Applied Deep Learning

Chapter 7: Sequence-to-Sequence Models

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

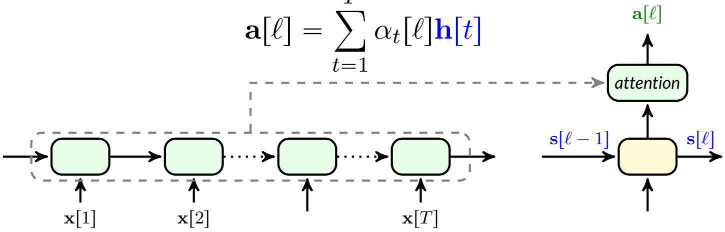
Fall 2025

Attention is All You Need!

This was the title of the paper introduced transformers[1]

The attention layer already computes a sort of memory

$$\mathbf{a}[\ell] = \sum_{t=1}^{T} \alpha_t[\ell]\mathbf{h}[t]$$



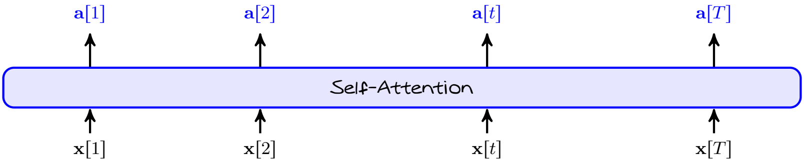
+ *Can't we then drop time connections?*
– This is the idea of self-attention

[1] Check the paper at this link

Self-Attention: *Basic Architecture*

In self-attention, *we*

1. *start with the complete sequence*
2. *capture all temporal correlation via attention mechanism*
3. *return a sequence which better represents time (positional) dependencies*



Typically, we have same time length for input and output

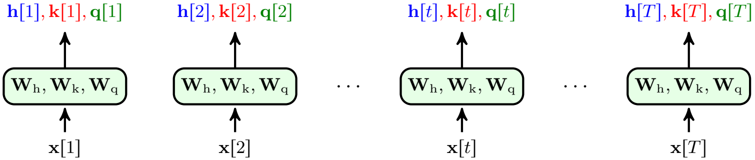
Self-Attention: *Basic Architecture*

+ *How can we do this via attention?*
– Input entries apply attention mechanism on the input sequence itself
+ *But, how can we do this?*
– Let's do it step by step

First, let's generate hidden features, keys and queries

```
Self-Attention():
```
1: **for** time $t = 1, \ldots, T$ **do**
2: Generate a hidden feature (value) $\mathbf{h}[t] = f(\mathbf{W}_{\mathrm{h}}\mathbf{x}[t])$
3: Generate a key $\mathbf{k}[t] = f(\mathbf{W}_{\mathrm{k}}\mathbf{x}[t])$
4: Generate a query $\mathbf{q}[t] = f(\mathbf{W}_{\mathrm{q}}\mathbf{x}[t])$
5: **end for**

Self-Attention: *Basic Architecture*



Self-Attention: *Basic Architecture*

Next we generate score for every time step

Self-Attention():

for time $t = 1, \ldots, T$ **do**

  Generate a hidden feature (value) $\mathbf{h}[t] = f\left(\mathbf{W}_{\mathrm{h}}\mathbf{x}[t]\right)$

  Generate a key $\mathbf{k}[t] = f\left(\mathbf{W}_{\mathrm{k}}\mathbf{x}[t]\right)$

  Generate a query $\mathbf{q}[t] = f\left(\mathbf{W}_{\mathrm{q}}\mathbf{x}[t]\right)$

