

CS 1678: Intro to Deep Learning
Transformers

Prof. Adriana Kovashka
University of Pittsburgh
March 18, 2024

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, GPT
- 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>

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:

signifier (symbol) \Leftrightarrow 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 0 1 0 0 0 0]

hotel = [0 0 0 0 0 0 0 1 0 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

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

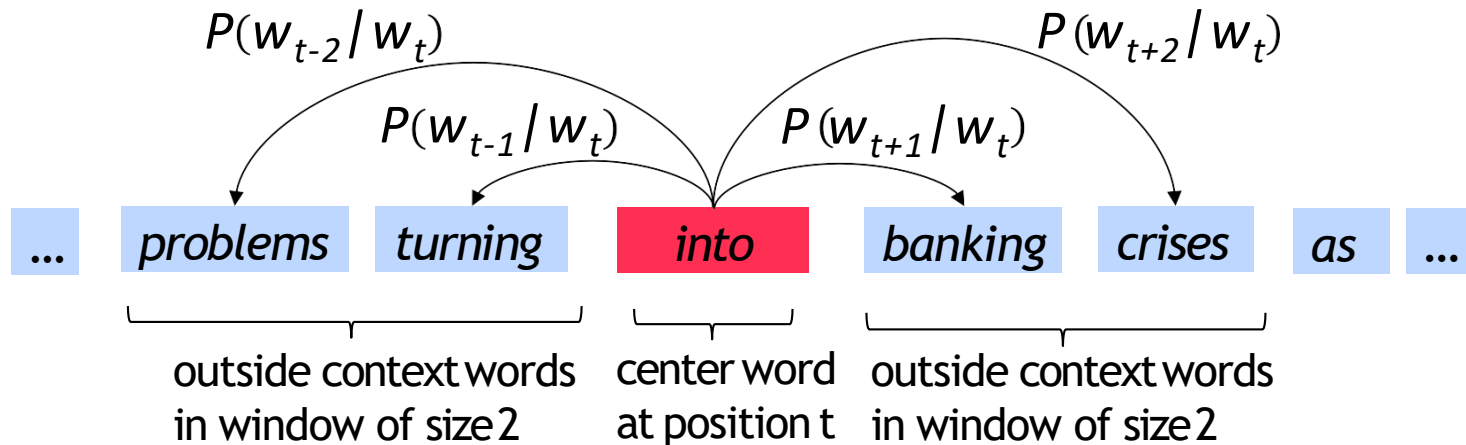
Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

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

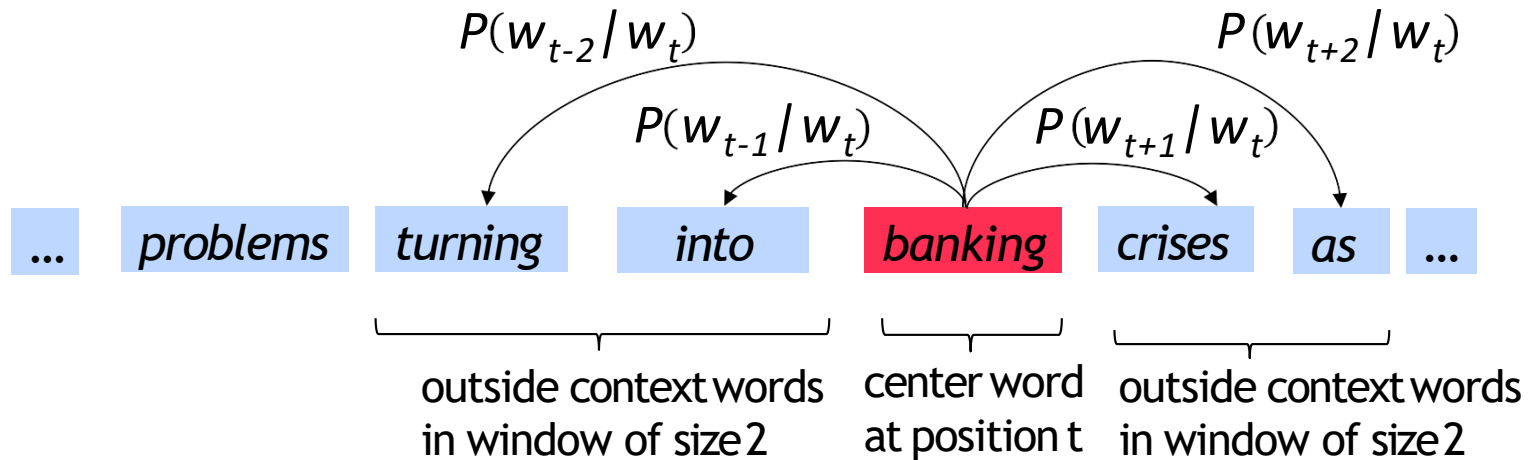
Word2Vec Overview

- 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 $t = 1, \dots, T$, predict context words within a window of fixed size m , given center word w_j .

