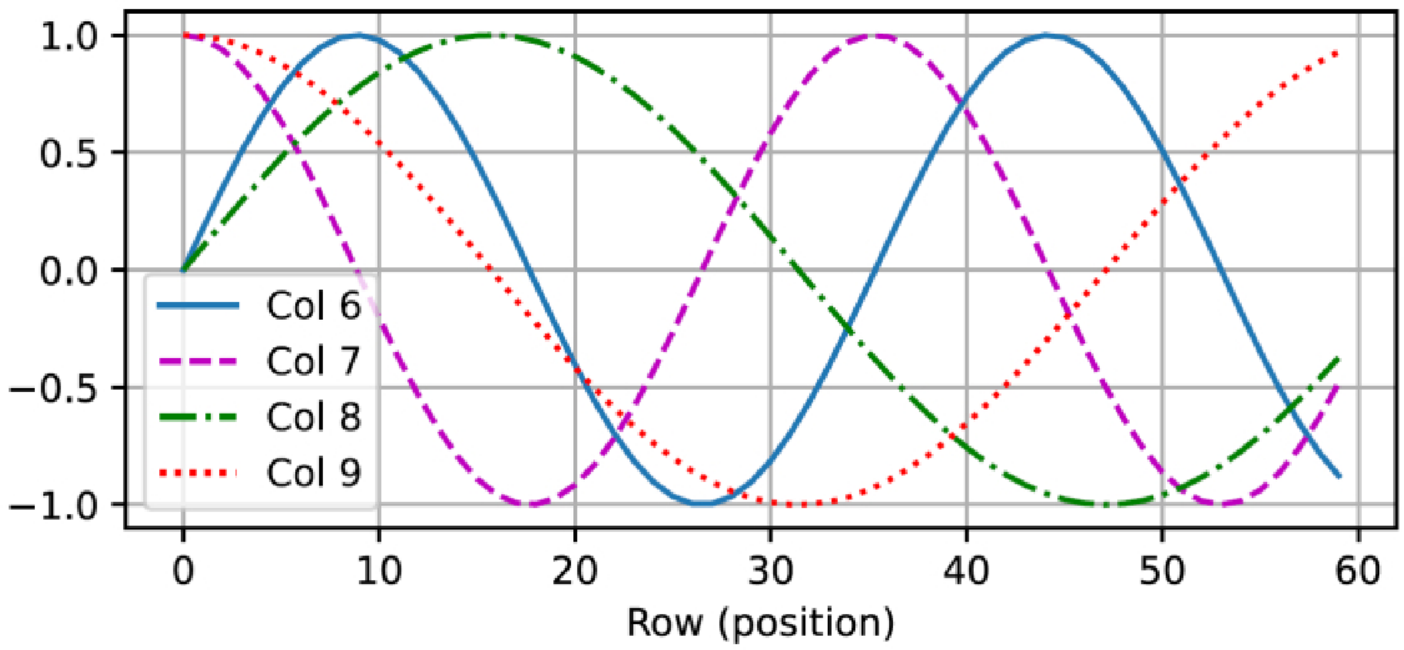

Figure 8: Positional encoding example.

5.6 Self-attention and positional encoding

- besides capturing absolute positional information, the above positional encoding also allows a model to easily learn to attend by relative positions
- this is because, for any fixed position offset δ, the positional encoding at position $i+\delta$ can be represented by a linear projection of that at position i
- this projection can be explained mathematically; denoting $\omega_j = 1/10000^{2j/d}$, any pair of $(p_{i,2j}, p_{i,2j+1})$ in (9) can be linearly projected to $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$, for any fixed offset δ:

$$\begin{aligned}
\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} &= \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} \sin((i+\delta)\omega_j) \\ \cos((i+\delta)\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned}$$

where the 2×2 projection matrix does not depend on any position index i

5.7 Transformer

- we have compared CNNs, RNNs, and self-attention in Section 5.6
- notably, self-attention enjoys both parallel computation and the shortest maximum path length
- therefore, naturally, it is appealing to design deep architectures by using self-attention
- unlike earlier self-attention models that still rely on RNNs for input representations, the *transformer* model is solely based on attention mechanisms, without any convolutional or recurrent layer

