

# EE-608: Deep Learning For Natural Language Processing: Transformers

James Henderson



DLNLP, Lecture 4

# Outline

Attention instead of Recurrence (review)

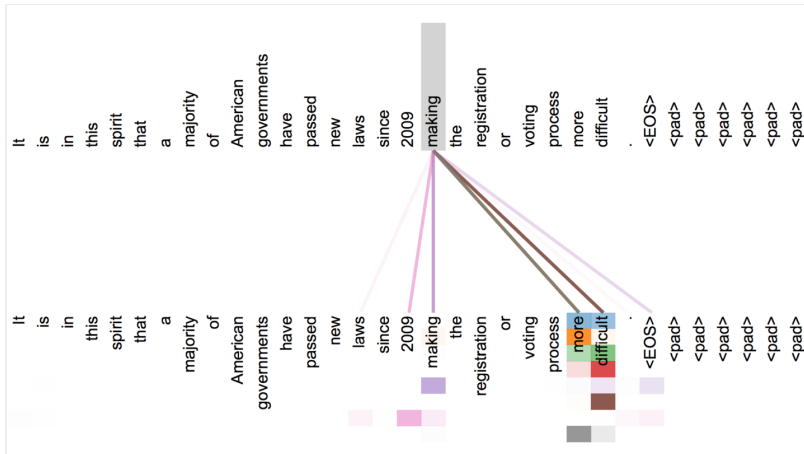
Transformer Architecture

Transformer Results

Transformer Variants

Pretrained Transformers (preview)

# Overview of Transformers



- ▶ Multiple layers, each outputting one vector per token
- ▶ Self-attention between layers (often bidirectional, depicted)
- ▶ All positions share the same parameters

# Outline

Attention instead of Recurrence (review)

Transformer Architecture

Transformer Results

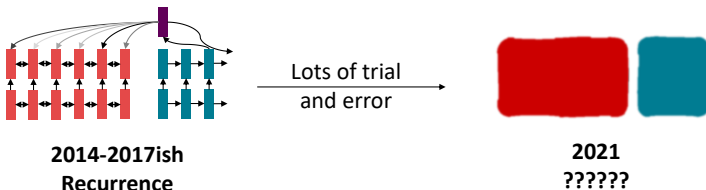
Transformer Variants

Pretrained Transformers (preview)



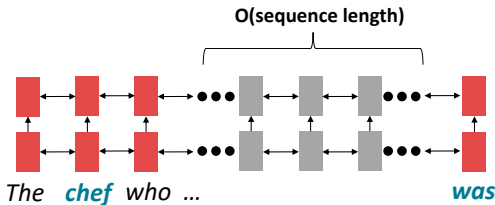
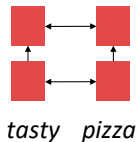
## Today: Same goals, different building blocks

- Last week, we learned about sequence-to-sequence problems and encoder-decoder models.
- Today, we're **not** trying to motivate entirely new ways of looking at problems (like Machine Translation)
- Instead, we're trying to find the best **building blocks** to plug into our models and enable broad progress.



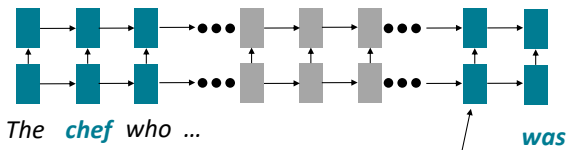
## Issues with recurrent models: Linear interaction distance

- RNNs are unrolled “left-to-right”.
- This encodes linear locality: a useful heuristic!
  - Nearby words often affect each other’s meanings
- **Problem:** RNNs take  $O(\text{sequence length})$  steps for distant word pairs to interact.



## Issues with recurrent models: Linear interaction distance

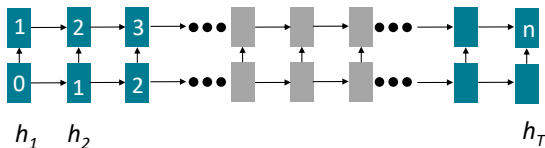
- **$O(\text{sequence length})$**  steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences...