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

θ is all variables
to be optimized

sometimes called *cost* or *loss* function

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

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

Minimizing objective function \Leftrightarrow Maximizing predictive accuracy

Word2Vec: objective function

- We want to minimize the objective function:

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

- 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

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

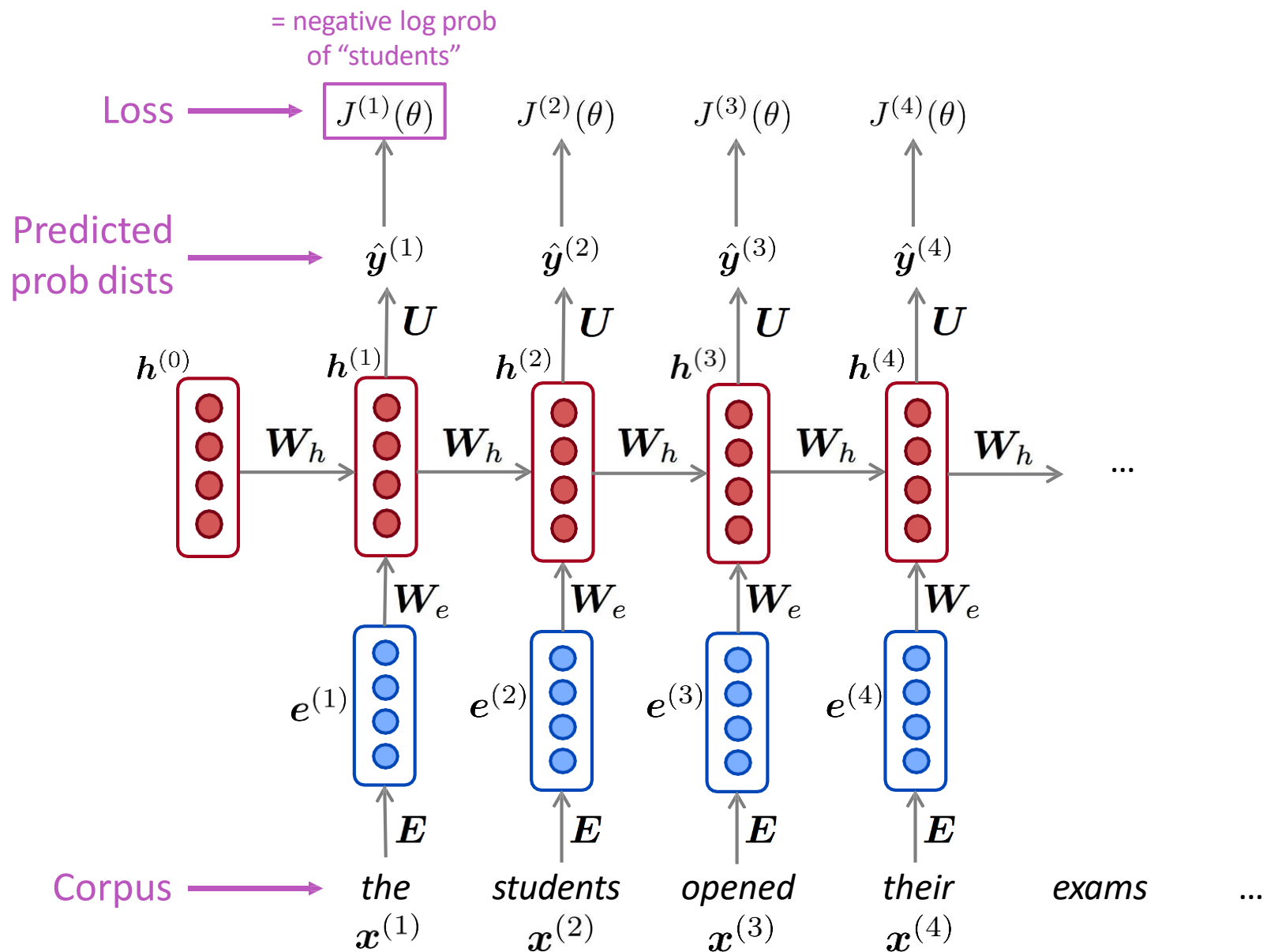
$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Normalize over entire vocabulary to give probability distribution

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

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

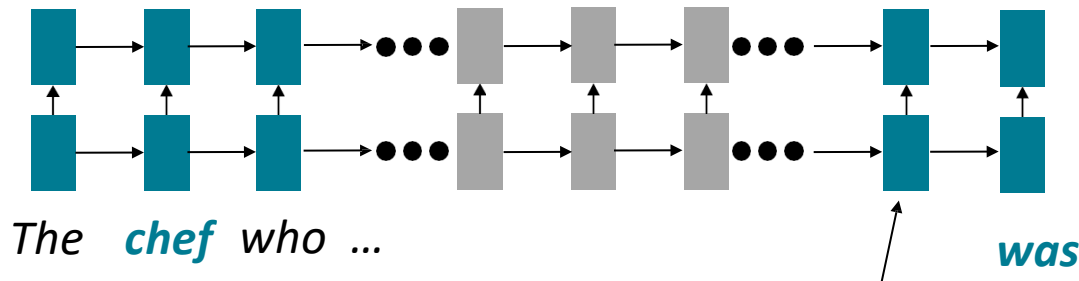
Recall: Recurrent Neural Networks (RNNs)



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”; not necessarily the right way to think about sentences...

