CS 1678: Intro to Deep Learning

Modeling Sequences/Sets: Transformers

Prof. Adriana Kovashka
University of Pittsburgh
October 27, 2021

Plan for this lecture

- Background
  - Context prediction, unsupervised learning
- Transformer models
  - Self-attention
  - Adapting self-attention for sequential data
  - The transformer architecture, encoder/decoder
  - Pre-training, BERT
- Transformers beyond language

Additional resources

- Learning about transformers on your own?
  - Key recommended resource:
    - http://nlp.seas.harvard.edu/2018/04/03/attention.html
    - The Annotated Transformer by Sasha Rush
    - Jupyter Notebook using PyTorch that explains everything!
  - The Illustrated Transformer
    - http://jalammar.github.io/illustrated-transformer/
  - Attention visualizer
    - https://github.com/jessevig/bertviz

Adapted from Christopher Manning

How do we represent the meaning of a word?

Definition: **meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Commonest linguistic way of thinking of meaning:

$\boxed{\text{signifier (symbol)} \Leftrightarrow \text{signified (idea or thing)}}$

= denotational semantics

How do we have usable meaning in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** (“is a” relationships).

e.g. synonym sets containing “good”:

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
              ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good  adj:
good
adj (sat): estimable, good, honorable, respectable  adj (sat):
beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. hypernyms of “panda”:

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")  hyper =
lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

Problems with resources like WordNet

- Great as a resource but missing nuance
  - e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts.
- Missing new meanings of words
  - e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
  - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can't compute accurate word similarity

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel – a localist representation

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocab (e.g. 500,000)

Problem with words as discrete symbols

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

There is no natural notion of **similarity** for one-hot vectors!

Solution:

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

Representing words by their context

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**
  - *"You shall know a word by the company it keeps"* (J. R. Firth 1957)
  - One of the most successful ideas of modern statistical NLP!
- When a word *w* appears in a text, its context is the set of words that appear nearby (within a fixed-size window).
- Use the many contexts of *w* to build up a representation of *w*

…government debt problems turning into	***banking***	*crises as happened in 2009…*
…saying that Europe needs unified	***banking***	*regulation to replace the hodgepodge…*
…India has just given its	***banking***	*system a shot in the arm…*

These context words will represent ***banking***

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, _____

Word vectors

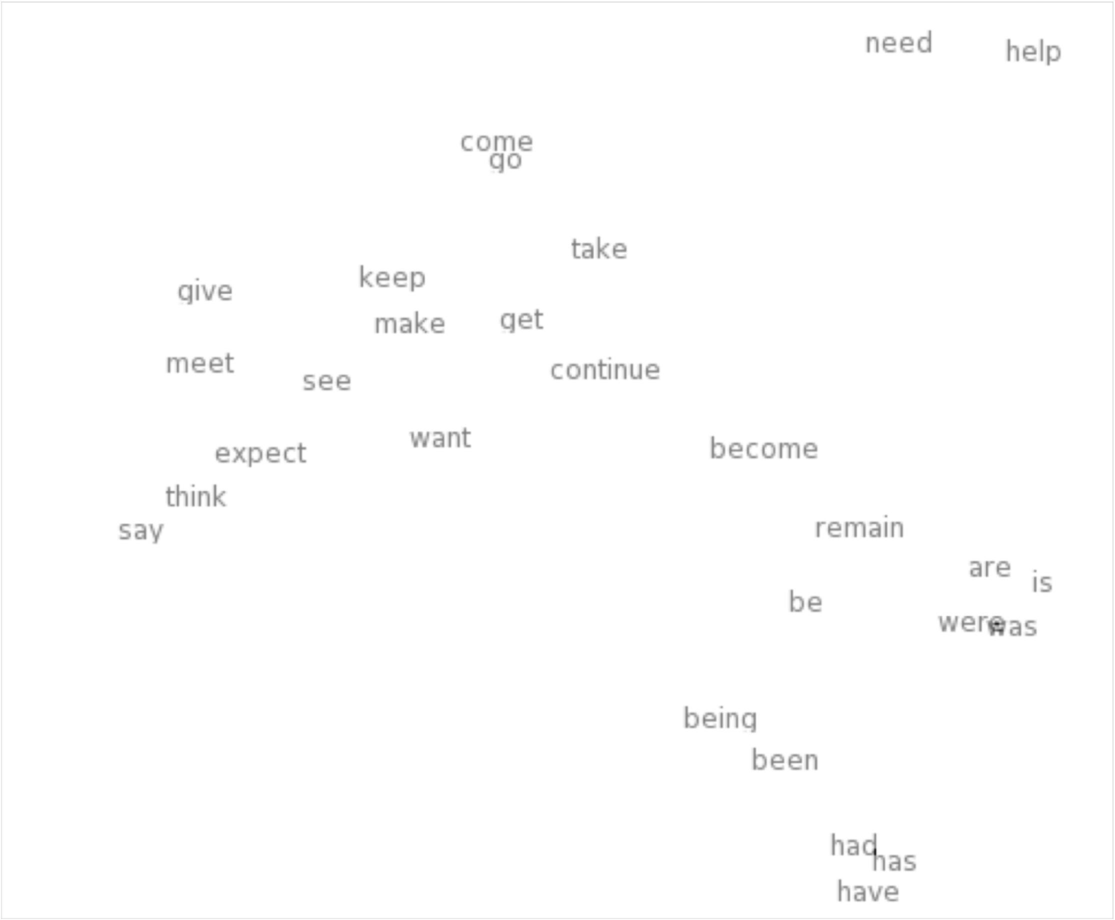
We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts

$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Note: word vectors are sometimes called word embeddings or word representations. They are a distributed representation.

Word meaning as a neural word vector - visualization

$$\textit{expect} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$



Word2Vec Overview

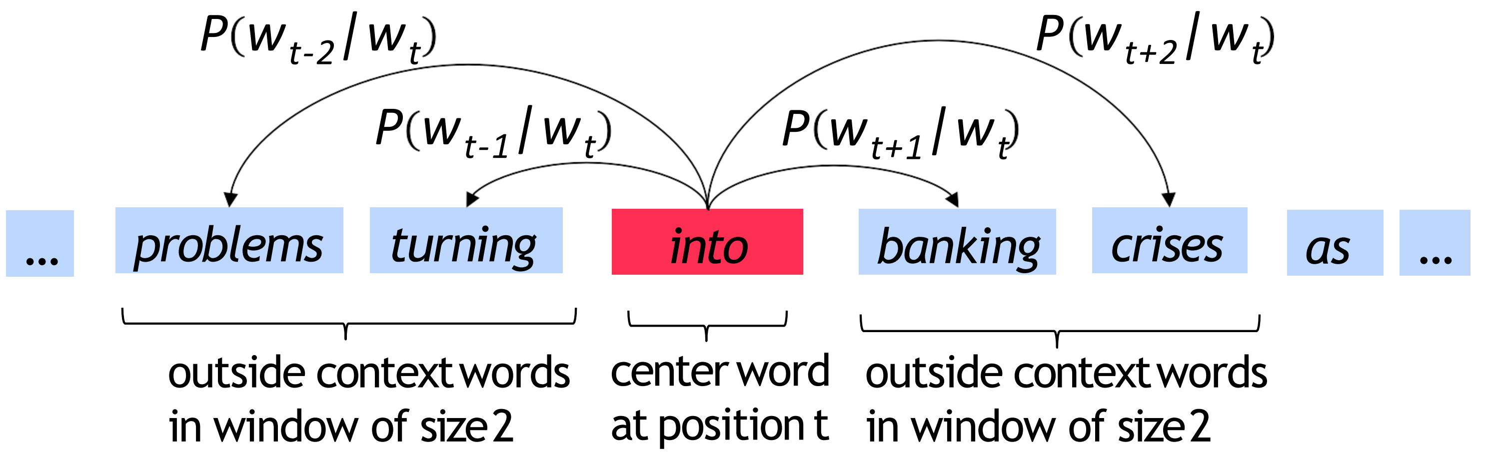
Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

Idea:

- We have a large corpus of text
- Every word in a fixed vocabulary is represented by a vector
- Go through each position *t* in the text, which has a center word *c* and context ("outside") words *o*
- Use the similarity of the word vectors for *c* and *o* to calculate the probability of *o* given *c* (or vice versa)
- Keep adjusting the word vectors to maximize this probability

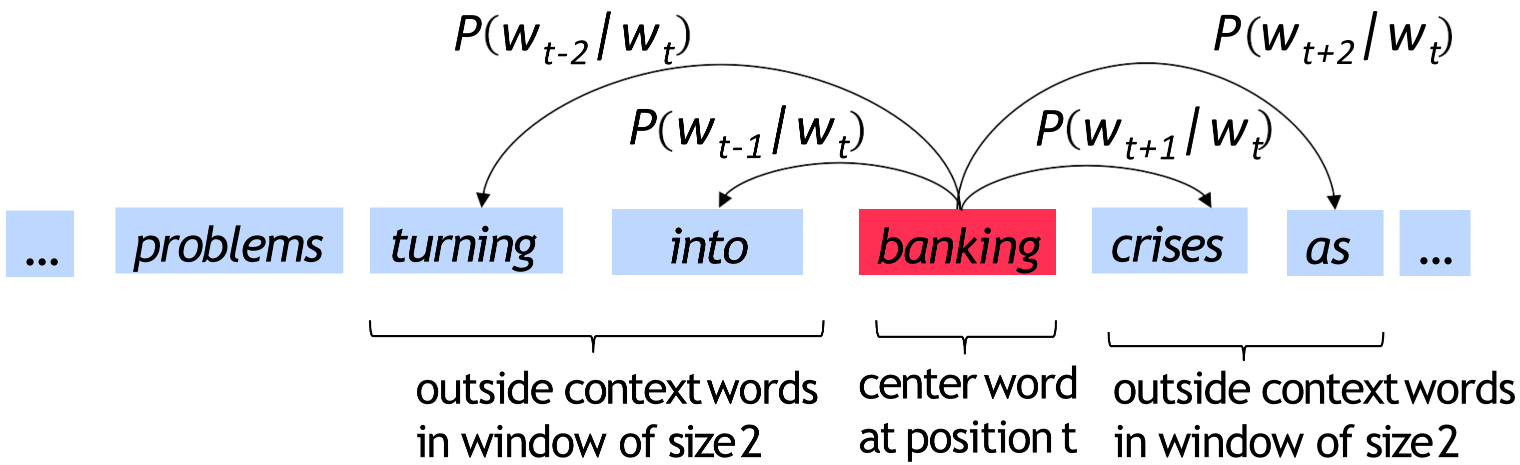
Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} | w_t)$



Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} | w_t)$



Word2Vec: objective function

For each position t = 1, … , T, predict context words within a window of fixed size *m*, given center word w_j.

Likelihood = $$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \ne 0}} P(w_{t+j} \mid w_t; \theta)$$

θ is all variables to be optimized

sometimes called *cost* or *loss* function

The objective function is the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \ne 0}} \log P(w_{t+j} \mid w_t; \theta)$$

Minimizing objective function ⟺ Maximizing predictive accuracy

Word2Vec: objective function

- We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T}\log L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\le j\le m\\ j\ne 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

- Question: How to calculate $P(w_{t+j} | w_t ; \theta)$?
- Answer: We will *use two* vectors per word *w*:
  - v_w when *w* is a center word
  - u_w when *w* is a context word
- Then for a center word *c* and a context word *o*:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

Word2Vec: prediction function

Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(\boxed{u_o^T v_c})}{\boxed{\sum_{w \in V} \exp(u_w^T v_c)}}$$

Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

Normalize over entire vocabulary
to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \to \mathbb{R}^n$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

- The softmax function maps arbitrary values to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i

ELMo: Embeddings from Language Models

Deep contextualized word representations. Peters et al. NAACL 2018. https://arxiv.org/abs/1802.05365

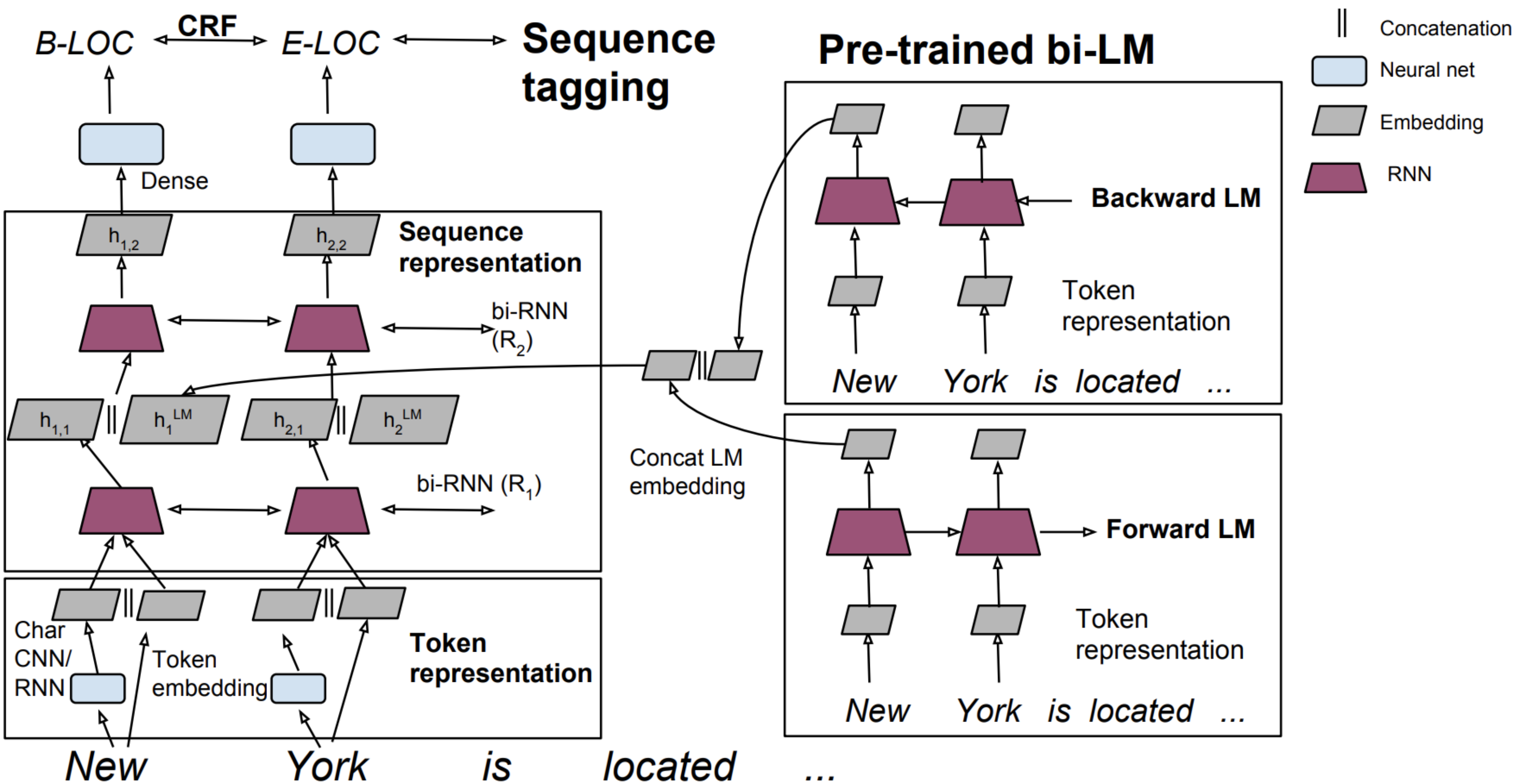
- Breakout version of **word token vectors** or **contextual word vectors**
- Learn word token vectors using long contexts not context windows (here, whole sentence, could be longer)
- Learn a deep Bi-NLM and use all its layers in prediction

ELMo: Embeddings from Language Models

- Train a bidirectional LM
- Aim at performant but not overly large LM:
  - Use 2 biLSTM layers
  - Use character CNN to build initial word representation
  - User 4096 dim hidden/cell LSTM states with 512 dim projections to next input
  - Use a residual connection
  - Tie parameters of token input and output (softmax) and tie these between forward and backward LMs

Adapted from Christopher Manning

ELMo used in a sequence tagger



$$\mathbf{h}_{k,1} = [\overrightarrow{\mathbf{h}}_{k,1}; \overleftarrow{\mathbf{h}}_{k,1}; \mathbf{h}_k^{LM}].$$

Christopher Manning, figure from https://tsenghungchen.github.io/posts/elmo/, paper at https://arxiv.org/pdf/1802.05365.pdf

