Recap: Attention

- Encoder hidden states: $h_1^{enc}, \ldots, h_n^{enc}$
  
  \( n: \# \text{ of words in source sentence} \)

- Decoder hidden state at time $t$: $h_t^{dec}$

- Attention scores:

  \[
  e^t = [g(h_1^{enc}, h_t^{dec}), \ldots, g(h_n^{enc}, h_t^{dec})] \in \mathbb{R}^n
  \]

- Attention distribution:

  \[
  \alpha^t = \text{softmax}(e^t) \in \mathbb{R}^n
  \]

- Weighted sum of encoder hidden states:

  \[
  a_t = \sum_{i=1}^{n} \alpha_i^{t} h_i^{enc} \in \mathbb{R}^h
  \]

Combine $a_t$ and $h_t^{dec}$ to predict next word

Note that $h_1^{enc}, \ldots, h_n^{enc}$ and $h_t^{dec}$ are hidden states from encoder and decoder RNNs.
Recap: Attention

- Attention addresses the “bottleneck” or fixed representation problem
- Attention learns the notion of **alignment**
  
  “Which source words are more relevant to the current target word?”

Attention as a soft, averaging lookup table

We can think of **attention** as performing fuzzy lookup a in **key-value store**

**Lookup table**: a table of keys that map to values. The query matches one of the keys, returning its value.

**Attention**: The query matches to all keys softly to a weight between 0 and 1. The keys’ values are multiplied by the weights and summed.

(So far, we assume key = value)
Understanding attention

Do you understand attention now?

(A) I understand the concept of attention and what it is for
(B) I understand the concept + its mathematical formulations
(C) I am still struggling
Attention Is All You Need

Transformers

(Vaswani et al., 2017)
Transformer encoder-decoder

- Transformer encoder + Transformer decoder
- First designed and experimented on NMT
- Can be viewed as a replacement for seq2seq + attention based on RNNs
Transformer encoder-decoder

- Transformer encoder = a stack of **encoder layers**
- Transformer decoder = a stack of **decoder layers**

**Transformer encoder:**  BERT, RoBERTa, ELECTRA

**Transformer decoder:**  GPT-3, ChatGPT, Palm

**Transformer encoder-decoder:**  T5, BART

- Key innovation: **multi-head, self-attention**
- Transformers don’t have any recurrence structures!

\[ h_t = f(h_{t-1}, x_t) \in \mathbb{R}^h \]
Issues with recurrent NNs

- Longer sequences can lead to vanishing gradients \( \implies \) It is hard to capture long-distance information

- RNNs lack of parallelizability
  - Forward and backward passes have \( O(\text{sequence length}) \) unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - Inhibits training on very large datasets

RNNs / LSTMs → seq2seq → seq2seq + attention → attention only = Transformers!
Transformers have become a new building block to replace RNNs
Transformers: roadmap

- From attention to self-attention
- From self-attention to multi-head self-attention
- Feedforward layers
- Positional encoding
- Residual connections + layer normalization
- Transformer encoder vs Transformer decoder

Reminder: we will ask you to implement Transformer encoder-decoder in A4!
Attention in a general form

- Assume that we have a set of values $v_1, \ldots, v_n \in \mathbb{R}^{d_v}$ and a query vector $q \in \mathbb{R}^{d_q}$

- Attention always involves the following steps:
  - Computing the attention scores $e = g(q, v_i) \in \mathbb{R}^n$
  - Taking softmax to get attention distribution $\alpha$:
    $$\alpha = \text{softmax}(e) \in \mathbb{R}^n$$
  - Using attention distribution to take weighted sum of values:
    $$a = \sum_{i=1}^{n} \alpha_i v_i \in \mathbb{R}^{d_v}$$
Attention in a general form

- A more general form: use a set of keys and values \((k_1, v_1), \ldots, (k_n, v_n)\), \(k_i \in \mathbb{R}^{d_k}, v_i \in \mathbb{R}^{d_v}\), keys are used to compute the attention scores and values are used to compute the output vector.

- Attention always involves the following steps:
  - Computing the attention scores \(e = g(q, k_i) \in \mathbb{R}^n\)
  - Taking softmax to get attention distribution \(\alpha\):
    \[
    \alpha = \text{softmax}(e) \in \mathbb{R}^n
    \]
  - Using attention distribution to take weighted sum of values:
    \[
    a = \sum_{i=1}^{n} \alpha_i v_i \in \mathbb{R}^{d_v}
    \]
Self-attention

- In NMT, `query = decoder hidden state, keys = values = encoder hidden states`
- Self-attention = attention from the sequence to itself
- Self-attention: let’s use each word in a sequence as the `query`, and all the other words in the sequence as `keys` and `values`.