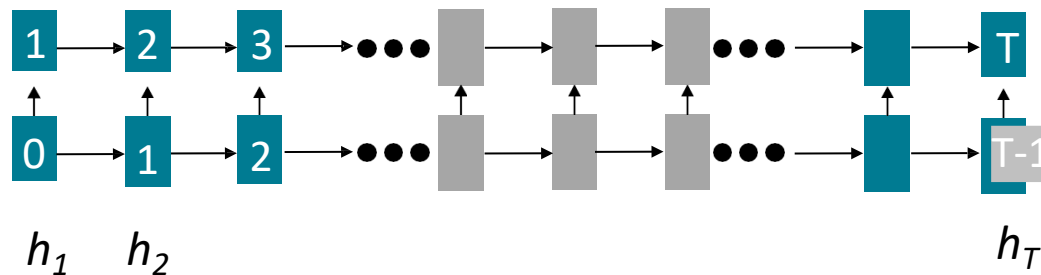


Info of *chef* has gone through $O(\text{sequence length})$ many layers!

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!

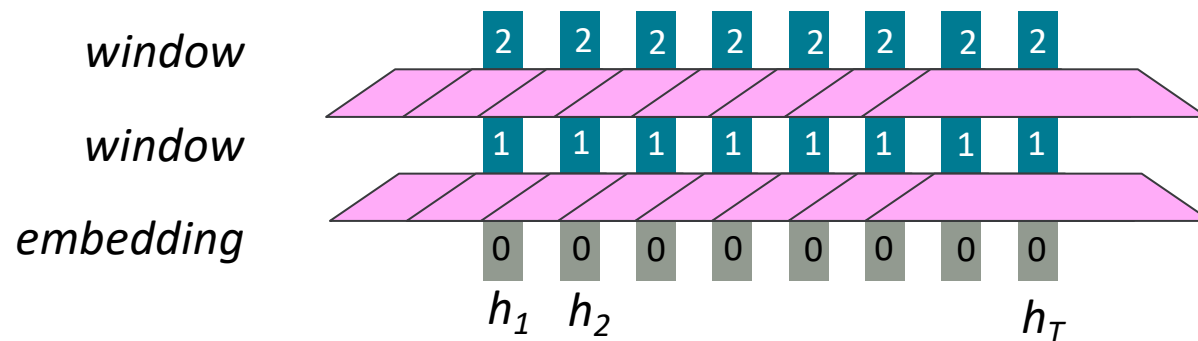


Numbers indicate min # of steps before a state can be computed

If 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!

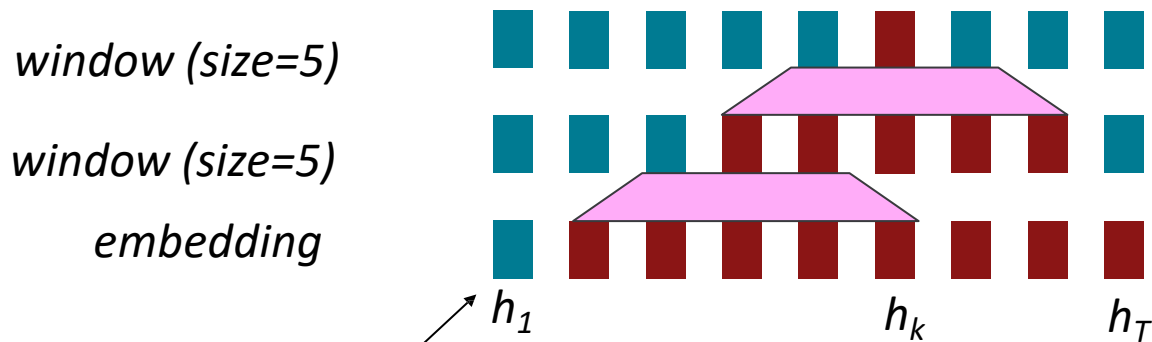


Numbers indicate min # of steps before a state can be computed

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



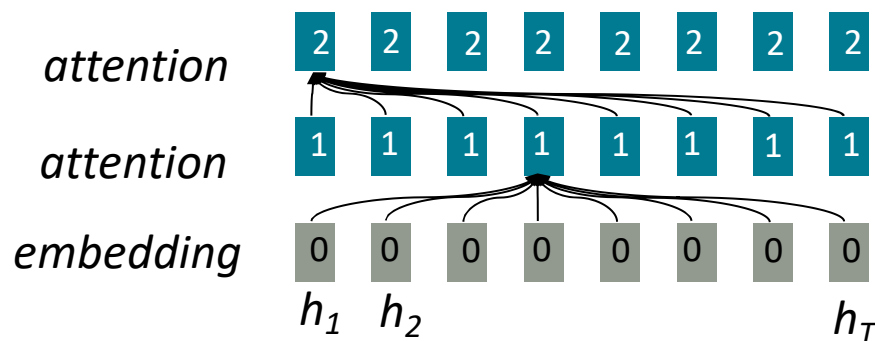
Red states
indicate those
“visible” to h_k

Too far from h_k to be considered

If not recurrence, then what?