Info of **chef** has gone through  $O(\text{sequence length})$  many layers!

## Issues with recurrent models: Lack of parallelizability

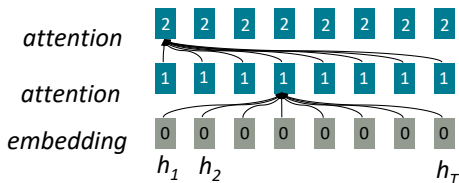
- Forward and backward passes have  **$O(\text{sequence length})$**  unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



Numbers indicate min # of steps before a state can be computed

## If no recurrence, then what? How about attention?

- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
  - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.
- Number of unparallelizable operations does not increase with sequence length.
- Maximum interaction distance:  $O(1)$ , since all words interact at every layer!

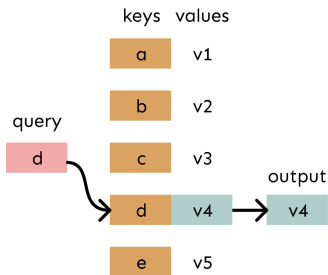


All words attend to all words in previous layer; most arrows here are omitted

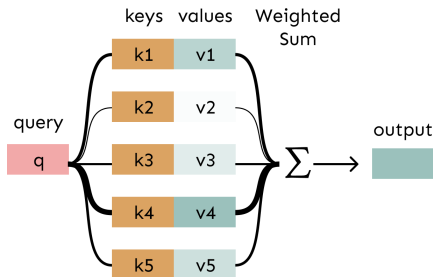
## Attention as a soft, averaging lookup table

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

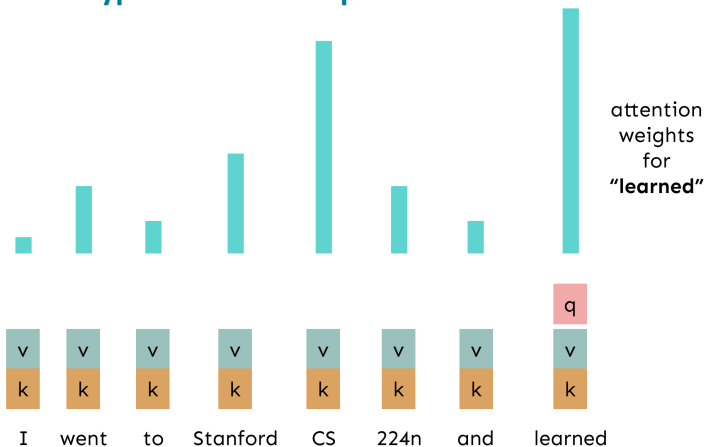
In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



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.



## Self-Attention Hypothetical Example



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

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

For each  $\mathbf{w}_i$ , let  $\mathbf{x}_i = E\mathbf{w}_i$ , 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  $\mathbb{R}^{d \times d}$

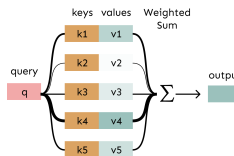
$$\mathbf{q}_i = Q\mathbf{x}_i \text{ (queries)} \quad \mathbf{k}_i = K\mathbf{x}_i \text{ (keys)} \quad \mathbf{v}_i = V\mathbf{x}_i \text{ (values)}$$

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

$$\mathbf{e}_{ij} = \mathbf{q}_i^\top \mathbf{k}_j \quad \alpha_{ij} = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})}$$

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

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_j$$





# 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**

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

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

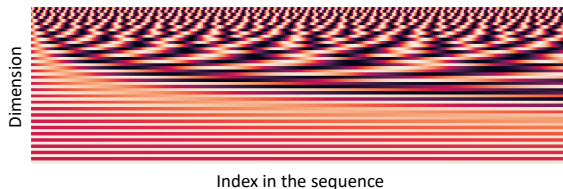
$$\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$$

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:

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



- 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!

# Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all  $p_i$  be learnable parameters!  
Learn a matrix  $\mathbf{p} \in \mathbb{R}^{d \times n}$ , and let each  $\mathbf{p}_i$  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  $1, \dots, n$ .
- 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

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning! It's all just weighted averages



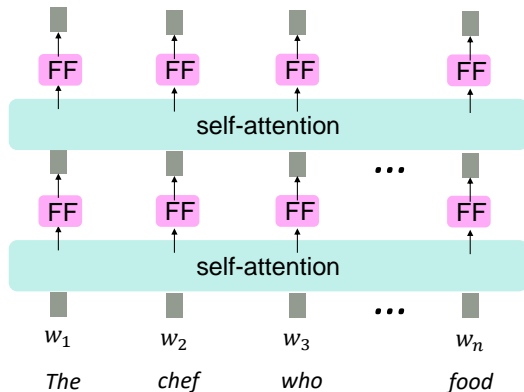
## Solutions

- Add position representations to the inputs

## 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.

$$\begin{aligned} m_i &= MLP(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \text{output}_i + b_1) + b_2 \end{aligned}$$



Intuition: the FF network processes the result of attention

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

## Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
  - Like in machine translation
  - Or language modeling



## Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.

# 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  $-\infty$ .

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

For encoding these words

We can look at these (not greyed out) words

	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				



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

## Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
  - Like in machine translation
  - Or language modeling

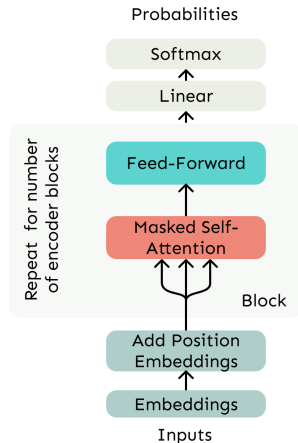


## Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.
- 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.



# Summary of Attention instead of Recurrence

- ▶ Attention is all you need
- ▶ Plus a representation of sequence order, with absolute (or relative) positions
- ▶ Plus layers of nonlinearity, for a fixed number of layers
- ▶ Plus causal masking, to simulate running multiple models on the same computation graph

# Outline

Attention instead of Recurrence (review)

**Transformer Architecture**

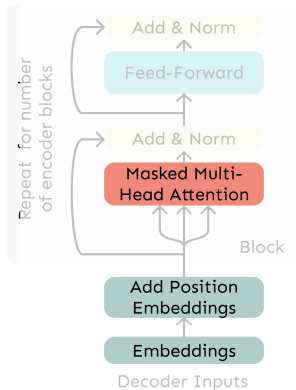
Transformer Results

Transformer Variants

Pretrained Transformers (preview)

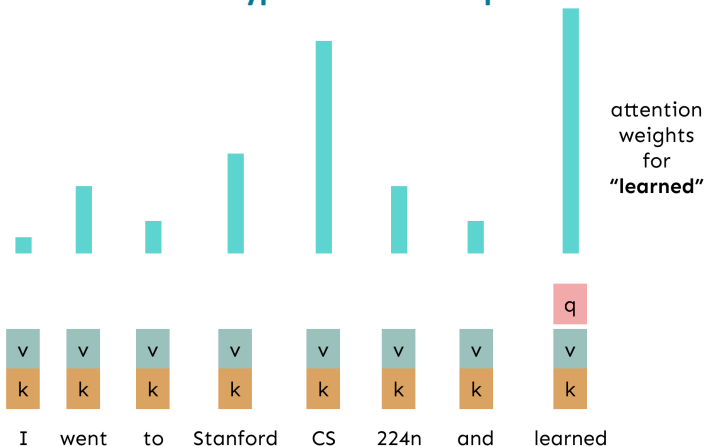
# The Transformer Decoder

- 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**.