ELMo results: Great for all tasks

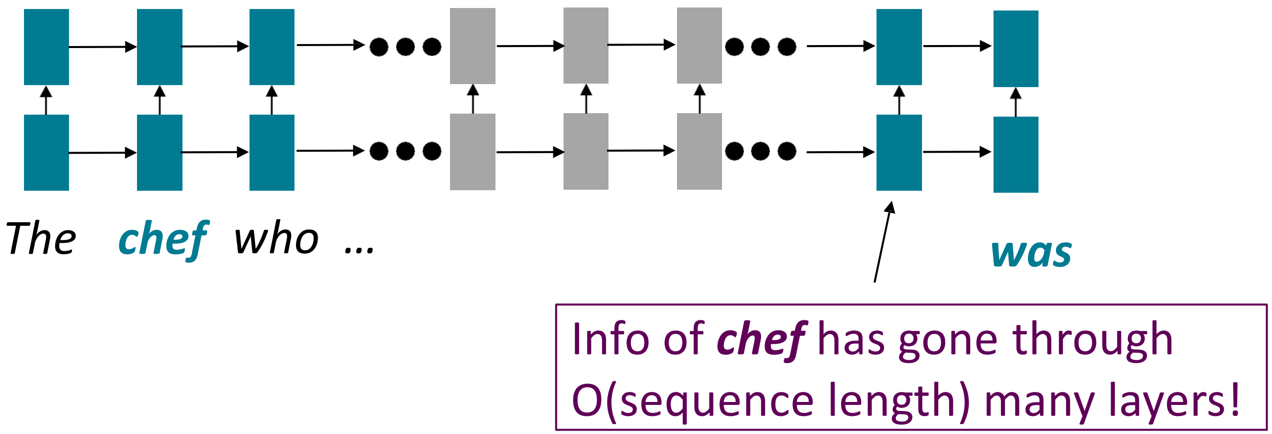
Task	Previous SOTA		Our baseline	ELMo + baseline	Increase (absolute/relative)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 $\pm$ 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 $\pm$ 0.19	90.15	92.22 $\pm$ 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 $\pm$ 0.5	3.3 / 6.8%

ELMo: Roles of different layers

- The two biLSTM NLM layers have differentiated uses/meanings
  - Lower layer is better for lower-level syntax, etc.
    - Part-of-speech tagging, syntactic dependencies, NER
  - Higher layer is better for higher-level semantics
    - Sentiment, Semantic role labeling, question answering, SNLI

Issues with recurrent models:
Linear interaction distance

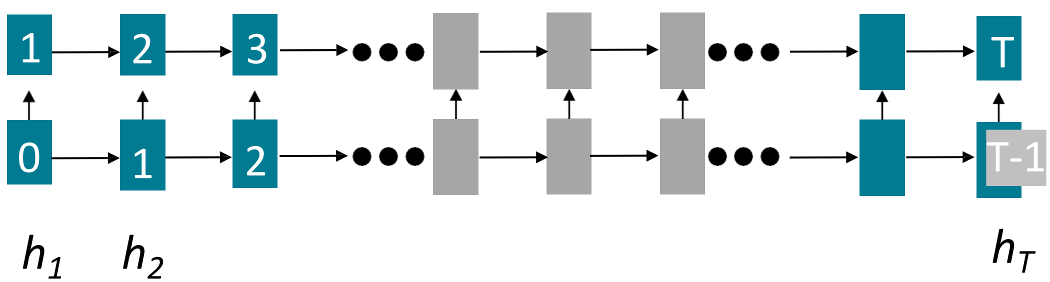
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is “baked in”; not necessarily the right way to think about sentences…



Adapted from John Hewitt

Issues with recurrent models: **Lack of parallelizability**

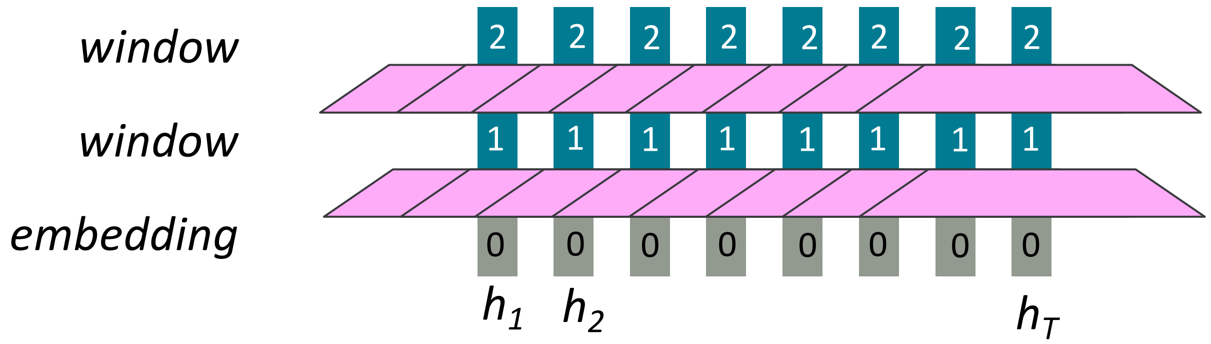
- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



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

If not recurrence, then what?
How about word windows?

- **Word window models aggregate local contexts**
  - Also known as 1D convolution
  - Number of unparallelizable operations not tied to sequence length!

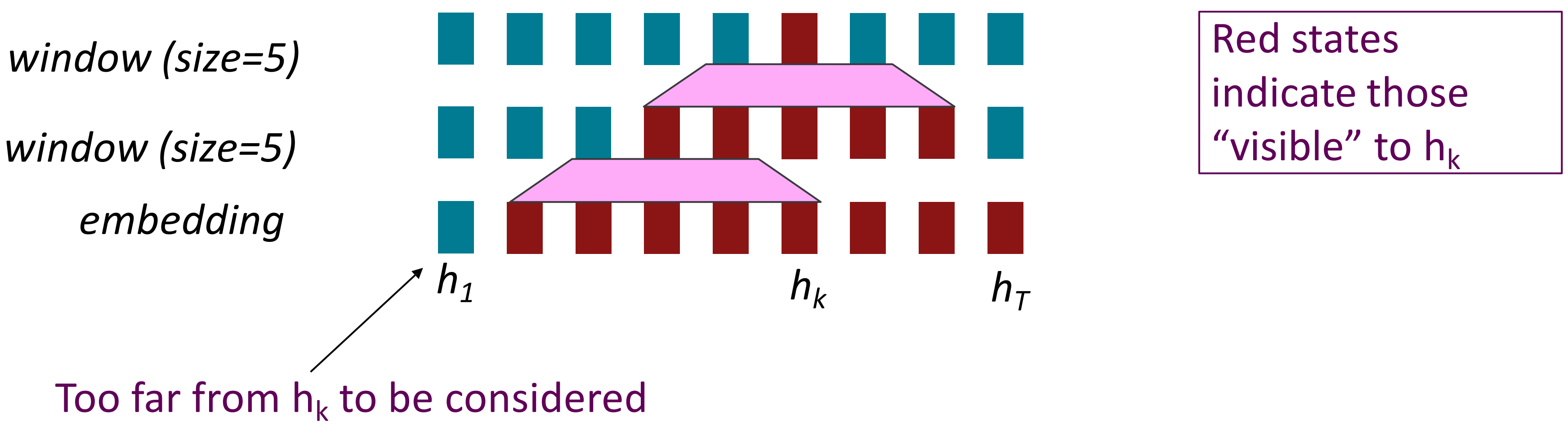


Numbers indicate min # of steps before a state can be computed

Adapted from John Hewitt

If not recurrence, then what?
How about word windows?

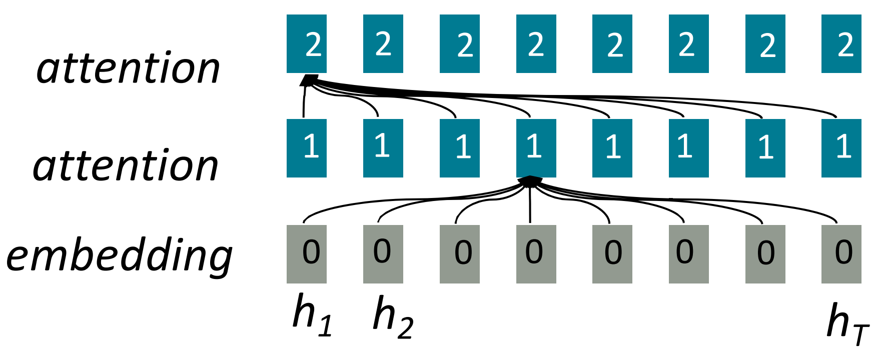
- **Word window models aggregate local contexts**
- What about long-distance dependencies?
  - Stacking word window layers allows interaction between farther words
  - But if your sequences are too long, you'll just ignore long-distance context



Adapted from John Hewitt

If not 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**.
    - If **attention** gives us access to any state… maybe we can just use attention and don't need the RNN?
- Number of unparallelizable operations not tied to sequence length.
- All words interact at every layer!



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

Adapted from John Hewitt

Self-Attention

- Attention operates on **queries**, **keys**, and **values.**
  - We have some **queries** $q_1, q_2, \dots, q_T$. Each query is $q_i \in \mathbb{R}^d$
  - We have some **keys** $k_1, k_2, \dots, k_T$. Each key is $k_i \in \mathbb{R}^d$
  - We have some **values** $v_1, v_2, \dots, v_T$. Each value is $v_i \in \mathbb{R}^d$
- In **self-attention**, the queries, keys, and values are drawn from the same source.
  - For example, if the output of the previous layer is $x_1, \dots, x_T$, (one vec per word) we could let $v_i = k_i = q_i = x_i$ (that is, use the same vectors for all of them!)
- The (dot product) self-attention operation is as follows:

The number of queries can differ from the number of keys and values in practice.

$$e_{ij} = q_i^{\top} k_j$$

Compute **key-query** affinities

$$\alpha_i = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

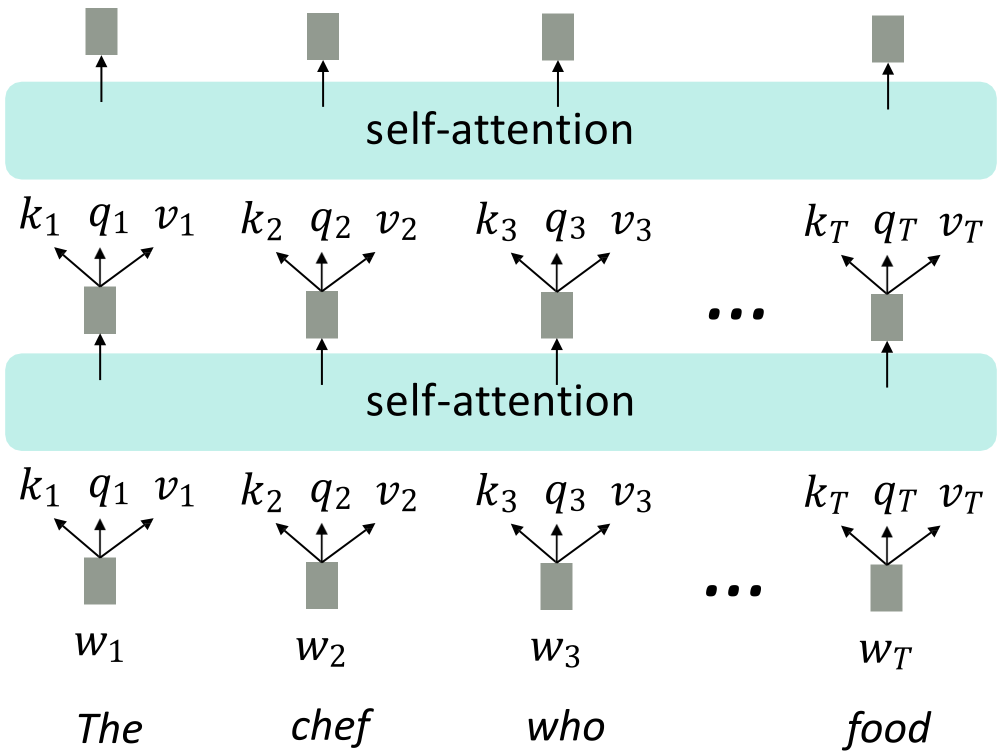
Compute attention weights from affinities (softmax)

$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

Compute outputs as weighted sum of **values**

Self-Attention

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.
- Can self-attention be a drop-in replacement for recurrence?
- No. It has a few issues, which we'll go through.
- First, self-attention is an operation on **sets**. It has no inherent notion of order.



Self-attention doesn't know the order of its inputs.

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**

$p_i \in \mathbb{R}^d$, for $i \in \{1,2,\dots,T\}$ 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 p_i to our inputs!
- Let v_i', k_i', q_i' be our old values, keys, and queries.

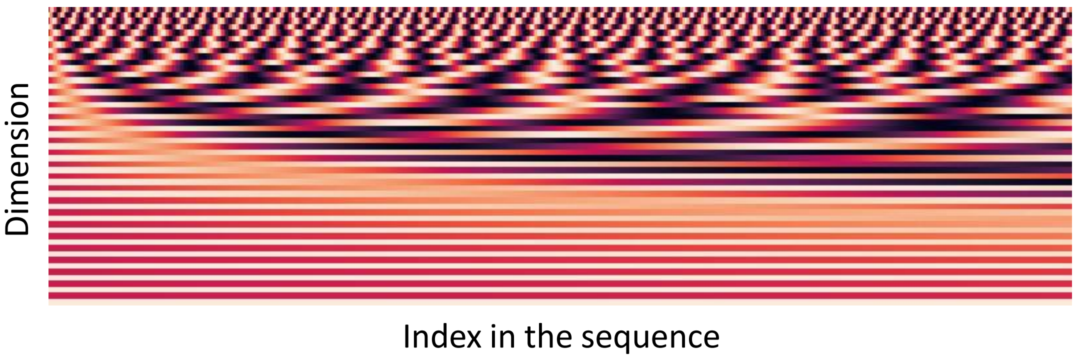
$$v_i = v_i' + p_i$$
$$q_i = q_i' + p_i$$
$$k_i = k_i' + p_i$$

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:

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



- 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/

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

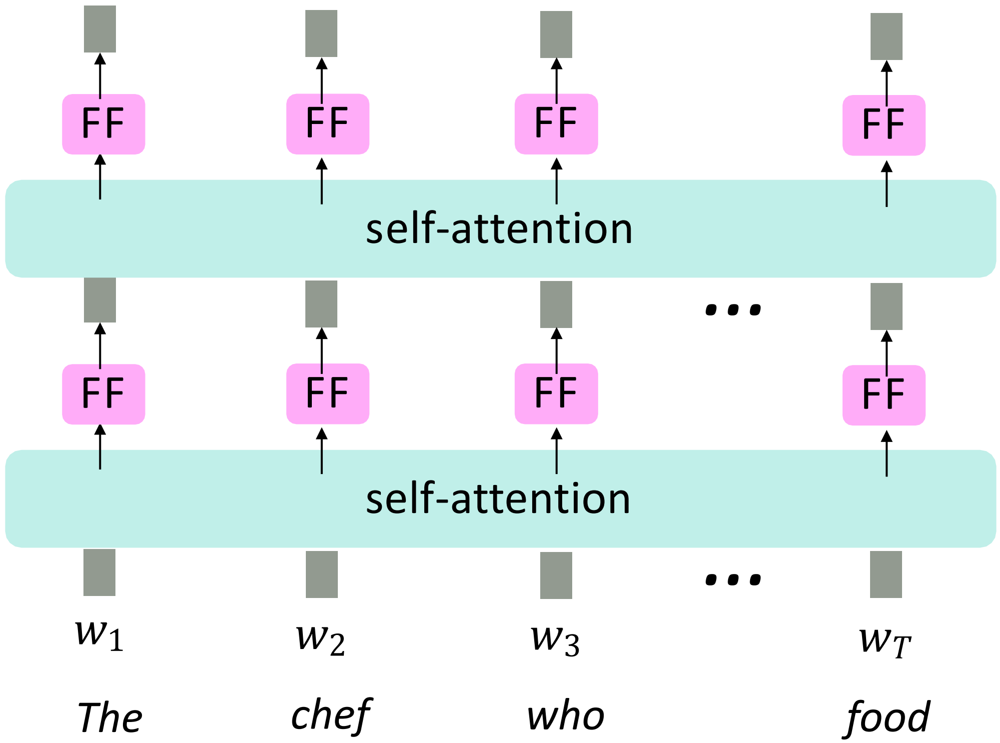
- Add position representations to the inputs

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

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

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



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

Barriers

- 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.

