L14: Self-Attention and Transformers

COS 484
Natural Language Processing

Spring 2022

(Some slides adapted from Stanford CS224N)
**Issues with RNNs**

- Sequential nature $\implies$ difficult to parallelize

$$h_t = f(h_{t-1}, x_t) \in \mathbb{R}^h$$

**LSTMs**

- **Input gate (how much to write):**
  $$i_t = \sigma(W^i h_{t-1} + U^i x_t + b^i) \in \mathbb{R}^h$$

- **Forget gate (how much to erase):**
  $$f_t = \sigma(W^f h_{t-1} + U^f x_t + b^f) \in \mathbb{R}^h$$

- **Output gate (how much to reveal):**
  $$o_t = \sigma(W^o h_{t-1} + U^o x_t + b^o) \in \mathbb{R}^h$$

- **New memory cell (what to write):**
  $$g_t = \tanh(W^g h_{t-1} + U^g x_t + b^g) \in \mathbb{R}^h$$

- **Final memory cell:**
  $$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

- **Final hidden cell:**
  $$h_t = o_t \odot \tanh(c_t)$$
Issues with RNNs

- Longer sequences can lead to vanishing gradients $\Rightarrow$ It is hard to capture long-distance information

Attention is the key to solving the problem!
This lecture

- Do we really need RNNs to model the arbitrary context?
- Maybe attention is all you need!

---

Attention Is All You Need

---

Ashish Vaswani*
Google Brain
avaswani@gmail.com

Noam Shazeer*
Google Brain
noam@gmail.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Liion Jones*
Google Research
liion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@gmail.com

Illia Polosukhin* †
illia.polosukhin@gmail.com

Vaswani et al., 2017: Attention Is All You Need
Transformer

- Consists of an encoder and a decoder
- Originally proposed for neural machine translation and later adapted for almost all the NLP tasks
  - For example, BERT only uses the encoder of the Transformer architecture (next lecture)
- Both encoder and decoder consist of $N$ layers
  - Each encoder layer has two sub-layers
  - Each decoder layer has three sublayers
  - Key innovation: multi-head self-attention

Vaswani et al., 2017: Attention Is All You Need
Transformers: roadmap

- From attention to self-attention
- From self-attention to multi-head self-attention
- Transformer encoder
- Transformer decoder
- Putting the pieces together
Recap: Attention in NMT

- Encoder hidden states: $h_1^{enc}, \ldots, h_n^{enc}$
- Decoder hidden state at time $t$: $h_t^{dec}$
- $g(\cdot)$ takes dot product in the simplest form!

First, get attention scores for this time step of decoder (we’ll define $g$ soon):

$$e^t = [g(h_1^{enc}, h_1^{dec}), \ldots, g(h_n^{enc}, h_t^{dec})]$$

Obtain the attention distribution using softmax:

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

Compute weighted sum of encoder hidden states:

$$a_t = \sum_{i=1}^{n} \alpha_t^i h_i^{enc} \in \mathbb{R}^h$$
Attention is a general deep learning technique

- Given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query.
  - We sometimes say that the query attends to the values.
  - In the NMT case, each decoder hidden state (query) attends to all the encoder hidden states (values).

- Intuition:
  - The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on.
  - Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).
Attention is a general deep learning technique

• 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 \( e = g(v_i, q) \in \mathbb{R}^n \)
  • Taking softmax to get attention distribution \( \alpha \):
    \[
    \alpha = \text{softmax}(e) \in \mathbb{R}^n
    \]
  • Using attention distribution to take weighted sum of values:
    \[
    a = \sum_{i=1}^{n} \alpha_i v_i \in \mathbb{R}^{d_v}
    \]

• 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 is a general deep learning technique

- Assume that we have a set of key-value pairs \((k_1, v_1), \ldots, (k_n, v_n)\), \(k_i \in \mathbb{R}^{d_k}\), \(v_i \in \mathbb{R}^{d_v}\) and a query vector \(q \in \mathbb{R}^{d_q}\)

- Attention always involves the following steps:
  - Computing the attention scores \(e = g(k_i, q) \in \mathbb{R}^n\)
  - 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

- We saw attention from the decoder (query) to the encoder (values), now we think about attention within one single sequence.
  - 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**.

- The queries, keys and values are drawn from the same source.