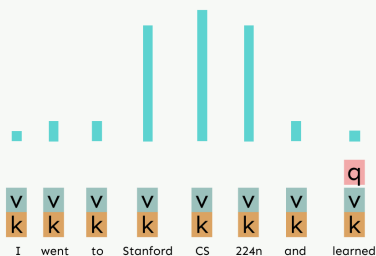
Transformer Decoder

## Recall the Self-Attention Hypothetical Example

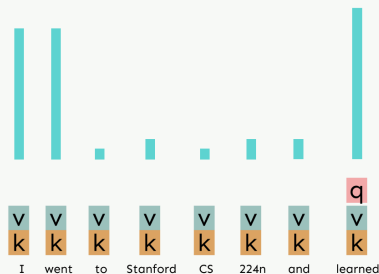


# Hypothetical Example of Multi-Head Attention

Attention head 1  
attends to entities



Attention head 2 attends to  
syntactically relevant words



I    went    to    Stanford    CS    224n    and    learned

## 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  $\text{output} = \text{softmax}(XQ(XK)^T)XV \in \mathbb{R}^{n \times d}$ .

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

$$\begin{matrix} \text{pink box} & \text{orange box} & = & \text{grey box} \\ XQ & K^T X^T & & XQK^T X^T \end{matrix} \in \mathbb{R}^{n \times n}$$

All pairs of attention scores!

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

$$\text{softmax} \left( \begin{matrix} \text{grey box} \\ XQK^T X^T \end{matrix} \right) \begin{matrix} \text{teal box} \\ XV \end{matrix} = \begin{matrix} \text{grey box} \\ \text{output} \end{matrix} \in \mathbb{R}^{n \times d}$$



# 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:
  - $\text{output}_\ell = \text{softmax}(X Q_\ell K_\ell^\top X^\top) * X V_\ell$ , where  $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \dots; \text{output}_h] Y$ , 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:  $XQ(XK)^\top$

$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{3 \times n \times n}$$

3 sets of all pairs of attention scores!

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

$$\text{softmax} \left( XQK^\top X^\top \right) XV = \text{mix} = \text{output} \in \mathbb{R}^{n \times n}$$

## 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:

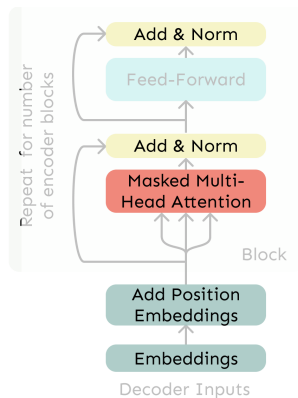
$$\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$$

- 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

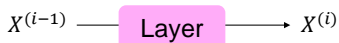
- Now that we've replaced self-attention with multi-head self-attention, we'll go through two **optimization tricks** that end up being :
  - **Residual Connections**
  - **Layer Normalization**
- In most Transformer diagrams, these are often written together as "Add & Norm"



Transformer Decoder

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

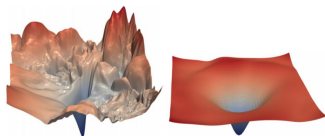
- **Residual connections** are a trick to help models train better.
  - Instead of  $X^{(i)} = \text{Layer}(X^{(i-1)})$  (where  $i$  represents the layer)



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



- 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:

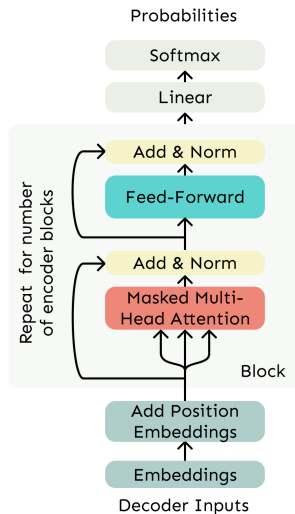
$$\text{output} = \frac{x - \mu}{\sqrt{\sigma} + \epsilon} * \gamma + \beta$$