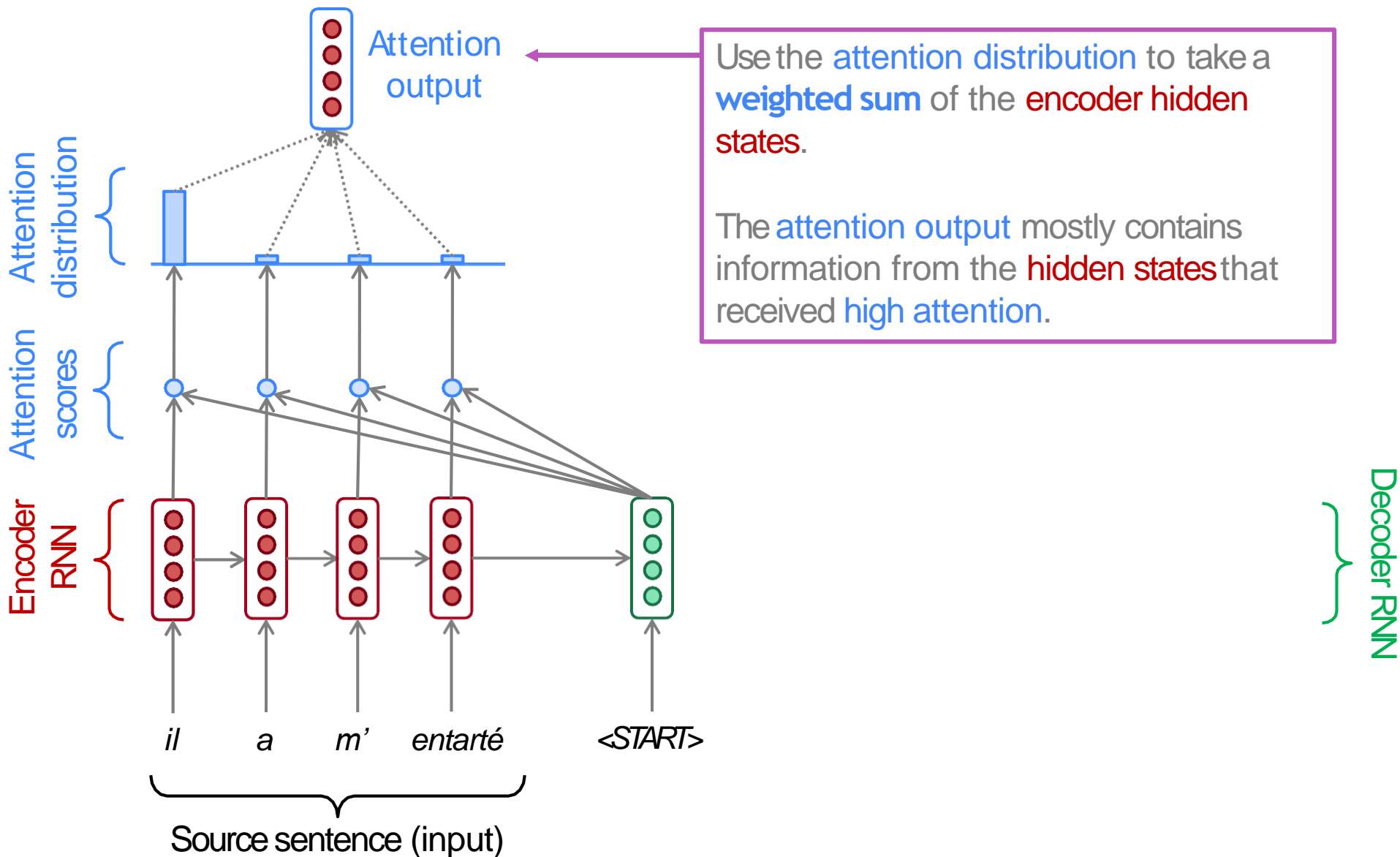
How about attention?

- **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 also 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

Recall: Sequence-to-sequence with attention



Recall: Attention in equations

- We have encoder hidden states $h_1, \dots, h_N \in \mathbb{R}^h$
- On timestep t , we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention score e^t for this step:

$$e^t = [s_t^T h_1, \dots, s_t^T h_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

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

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output a_t

$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output a_t with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

Attention Notation: Queries, Keys, Values

- 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

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

Compute attention weights from affinities (softmax)

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

Compute outputs as weighted sum of **values**

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 $Q \in \mathbb{R}^{d \times d}$ is the query matrix.
 - $v_i = Vx_i$, where $V \in \mathbb{R}^{d \times d}$ is the value matrix.
- These matrices allow *different aspects* of the x vectors to be used/emphasized in each of the three roles.

Key-Query-Value Attention

- Let's look at how key-query-value attention is computed, in matrices.
 - Let $X = [x_1; \dots; x_T] \in \mathbb{R}^{T \times d}$ 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 $\text{output} = \text{softmax}(XQ(XK)^T) \times XV$.

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

A diagram illustrating the first step of the attention mechanism. It shows a pink box labeled XQ followed by a pink box labeled $K^T X^T$, with an equals sign and a larger pink box labeled $XQK^T X^T$. To the right of the final box is the text $\in \mathbb{R}^{T \times T}$. A teal-colored text label "All pairs of attention scores!" is positioned to the right of the equation. An arrow points from the $XQK^T X^T$ box down to the next equation.

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

A diagram illustrating the second step of the attention mechanism. It shows the word "softmax" to the left of a large pink box containing $XQK^T X^T$, which is enclosed in large square brackets. This is followed by a pink box labeled XV , an equals sign, and a final pink box representing the output. To the right of the output box is the text $\text{output} \in \mathbb{R}^{T \times d}$.

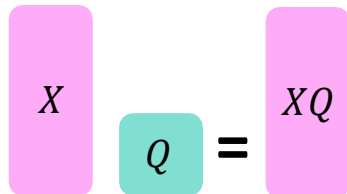
Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x^T Q^T K x_j$ 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:
 - $\text{output}_P = \text{softmax}(X Q_P K_P^T X^T) * X V_P$, where $\text{output}_P \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
 - $\text{output} = Y[\text{output}_1; \dots; \text{output}_h]$, where $Y \in \mathbb{R}^{d \times d}$
- Each head gets to “look” at different things, and construct value vectors differently.