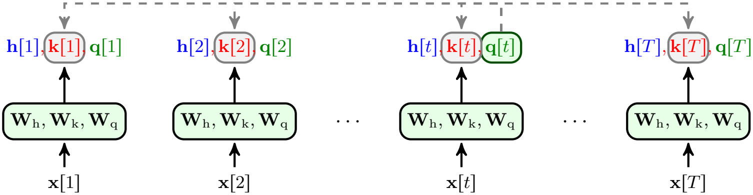
  for time $j = 1, \ldots, T$ **do**

    Compare query of time t with key j by computing $\xi_j[t] = \mathbf{q}^{\mathsf{T}}[t]\mathbf{k}[j]$

  end for

end for

Self-Attention: *Basic Architecture*



Self-Attention: *Basic Architecture*

Finally, we find the weights and use them to compute attention features

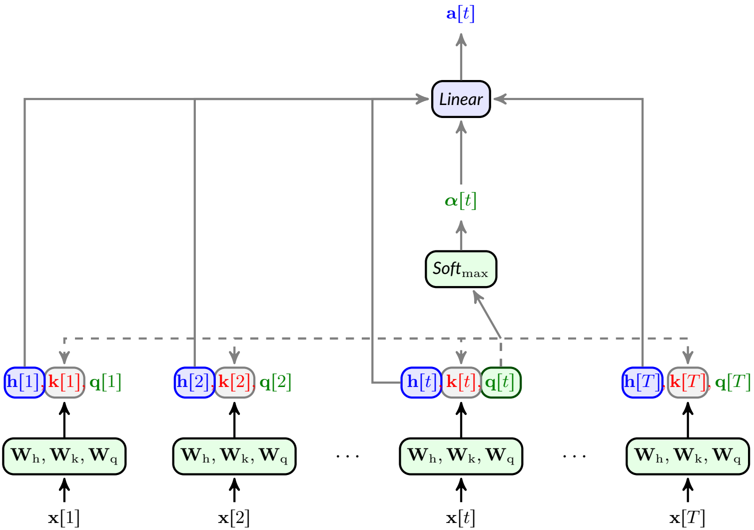
Self-Attention():

1: **for** time $t = 1, \ldots, T$ **do**
2: Generate a hidden feature (value) $\mathbf{h}[t] = f(\mathbf{W}_{\mathrm{h}} \mathbf{x}[t])$
3: Generate a key $\mathbf{k}[t] = f(\mathbf{W}_{\mathrm{k}} \mathbf{x}[t])$
4: Generate a query $\mathbf{q}[t] = f(\mathbf{W}_{\mathrm{q}} \mathbf{x}[t])$
5: **for** time $j = 1, \ldots, T$ **do**
6: Compare query of time t with key j by computing $\xi_j[t] = \mathbf{q}^{\mathsf{T}}[t]\mathbf{k}[j]$
7: **end for**
8: Pass score $\boldsymbol{\xi}[t]$ through *Softmax* to find attention weights $\boldsymbol{\alpha}[t]$
9: Compute attention features from values and attention weights

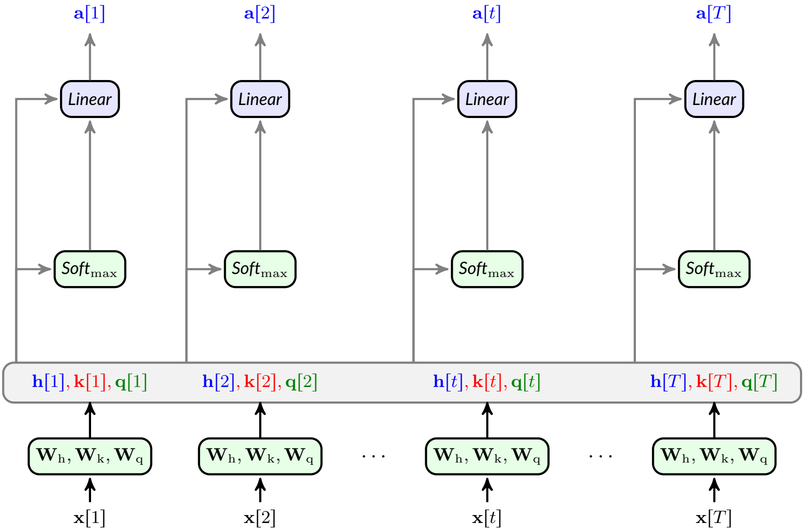
$$\mathbf{a}[t] = \sum_{j=1}^{T} \alpha_j[t] \mathbf{h}[j]$$