Normalize by scalar mean and variance  $\rightarrow$

$\leftarrow$  Modulate by learned elementwise gain and bias

# The Transformer Decoder

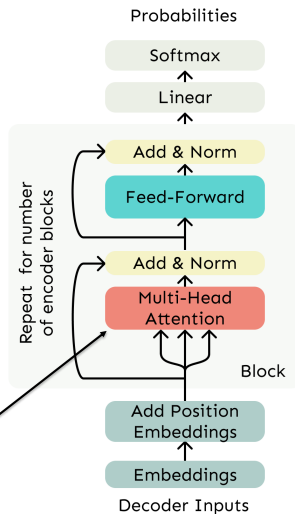
- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



# 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.

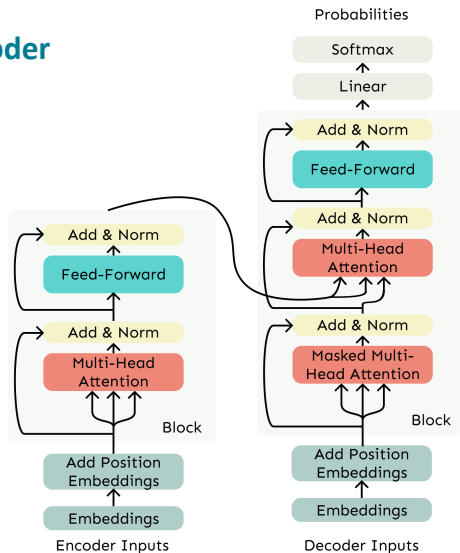
**No Masking!**





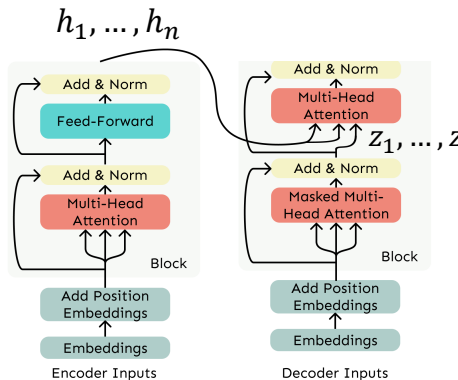
# The Transformer Encoder-Decoder

- 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  $h_1, \dots, h_n$  be **output** vectors **from** the Transformer **encoder**;  $x_i \in \mathbb{R}^d$
- Let  $z_1, \dots, z_n$  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**,  $q_i = Qz_i$ .



# Outline

Attention instead of Recurrence (review)

Transformer Architecture

**Transformer Results**

Transformer Variants

Pretrained Transformers (preview)

# 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	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$

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

[Vaswani et al., 2017]

*Slide from John Hewitt*

# Great Results with Transformers

Next, document generation!

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

The old standard

Transformers all the way down.

# Great Results with Transformers

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

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

On this popular aggregate benchmark, for example:



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

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

More results Thursday when we discuss pretraining.

[[Liu et al., 2018](#)]

# Outline

Attention instead of Recurrence (review)

Transformer Architecture

Transformer Results

**Transformer Variants**

Pretrained Transformers (preview)

# What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!
- **Position representations:**
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [\[Shaw et al., 2018\]](#)
  - Dependency syntax-based position [\[Wang et al., 2019\]](#)



# 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  $O(n^2d)$ , where  $n$  is the sequence length, and  $d$  is the dimensionality.

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

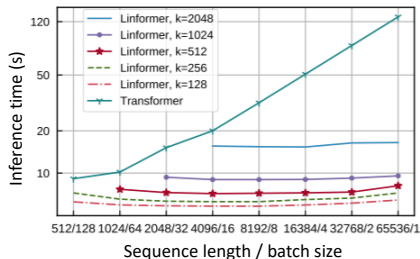
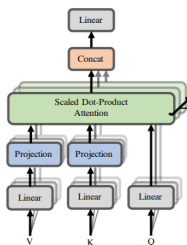
Need to compute all pairs of interactions!  
 $O(n^2d)$

- Think of  $d$  as around **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?

