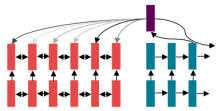
# Natural Language Processing with Deep Learning CS224N/Ling284

Tatsunori Hashimoto

Lecture 8: Self-Attention and Transformers

## Do we even need recurrence at all?

- Abstractly: Attention is a way to pass information from a sequence ($x$) to a neural network input. ($h_t$)
  - This is also *exactly* what RNNs are used for – to pass information!
  - **Can we just get rid of the RNN entirely?** Maybe attention is just a better way to pass information!



Lots of trial and error

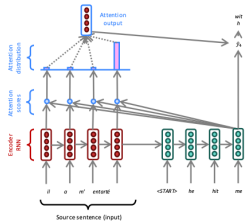
**2014-2017ish Recurrence**

**2021 ??????**



## The building block we need: *self* attention

- What we talked about – **Cross** attention: paying attention to the input x to generate $y_t$



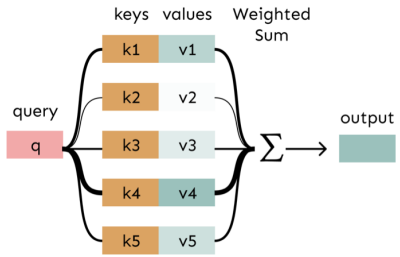
- What we need – **Self** attention: to generate $y_t$, we need to pay attention to $y_{<t}$



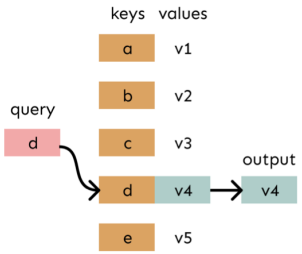
## Attention is *weighted* averaging, which lets you do lookups!

Attention is just a **weighted** average – this is very powerful if the weights are learned!

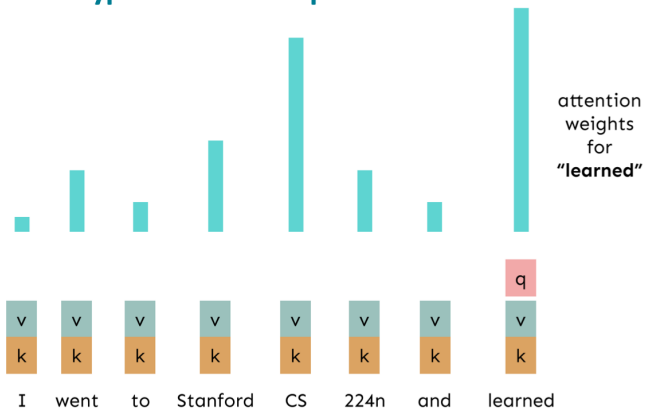
In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.




In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



## Self-Attention Hypothetical Example






## Self-Attention: keys, queries, values from the same sequence

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary $V$, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V, each in $\mathbb{R}^{d \times d}$

$\boldsymbol{q}_i = Q\boldsymbol{x}_i$ (queries) $\quad \boldsymbol{k}_i = K\boldsymbol{x}_i$ (keys) $\quad \boldsymbol{v}_i = V\boldsymbol{x}_i$ (values)

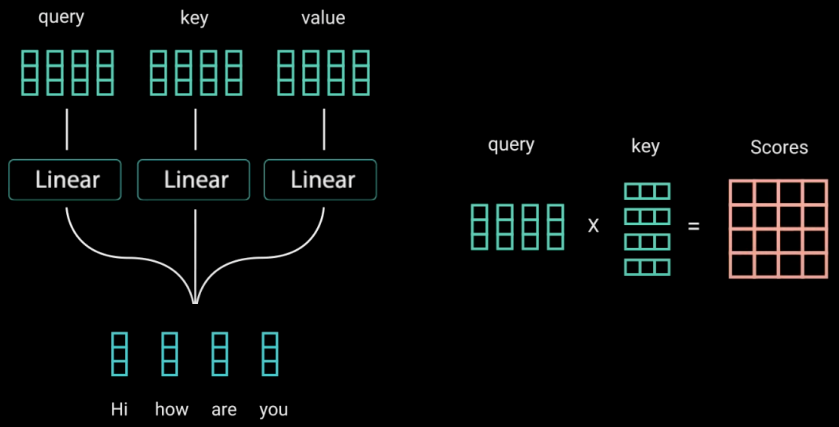
2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