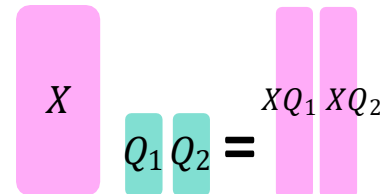
Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x^T Q^T K x_j$ 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 .

Single-head attention (just the query matrix)



Multi-head attention (just two heads here)



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

Dot-Product Attention (alternative slide)

- 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

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

Dot-Product Attention – Matrix notation (alternative slide)

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

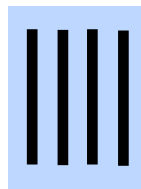
$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- becomes:

$$A(Q, K, V) = \text{softmax}(QK^T)V$$

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

softmax
row-wise



$$=[|Q| \times d_v]$$

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!

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.

$$\begin{aligned}v_i &= v_i' + p_i \\q_i &= q_i' + p_i \\k_i &= k_i' + p_i\end{aligned}$$

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:

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

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

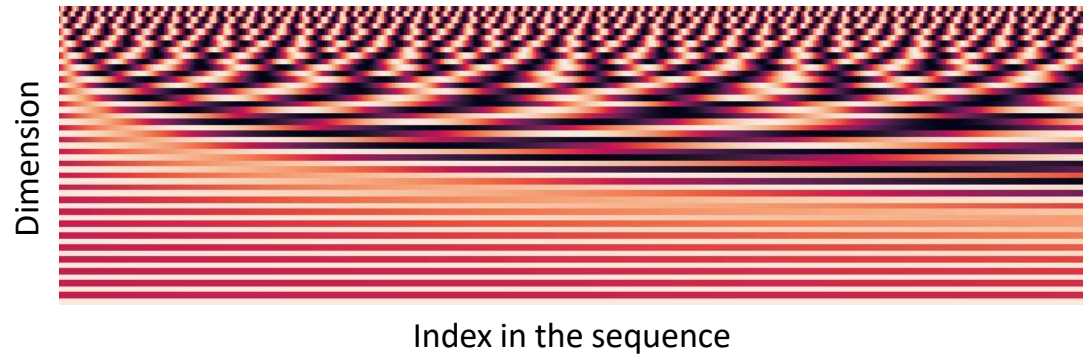
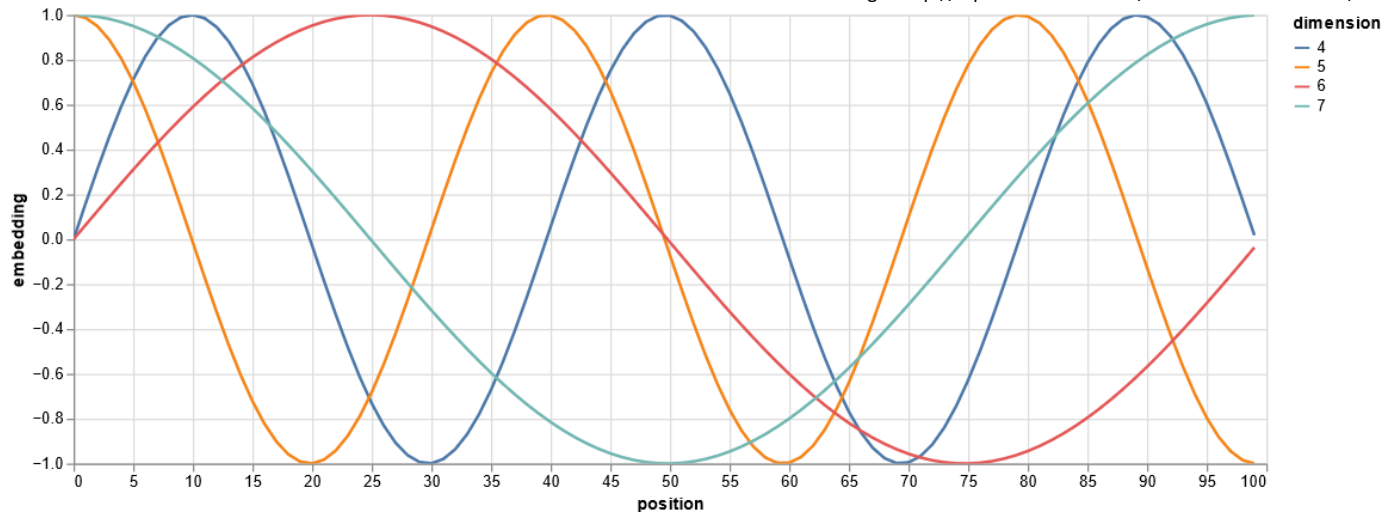