# Work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the  $O(T^2)$  all-pairs self-attention cost?*
- For example, **Linformer** [\[Wang et al., 2020\]](#)

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



## 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, **almost no large Transformer language models use anything but the quadratic cost attention we've presented here.**
  - The cheaper methods tend not to work as well at scale.
- So, is there no point in trying to design cheaper alternatives to self-attention?
- Or would we unlock much better models with much longer contexts (>100k tokens?) if we were to do it right?

## Do Transformer Modifications Transfer?

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

[illegible]

## Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang\* Hyung Won Chung Yi Tay William Fedus

Thibault Fevry<sup>†</sup>   Michael Matena<sup>†</sup>   Karishma Malkan<sup>†</sup>   Noah FiedelNoam Shazeer      Zhenzhong Lan<sup>†</sup>      Yanqi Zhou      Wei Li

Nan Ding      Jake Marcus      Adam Roberts      Colin Raffel†

# Fixing position representations:

## Relative position encoding

In language, absolute positions are not very meaningful, but relative positions are important

- ▶ Relative positions are properties of pairs of tokens
  - ▶ The attention mechanism processes pairs of tokens
- ⇒ Input relative position to the attention weight computation
- ▶ Given relative position embeddings  $R \in \mathbb{R}^{n \times d_k}$ , where  $R_{ij}$  is the embedding of length  $j-i$ , [\[Shaw et al 2018\]](#)

$$E_{ij}(Q, K, R) = Q_i(K_j + R_{ij})^T$$

$$A(Q, K, V, R) = \text{softmax} \left( \frac{E(Q, K, R)}{\sqrt{d_k}} \right) V$$

with queries  $Q \in \mathbb{R}^{n \times d_k}$ , keys  $K \in \mathbb{R}^{n \times d_k}$ , values  $V \in \mathbb{R}^{n \times d_v}$

# Machine Translation (Shaw et al, 2018)

Model	Position Representation	BLEU En-De	BLEU En-Fr
Transformer Big	Absolute	27.9	41.3
Transformer Big	Relative	<b>29.2</b>	<b>41.5</b>

# Fixing sequence inputs: Graph-to-Graph Transformers

Instead of sequence-to-sequence Transformers, can we have graph-to-graph Transformers?

Yes! [Mohammadshahi and Henderson 2021, 2020]

- ▶ One “token” for each node of the graph
- ▶ Graph edges are properties of pairs of tokens
- ⇒ Input edge labels to the attention weight computation
- ▶ Given edge label embeddings  $R \in \mathbb{R}^{n \times d_k}$ , where  $R_{ij}$  is the embedding of length  $j-i$ ,

$$E_{ij}(Q, K, R) = Q_i(K_j + R_{ij})^T$$

$$A(Q, K, V, R) = \text{softmax} \left( \frac{E(Q, K, R)}{\sqrt{d_k}} \right) V$$

with queries  $Q \in \mathbb{R}^{n \times d_k}$ , keys  $K \in \mathbb{R}^{n \times d_k}$ , values  $V \in \mathbb{R}^{n \times d_v}$

## Fixing unknown words: subword inputs

There are always new words you have never seen before. And for some languages words are not the natural way to segment the input.

- ▶ input subwords rather than whole words
- ▶ choose the subword vocabulary based on frequency, so there are never any unknown subwords








# Word structure and subword models

Let's take a look at the assumptions we've made about a language's vocabulary.

We assume a fixed vocab of tens of thousands of words, built from the training set.

All *novel* words seen at test time are mapped to a single UNK.

	word		vocab mapping	embedding
Common words	hat	→	pizza (index)	
	learn	→	tasty (index)	
Variations	taaaaasty	→	UNK (index)	
misspellings	laern	→	UNK (index)	
novel items	Transformerify	→	UNK (index)	

## Word structure and subword models

Finite vocabulary assumptions make even *less* sense in many languages.