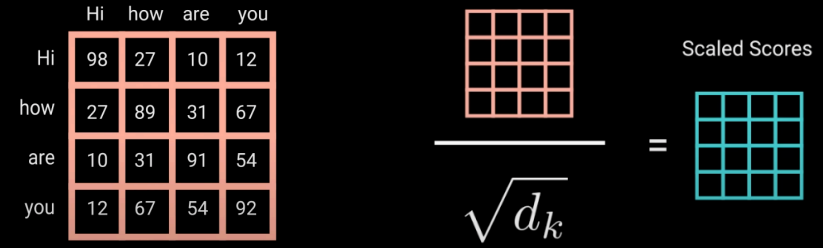
3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

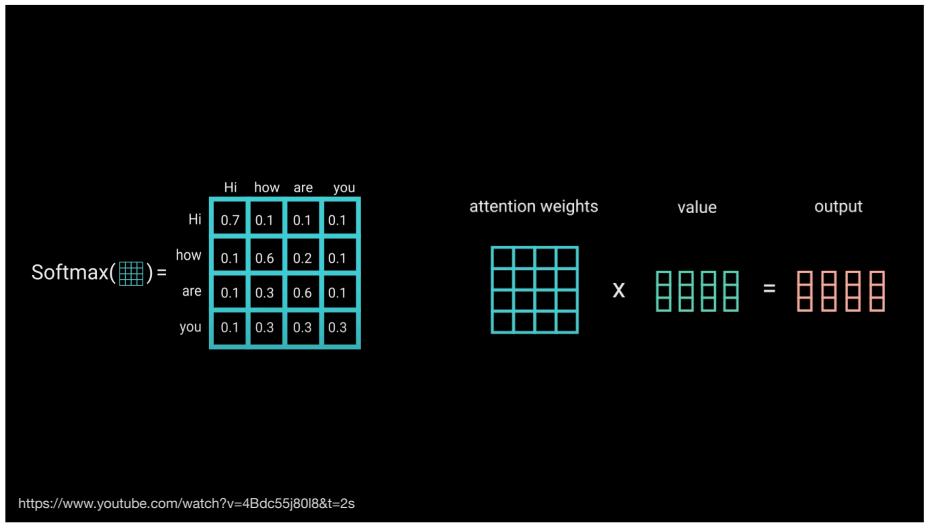





https://www.youtube.com/watch?v=4Bdc55j80l8&t=2s




https://www.youtube.com/watch?v=4Bdc55j80l8&t=2s




## Sequence-Stacked form of Attention

- Let's look at how key-query-value attention is computed, in matrices.
  - Let $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors.
  - First, note that $XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$.
  - The output is defined as output $= \text{softmax}(XQ(XK)^\top)XV \in \in \mathbb{R}^{n \times d}$.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$

$XQ$ $K^\top X^\top$ $= XQK^\top X^\top$ $\in \mathbb{R}^{n \times n}$

All pairs of attention scores!

Next, softmax, and compute the weighted average with another matrix multiplication.

softmax$(XQK^\top X^\top)$ $XV$ $=$ output $\in \mathbb{R}^{n \times d}$



## Barriers and solutions for Self-Attention as a building block

**Barriers**

- Doesn't have an inherent notion of order!

**Solutions**



## Fixing the first self-attention problem: **sequence order**

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1, 2, \dots, n\} \text{ are position vectors}$$

- Don't worry about what the $p_i$ are made of yet!
- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index $i$. The positioned embedding is:

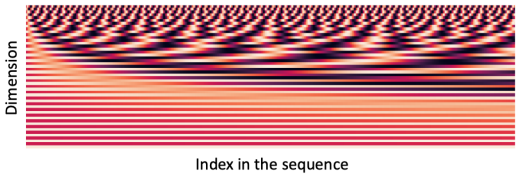
$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

## Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$




- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation doesn't really work!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/



## Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all $p_i$ be learnable parameters!
  Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, *and let each* $\boldsymbol{p}_i$ *be a column of that matrix!*
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \ldots, n$.
- Most systems use this!
- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]