Self-attention doesn’t know the order of the inputs - we will come back to this later!
A self-attention layer maps a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of $n$ vectors: $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^{d_2}$

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

First, construct a set of queries, keys and values:

$$
\mathbf{q}_i = W^Q \mathbf{x}_i, \quad \mathbf{k}_i = W^K \mathbf{x}_i, \quad \mathbf{v}_i = W^V \mathbf{x}_i
$$

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

Second, for each $\mathbf{q}_i$, compute attention scores and attention distribution:

$$
\alpha_{i,j} = \text{softmax}\left( \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \right) \quad \text{aka. “scaled dot product”}
$$

- It must be $d_q = d_k$ in this case

Finally, compute the weighted sum:

$$
\mathbf{y}_i = \sum_{j=1}^n \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v} \quad \quad \text{($d_v = d_2$)}
$$
Self-attention: illustration

http://jalammar.github.io/illustrated-transformer/
Self-attention: illustration

http://jalammar.github.io/illustrated-transformer/
Self-attention: matrix notations

\[ X \in \mathbb{R}^{n \times d_1} \]

Note: the notations we use here are following the original paper (= the transpose of the matrices in previous notations)

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

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

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

Q: What is this softmax operation?
Self-Attention

What is self-attention? Self-attention calculates a weighted average of feature representations with the weight proportional to a similarity score between pairs of representations. Formally, an input sequence of $n$ tokens of dimensions $d$, $X \in \mathbb{R}^{n \times d}$, is projected using three matrices $W_Q \in \mathbb{R}^{d \times d_q}$, $W_K \in \mathbb{R}^{d \times d_k}$, and $W_V \in \mathbb{R}^{d \times d_v}$ to extract feature representations $Q$, $K$, and $V$, referred to as query, key, and value respectively with $d_k = d_q$. The outputs $Q$, $K$, $V$ are computed as

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V. \quad (1)$$

So, self-attention can be written as,

$$S = D(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_q}} \right)V, \quad (2)$$

where softmax denotes a row-wise softmax normalization function. Thus, each element in $S$ depends on all other elements in the same row.
Multi-head attention

- It is better to use multiple attention functions instead of one!
  - Each attention function ("head") can focus on different positions.
- How to do this? Use different sets of query, key and value matrices!

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

• It is better to use multiple attention functions instead of one!

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

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\
\text{head}_i = \text{Attention}(XW^Q_i, XW^K_i, XW^V_i)
\]
Multi-head attention

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \\
\text{head}_i = \text{Attention}(X W^Q_i, X W^K_i, X W^V_i)
\]

- In practice, we use a reduced dimension for each head.

\[
W^Q_i \in \mathbb{R}^{d_1 \times d_q}, W^K_i \in \mathbb{R}^{d_1 \times d_k}, W^V_i \in \mathbb{R}^{d_1 \times d_v} \\
d_q = d_k = d_v = d/h \quad d = \text{hidden size}, h = \# \text{ of heads} \]

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

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

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

https://github.com/jessevig/bertviz
Unlike RNNs, self-attention doesn’t build in order information, we need to encode the order of the sentence.

Solution: Add “position encoding” to the input embeddings

\[ \mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{p}_i \]

Use sine and cosine functions of different frequencies (not learnable):

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

Later, people just use a learnable embedding \( \mathbf{p}_i \in \mathbb{R}^{d_1} \) for every unique position.
Adding nonlinearities

- There is 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

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

- In practice, they use \( d_{ff} = 4d \)
Transformers: pros and cons

- Easier to capture dependencies: we draw attention between every pair of words!

- Easier to parallelize: 
  \[ Q = XW^Q \quad K = XW^K \quad V = XW^V \]
  \[
  \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V
  \]

- Quadratic computation in self-attention:
  - Can become very slow when the sequence length is large

- Are these positional representations enough to capture positional information?
Transformer encoder

Each encoder layer has two sub-layers:

- A multi-head self-attention layer
- A feedforward layer

Add & Norm:

\[
\text{LayerNorm}(x + \text{Sublayer}(x))
\]

- Residual connection (He et al., 2016)
- Layer normalization (Ba et al., 2016)

In (Vaswani et al., 2017), \( N = 6 \)
Transformer decoder

Each decoder layer has three sub-layers:

- A masked multi-head attention layer
- A multi-head cross-attention layer
- A feedforward layer

Masked multi-head attention:
self-attention on the decoder states

However, you can’t see the future!

Multi-head cross-attention:
Decoder attends to encoder states

encoder: keys/values, decoder: queries

In (Vaswani et al., 2017), N = 6
Transformer decoder

http://jalammar.github.io/illustrated-transformer/
Masked multi-head attention

- Key point: you can’t see the future words for the decoder!

- Solution: for every $q_i$, only attend to $\{(k_j, v_j)\}; j \leq i$
Masked multi-head attention

\[ q_i = W^Q x_i, \quad k_i = W^K x_i, \quad v_i = W^V x_i \]

\[ \alpha_{i,j} = \text{softmax}\left( \frac{q_i \cdot k_j}{\sqrt{d_k}} \right) \]