- Many languages exhibit complex **morphology**, or word structure.
  - The effect is more word types, each occurring fewer times.

Example: Swahili verbs can have hundreds of conjugations, each encoding a wide variety of information. (Tense, mood, definiteness, negation, information about the object, ++)

Here's a small fraction of the conjugations for *ambia* – to tell.

[illegible]

# The byte-pair encoding algorithm

Subword modeling in NLP encompasses a wide range of methods for reasoning about structure below the word level. (Parts of words, characters, bytes.)

- The dominant modern paradigm is to learn a vocabulary of **parts of words (subword tokens)**.
- At training and testing time, each word is split into a sequence of known subwords.

**Byte-pair encoding** is a simple, effective strategy for defining a subword vocabulary.










1. Start with a vocabulary containing only characters and an “end-of-word” symbol.
2. Using a corpus of text, find the most common adjacent characters “a,b”; add “ab” as a subword.
3. Replace instances of the character pair with the new subword; repeat until desired vocab size.

Originally used in NLP for machine translation; now a similar method (WordPiece) is used in pretrained models.

## Word structure and subword models

Common words end up being a part of the subword vocabulary, while rarer words are split into (sometimes intuitive, sometimes not) components.

In the worst case, words are split into as many subwords as they have characters.

	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	taa## aaa## sty	  
misspellings	laern	→	la## ern##	 
novel items	Transformerify	→	Transformer## ify	 

# Summary of Transformers

- ▶ Transformers are multi-layer attention-based sequence models
- ▶ Use bidirectional self-attention: Each token at one layer uses attention over all tokens at the layer below
- ▶ Use multiple attention heads
- ▶ Use bag-of-vector representations, with sequential position encoded in the input vectors (or relative position in attention vectors)
- ▶ Decoder transformers can be used to generate text
- ▶ Transformers can be generalised to graphs

# Outline

Attention instead of Recurrence (review)

Transformer Architecture

Transformer Results

Transformer Variants

Pretrained Transformers (preview)

# Overview of Pretraining Models

We saw with word embeddings that the distributions in large text corpora have a lot of information about the meaning of text. Can we also exploit this information in contextualised word representations for transfer to a new task?

- ▶ BERT:

- ▶ Transformer trained on predicting masked words
- ▶ Fine-tuned to perform a new task

- ▶ GPT (2,3):

- ▶ Transformer trained as a left-to-right language model
- ▶ Mined for information implicit in the language model

# Outline

1. A brief note on subword modeling
2. Motivating model pretraining from word embeddings
3. Model pretraining three ways
  1. Encoders
  2. Encoder-Decoders
  3. Decoders
4. What do we think pretraining is teaching?



# Motivating word meaning and context

Recall the adage we mentioned at the beginning of the course:

*“You shall know a word by the company it keeps”* (J. R. Firth 1957: 11)

This quote is a summary of **distributional semantics**, and motivated **word2vec**. But:

*“... the complete meaning of a word is always contextual,  
and no study of meaning apart from a complete context  
can be taken seriously.”* (J. R. Firth 1935)

Consider *I **record** the **record***: the two instances of **record** mean different things.

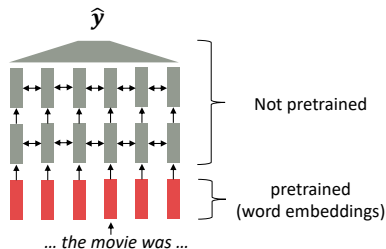
## Where we were: pretrained word embeddings

Circa 2017:

- Start with pretrained word embeddings (no context!)
- Learn how to incorporate context in an LSTM or Transformer while training on the task.

### Some issues to think about:

- The training data we have for our **downstream task** (like question answering) must be sufficient to teach all contextual aspects of language.
- Most of the parameters in our network are randomly initialized!