## Barriers and solutions for Self-Attention as a building block

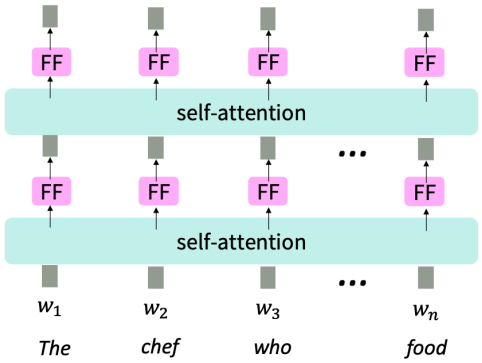
| Barriers | Solutions |
|---|---|
| Doesn't have an inherent notion of order! | Add position representations to the inputs |
| No nonlinearities for deep learning! It's all just weighted averages | |



## Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1\, \text{output}_i + b_1) + b_2$$




Intuition: the FF network processes the result of attention

## Barriers and solutions for Self-Attention as a building block

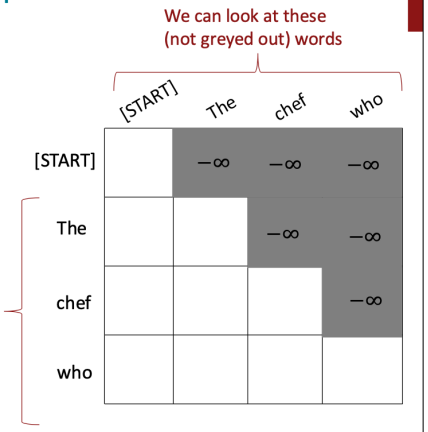
| Barriers | Solutions |
|---|---|
| • Doesn't have an inherent notion of order! | • Add position representations to the inputs |
| • No nonlinearities for deep learning magic! It's all just weighted averages | • Easy fix: apply the same feedforward network to each self-attention output. |
| • Need to ensure we don't "look at the future" when predicting a sequence<br>  • Like in machine translation<br>  • Or language modeling | |



## Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$





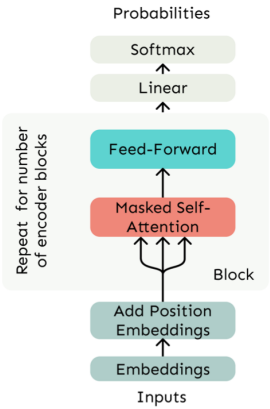

## Barriers and solutions for Self-Attention as a building block

| Barriers | Solutions |
|---|---|
| • Doesn't have an inherent notion of order! | • Add position representations to the inputs |
| • No nonlinearities for deep learning magic! It's all just weighted averages | • Easy fix: apply the same feedforward network to each self-attention output. |
| • Need to ensure we don't "look at the future" when predicting a sequence<br>  • Like in machine translation<br>  • Or language modeling | • Mask out the future by artificially setting attention weights to 0! |



## Necessities for a self-attention building block:

- **Self-attention**:
  - the basis of the method.
- **Position representations**:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities**:
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- **Masking**:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



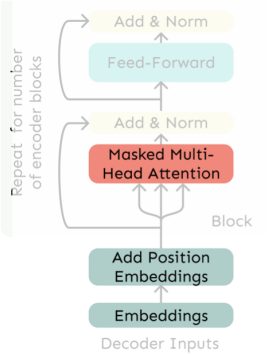
## Outline

1. From recurrence (RNN) to attention-based NLP models
2. The Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers



## The Transformer Decoder

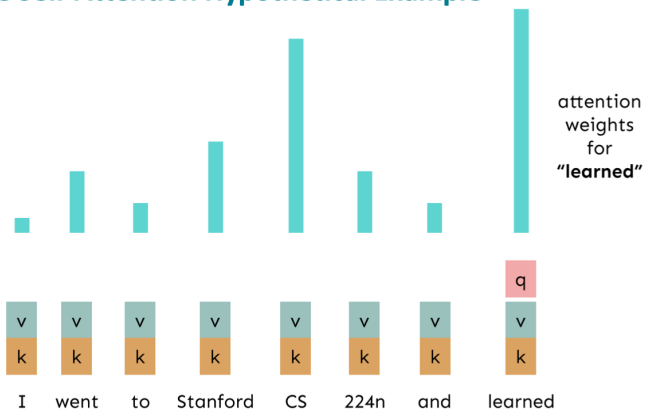
- A Transformer decoder is how we'll build systems like **language models**.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- The embeddings and position embeddings are identical.
- We'll next replace our self-attention with **multi-head self-attention.**