10: **end for**

Self-Attention: *Basic Architecture*



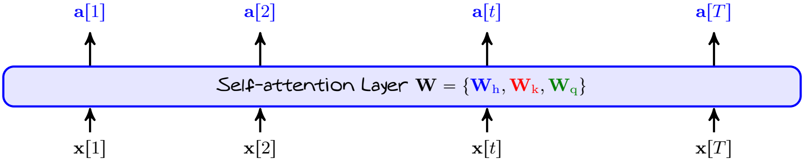
Self-Attention: *Complete Architecture*



Self-Attention as a Layer

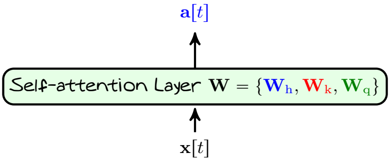
We can look at the whole process as a single layer

1. *it takes a sequence of time as input*
2. *it processes it via attention mechanism*
3. *it returns an output time sequence*



Self-Attention as a Layer

Note that *all the weights are fixed through time: think of it as in RNNs*



However, we should pay attention that

> *Self-attention returns the complete output sequence only after going through the complete input sequence: No sequential order!*

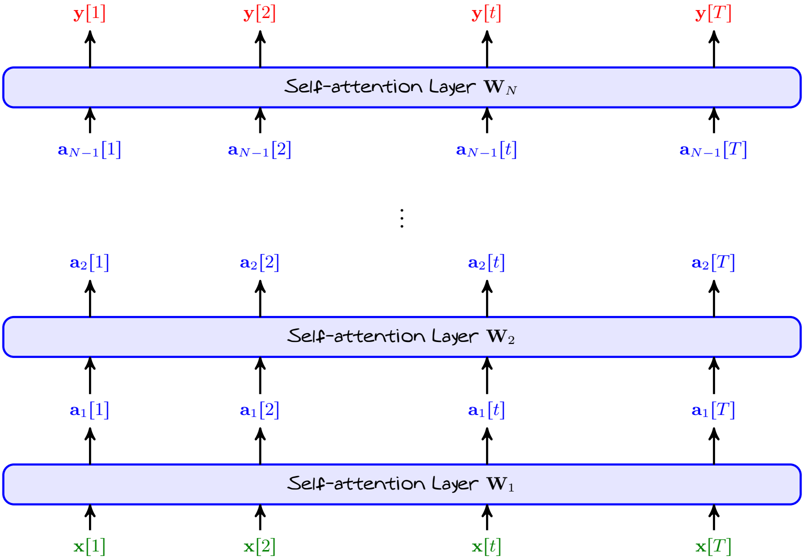
Processing Sequences Only with Attention

At *this point*, people started to think: *what if*

> *we process the time sequence via multiple layers of self-attention*

- *Start with input $\mathbf{x}[t]$ and compute an attention sequence $\mathbf{a}_1[t]$*
  ↳ $\mathbf{a}_1[t]$ *has captured temporal features of* $\mathbf{x}[t]$
- *From attention $\mathbf{a}_1[t]$, compute a new attention sequence $\mathbf{a}_2[t]$*
  ↳ $\mathbf{a}_2[t]$ *has captured better temporal features of* $\mathbf{x}[t]$
- $\cdots \times N$
- *From last attention $\mathbf{a}_{N-1}[t]$, we compute an output sequence $\mathbf{y}[t]$*
  ↳ $\mathbf{y}[t]$ *has captured almost all temporal features of* $\mathbf{x}[t]$

Processing Sequences Only with Attention



Processing Sequences Only with Attention

Intuitive Observation

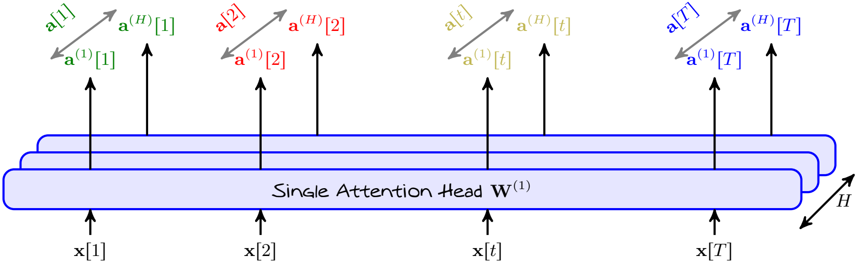
Self-attention does the same thing as convolution