**Efficient implementation:** compute attention as we normally do, mask out attention to future words by setting attention scores to \( -\infty \)

```
-dot = torch.bmm(queries, keys.transpose(1, 2))

-indices = torch.triu_indices(t, t, offset=1)
-dot[:, indices[0], indices[1]] = float('-inf')

-dot = F.softmax(dot, dim=2)
```

http://peterbloem.nl/blog/transformers
Multi-head cross-attention

\[ q_i = W^Q x_i, \quad k_i = W^K x_i, \quad v_i = W^V x_i \]

\[ \alpha_{i,j} = \text{softmax} \left( \frac{q_i \cdot k_j}{\sqrt{d_k}} \right) \]

\[ q_i = W^Q x_i, \quad k_j = W^K h_j, \quad v_j = W^V h_j \]

\[ \alpha_{i,j} = \text{softmax} \left( \frac{q_i \cdot k_j}{\sqrt{d_k}} \right) \]

Q: What is the size of \( \alpha \)?

\[ y_i = \sum_{j=1}^{m} \alpha_{i,j} v_j \]

\( \cdot \): hidden states from encoder

\( x_1, \ldots, x_n \): hidden states from decoder
Putting the pieces together
Putting the pieces together

Looking back at the whole model, zooming in on an Encoder block:

Transformer Encoder
- Residual + LayerNorm
- Feed-Forward
- Residual + LayerNorm
- Multi-Head Attention

Transformer Decoder
- [predictions!]
- [decoder attends to encoder states]

Word Embeddings + Position Representations

[Input sequence]

Word Embeddings + Position Representations

[Output sequence]
Putting the pieces together

Looking back at the whole model, zooming in on a Decoder block:

Transformer Encoder

Transformer Encoder

Word Embeddings + Position Representations

[output sequence]

Transformer Decoder

Residual + LayerNorm

Feed-Forward

Residual + LayerNorm

Multi-Head Cross-Attention

Residual + LayerNorm

Masked Multi-Head Self-Attention

[predictions!]

Word Embeddings + Position Representations

[input sequence]
Transformers: machine translation

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>Training Cost (FLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EN-DE</td>
<td>EN-FR</td>
</tr>
<tr>
<td></td>
<td>EN-DE</td>
<td>EN-FR</td>
</tr>
<tr>
<td>ByteNet [15]</td>
<td>23.75</td>
<td>1.0 \times 10^{20}</td>
</tr>
<tr>
<td>Deep-Att + PosUnk [32]</td>
<td>39.2</td>
<td>1.4 \times 10^{20}</td>
</tr>
<tr>
<td>GNMT + RL [31]</td>
<td>24.6</td>
<td>2.3 \times 10^{19}</td>
</tr>
<tr>
<td>ConvS2S [8]</td>
<td>39.92</td>
<td>1.5 \times 10^{20}</td>
</tr>
<tr>
<td>MoE [26]</td>
<td>26.03</td>
<td>2.0 \times 10^{19}</td>
</tr>
<tr>
<td></td>
<td>40.56</td>
<td>1.2 \times 10^{20}</td>
</tr>
<tr>
<td>Deep-Att + PosUnk Ensemble [32]</td>
<td>40.4</td>
<td>8.0 \times 10^{20}</td>
</tr>
<tr>
<td>GNMT + RL Ensemble [31]</td>
<td>26.30</td>
<td>1.8 \times 10^{20}</td>
</tr>
<tr>
<td>ConvS2S Ensemble [8]</td>
<td>41.16</td>
<td>1.1 \times 10^{21}</td>
</tr>
<tr>
<td></td>
<td>40.4</td>
<td>7.7 \times 10^{19}</td>
</tr>
<tr>
<td></td>
<td>41.29</td>
<td>1.2 \times 10^{21}</td>
</tr>
<tr>
<td>Transformer (base model)</td>
<td>27.3</td>
<td>3.3 \times 10^{18}</td>
</tr>
<tr>
<td>Transformer (big)</td>
<td>38.1</td>
<td>2.3 \times 10^{19}</td>
</tr>
<tr>
<td></td>
<td>28.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41.0</td>
<td></td>
</tr>
</tbody>
</table>

Vaswani et al., 2017: Attention Is All You Need
## Transformers: document generation

<table>
<thead>
<tr>
<th>Model</th>
<th>Test perplexity</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq2seq-attention, $L = 500$</td>
<td>5.04952</td>
<td>12.7</td>
</tr>
<tr>
<td>Transformer-ED, $L = 500$</td>
<td>2.46645</td>
<td>34.2</td>
</tr>
<tr>
<td>Transformer-D, $L = 4000$</td>
<td>2.22216</td>
<td>33.6</td>
</tr>
<tr>
<td>Transformer-DMCA, no MoE-layer, $L = 11000$</td>
<td>2.05159</td>
<td>36.2</td>
</tr>
<tr>
<td>Transformer-DMCA, MoE-128, $L = 11000$</td>
<td>1.92871</td>
<td>37.9</td>
</tr>
<tr>
<td>Transformer-DMCA, MoE-256, $L = 7500$</td>
<td>1.90325</td>
<td>38.8</td>
</tr>
</tbody>
</table>

Very large gains compared to seq2seq-attention with LSTMs!

Liu et al., 2018: Generating Wikipedia by Summarizing Long Sequences