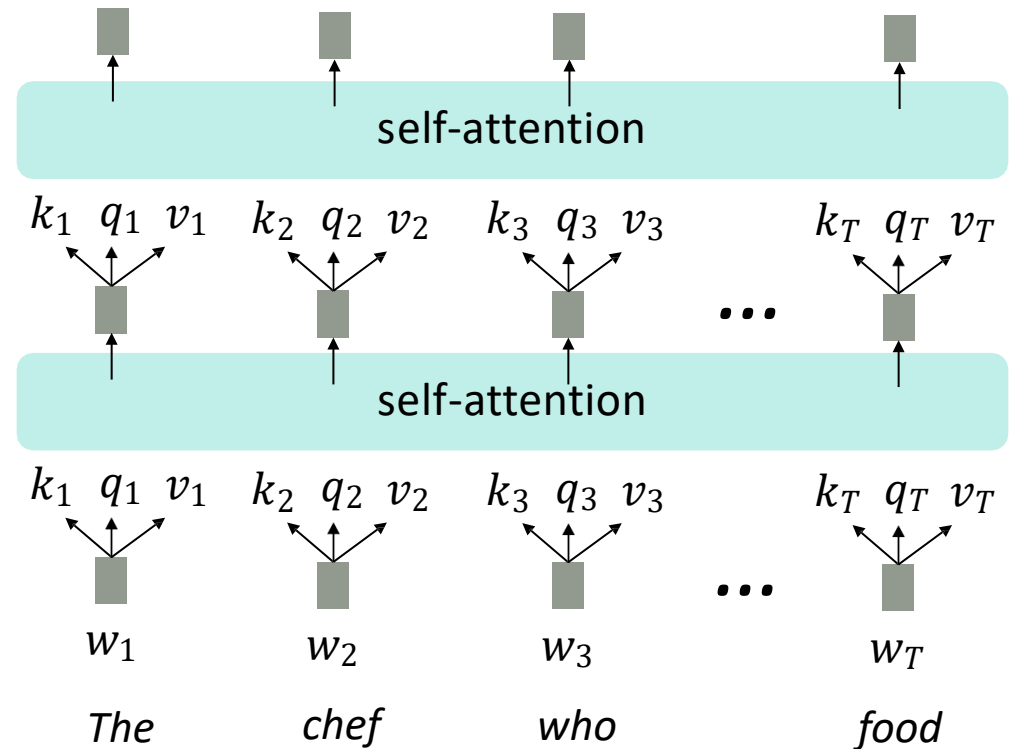
Image: <http://nlp.seas.harvard.edu/annotated-transformer/>



Pros: Periodicity indicates that maybe “absolute position” isn’t as important; maybe can extrapolate to longer sequences as periods restart. Cons: Not learnable.

Stacking self-attention

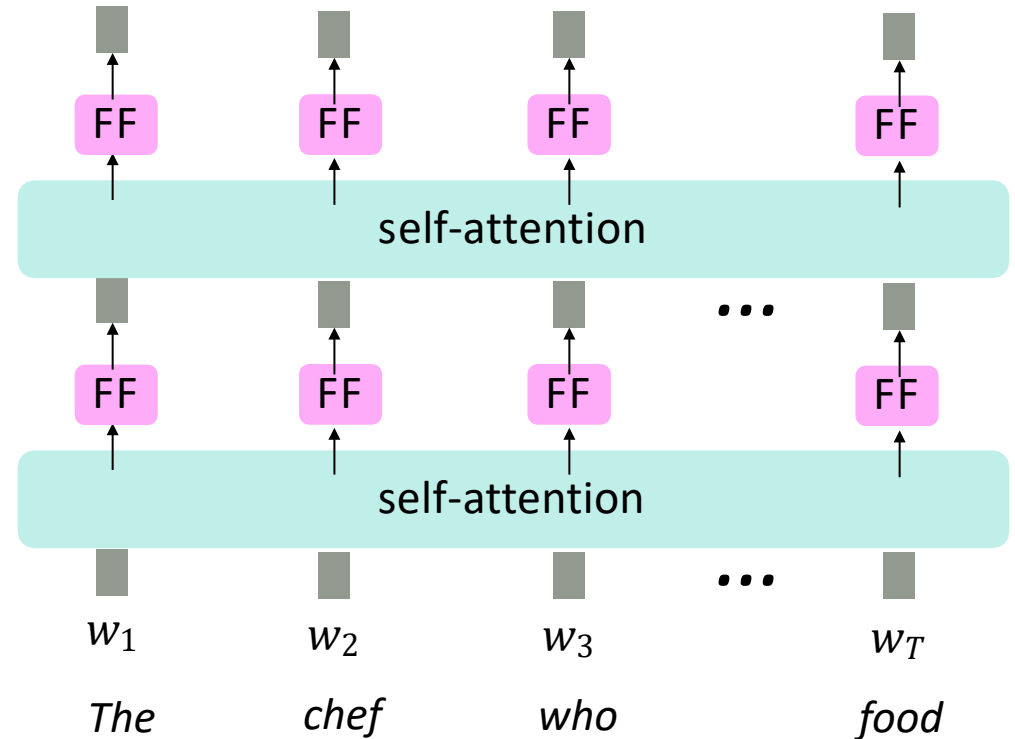
- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.
- The different layers capture a hierarchy of relationships within the data, similar to how convolutional networks capture a hierarchy of patterns that range from low- to high-level.



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.