- *It has a fixed set of weights to compute values, keys, and queries*
  - ↳ *Convolutional layers have fixed filters (kernels)*
- *It goes through the entries sequence with these weights using attention*
  - ↳ *Convolutional layers apply convolution with the same filter*
- *It returns a output sequence that captures temporal correlation*
  - ↳ *Convolutional layers return maps that capture spatial correlation*

\+ *But, we compute multiple maps via multiple filters in convolutional layers!*

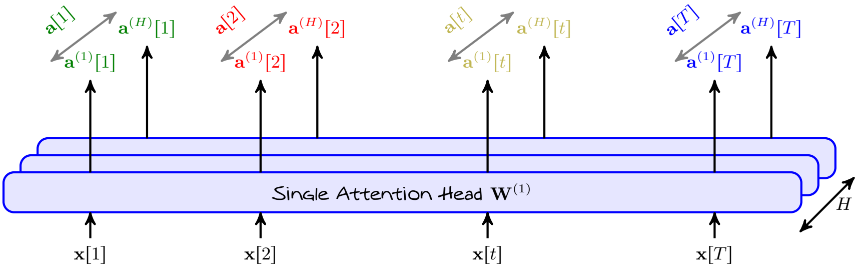
– Well, we could do the same here!

Self-Attention: *Multi-headed Architecture*



- *We can compute multiple self-attention layers in parallel*
  - ↳ *A layer in this context is called a head*
- *Each head computes a separate attention feature*
  - ↳ *Head h computes the attention feature $\mathbf{a}^{(h)}[t] \in \mathbb{R}^{D_0}$*
- *The concatenation of these features is the final attention feature*
  - ↳ *Attention feature at time t is $\mathbf{a}[t] = \left[\mathbf{a}^{(1)}[t], \ldots, \mathbf{a}^{(H)}[t]\right] \in \mathbb{R}^{HD_0}$*

Self-Attention: *Multi-headed Architecture*



Typical Practice

In practice, we often choose D_0 and H such that dimensionality is preserved: say every $\mathbf{x}[t] \in \mathbb{R}^N$

- *We set $D_0 = N/H$ for an H-head self-attention layer*
- *The concatenated attention feature is then of length $HD_0 = N$*

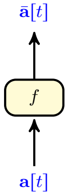
The key reason is that we often use this layer with skip connection

Self-Attention: *Adding Nonlinearity*

It turns out that although self-attention captures temporal correlation

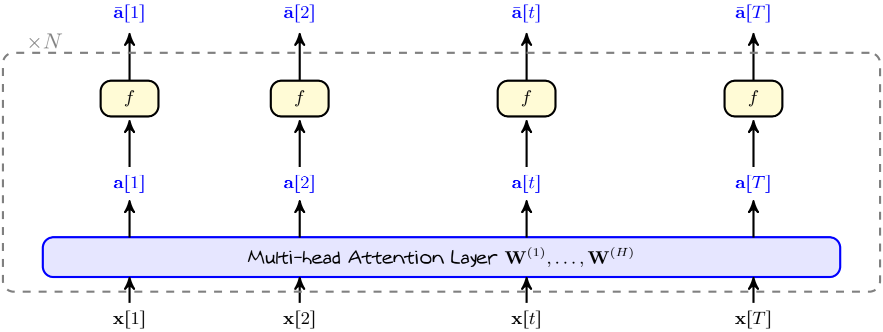
it is not complex enough to capture content features accurately

+ *Can't we do it by having more complex layers!*
- *Sure! We often process the attention features with a nonlinear layer*



This layer further captures content features

Self-Attention: *A Unit*



Self-Attention Unit: *Generic Form*

A self-attention unit consists of a multi-head attention layer followed by an activated layer. These layers are all fixed through time