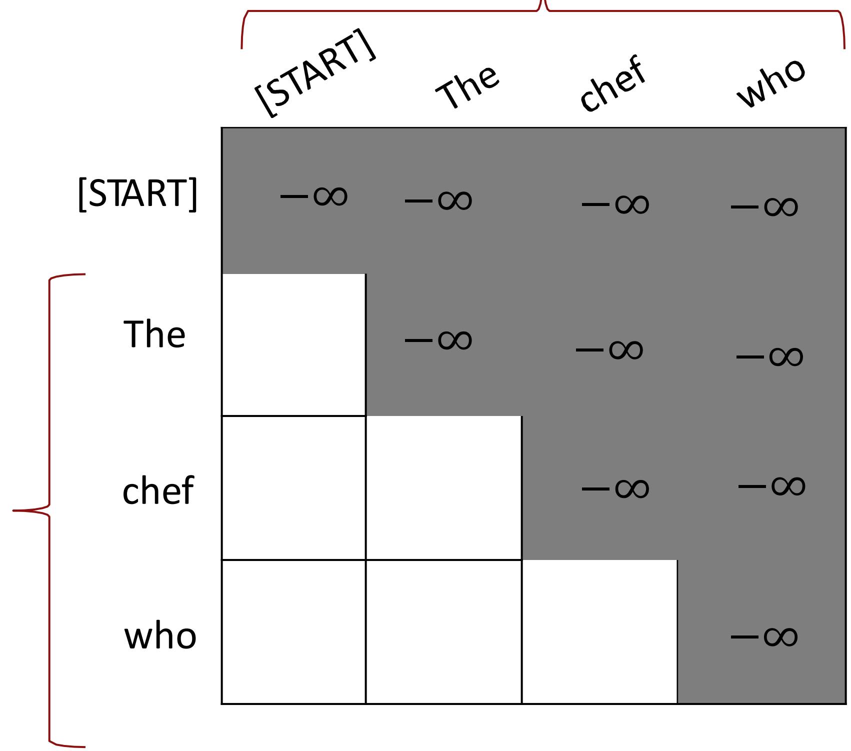
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)

- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

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

We can look at these (not greyed out) words

For encoding these words



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.

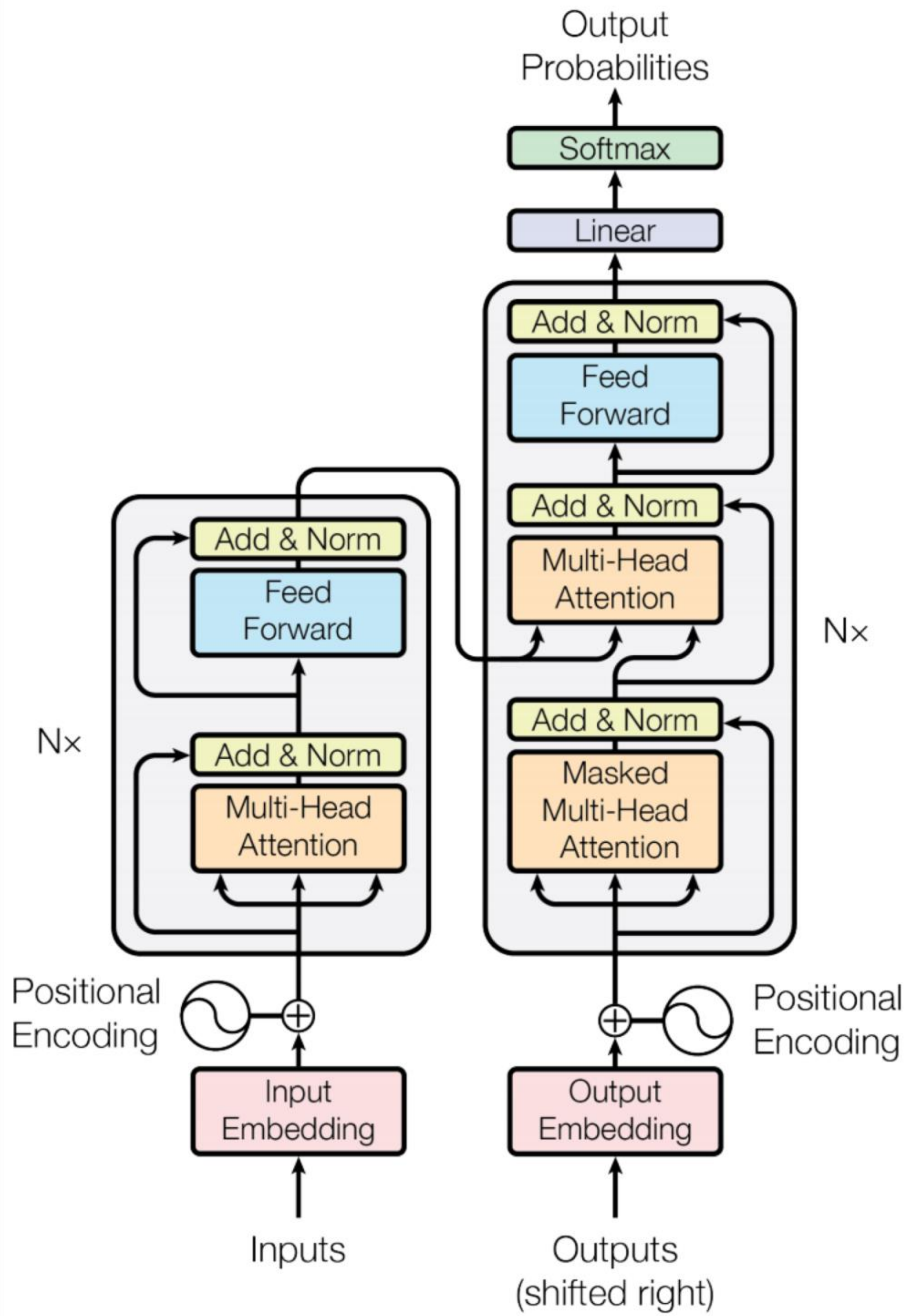
- That’s it! But this is not the **Transformer** model we’ve been hearing about.

Transformer Overview

Attention is all you need. 2017. Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin
https://arxiv.org/pdf/1706.03762.pdf

- Non-recurrent sequence-to-sequence encoder-decoder model
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier

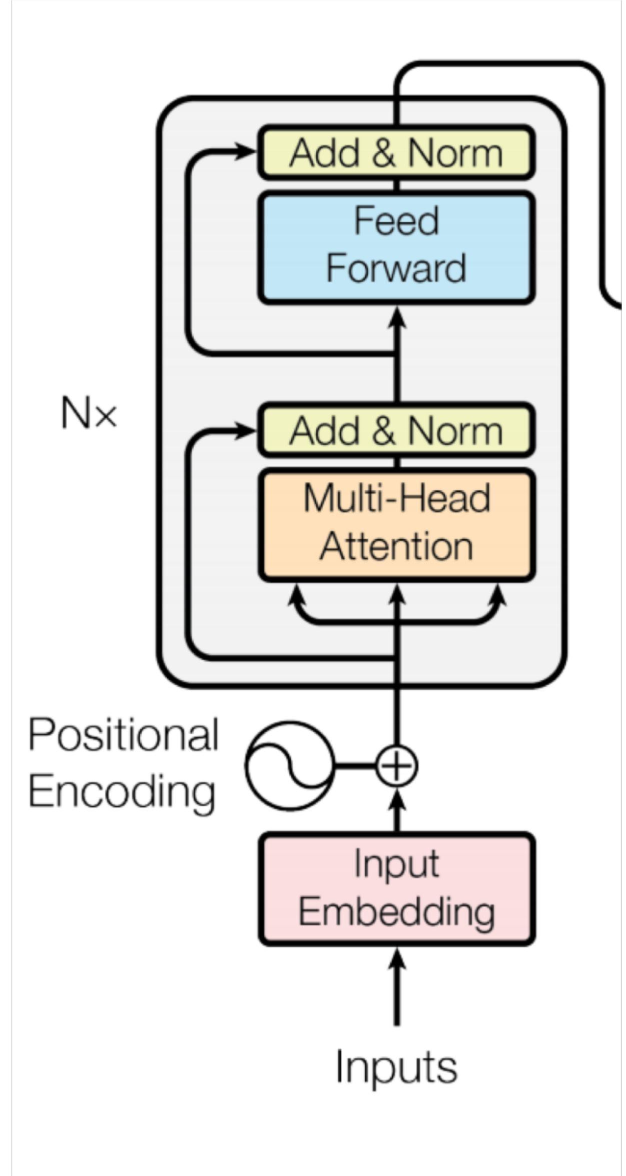
This and related figures from paper ⇑



Transformer Encoder

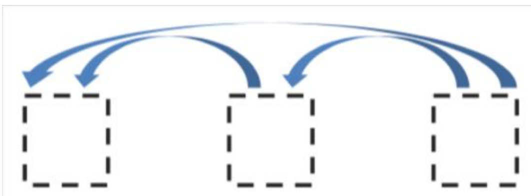
- For encoder, at each block, we use the same Q, K and V
  from the previous layer

- Blocks are repeated 6 times (in vertical stack)

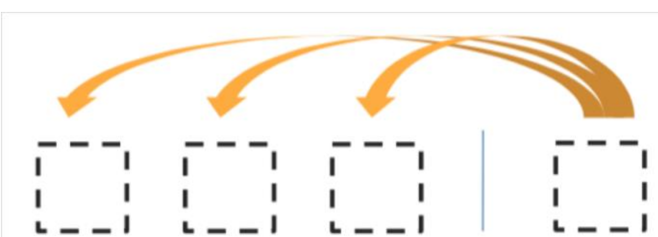


Transformer Decoder

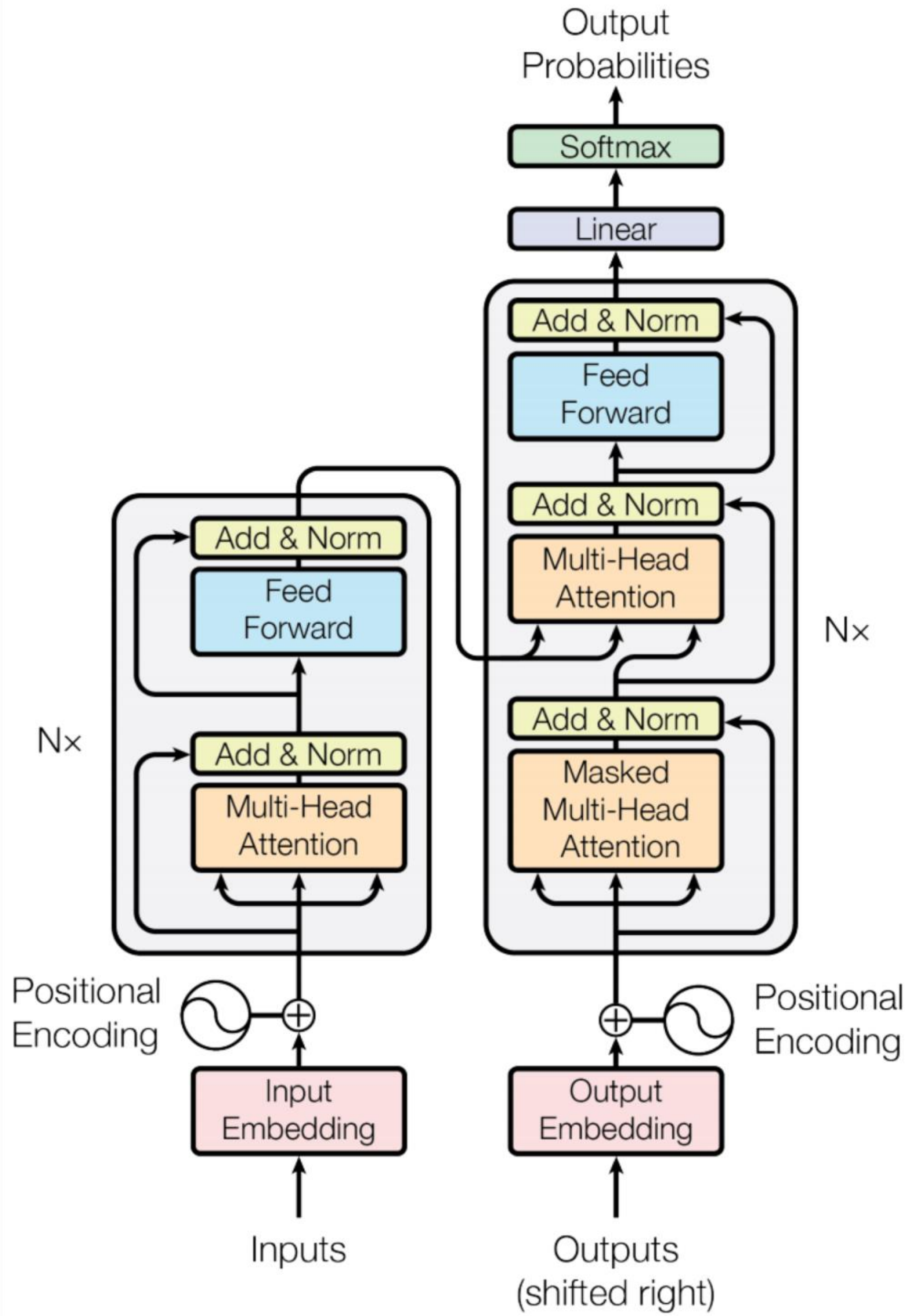
- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:



- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

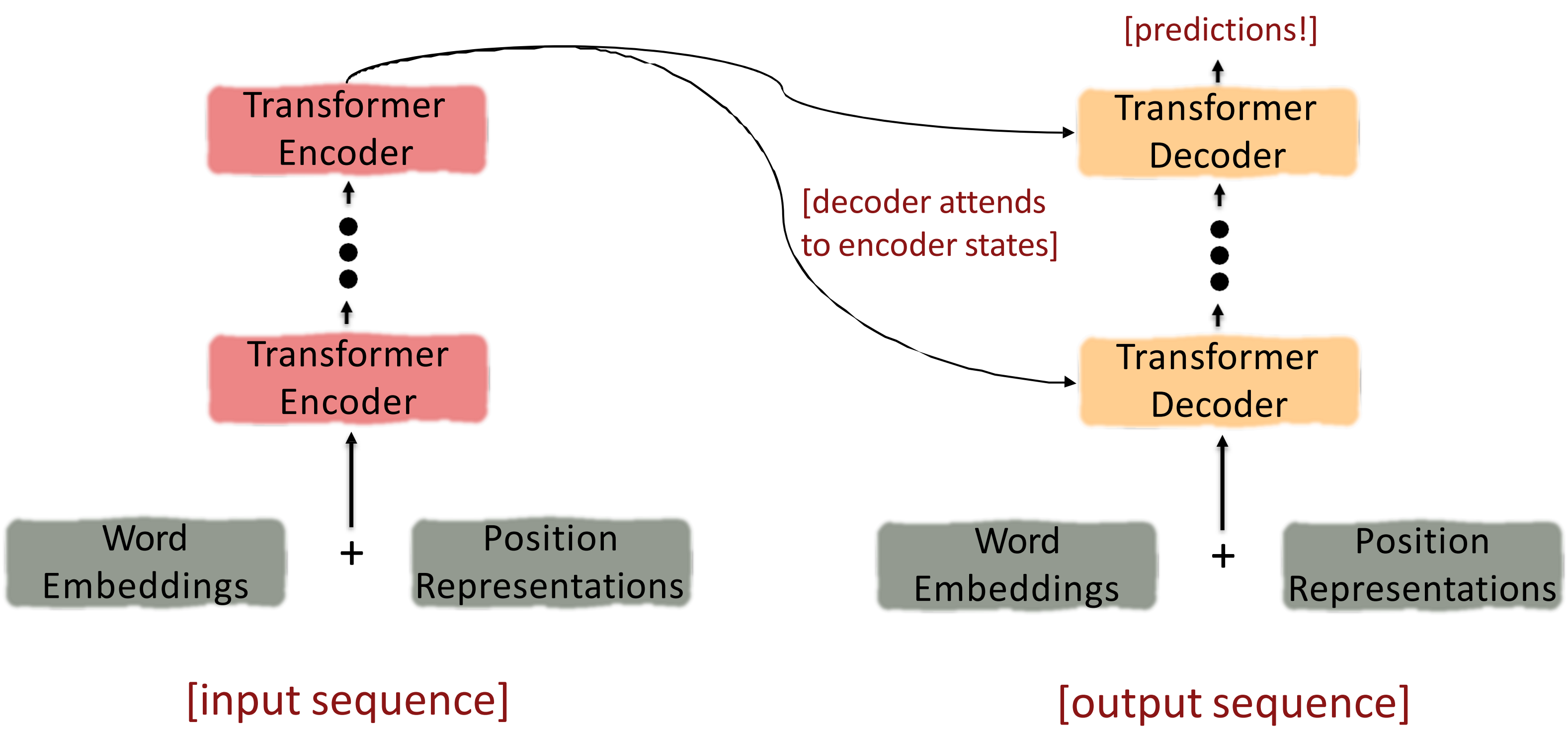


Blocks repeated 6 times also



The Transformer Encoder-Decoder [Vaswani et al., 2017]

Another look at the Transformer Encoder and Decoder Blocks at a high level



The Transformer Encoder-Decoder [Vaswani et al., 2017]

Next, let's look at the Transformer Encoder and Decoder Blocks

What's left in a Transformer Encoder Block that we haven't covered?