Transformer Decoder

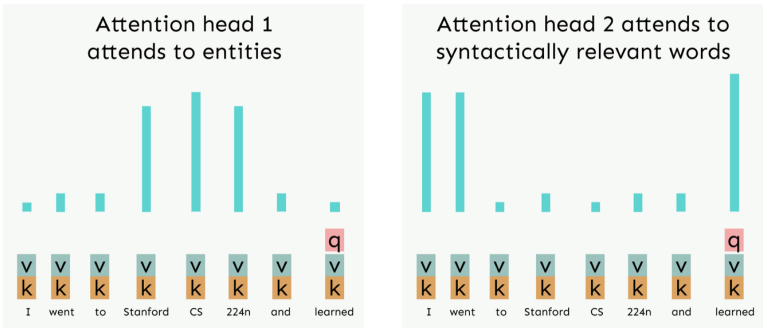


## Recall the Self-Attention Hypothetical Example

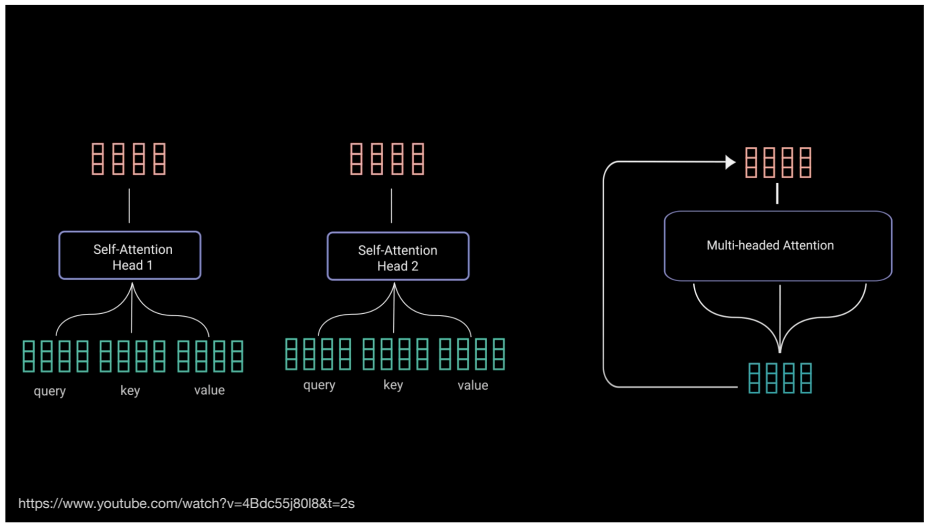





## Hypothetical Example of Multi-Head Attention




I went to Stanford CS 224n and learned




## Multi-headed attention

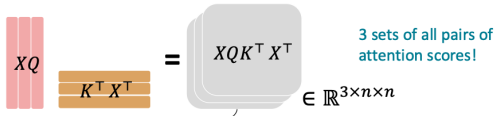
- What if we want to look in multiple places in the sentence at once?
  - For word $i$, self-attention "looks" where $x_i^\top Q^\top K x_j$ is high, but maybe we want to focus on different $j$ for different reasons?
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where $h$ is the number of attention heads, and $\ell$ ranges from 1 to $h$.
- Each attention head performs attention independently:
  - $\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \ldots; \text{output}_h]Y$, where $Y \in \mathbb{R}^{d \times d}$
- Each head gets to "look" at different things, and construct value vectors differently.



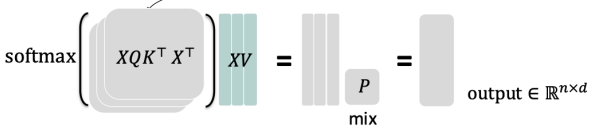
## Multi-head self-attention is computationally efficient

- Even though we compute $h$ many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for $XK$, $XV$.)
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.
  - Almost everything else is identical, and the **matrices are the same sizes.**

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$




Next, softmax, and compute the weighted average with another matrix multiplication.