A deep self-attention architecture is made by cascading self-attention units

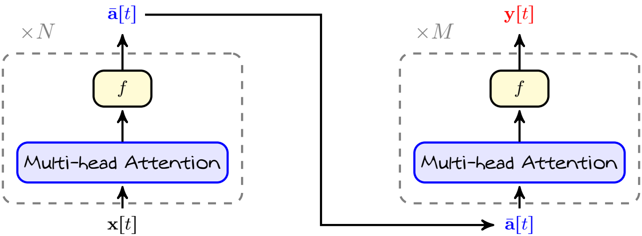
Transformer: *Encoding and Decoding via Self-Attention*

Experiments showed that

self-attention can better capture temporal and content features

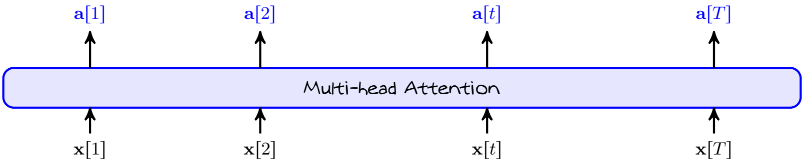
---

+ *Can't we replace them with RNNs in an encoder-decoder architecture?*
- Sure! This is indeed a basic transformer!



Transformer: *Encoding and Decoding via Self-Attention*

+ *Does this idea work that easily?!*
– Well! Pretty much Yes, after some basic modifications

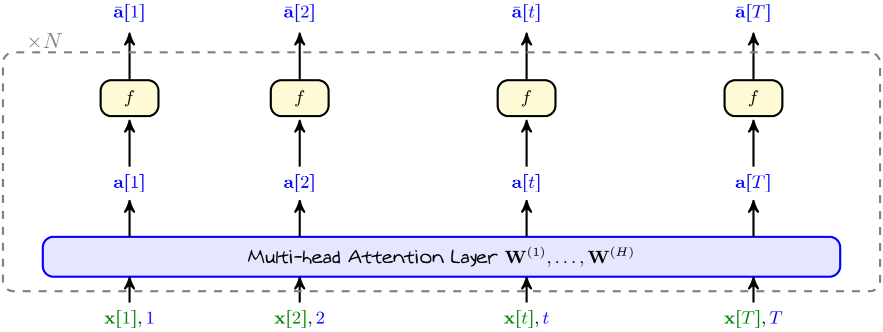


There are two challenges in using self-attention for encoding and decoding

- *For encoding: changing time order at input does not impact the output*
  ↳ *Self-attention captures temporal correlation, but not sequential order!*
- *For decoding: we should generate the entire output at once*
  ↳ *We often want to generate $\mathbf{y}[t]$ depending on previous outputs $\mathbf{y}[1:t-1]$*

Encoding via Self-Attention: *Positional Encoding*

A simple remedy is to include time index in the input!

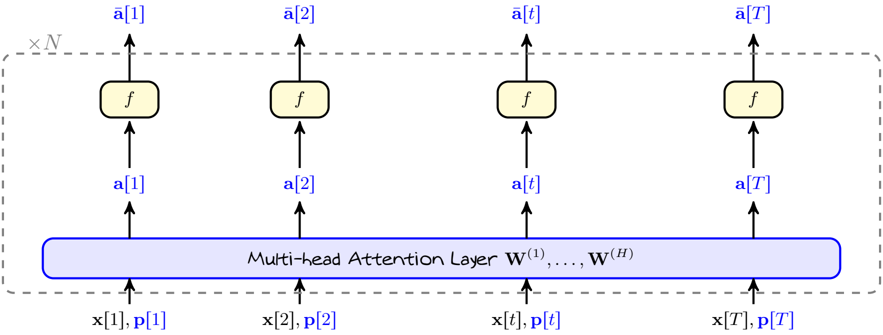


We replace the input with (input, time), i.e.,

$$\mathbf{x}[t] \leftarrow (\mathbf{x}[t], t)$$

Now, time permutation changes output $\rightsquigarrow$ *output depends on sequential order*