1. **Key-query-value attention:** How do we get the k, q, v vectors from a single word embedding?
2. **Multi-headed attention**: Attend to multiple places in a single layer!
3. **Tricks to help with training!**
    1. Residual connections
    2. Layer normalization
    3. Scaling the dot product
    4. These tricks **don't improve** what the model is able to do; they help improve the training process

The Transformer Encoder: **Dot-Product Attention**

- Inputs: a query q and a set of key-value (k-v) pairs to an output
- Query, keys, values, and output are all vectors

- Output is weighted sum of values, where
- Weight of each value is computed by an inner product of query and corresponding key
- Queries and keys have same dimensionality d_k , value have d_v

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

The Transformer Encoder:
Dot-Product Attention – Matrix notation

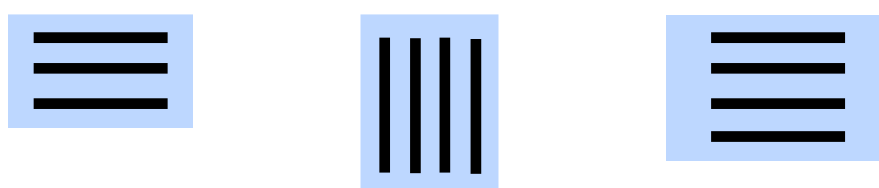
- When we have multiple queries q, we stack them in a matrix Q:

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- Becomes: $A(Q, K, V) = softmax(QK^T)V$

$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$

softmax
row-wise



$= [|Q| \times d_v]$

The Transformer Encoder: **Key-Query-Value Attention**

- We saw that self-attention is when keys, queries, and values come from the same source. The Transformer does this in a particular way:
  - Let $x_1, \dots, x_T$ be input vectors to the Transformer encoder; $x_i \in \mathbb{R}^d$
- Then keys, queries, values are:
  - $k_i = Kx_i$, where $K \in \mathbb{R}^{d\times d}$ is the key matrix.
  - $q_i = Qx_i$, where $\mathrm{Q} \in \mathbb{R}^{d\times d}$ is the query matrix.
  - $v_i = Vx_i$, where $\mathrm{V} \in \mathbb{R}^{d\times d}$ is the value matrix.
- These matrices allow *different aspects* of the x vectors to be used/emphasized in each of the three roles.

The Transformer Encoder: **Key-Query-Value Attention**

- Let's look at how key-query-value attention is computed, in matrices.
  - Let $X = [x_1; \ldots; x_T] \in \mathbb{R}^{T\times d}$ be the concatenation of input vectors.
  - First, note that $XK \in \mathbb{R}^{T\times d}$, $XQ \in \mathbb{R}^{T\times d}$, $XV \in \mathbb{R}^{T\times d}$.
  - The output is defined as $\text{output} = \text{softmax}(XQ(XK)^\top) \times XV$.

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

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

All pairs of attention scores!

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

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

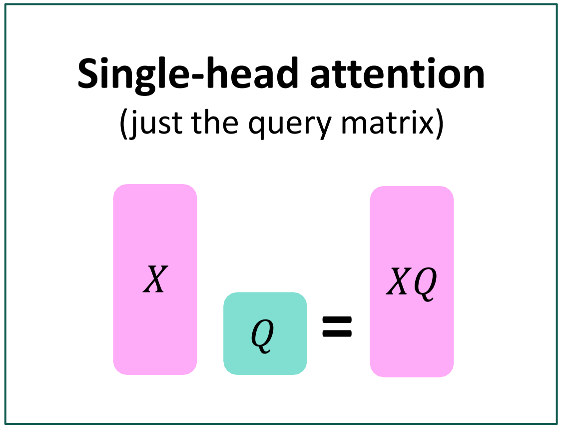
The Transformer Encoder: **Multi-headed attention**

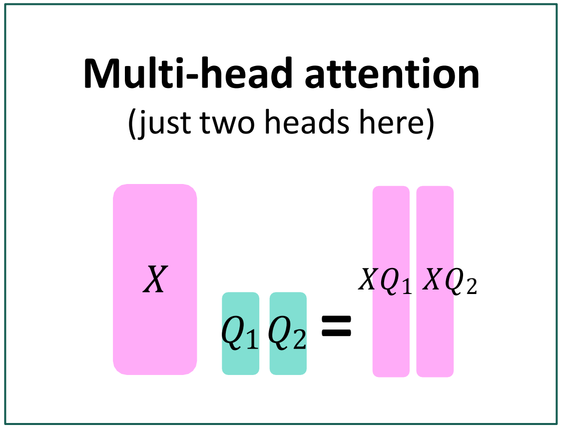
- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices
- Let, $Q_P,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

- Each head gets to "look" at different things, and construct value vectors differently.

The Transformer Encoder: **Multi-headed attention**

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x^\top Q^\top K x_j$ 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, $Q_P, K_P, V_P \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and P ranges from 1 to h.

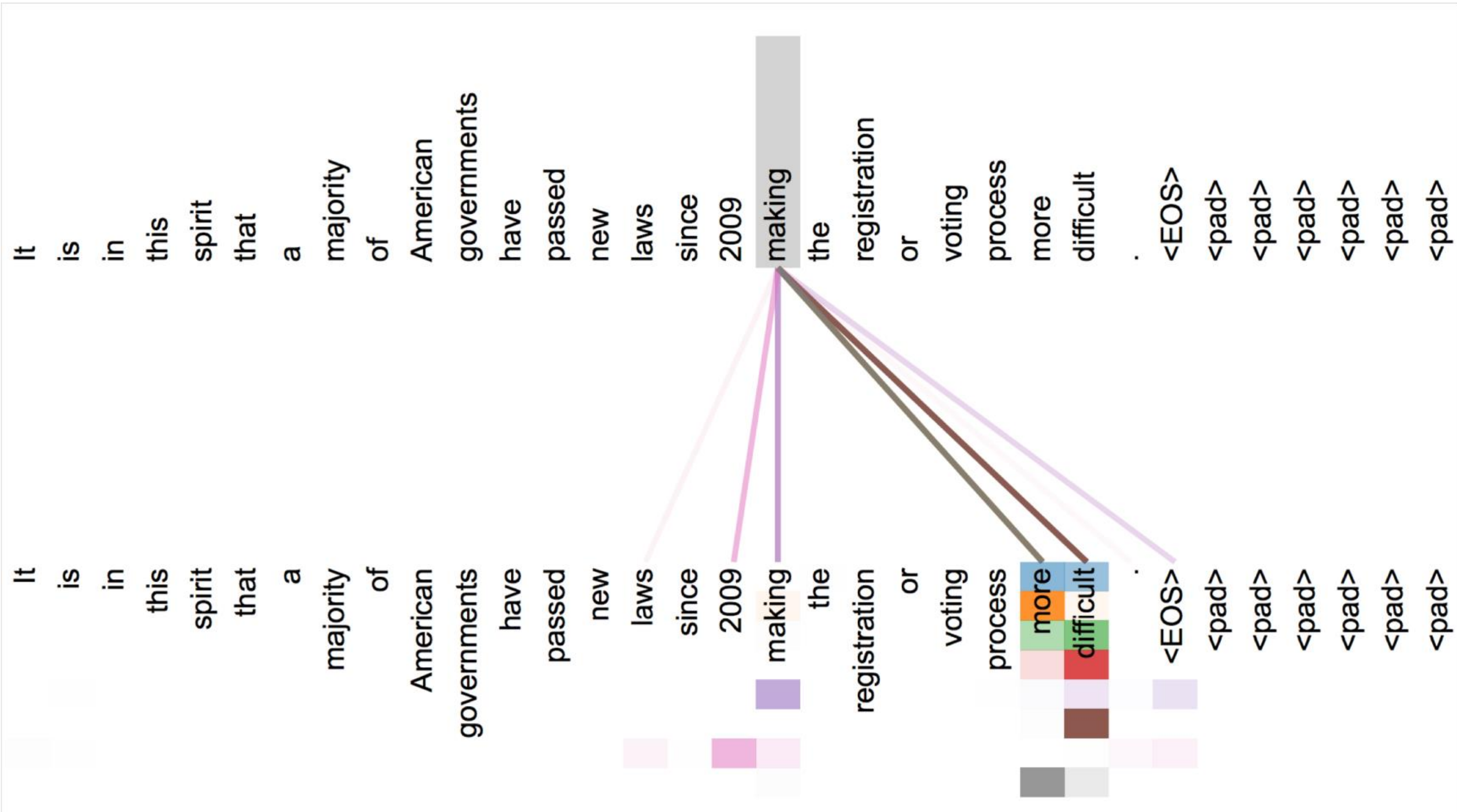




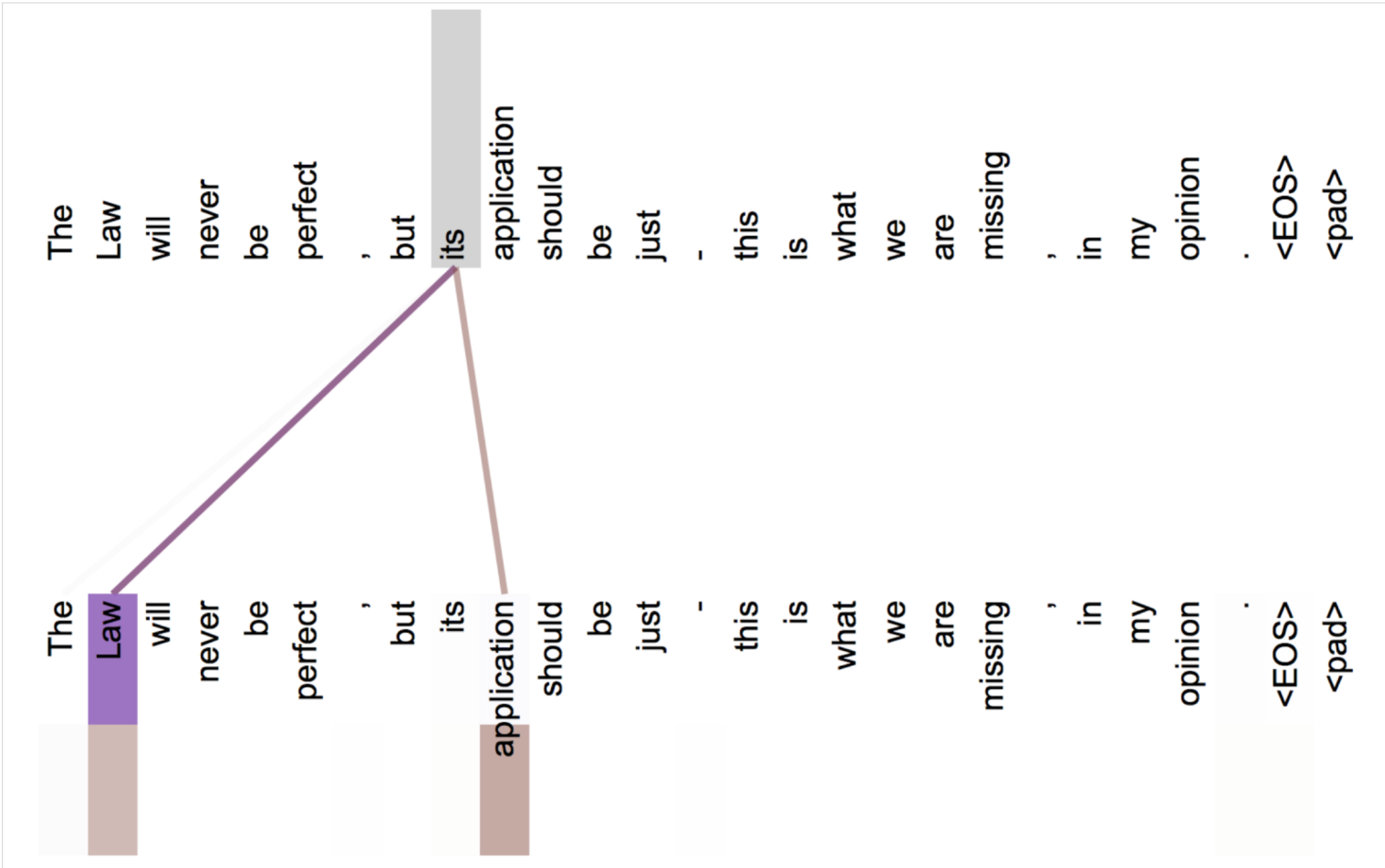
Same amount of computation as single-head self-attention!

Attention visualization in layer 5

- Words start to pay attention to other words in sensible ways

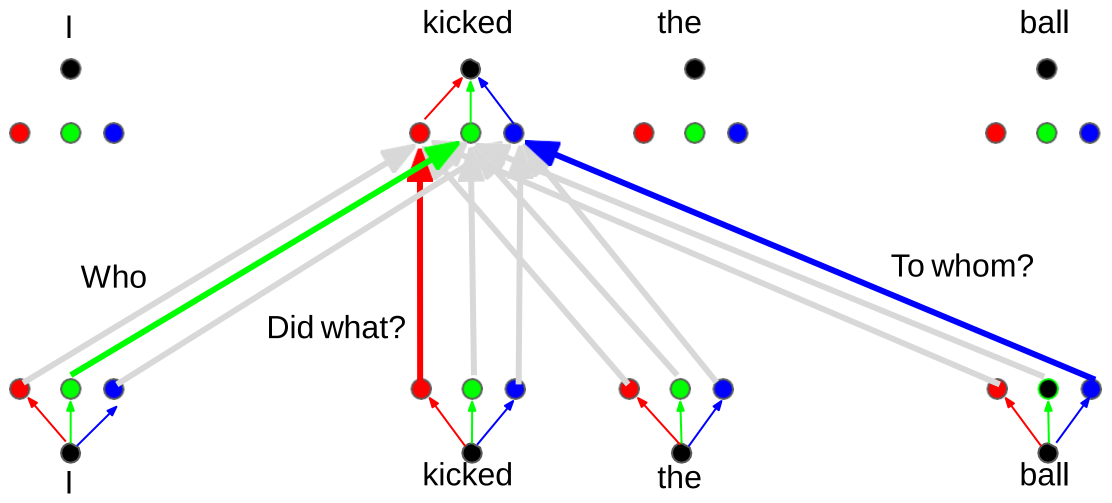


Attention visualization: Implicit anaphora resolution



In 5th layer. Isolated attentions from just the word 'its' for attention heads 5 and 6. Note that the attentions are very sharp for this word.

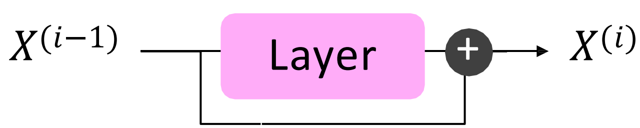
Parallel attention heads



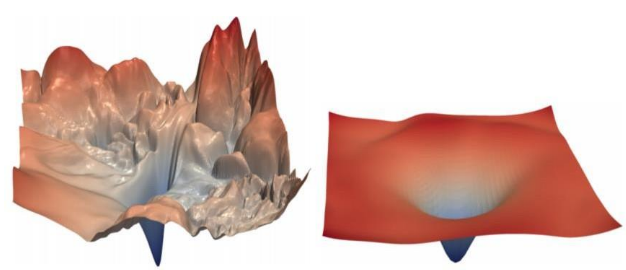
The Transformer Encoder: **Residual connections** [He et al., 2016]

- **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)



  - Residual connections are thought to make the loss landscape considerably smoother (thus easier training!)



[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 $\mu = \Sigma_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
  - Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x - \mu)^2}$; 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 Encoder: **Scaled Dot Product** [Vaswani et al., 2017]

- **"Scaled Dot Product"** attention is a final variation to aid in Transformer training.
- When dimensionality d becomes large, dot products between vectors become large, inputs to the softmax function can be large, making gradients small.