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



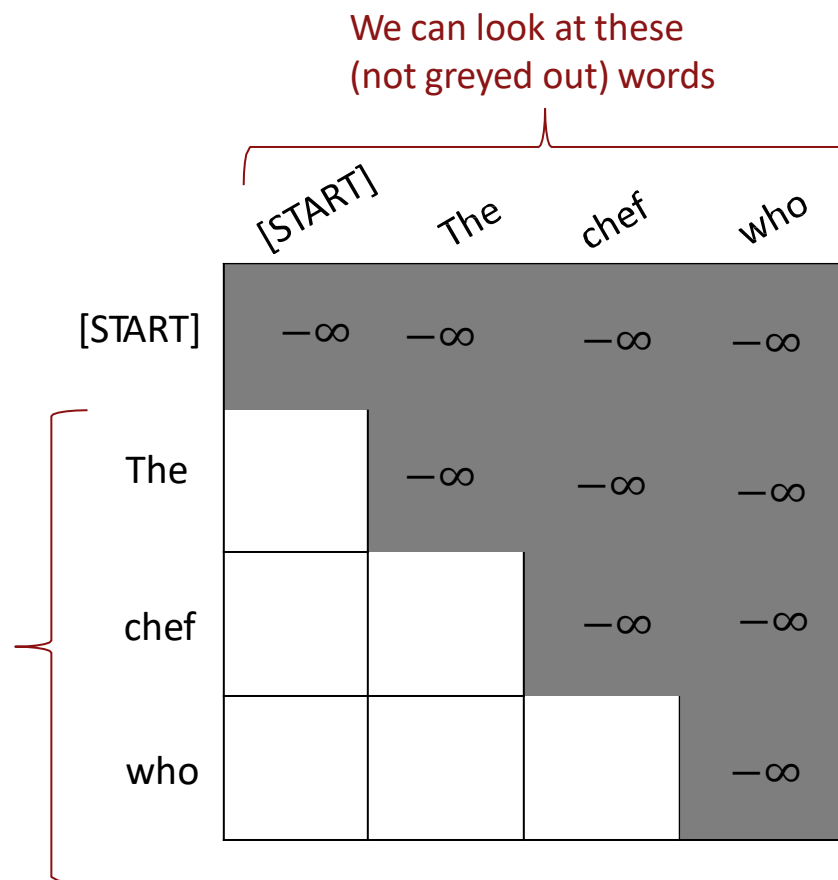
Intuition: the FF network processes the result of attention

Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$e_{ij} =$ For encoding these words

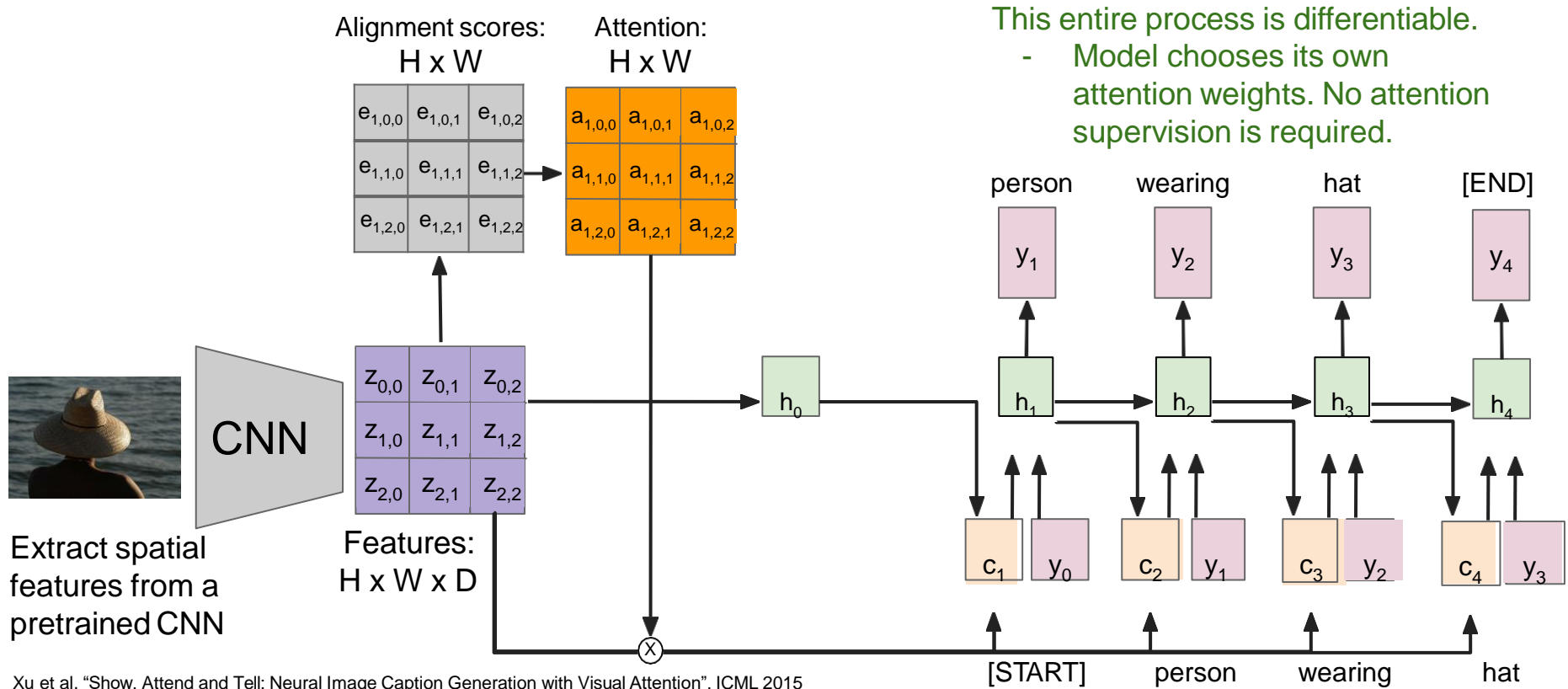
$$q_i^T k_j, j < i$$
$$-\infty, j \geq i$$