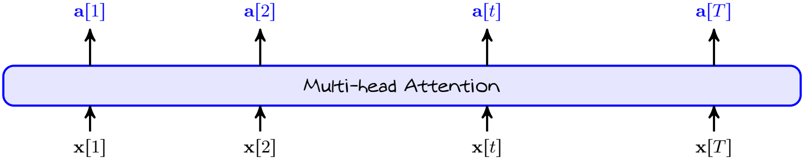
Positional Encoding



Positional Encoding

To use self-attention at encoder we concatenate input with a time-dependent (positional) feature $\mathbf{p}[t]$ that can be potentially learned from data itself

Decoding via Self-Attention: *Masked Decoding*



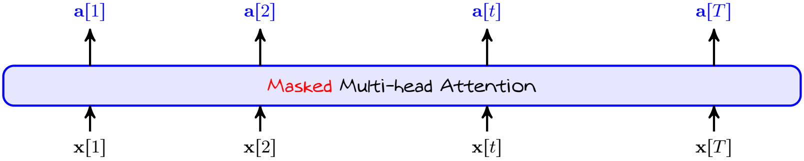
Recall that attention at time t is computed as

$$\mathbf{a}[t] = \sum_{j=1}^{T} \alpha_j[t]\mathbf{h}[j]$$

which depends on the future data through $\mathbf{h}[j]$ for $j = t+1, \ldots, T$

We could simply ignore them by masking $\alpha_j[t]$ for $j = t+1, \ldots, T$

Decoding via Self-Attention: *Masked Decoding*

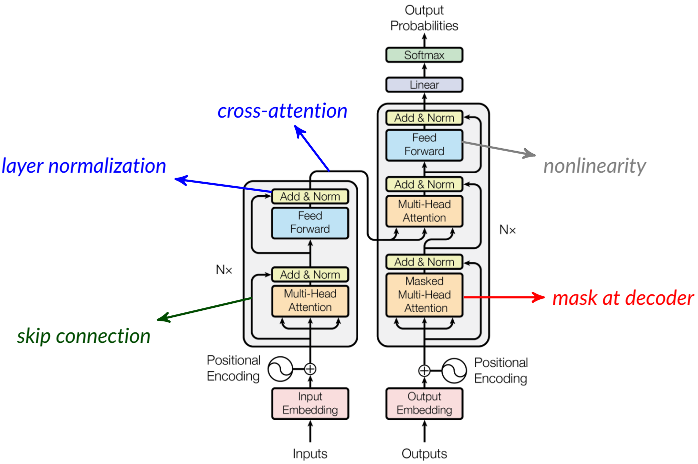


We replace $\alpha_j[t] \leftarrow 0$ *for* $j = t+1, \dots, T$ *and compute*

$$\mathbf{a}[t] = \sum_{j=1}^{T} \alpha_j[t]\mathbf{h}[j] = \sum_{j=1}^{t} \alpha_j[t]\mathbf{h}[j]$$

A Classic Transformer

A classic transformer looks like this[2]



[2] *The diagram is from the paper Attention is All You Need!*

Why Transformers?

+ *Transformers seem to be more challenging to implement! Why should we use them?*
– That's right! But they come with some benefits!

In general transformers

- *process sequences in parallel* $\rightsquigarrow$ *room for parallel computing*
- *preserve much better long-term connections*
- *can be deepened much more straightforward*

But, it's true that

- *they pose more complexity* $\rightsquigarrow$ *we can handle it nowadays*

Final Notes

Transformer is a core element of many modern learning models

- *GPT stands for Generative Pre-trained Transformer*
- *They are the better choice for complicated Seq2Seq problems*
- *Pre-implemented transformer is available in PyTorch*

```
torch.nn.Transformer(d_model, nhead, num_encoder_layers,
                     num_decoder_layers, ...)
```

If you are interested to know more about transformers

↳ *Consider taking NLP course next fall semester*

↳ *We also revisit it in the Generative Models course this summer*

↳ *Take a look at this nice online resource*