## Scaled Dot Product [Vaswani et al., 2017]

- **"Scaled Dot Product"** attention aids in training.
- When dimensionality $d$ becomes large, dot products between vectors tend to become large.
  - Because of this, inputs to the softmax function can be large, making the gradients small.
- Instead of the self-attention function we've seen:

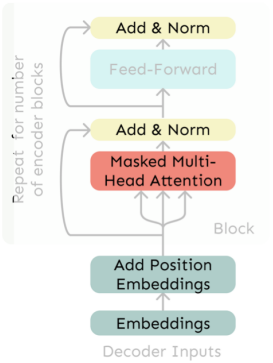
$$\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$$

- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of $d/h$ (The dimensionality divided by the number of heads.)

$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

## The Transformer Decoder

- Now that we've replaced self-attention with multi-head self-attention, we'll go through two **optimization tricks** that end up being :
  - **Residual Connections**
  - **Layer Normalization**
- In most Transformer diagrams, these are often written together as "Add & Norm"




Transformer Decoder



## The Transformer Encoder: **Residual connections** [He et al., 2016]

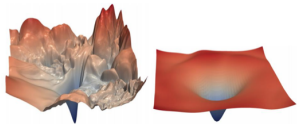
- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where $i$ represents the layer)

$X^{(i-1)}$ → Layer → $X^{(i)}$

  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn "the residual" from the previous layer)

$X^{(i-1)}$ → Layer → (+) → $X^{(i)}$

  - Gradient is **great** through the residual connection; it's 1!
  - Bias towards the identity function!



[no residuals] [residuals]

[Loss landscape visualization, Li et al., 2018, on a ResNet]



## The Transformer Encoder: **Layer normalization** [Ba et al., 2016]

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)
- Then layer normalization computes:

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

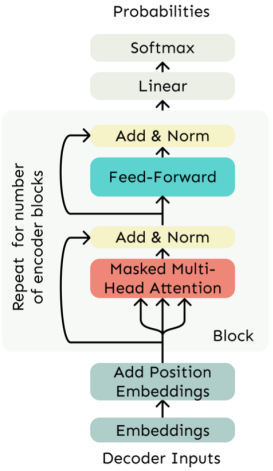
Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias



## The Transformer Decoder

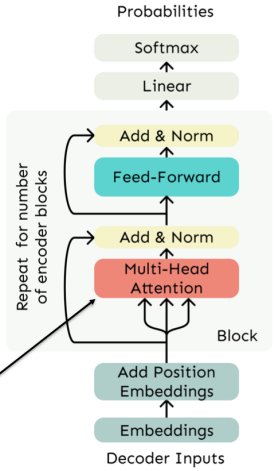
- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



## The Transformer Encoder

- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

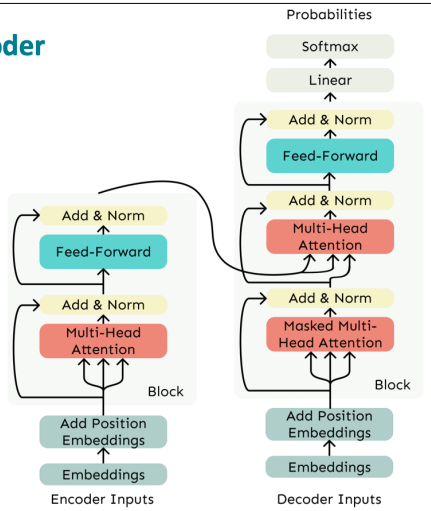
**No Masking!**






## The Transformer Encoder-Decoder

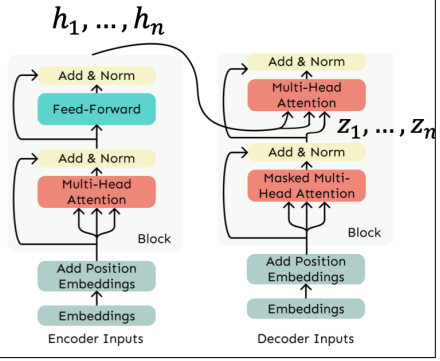
- Recall that in machine translation, we processed the source sentence with a **bidirectional** model and generated the target with a **unidirectional model**.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.






## Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $h_1, \ldots, h_n$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \ldots, z_n$ be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
  - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.