- Instead of the self-attention function we've seen:

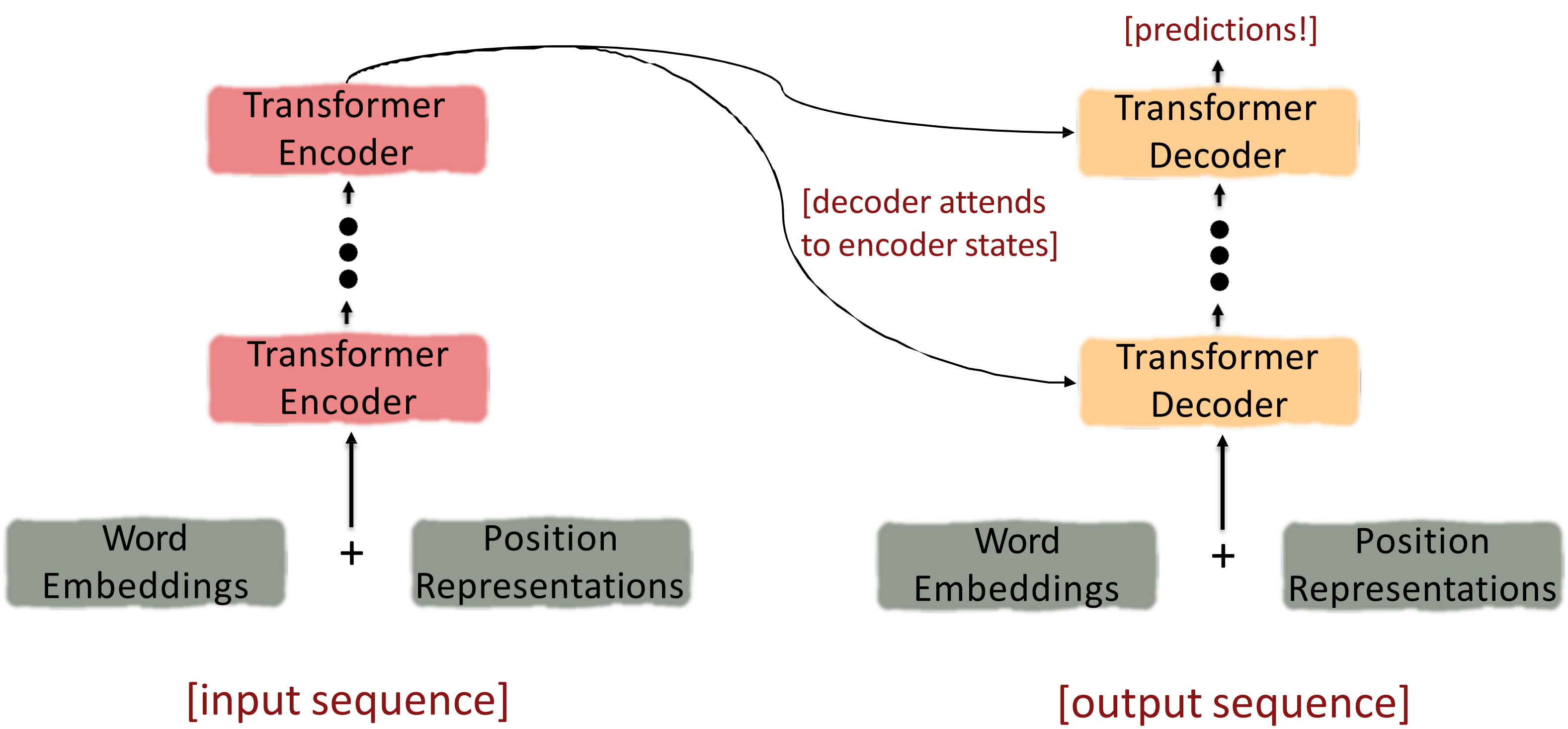
$$\text{output}_P = \text{softmax}\left(XQ_P K_P^\top X^\top\right) * XV_P$$

- 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.)

$$\text{output}_P = \text{softmax}\left(\frac{XQ_P K_P^\top X^\top}{\sqrt{d/h}}\right) * XV_P$$

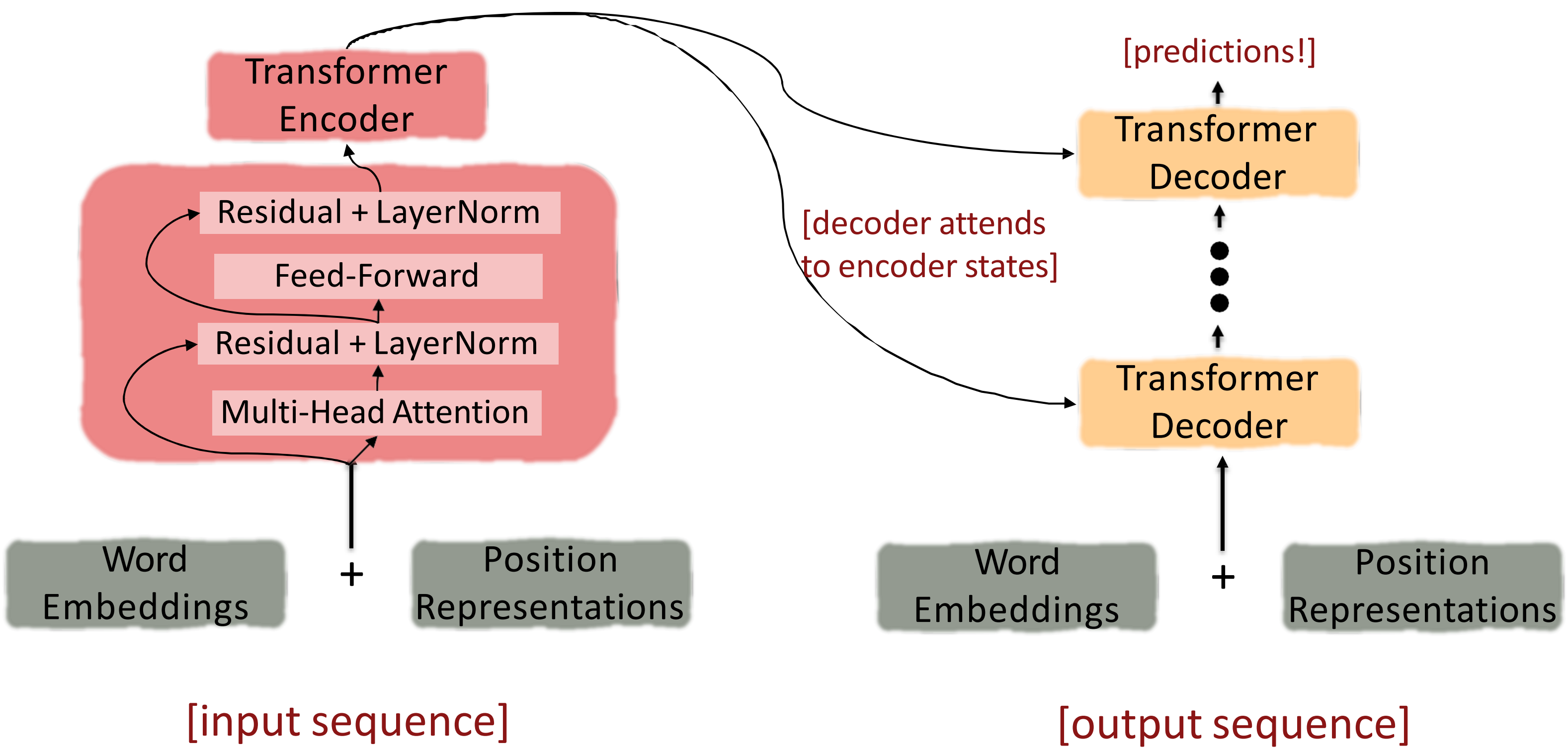
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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



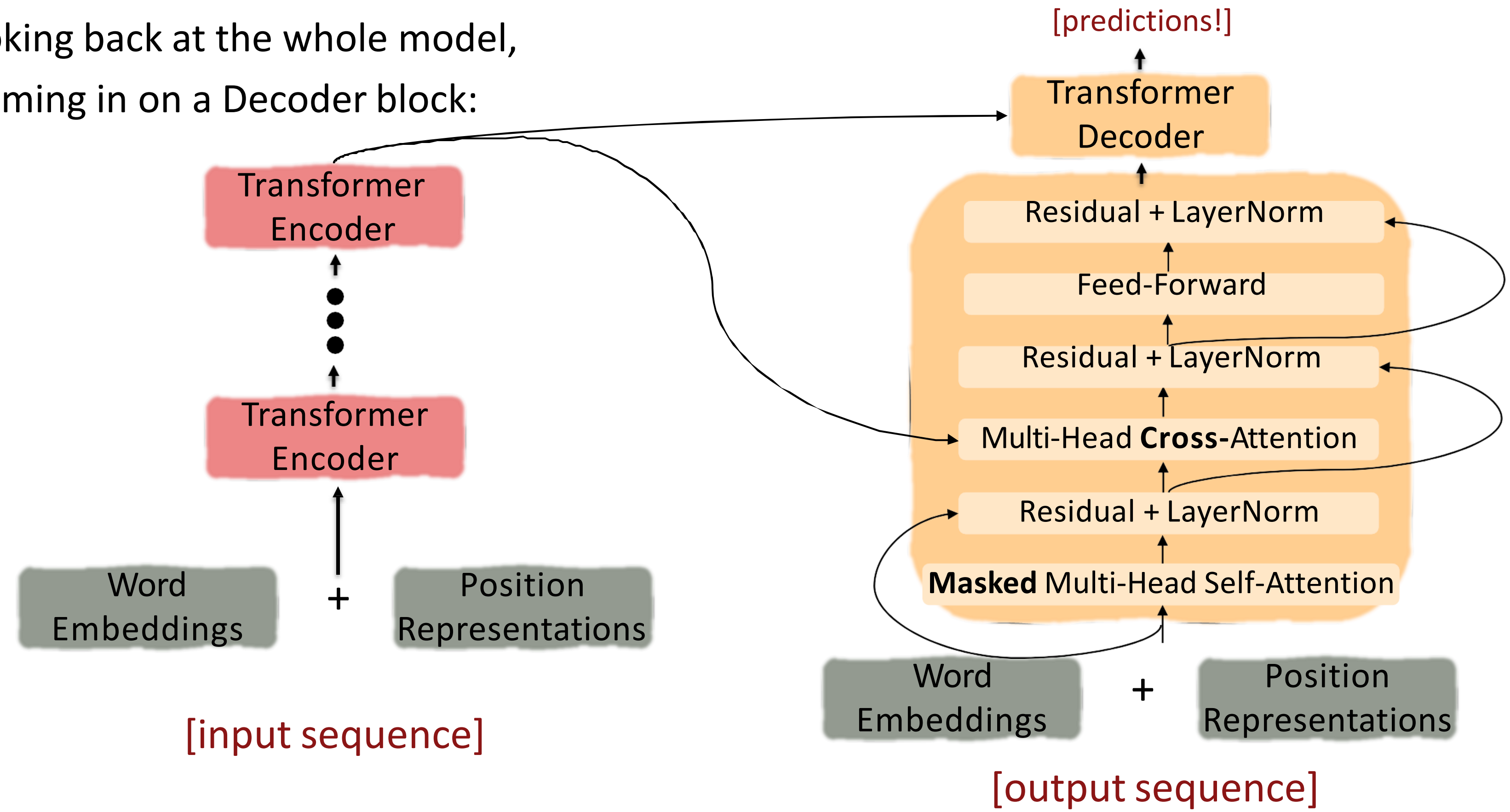
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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



The Transformer Encoder-Decoder [Vaswani et al., 2017]

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



The Transformer Decoder: **Cross-attention (details)**

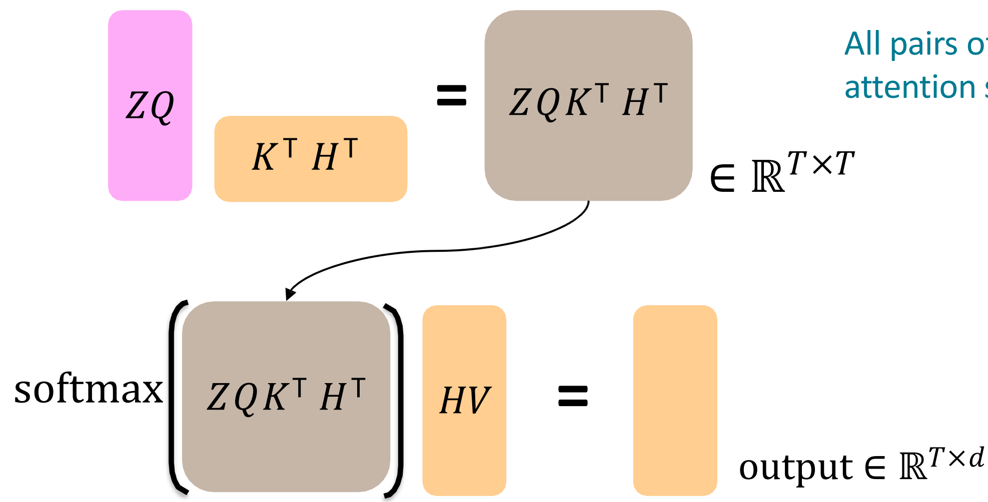
- We saw self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $h_1, \dots, h_T$ be **output** vectors from the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \dots, z_T$ be input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

The Transformer Encoder: **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 $=$ $\text{softmax}(ZQ(HK)\ \ ) \times HV$.

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

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



All pairs of attention scores!

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

What would we like to fix about the Transformer?

- **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!
- **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 function of seq. length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O\left(T^2 d\right)$, where T is the sequence length, and d is the dimensionality.

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

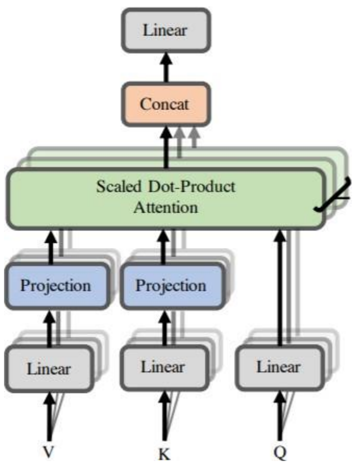
Need to compute all pairs of interactions! $O(T^2 d)$

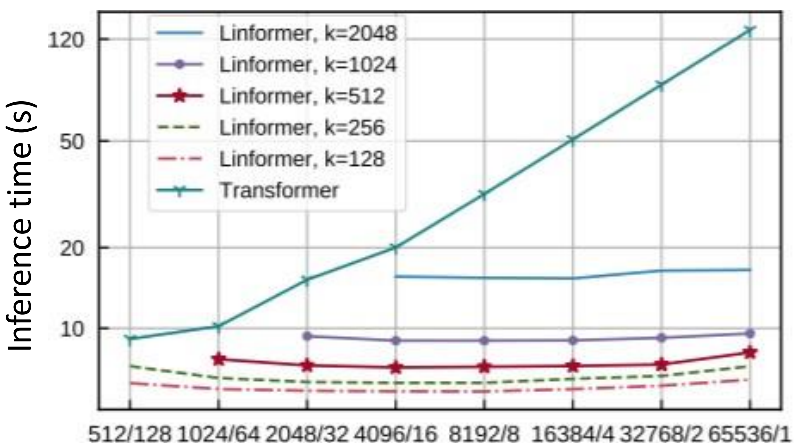
- Think of d as around $\mathbf{1,000}$.
  - So, for a single (shortish) sentence, $T \leq 30; T^2 \leq \mathbf{900}.$
  - In practice, we set a bound like $T = 512$.
  - **But what if we'd like $\boldsymbol{T \geq 10,000}$?** For example, to work on long documents?

Recent 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

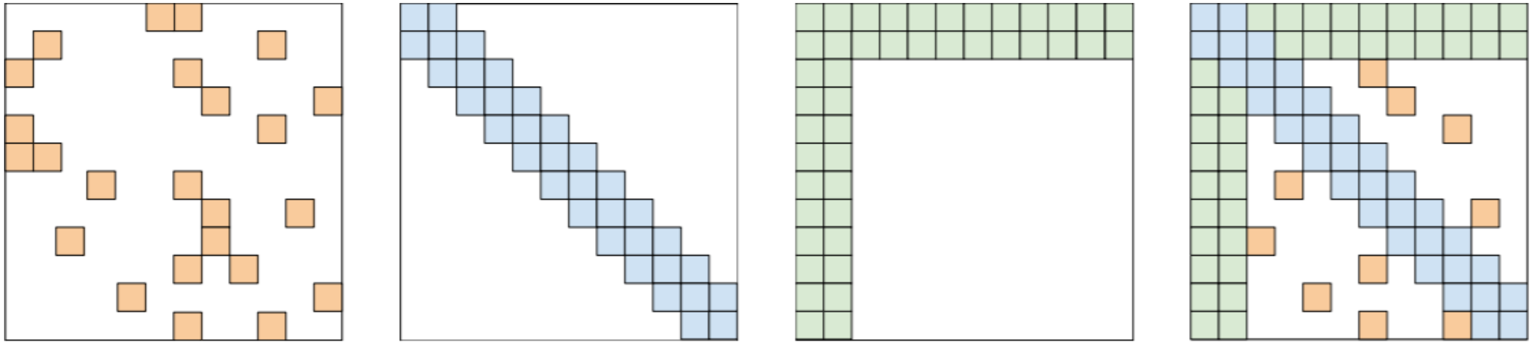




Recent work on improving on quadratic self-attention cost

- 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, **BigBird** [Zaheer et al., 2021]

Key idea: replace all-pairs interactions with a family of other interactions, **like local windows**, **looking at everything**, and **random interactions**.