https://jalammar.github.io/illustrated-transformer/
Self-attention

A self-attention layer maps a sequence of input vectors $x_1, \ldots, x_n \in \mathbb{R}^{d_1}$ to a sequence of $n$ vectors: $h_1, \ldots, h_n \in \mathbb{R}^{d_2}$

- The same abstraction as RNNs - used as a drop-in replacement for an RNN layer

\[ h_t = g(W h_{t-1} + U x_t + b) \in \mathbb{R}^h \]

Self-attention:

\[ q_i = W^{(q)} x_i, \quad k_i = W^{(k)} x_i, \quad v_i = W^{(v)} x_i, \]
\[ h_i = W^{(o)} \sum_{j=1}^n \left( \frac{\exp(q_i \cdot k_j / \sqrt{d})}{\sum_{j'=1}^n \exp(q_i \cdot k_{j'}/\sqrt{d})} v_j \right) \]

where $W^{(q)}, W^{(k)}, W^{(v)}, W^{(o)} \in \mathbb{R}^{d \times d}$. 
Self-attention

Step #1: Transform each input vector into three vectors: query, key, and value vectors

\[
q_i = x_i W^Q \in \mathbb{R}^{d_q} \quad k_i = x_i W^K \in \mathbb{R}^{d_k} \quad v_i = x_i W^V \in \mathbb{R}^{d_v}
\]

\[
W^Q \in \mathbb{R}^{d_1 \times d_q} \quad W^K \in \mathbb{R}^{d_1 \times d_k} \quad W^V \in \mathbb{R}^{d_1 \times d_v}
\]

Note that we use row vectors here; it is also common to write

\[
q_i = W^Q x_i \in \mathbb{R}^{d_q}
\]

for \( x_i \) = a column vector

https://jalammar.github.io/illustrated-transformer/
Self-attention

Step #2: Compute pairwise similarities between keys and queries; normalize with softmax
For each $q_i$, compute attention scores and attention distribution:

$$\alpha_{i,j} = \text{softmax}(\frac{q_i \cdot k_j}{\sqrt{d_k}})$$

aka. “scaled dot product”
It must be $d_q = d_k$ in this case

Q. Why scaled dot product?
To avoid the dot product to become too large for larger $d_k$; scaling the dot product by $\frac{1}{\sqrt{d_k}}$ is easier for optimization

https://jalammar.github.io/illustrated-transformer/
Self-attention

Step #3: Compute output for each input as weighted sum of values

\[ h_i = \sum_{j=1}^{n} \alpha_{i,j} v_j \in \mathbb{R}^{d_v} \]

\[ (d_v = d_2) \]
What would be the output vector for the word “Thinking” approximately?

(a) $0.5v_1 + 0.5v_2$

(b) $0.54v_1 + 0.46v_2$

(c) $0.88v_1 + 0.12v_2$

(d) $0.12v_1 + 0.88v_2$

(c) is correct.
Self-attention: matrix notations

\[ X \in \mathbb{R}^{n \times d_1} \quad \text{(n = input length)} \]

\[ Q = XW^Q \quad K = XW^K \quad V = XW^V \]

\[ W^Q \in \mathbb{R}^{d_1 \times d_q}, W^K \in \mathbb{R}^{d_1 \times d_k}, W^V \in \mathbb{R}^{d_1 \times d_v} \]

Attention(\(Q, K, V\)) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V

Q: What is this softmax operation?

https://jalammar.github.io/illustrated-transformer/
Multi-head attention

“The Beast with Many Heads”

• It is better to use multiple attention functions instead of one!
  • Each attention function (“head”) can focus on different positions.

https://jalammar.github.io/illustrated-transformer/
Multi-head attention

“The Beast with Many Heads”

Finally, we just concatenate all the heads and apply an output projection matrix.

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O
\]

\[
\text{head}_i = \text{Attention}(XW^Q_i, XW^K_i, XW^V_i)
\]

- In practice, we use a \textit{reduced} dimension for each head.

\[
W^Q_i \in \mathbb{R}^{d_1 \times d_q}, W^K_i \in \mathbb{R}^{d_1 \times d_k}, W^V_i \in \mathbb{R}^{d_1 \times d_v}
\]

\[
d_q = d_k = d_v = d/m \quad d = \text{hidden size}, \quad m = \text{# of heads}
\]

\[
W^O \in \mathbb{R}^{d \times d_2} \quad \text{If we stack multiple layers, usually } d_1 = d_2 = d
\]

- The total computational cost is similar to that of single-head attention with full dimensionality.

https://jalammar.github.io/illustrated-transformer/
What does multi-head attention learn?

https://github.com/jessevig/bertviz
Missing piece: positional encoding

• Unlike RNNs, self-attention doesn’t build in order information, we need to encode the order of the sentence in our keys, queries, and values.