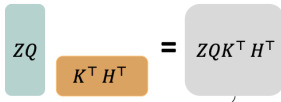





## Cross-attention (details)

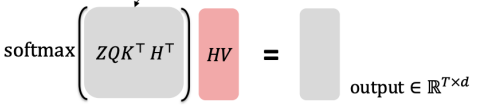
- Let's look at how cross-attention is computed, in matrices.
  - Let $\mathrm{H} = [h_1; \ldots; h_T] \in \mathbb{R}^{T \times d}$ be the concatenation of encoder vectors.
  - Let $\mathrm{Z} = [z_1; \ldots; z_T] \in \mathbb{R}^{T \times d}$ be the concatenation of decoder vectors.
  - The output is defined as output $= \mathrm{softmax}(ZQ(HK)^\top) \times HV$.

First, take the query-key dot products in one matrix multiplication: $ZQ(HK)^\top$




All pairs of attention scores!

Next, softmax, and compute the weighted average with another matrix multiplication.



## Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers



## Great Results with Transformers

First, Machine Translation from the original Transformers paper!

| Model | BLEU | | Training Cost (FLOPs) | |
|---|---|---|---|---|
| | EN-DE | EN-FR | EN-DE | EN-FR |
| ByteNet [18] | 23.75 | | | |
| Deep-Att + PosUnk [39] | | 39.2 | | $1.0 \cdot 10^{20}$ |
| GNMT + RL [38] | 24.6 | 39.92 | $2.3 \cdot 10^{19}$ | $1.4 \cdot 10^{20}$ |
| ConvS2S [9] | 25.16 | 40.46 | $9.6 \cdot 10^{18}$ | $1.5 \cdot 10^{20}$ |
| MoE [32] | 26.03 | 40.56 | $2.0 \cdot 10^{19}$ | $1.2 \cdot 10^{20}$ |
| Deep-Att + PosUnk Ensemble [39] | | 40.4 | | $8.0 \cdot 10^{20}$ |
| GNMT + RL Ensemble [38] | 26.30 | 41.16 | $1.8 \cdot 10^{20}$ | $1.1 \cdot 10^{21}$ |
| ConvS2S Ensemble [9] | 26.36 | **41.29** | $7.7 \cdot 10^{19}$ | $1.2 \cdot 10^{21}$ |

 [Test sets: WMT 2014 English-German and English-French]

[Vaswani et al., 2017]

## Great Results with Transformers

Next, document generation!

| Model | Test perplexity | ROUGE-L |
|---|---|---|
| *seq2seq-attention, L = 500* | 5.04952 | 12.7 |
| *Transformer-ED, L = 500* | 2.46645 | 34.2 |
| *Transformer-D, L = 4000* | 2.22216 | 33.6 |
| *Transformer-DMCA, no MoE-layer, L = 11000* | 2.05159 | 36.2 |
| *Transformer-DMCA, MoE-128, L = 11000* | 1.92871 | 37.9 |
| *Transformer-DMCA, MoE-256, L = 7500* | 1.90325 | 38.8 |

The old standard

Transformers all the way down.



[Liu et al., 2018]; WikiSum dataset

## Great Results with Transformers

Before too long, most Transformers results also included **pretraining**, a method we'll go over next.

Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.

On this popular aggregate benchmark, for example:

GLUE

**All** top models are Transformer (and pretraining)-based.

| Rank | Name | Model | URL | Score |
|---|---|---|---|---|
| 1 | DeBERTa Team - Microsoft | DeBERTa / TuringNLRv4 | | 90.8 |
| 2 | HFL iFLYTEK | MacALBERT + DKM | | 90.7 |
| 3 | Alibaba DAMO NLP | StructBERT + TAPT | | 90.6 |
| 4 | PING-AN Omni-Sinitic | ALBERT + DAAF + NAS | | 90.6 |
| 5 | ERNIE Team - Baidu | ERNIE | | 90.4 |
| 6 | T5 Team - Google | T5 | | 90.3 |

**More results Thursday when we discuss pretraining.**



[Liu et al., 2018]

## Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers



## What would we like to fix about the Transformer?

- **Training instabilities (Pre vs Post norm)**
- **Quadratic compute in self-attention** :
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!