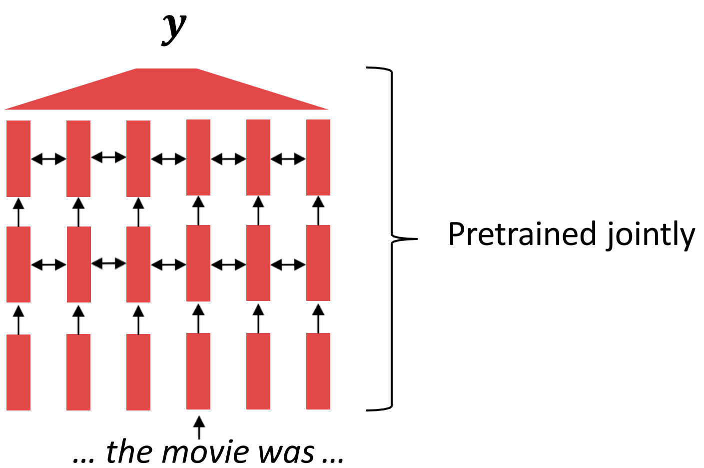
(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

Pretraining 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.

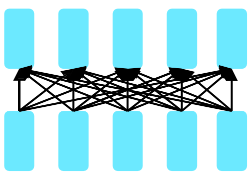


[This model has learned how to represent entire sentences through pretraining]

Adapted from John Hewitt

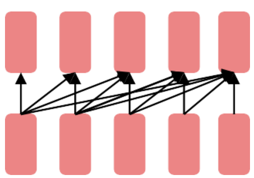
Pretraining for three types of architectures

The neural architecture influences the type of pretraining, and natural use cases.



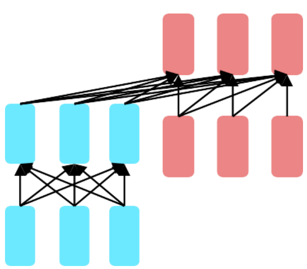
Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?



Decoders

- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words



Encoder-Decoders

- Good parts of decoders and encoders?
- What's the best way to pretrain them?

Adapted from John Hewitt

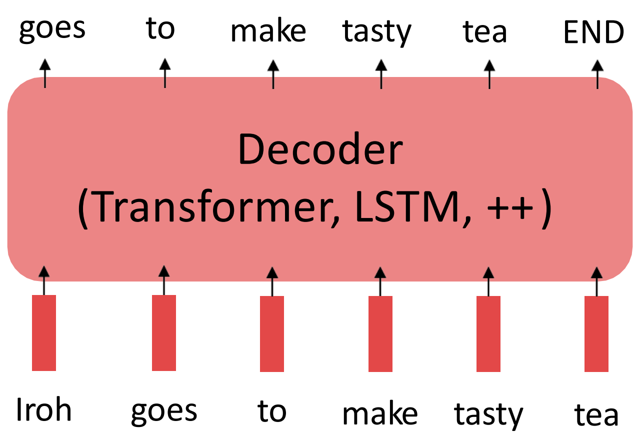
Pretraining through language modeling [Dai and Le, 2015]

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

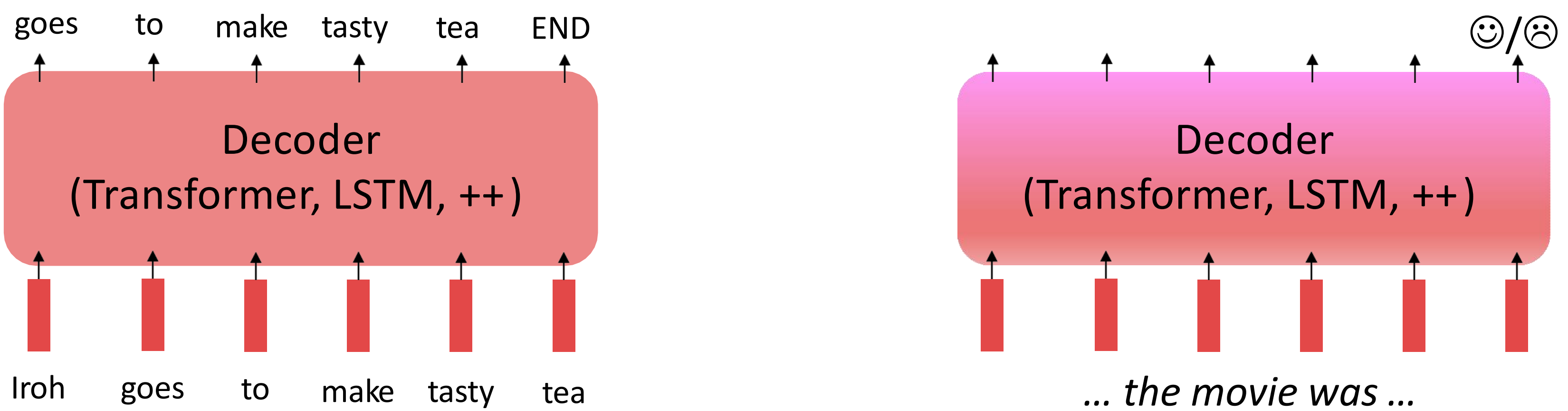
Pretraining can improve NLP applications by serving as parameter initialization.

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!



Capturing meaning via context: What kinds of things does pretraining learn?

There's increasing evidence that pretrained models learn a wide variety of things about the statistical properties of language:

- *Stanford University is located in__________, California.* [Trivia]
- *I put___ fork down on the table.* [syntax]
- *The woman walked across the street, checking for traffic over___ shoulder.* [coreference]
- *I went to the ocean to see the fish, turtles, seals, and_____.* [lexical semantics/topic]
- *Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was___.* [sentiment]
- Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the______. [some reasoning – this is harder]
- I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21,____ [some basic arithmetic; they don't learn the Fibonnaci sequence]
- Models also learn – and can exacerbate racism, sexism, all manner of bad biases.

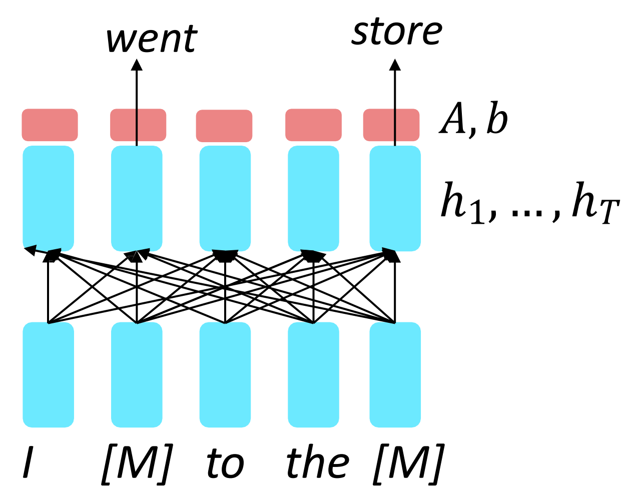
Pretraining encoders: What pretraining objective to use?

So far, we've looked at language model pretraining. But **encoders get bidirectional context,** so we can't do language modeling!

Idea: replace some fraction of words in the input with a special [MASK] token; predict these words.

$$h_1, \dots, h_T = \text{Encoder}(w_1, \dots, w_T)$$
$$y_i \sim Aw_i + b$$

Only add loss terms from words that are "masked out." If x' is the masked version of x, we're learning $p_\theta(x|x')$. Called **Masked LM**.



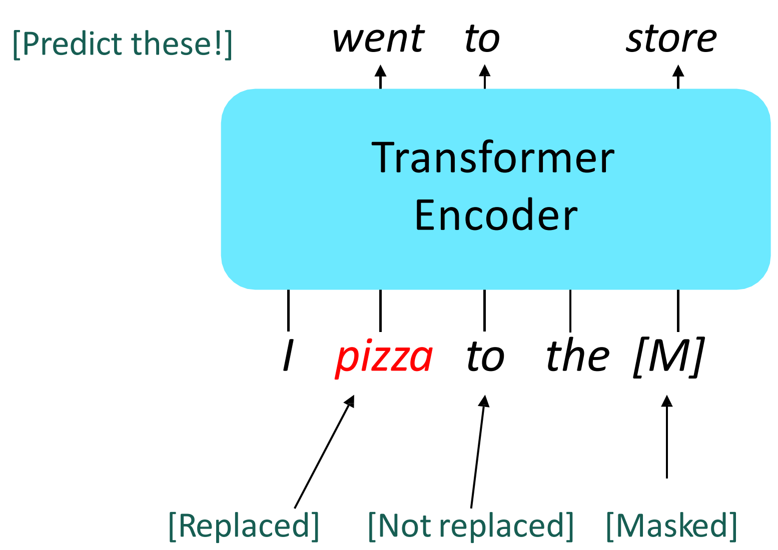
[Devlin et al., 2018]

BERT: Bidirectional Encoder Representations from Tranformers

Devlin et al., 2018 proposed the “Masked LM” objective and **released the weights of a pretrained Transformer**, a model they labeled BERT.

Some more details about Masked LM for BERT:

- Predict a random 15% of (sub)word tokens.
  - Replace input word with [MASK] 80% of the time
  - Replace input word with a random token 10% of the time
  - Leave input word unchanged 10% of the time (but still predict it!)
- Why? Doesn’t let the model get complacent and not build strong representations of non-masked words. (No masks are seen at fine-tuning time!)



[Devlin et al., 2018]

BERT: Bidirectional Encoder Representations from Tranformers

- Mask out *k*% of the input words, and then predict the masked words
  - They always use k = 15%

store ↑ gallon ↑

the man went to the [MASK] to buy a [MASK] of milk

- Too little masking: Too expensive to train
- Too much masking: Not enough context

BERT: Bidirectional Encoder Representations from Tranformers

- Additional task: Next sentence prediction
- To learn *relationships* between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

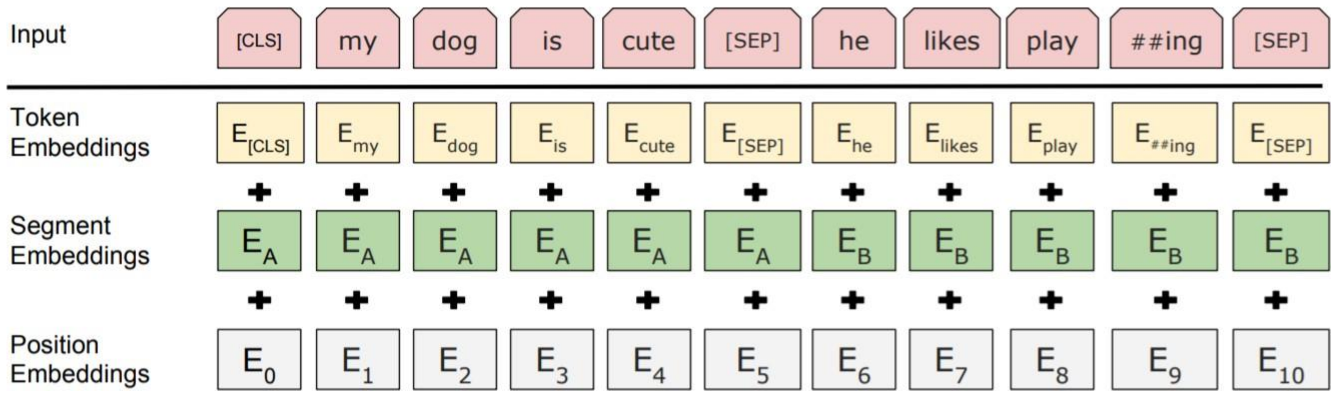
Sentence A = `The man went to the store.`
Sentence B = `He bought a gallon of milk.`
Label = `IsNextSentence`

Sentence A = `The man went to the store.`
Sentence B = `Penguins are flightless.`
Label = `NotNextSentence`

Adapted from Christopher Manning

BERT: Bidirectional Encoder Representations from Tranformers

- The pretraining input to BERT was two separate contiguous chunks of text:



- In addition to masked input reconstruction, BERT was trained to predict whether one chunk follows the other or is randomly sampled.
- Later work has argued this “next sentence prediction” is not necessary.

[Devlin et al., 2018, Liu et al., 2019]

Adapted from John Hewitt

BERT: Bidirectional Encoder Representations from Tranformers

Details about BERT

- Two models were released:
  - BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
  - BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.
- Trained on:
  - BooksCorpus (800 million words)
  - English Wikipedia (2,500 million words)
- Pretraining is expensive and impractical on a single GPU.
  - BERT was pretrained with 64 TPU chips for a total of 4 days.
  - (TPUs are special tensor operation acceleration hardware)
- Finetuning is practical and common on a single GPU
  - "Pretrain once, finetune many times."

[Devlin et al., 2018]

BERT: Bidirectional Encoder Representations from Tranformers

BERT was massively popular and hugely versatile; finetuning BERT led to new state-of- the-art results on a broad range of tasks.

- **QQP:** Quora Question Pairs (detect paraphrase questions)
- **QNLI**: natural language inference over question answering data
- **SST-2**: sentiment analysis
- **CoLA**: corpus of linguistic acceptability (detect whether sentences are grammatical.)
- **STS-B**: semantic textual similarity
- **MRPC**: microsoft paraphrase corpus
- **RTE**: small natural language inference corpus

System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	**Average**
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

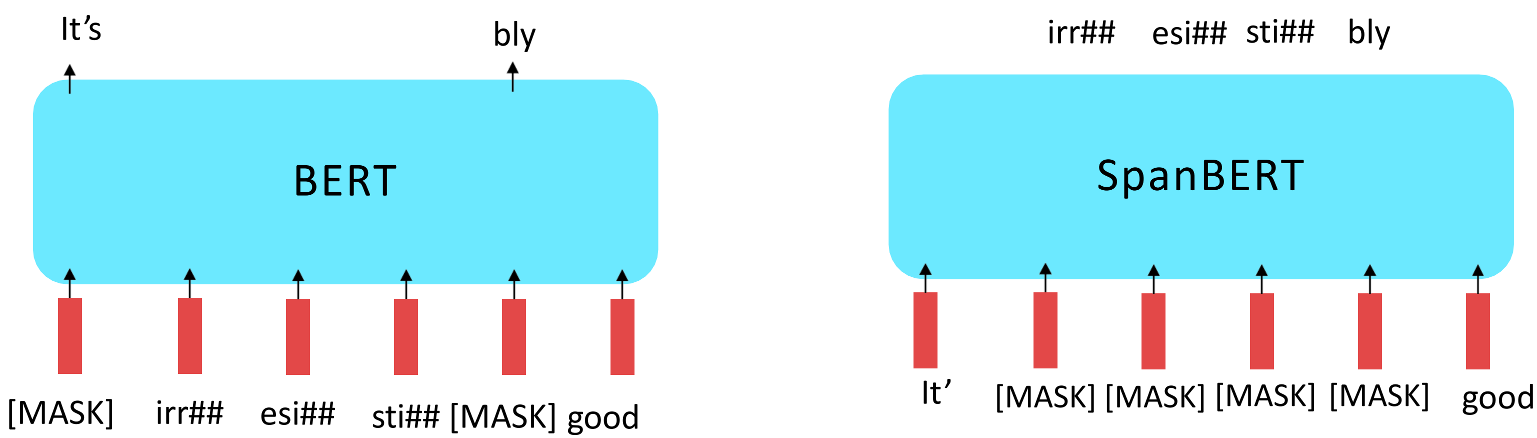
[Devlin et al., 2018]

Extensions of BERT

You'll see a lot of BERT variants like RoBERTa, SpanBERT, +++

Some generally accepted improvements to the BERT pretraining formula:

- RoBERTa: mainly just train BERT for longer and remove next sentence prediction!
- SpanBERT: masking contiguous spans of words makes a harder, more useful pretraining task



[Liu et al., 2019; Joshi et al., 2020]

Extensions of BERT

A takeaway from the RoBERTa paper: more compute, more data can improve pretraining even when not changing the underlying Transformer encoder.

Model	data	bsz	steps	SQuAD (v1.1/2.0)	MNLI-m	SST-2
RoBERTa						
with BOOKS + WIKI	16GB	8K	100K	93.6/87.3	89.0	95.3
+ additional data (§3.2)	160GB	8K	100K	94.0/87.7	89.3	95.6
+ pretrain longer	160GB	8K	300K	94.4/88.7	90.0	96.1
+ pretrain even longer	160GB	8K	500K	**94.6/89.4**	**90.2**	**96.4**
$\text{BERT}_{\text{LARGE}}$						
with BOOKS + WIKI	13GB	256	1M	90.9/81.8	86.6	93.7

[Liu et al., 2019; Joshi et al., 2020]

Pretraining decoders

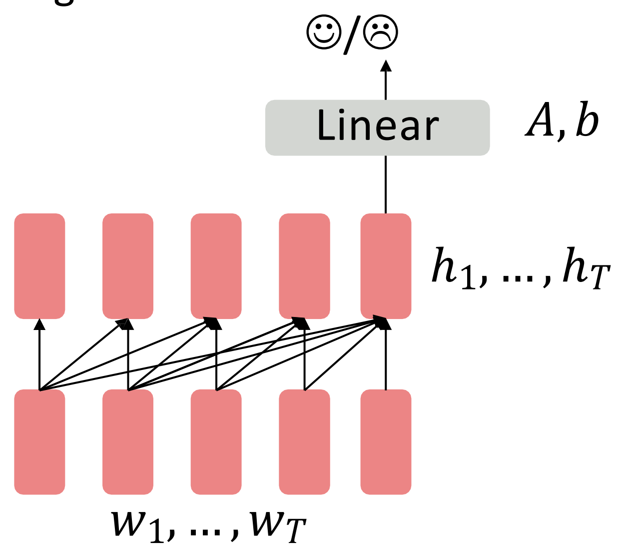
When using language model pretrained decoders, we can ignore that they were trained to model $p\left(w_t|w_{1:t-1}\right)$.