5.7 Transformer

- though originally proposed for sequence to sequence learning on text data, transformers have been pervasive in a wide range of modern deep learning applications, such as in areas of language, vision, speech, and reinforcement learning
- as an instance of the encoder-decoder architecture, the overall architecture of the transformer is presented in Figure 9
- as we can see, the transformer is composed of an encoder and a decoder
- different from Bahdanau attention for sequence to sequence learning in Figure 5, the input (source) and output (target) sequence embeddings are added with positional encoding, before being fed into the encoder and the decoder that stack modules based on self-attention

5.7 Transformer

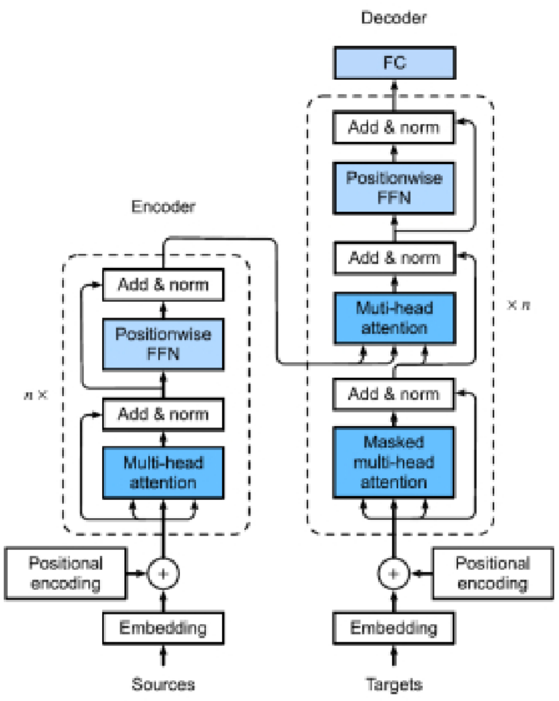

Figure 9: The transformer architecture.

5.7 Transformer

- now, we provide an overview of the transformer architecture in Figure 9
- 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 position-wise feed-forward network
- specifically, in the encoder self-attention, queries, keys, and values are all from the outputs of the previous encoder layer

5.7 Transformer

- inspired by the ResNet design in Section 3.10, a residual connection is employed around both sublayers
- in the transformer, for any input $\boldsymbol{x} \in \mathbb{R}^d$, at any position of the sequence, we require that sublayer$(\boldsymbol{x}) \in \mathbb{R}^d$, so that the residual connection $\boldsymbol{x} +$ sublayer$(\boldsymbol{x}) \in \mathbb{R}^d$ is feasible
- this addition from the residual connection is immediately followed by *layer normalization*
- as a result, the transformer encoder outputs a d-dimensional vector representation for each position of the input sequence

5.7 Transformer

- 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
- in the 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

5.7 Transformer

- the position-wise feed-forward network transforms the representation at all the sequence positions using the same MLP; this is why we call it *position-wise*
- now, let us focus on the “add & norm” component in Figure 9
- as we described at the beginning of this section, this is a residual connection immediately followed by layer normalization; both are key to effective deep architectures
- in Section 3.9, we explained how batch normalization recenters and rescales across the examples within a mini-batch
- layer normalization is the same as batch normalization, except that the former normalizes across the feature dimension
- 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 variable-length sequences

5.7 Transformer

- when training sequence-to-sequence models, tokens at all the positions (time steps) of the output sequence are known
- however, 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 that any query should only attend to all positions in the decoder up to the query position
- although the transformer architecture was originally proposed for sequence-to-sequence learning, either the transformer encoder or the transformer decoder is often individually used for different deep learning tasks

Thank you!