## Pre vs Post norm

The one thing *everyone* agrees on (in 2024)

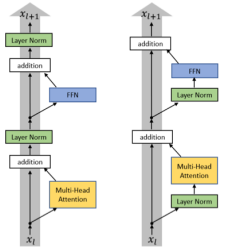



Figure from Xiong 2020

| Post-LN Transformer | Pre-LN Transformer |
|---|---|
| $x_{l,i}^{post,1} = \text{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}])$ | $x_{l,i}^{pre,1} = \text{LayerNorm}(x_{l,i}^{pre})$ |
| $x_{l,i}^{post,2} = x_{l,i}^{post} + x_{l,i}^{post,1}$ | $x_{l,i}^{pre,2} = \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}])$ |
| $x_{l,i}^{post,3} = \text{LayerNorm}(x_{l,i}^{post,2})$ | $x_{l,i}^{pre,3} = x_{l,i}^{pre} + x_{l,i}^{pre,2}$ |
| $x_{l,i}^{post,4} = \text{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$ | $x_{l,i}^{pre,4} = \text{LayerNorm}(x_{l,i}^{pre,3})$ |
| $x_{l,i}^{post,5} = x_{l,i}^{post,3} + x_{l,i}^{post,4}$ | $x_{l,i}^{pre,5} = \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$ |
| $x_{l+1,i}^{post} = \text{LayerNorm}(x_{l,i}^{post,5})$ | $x_{l+1,i}^{pre} = x_{l,i}^{pre,5} + x_{l,i}^{pre,3}$ |
| | Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$ |

Set up LayerNorm so that it doesn't affect the main residual signal path (on the left)

**Almost all modern LMs use pre-norm (but BERT was post-norm)**

(One somewhat funny exception – OPT350M. I don't know why this is post-norm)



## Quadratic computation as a function of sequence length

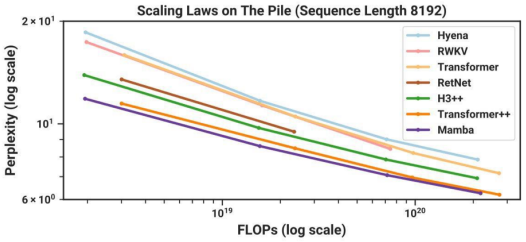
- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(n^2 d)$, where $n$ is the sequence length, and $d$ is the dimensionality.

$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{n \times n}$$

Need to compute all pairs of interactions! $O(n^2 d)$

- Think of $d$ as around $\mathbf{1,000}$ (though for large language models it's much larger!).
  - So, for a single (shortish) sentence, $n \leq 30$; $n^2 \leq \mathbf{900}$.
  - In practice, we set a bound like $n = 512$.
  - **But what if we'd like $n \geq 50,000$?** For example, to work on long documents?

## Back to the future – RNNs are back!



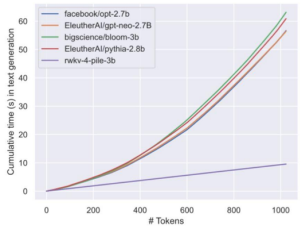



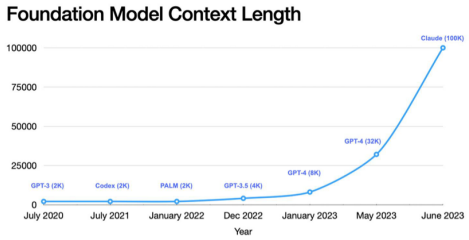

Figure 7: Cumulative time on text generation for LLM. Unlike transformers, RWKV exhibits linear scaling.

If you want *really* long context, RNNs provide this (linear complexity).
Modern RNNs (RWKV, Mamba, etc) are getting better!



## Do we even need to remove the quadratic cost of attention?

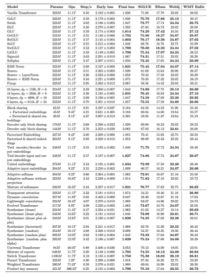
- As Transformers grow larger, a larger and larger percent of compute is **outside** the self-attention portion, despite the quadratic cost.
- In practice, **production Transformer language models use quadratic cost attention**
  - The cheaper methods tend not to work as well at scale.
  - Systems optimizations work well (Flash attention – Jun 2022)






## Do Transformer Modifications Transfer?

- "Surprisingly, we find that most modifications do not meaningfully improve performance."



Do Transformer Modifications Transfer Across Implementations and Applications?


Sharan Narang* Hyung Won Chung Yi Tay William Fedus
Thibault Fevry† Michael Matena† Karishma Malkan† Noah Fiedel
Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li
Nan Ding Jake Marcus Adam Roberts Colin Raffel†