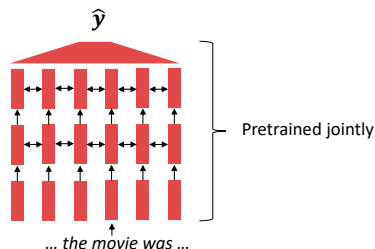


[Recall, *movie* gets the same word embedding, no matter what sentence it shows up in]

## Where we're going: **pretraining whole models**

In modern NLP:

- All (or almost all) parameters in NLP networks are initialized via **pretraining**.
- Pretraining methods hide parts of the input from the model, and train the model to reconstruct those parts.
- This has been exceptionally effective at building strong:
  - **representations of language**
  - **parameter initializations** for strong NLP models.
  - **Probability distributions** over language that we can sample from



[This model has learned how to represent entire sentences through pretraining]

## What can we learn from reconstructing the input?

Stanford University is located in \_\_\_\_\_, California.

## What can we learn from reconstructing the input?

I put \_\_\_\_ fork down on the table.

## What can we learn from reconstructing the input?

The woman walked across the street,  
checking for traffic over \_\_\_\_ shoulder.

## What can we learn from reconstructing the input?

I went to the ocean to see the fish, turtles, seals, and \_\_\_\_\_.

## What can we learn from reconstructing the input?

Overall, the value I got from the two hours watching  
it was the sum total of the popcorn and the drink.  
The movie was \_\_\_\_.



## What can we learn from reconstructing the input?

Iroh went into the kitchen to make some tea.  
Standing next to Iroh, Zuko pondered his destiny.  
Zuko left the \_\_\_\_\_.

## What can we learn from reconstructing the input?

I was thinking about the sequence that goes

1, 1, 2, 3, 5, 8, 13, 21, \_\_\_\_\_

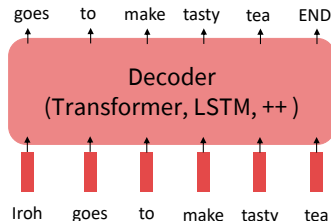
## Pretraining through language modeling [\[Dai and Le, 2015\]](#)

Recall the **language modeling** task:

- Model  $p_{\theta}(w_t | w_{1:t-1})$ , the probability distribution over words given their past contexts.
- There's lots of data for this! (In English.)

**Pretraining through language modeling:**

- Train a neural network to perform language modeling on a large amount of text.
- Save the network parameters.

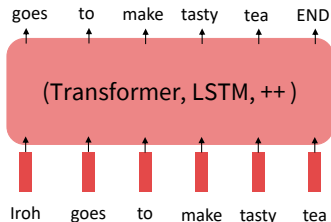


# The Pretraining / Finetuning Paradigm

Pretraining can improve NLP applications by serving as parameter initialization.

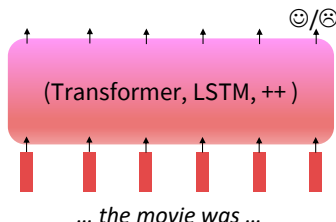
## Step 1: Pretrain (on language modeling)

Lots of text; learn general things!



## Step 2: Finetune (on your task)

Not many labels; adapt to the task!



## Stochastic gradient descent and pretrain/finetune

Why should pretraining and finetuning help, from a “training neural nets” perspective?

- Consider, provides parameters  $\hat{\theta}$  by approximating  $\min_{\theta} \mathcal{L}_{\text{pretrain}}(\theta)$ .
  - (The pretraining loss.)
- Then, finetuning approximates  $\min_{\theta} \mathcal{L}_{\text{finetune}}(\theta)$ , starting at  $\hat{\theta}$ .
  - (The finetuning loss)
- The pretraining may matter because stochastic gradient descent sticks (relatively) close to  $\hat{\theta}$  during finetuning.
  - So, maybe the finetuning local minima near  $\hat{\theta}$  tend to generalize well!
  - And/or, maybe the gradients of finetuning loss near  $\hat{\theta}$  propagate nicely!