We can finetune them by training a classifier on the last word's hidden state.

$$h_1, \dots, h_T = \text{Decoder}(w_1, \dots, w_T)$$
$$y \sim Aw_T + b$$

Where A and b are randomly initialized and specified by the downstream task.

Gradients backpropagate through the whole network.



[Note how the linear layer hasn't been pretrained and must be learned from scratch.]

Pretraining decoders

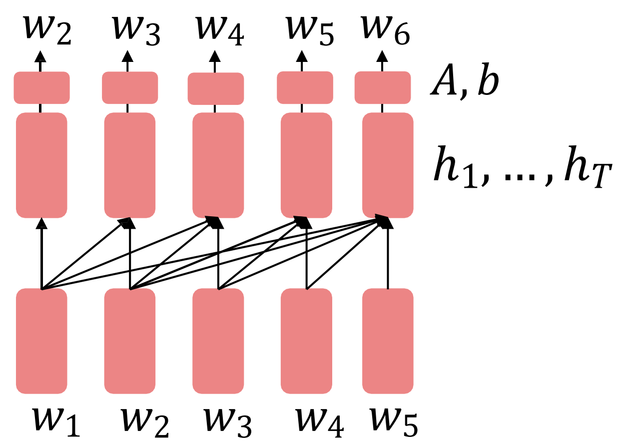
It's natural to pretrain decoders as language models and then use them as generators, finetuning their $p_\theta(w_t|w_{1:t-1})$!

This is helpful in tasks **where the output is a sequence** with a vocabulary like that at pretraining time!

- Dialogue (context=dialogue history)
- Summarization (context=document)

$$h_1, \dots, h_T = \text{Decoder}(w_1, \dots, w_T)$$
$$w_t \sim A w_{t-1} + b$$

Where A, b were pretrained in the language model!



[Note how the linear layer has been pretrained.]

Generative Pretrained Transformer (GPT) [Radford et al., 2018]

2018's GPT was a big success in pretraining a decoder!

- Transformer decoder with 12 layers.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained on BooksCorpus: over 7000 unique books.
  - Contains long spans of contiguous text, for learning long-distance dependencies.
- The acronym "GPT" never showed up in the original paper; it could stand for "Generative PreTraining" or "Generative Pretrained Transformer"

[Devlin et al., 2018]

Generative Pretrained Transformer (GPT) [Radford et al., 2018]

How do we format inputs to our decoder for **finetuning tasks?**

Natural Language Inference: Label pairs of sentences as *entailing/contradictory/neutral*

Premise: *The man is in the doorway*
Hypothesis: *The person is near the door*
entailment

Radford et al., 2018 evaluate on natural language inference.
Here's roughly how the input was formatted, as a sequence of tokens for the decoder.

[START] *The man is in the doorway* [DELIM] *The person is near the door* [EXTRACT]

The linear classifier is applied to the representation of the [EXTRACT] token.

Generative Pretrained Transformer (GPT) [Radford et al., 2018]

GPT results on various *natural language inference* datasets.

Method	MNLI-m	MNLI-mm	SNLI	SciTail	QNLI	RTE
ESIM + ELMo [44] (5x)	-	-	<u>89.3</u>	-	-	-
CAFE [58] (5x)	80.2	79.0	<u>89.3</u>	-	-	-
Stochastic Answer Network [35] (3x)	<u>80.6</u>	<u>80.1</u>	-	-	-	-
CAFE [58]	78.7	77.9	88.5	<u>83.3</u>		
GenSen [64]	71.4	71.3	-	-	<u>82.3</u>	59.2
Multi-task BiLSTM + Attn [64]	72.2	72.1	-	-	82.1	**61.7**
Finetuned Transformer LM (ours)	**82.1**	**81.4**	**89.9**	**88.3**	**88.1**	56.0

Increasingly convincing generations (GPT2) [Radford et al., 2018]

We mentioned how pretrained decoders can be used **in their capacities as language models.** **GPT-2,** a larger version of GPT trained on more data, was shown to produce relatively convincing samples of natural language.

> **Context (human-written):** In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.
>
> **GPT-2:** The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.
>
> Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.
>
> Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Aside: 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)	

Aside: 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.

Conjugation of *-ambia* [less ▲]

Non-finite forms		
Form	Positive	Negative
Infinitive	kuambia	kutoambia
Simple finite forms		
Positive form	Singular	Plural
Imperative	ambia	ambieni
Habitual	huambia	

Complex finite forms

Polarity	Persons: 1st Sg.	1st Pl.	2nd Sg.	2nd Pl.	Persons / Classes: 3rd / M-wa Sg. / 1	Pl. / 2	Classes: M-mi 3	4	Ma 5	6	Ki-vi 7	8	N 9	10	U 11 / 14	Ku 15 / 17	Pa 16	Mu 18
Past [less ▲]																		
Positive	niliambia naliambia	tuliambia twaliambia	uliambia waliambia	mliambia mwaliambia	aliambia	waliambia	uliambia	iliambia	liliambia	yaliambia	kiliambia	viliambia	iliambia	ziliambia	uliambia	kuliambia	paliambia	muliambia
Negative	sikuambia	hatukuambia	hukuambia	hamkuambia	hakuambia	hawakuambia	haukuambia	haikuambia	halikuambia	hayakuambia	hakikuambia	havikuambia	haikuambia	hazikuambia	haukuambia	hakukuambia	hapakuambia	hamukuambia
Present [less ▲]																		
Positive	ninaambia naambia	tunaambia	unaambia	mnaambia	anaambia	wanaambia	unaambia	inaambia	linaambia	yanaambia	kinaambia	vinaambia	inaambia	zinaambia	unaambia	kunaambia	panaambia	munaambia
Negative	siambii	hatuambii	huambii	hamambii	haambii	hawaambii	hauambii	haiambii	haliambii	hayaambii	hakiambii	haviambii	haiambii	haziambii	hauambii	hakuambii	hapaambii	hamuambii
Future [less ▲]																		
Positive	nitaambia	tutaambia	utaambia	mtaambia	ataambia	wataambia	utaambia	itaambia	litaambia	yataambia	kitaambia	vitaambia	itaambia	zitaambia	utaambia	kutaambia	pataambia	mutaambia
Negative	sitaambia	hatutaambia	hutaambia	hamtaambia	hataambia	hawataambia	hautaambia	haitaambia	halitaambia	hayataambia	hakitaambia	havitaambia	haitaambia	hazitaambia	hautaambia	hakutaambia	hapataambia	hamutaambia
Subjunctive [less ▲]																		
Positive	niambie	tuambie	uambie	mambie	aambie	waambie	uambie	iambie	liambie	yaambie	kiambie	viambie	iambie	ziambie	uambie	kuambie	paambie	muambie
Negative	nisiambie	tusiambie	usiambie	msiambie	asiambie	wasiambie	usiambie	isiambie	lisiambie	yasiambie	kisiambie	visiambie	isiambie	zisiambie	usiambie	kusiambie	pasiambie	musiambie
Present Conditional [less ▲]																		
Positive	ningeambia	tungeambia	ungeambia	mngeambia	angeambia	wangeambia	ungeambia	ingeambia	lingeambia	yangeambia	kingeambia	vingeambia	ingeambia	zingeambia	ungeambia	kungeambia	pangeambia	mungeambia
Negative	nisingeambia singeambia	tusingeambia hatungeambia	usingeambia hungeambia	msingeambia hamngeambia	asingeambia hangeambia	wasingeambia hawangeambia	usingeambia haungeambia	isingeambia haingeambia	lisingeambia halingeambia	yasingeambia hayangeambia	kisingeambia hakingeambia	visingeambia havingeambia	isingeambia haingeambia	zisingeambia hazingeambia	usingeambia haungeambia	kusingeambia hakungeambia	pasingeambia hapangeambia	musingeambia hamungeambia
Past Conditional [less ▲]																		
Positive	ningaliambia	tungaliambia	ungaliambia	mngaliambia	angaliambia	wangaliambia	ungaliambia	ingaliambia	lingaliambia	yangaliambia	kingaliambia	vingaliambia	ingaliambia	zingaliambia	ungaliambia	kungaliambia	pangaliambia	mungaliambia
Negative	nisingaliambia singaliambia	tusingaliambia hatungaliambia	usingaliambia hungaliambia	msingaliambia hamngaliambia	asingaliambia hangaliambia	wasingaliambia hawangaliambia	usingaliambia haungaliambia	isingaliambia haingaliambia	lisingaliambia halingaliambia	yasingaliambia hayangaliambia	kisingaliambia hakingaliambia	visingaliambia havingaliambia	isingaliambia haingaliambia	zisingaliambia hazingaliambia	usingaliambia haungaliambia	kusingaliambia hakungaliambia	pasingaliambia hapangaliambia	musingaliambia hamungaliambia
Conditional Contrary to Fact [less ▲]																		
Positive	ningeliambia	tungeliambia	ungeliambia	mngeliambia	angeliambia	wangeliambia	ungeliambia	ingeliambia	lingeliambia	yangeliambia	kingeliambia	vingeliambia	ingeliambia	zingeliambia	ungeliambia	kungeliambia	pangeliambia	mungeliambia
Gnomic [less ▲]																		
Positive	naambia	twaambia	waambia	mwaambia	aambia	waambia	waambia	yaambia	laambia	yaambia	chaambia	vyaambia	yaambia	zaambia	waambia	kwaambia	paambia	mwaambia
Perfect [less ▲]																		

[Wiktionary]

Aside: 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.

[Sennrich et al., 2016, Wu et al., 2016]

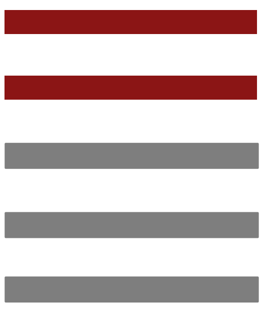
Aside: Word structure and subword models

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
Common words	hat	hat
	learn	learn
Variations	taaaaasty	taa## aaa## sty
misspellings	laern	la## ern##
novel items	Transformerify	Transformer## ify

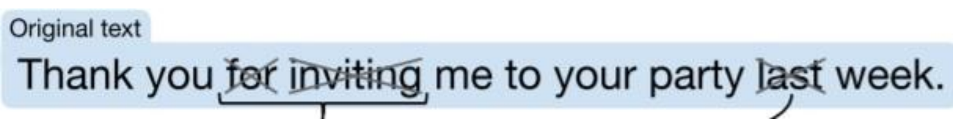
embedding



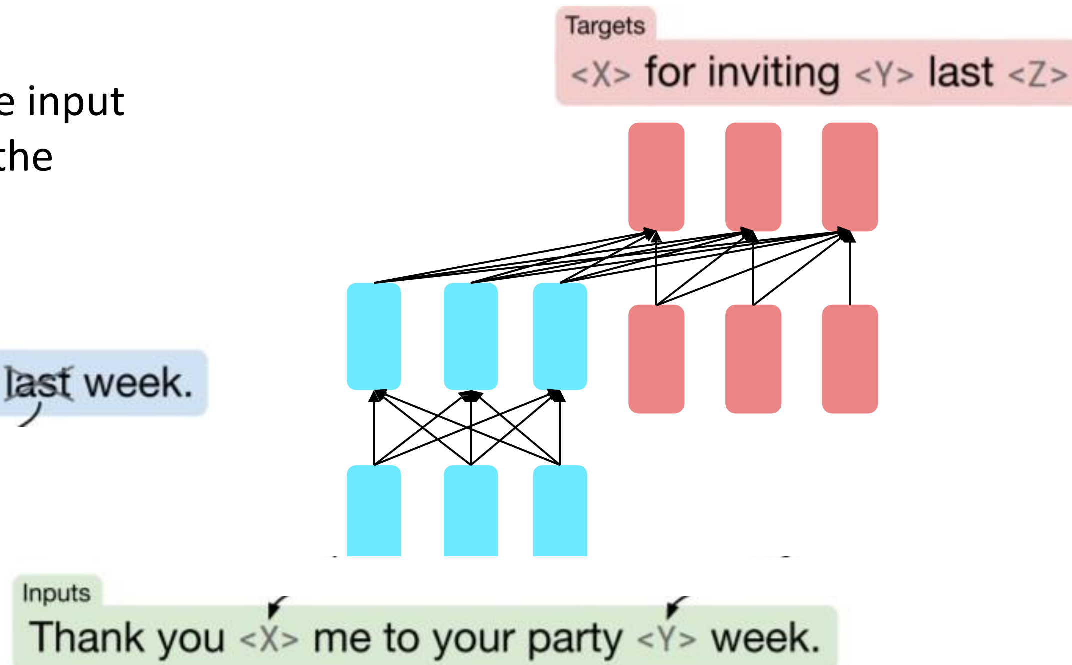
Pretraining encoder-decoders: What pretraining objective to use?

What Raffel et al., 2018 found to work best was **span corruption.** Their model: **T5**.

Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!

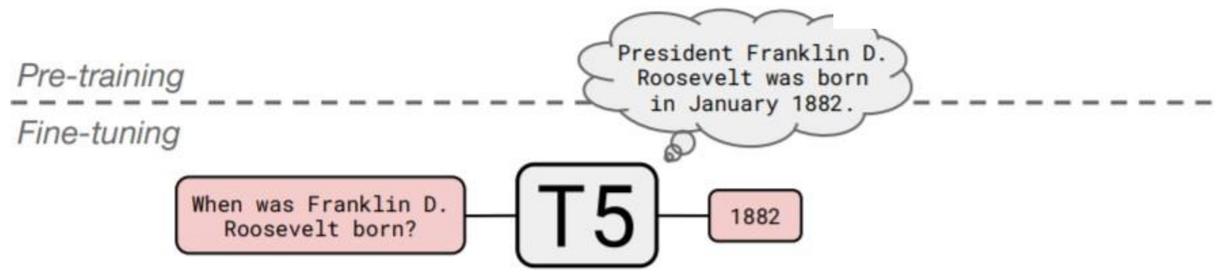


This is implemented in text preprocessing: it's still an objective that looks like **language modeling** at the decoder side.



Pretraining encoder-decoders: What pretraining objective to use?

A fascinating property of T5: it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters.



NQ: Natural Questions

WQ: WebQuestions

TQA: Trivia QA

All "open-domain" versions

	NQ	WQ	TQA dev	TQA test	
Karpukhin et al. (2020)	**41.5**	42.4	**57.9**	–	
T5.1.1-Base	25.7	28.2	24.2	30.6	**220 million params**
T5.1.1-Large	27.3	29.5	28.5	37.2	**770 million params**
T5.1.1-XL	29.5	32.4	36.0	45.1	**3 billion params**
T5.1.1-XXL	32.8	35.6	42.9	52.5	**11 billion params**
T5.1.1-XXL + SSM	35.2	**42.8**	51.9	**61.6**	

[Raffel et al., 2018]

GPT-3, in-context learning, very large models

So far, we've interacted with pretrained models in two ways:

- Sample from the distributions they define (maybe providing a prompt)
- Fine-tune them on a task we care about, and take their predictions.

Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.

GPT-3 is the canonical example of this. The largest T5 model had 11 billion parameters.

GPT-3 has 175 billion parameters.

GPT-3, in-context learning, very large models

Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.

The in-context examples seem to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.