• Solution: Add “position encoding” to the input embeddings: \( p_i \in \mathbb{R}^d \) for \( i = 1, 2, \ldots, n \)

\[
x_i \leftarrow x_i + p_i
\]

• **Sinusoidal position encoding**: sine and cosine functions of different frequencies:

\[
p_i = \begin{pmatrix}
sin(i/10000^{2^{1/d}}) \\
cos(i/10000^{2^{1/d}}) \\
\vdots \\
sin(i/10000^{2^{d/2}}) \\
cos(i/10000^{2^{d/2}})
\end{pmatrix}
\]

• **Pros**: Periodicity + can extrapolate to longer sequences

• **Cons**: Not learnable
Missing piece: positional encoding

- **Learned absolute position encoding:** let all $p_i$ be learnable parameters
  - $P \in \mathbb{R}^{d \times L}$ for $L = \text{max sequence length}$

- **Pros:** each position gets to be learned to fit the data
- **Cons:** can’t extrapolate to indices outside of max sequence length $L$
- Most systems use this!

---

**RoFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING**

---

**Self-Attention with Relative Position Representations**

<table>
<thead>
<tr>
<th>Peter Shaw</th>
<th>Jakob Uszkoreit</th>
<th>Ashish Vaswani</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Google Brain</td>
<td>Google Brain</td>
</tr>
<tr>
<td><a href="mailto:petershaw@google.com">petershaw@google.com</a></td>
<td><a href="mailto:usz@google.com">usz@google.com</a></td>
<td><a href="mailto:avaswani@google.com">avaswani@google.com</a></td>
</tr>
</tbody>
</table>

---

**Jianlin Su**
Zhuiyi Technology Co., Ltd.
Shenzhen
bojonesu@wezhuiyi.com

**Yu Lu**
Zhuiyi Technology Co., Ltd.
Shenzhen
julianlu@wezhuiyi.com

**Ahmed Murtadha**
Zhuiyi Technology Co., Ltd.
Shenzhen
nengjinyi@wezhuiyi.com

**Bo Wen**
Zhuiyi Technology Co., Ltd.
Shenzhen
brucewen@wezhuiyi.com

**Shengfeng Pan**
Zhuiyi Technology Co., Ltd.
Shenzhen
nickpan@wezhuiyi.com

**Yunfeng Liu**
Zhuiyi Technology Co., Ltd.
Shenzhen
glenliu@wezhuiyi.com
Adding nonlinearities

- There are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors.

- Simple fix: add a feed-forward network to post-process each output vector.

\[
\text{FFN}(x_i) = \text{ReLU}(x_i W_1 + b_1) W_2 + b_2
\]

\[
W_1 \in \mathbb{R}^{d \times d_{ff}}, \quad b_1 \in \mathbb{R}^{d_{ff}}
\]

\[
W_2 \in \mathbb{R}^{d_{ff} \times d}, \quad b_2 \in \mathbb{R}^{d}
\]

In practice, they use \(d_{ff} = 4d\).
Which of the following statements is correct?

(a) Transformers have less operations compared to LSTMs
(b) Transformers are easier to parallelize compared to LSTMs
(c) Transformers have less parameters compared to LSTMs
(d) Transformers are better at capturing positional information than LSTMs

(b) is correct.
Transformer encoder: let’s put things together

From the bottom to the top:
- Input embedding
- Positional encoding
- A stack of Transformer encoder layers

Transformer encoder is a stack of \( N \) layers, which consists of two sub-layers:
- Multi-head attention layer
- Feed-forward layer

\[
x_1, \ldots, x_n \in \mathbb{R}^{d_1} \quad \rightarrow \quad h_1, \ldots, h_n \in \mathbb{R}^{d_2}
\]
Residual connection & layer normalization

Add & Norm: \( \text{LayerNorm}(x + \text{Sublayer}(x)) \)

Residual connections (He et al., 2016)

Instead of \( X^{(i)} = \text{Layer}(X^{(i-1)}) \) (\( i \) represents the layer)

We let \( X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)}) \), so we only need to learn “the residual” from the previous layer

Gradient through the residual connection is 1 - good for propagating information through layers
Residual connection & layer normalization

Add & Norm:  \( \text{LayerNorm}(x + \text{Sublayer}(x)) \)

**Layer normalization** (Ba et al., 2016) helps train model faster

Idea: normalize the hidden vector values to unit mean and stand deviation within each layer

\[
y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \ast \gamma + \beta \quad \gamma, \beta \in \mathbb{R}^d \text{ are learnable parameters}
\]
Transformer decoder

From the bottom to the top:
- Output embedding
- Positional encoding
- A stack of Transformer decoder layers
- Linear + softmax

Transformer decoder is a stack of $N$ layers, which consists of three sub-layers:
- Masked multi-head attention
- Multi-head cross-attention
- Feed-forward layer
- (W/ Add & Norm between sub-layers)