Input (prefix within a single Transformer decoder context):

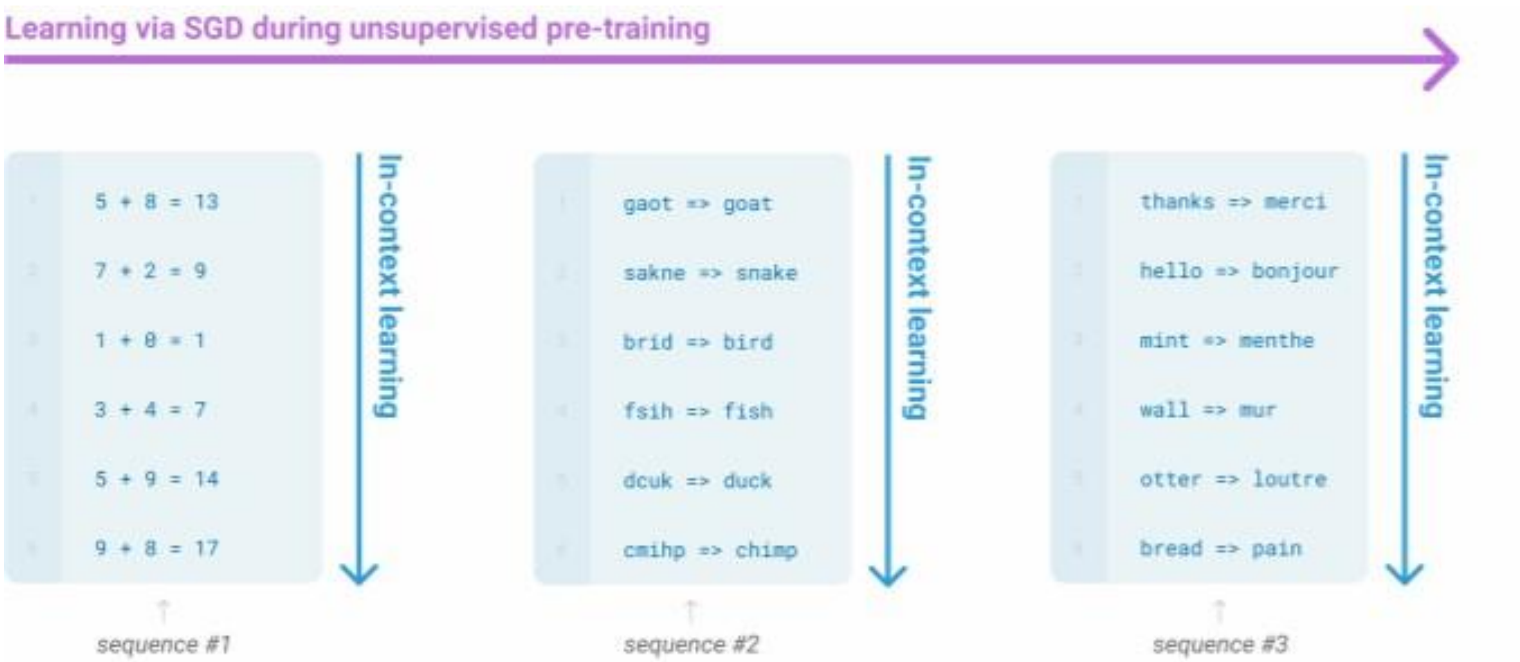
"　thanks -> merci
hello -> bonjour
mint -> menthe
otter -> "

Output (conditional generations):

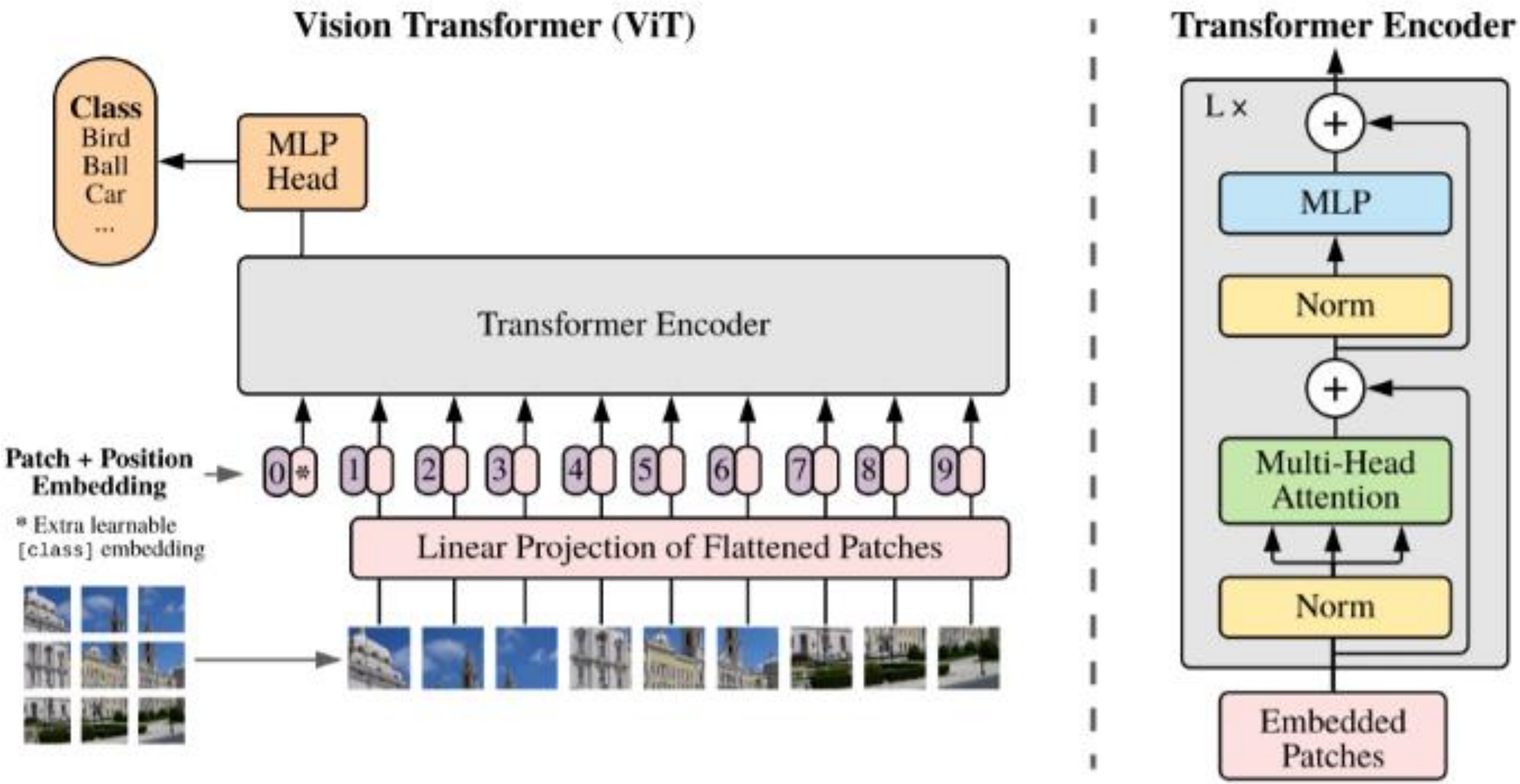
loutre…"

GPT-3, in-context learning, very large models

Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.



Transformers in vision



Dosovitskiy, ICLR 2021, https://github.com/google-research/vision_transformer https://www.youtube.com/watch?v=TrdevFK_am4

Cross-modal transformers

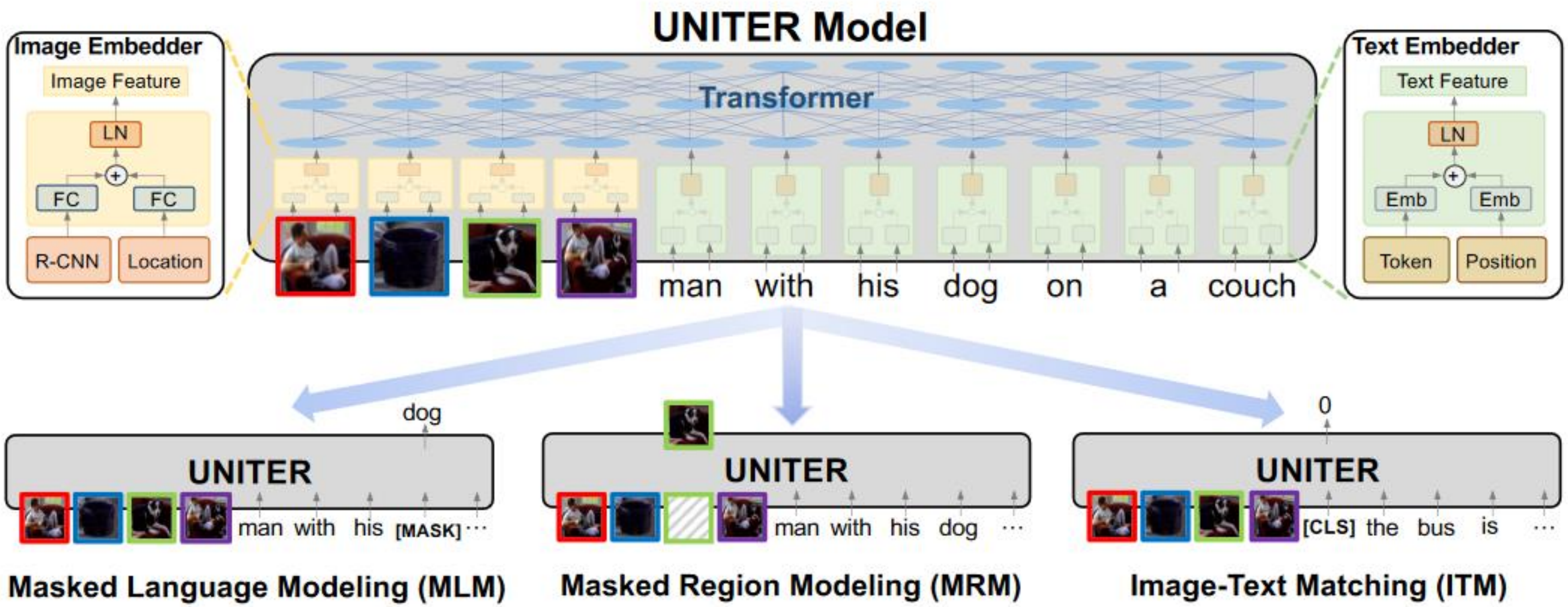


Figure 1: Overview of the proposed UNITER model (best viewed in color), consisting of an Image Embedder, a Text Embedder and a multi-layer self-attention Transformer, learned through three pre-training tasks.

Chen et al., “UNITER: Learning UNiversal Image-TExt Representations”, arxiv 2019

Cross-modal transformers

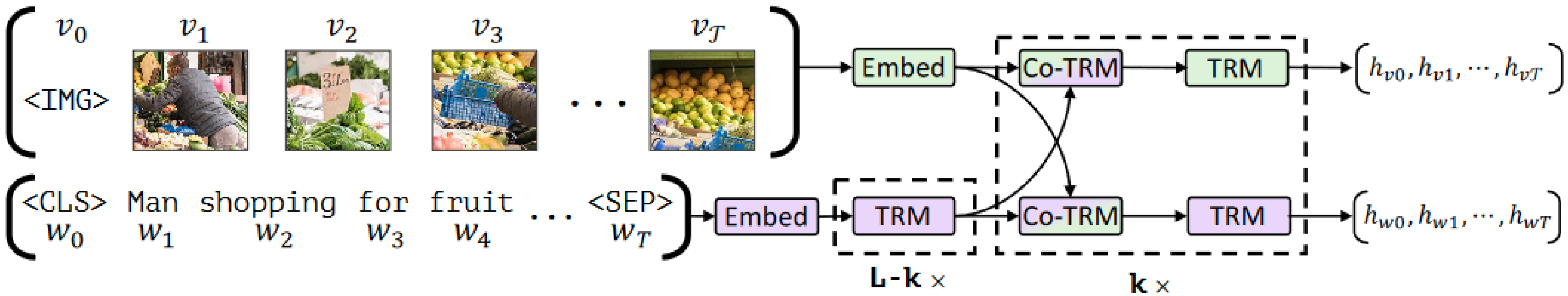


Figure 1: Our ViLBERT model consists of two parallel streams for visual (green) and linguistic (purple) processing that interact through novel co-attentional transformer layers. This structure allows for variable depths for each modality and enables sparse interaction through co-attention. Dashed boxes with multiplier subscripts denote repeated blocks of layers.

Lu et al., "ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks", NeurIPS 2019

Cross-modal transformers

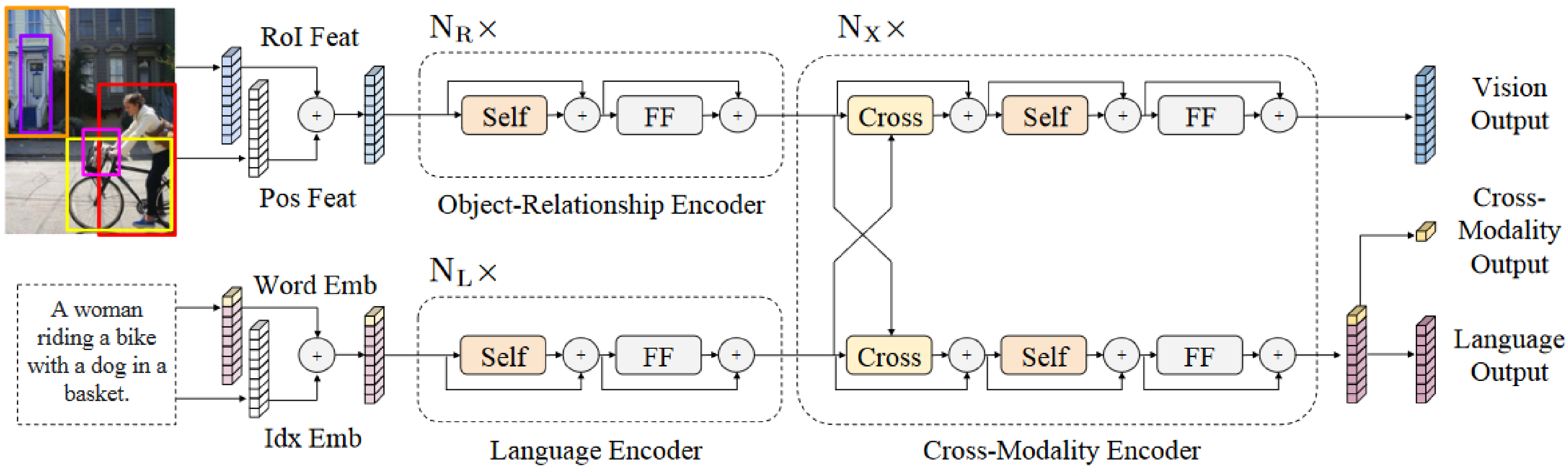
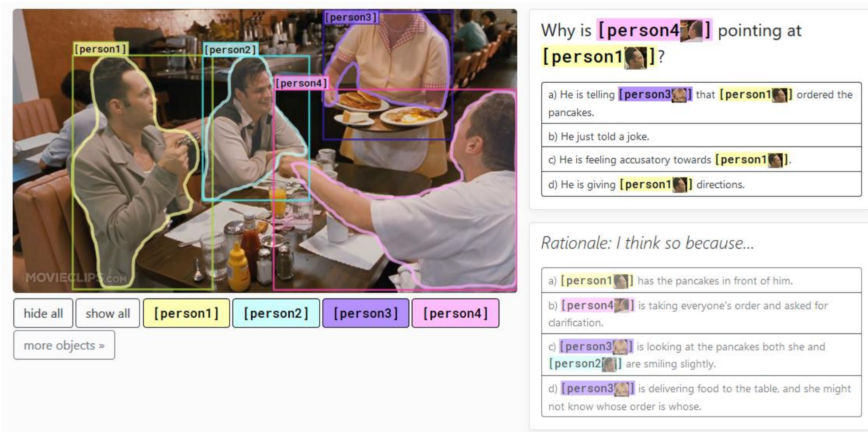


Figure 1: The LXMERT model for learning vision-and-language cross-modality representations. 'Self' and 'Cross' are abbreviations for self-attention sub-layers and cross-attention sub-layers, respectively. 'FF' denotes a feed-forward sub-layer.

Tan and Bansal, "LXMERT: Learning Cross-Modality Encoder Representationsfrom Transformers", EMNLP 2019

Visual Commonsense Reasoning leaderboard



Rank	Model	Q->A	QA->R	Q->AR
	Human Performance *University of Washington* (Zellers et al. '18)	91.0	93.0	85.0
September 30, 2019	UNITER-large (ensemble) *MS D365 AI* https://arxiv.org/abs/1909.11740	**79.8**	**83.4**	**66.8**
2 September 23, 2019	UNITER-large (single model) *MS D365 AI* https://arxiv.org/abs/1909.11740	77.3	80.8	62.8
3 August 9,2019	ViLBERT (ensemble of 10 models) *Georgia Tech & Facebook AI Research* https://arxiv.org/abs/1908.02265	76.4	78.0	59.8
4 September 23,2019	VL-BERT (single model) *MSRA & USTC* https://arxiv.org/abs/1908.08530	75.8	78.4	59.7
5 August 9,2019	ViLBERT (ensemble of 5 models) *Georgia Tech & Facebook AI Research* https://arxiv.org/abs/1908.02265	75.7	77.5	58.8

https://visualcommonsense.com/leaderboard/

Cost of training

ULMfit
Jan 2018
Training: 1 GPU day

GPT
June 2018
Training
240 GPU days

BERT
Oct 2018
Training
256 TPU days
~320–560 GPU days

GPT-2
Feb 2019
Training
~2048 TPU v3 days according to a reddit thread