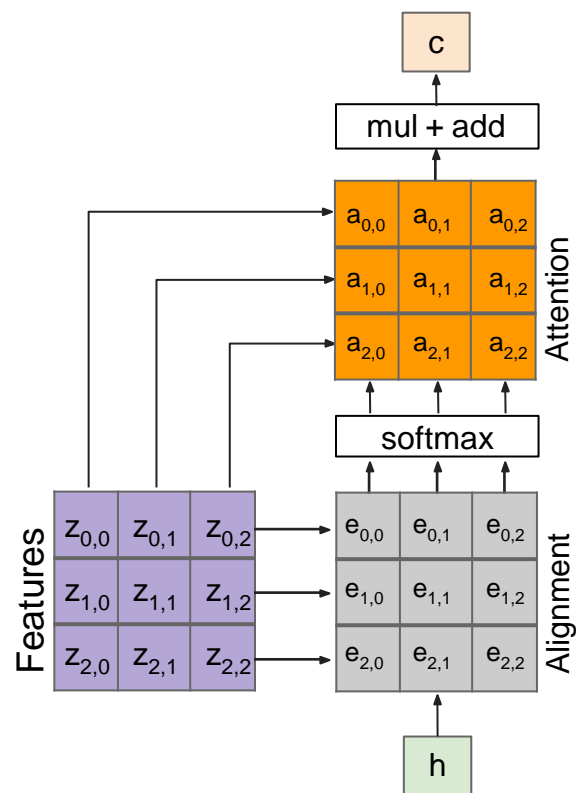
Necessities for a self-attention building block:

- **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 (yet).

Recall: Image Captioning with RNNs and Attention



Attention we saw in image captioning

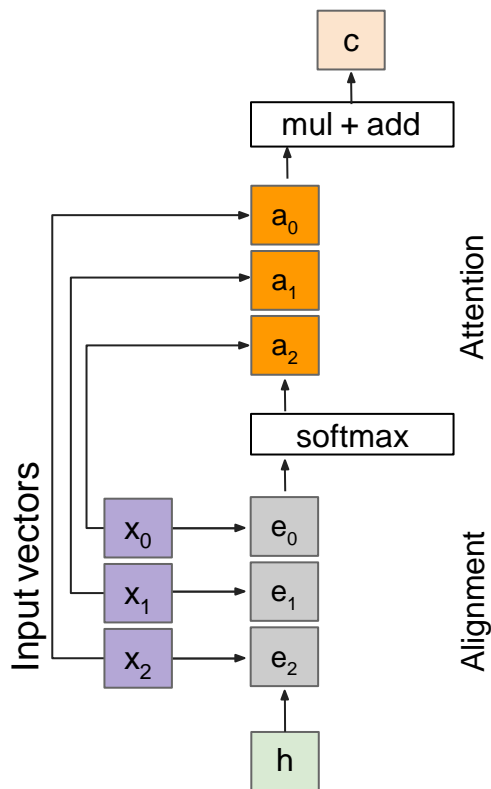


Outputs:
context vector: \mathbf{c} (shape: D)

Operations:
Alignment: $e_{i,j} = f_{\text{att}}(\mathbf{h}, \mathbf{z}_{i,j})$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $\mathbf{c} = \sum_{i,j} a_{i,j} \mathbf{z}_{i,j}$

Inputs:
Features: \mathbf{z} (shape: H x W x D)
Query: \mathbf{h} (shape: D)

General attention layer (alternative slide)



Outputs:

context vector: \mathbf{c} (shape: D)

Operations:

Alignment: $e_i = f_{\text{att}}(h, x_i)$

Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

Output: $\mathbf{c} = \sum_i a_i x_i$

Attention operation is **permutation invariant**.

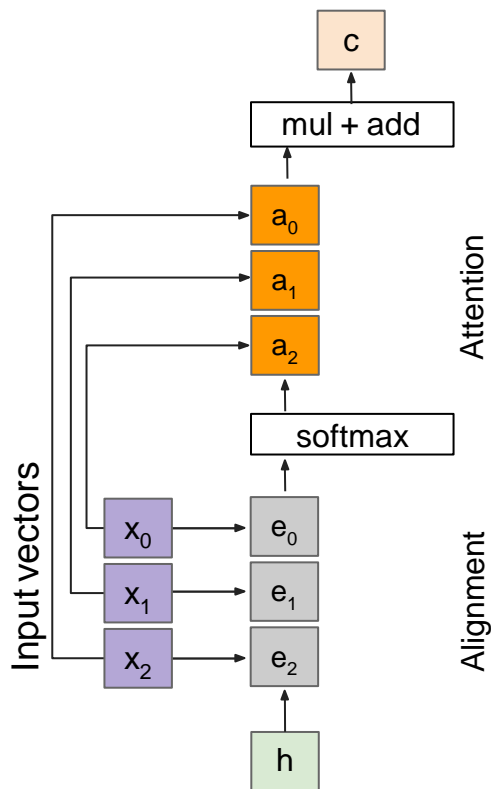
- Doesn't care about ordering of the features
- Stretch $H \times W = N$ into N vectors

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

Query: \mathbf{h} (shape: D)

General attention layer (alternative slide)



Outputs:

context vector: \mathbf{c} (shape: D)

Operations:

Alignment: $e_i = h \cdot x_i$

Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

Output: $\mathbf{c} = \sum_i a_i x_i$

Change $f_{\text{att}}(\cdot)$ to a simple dot product

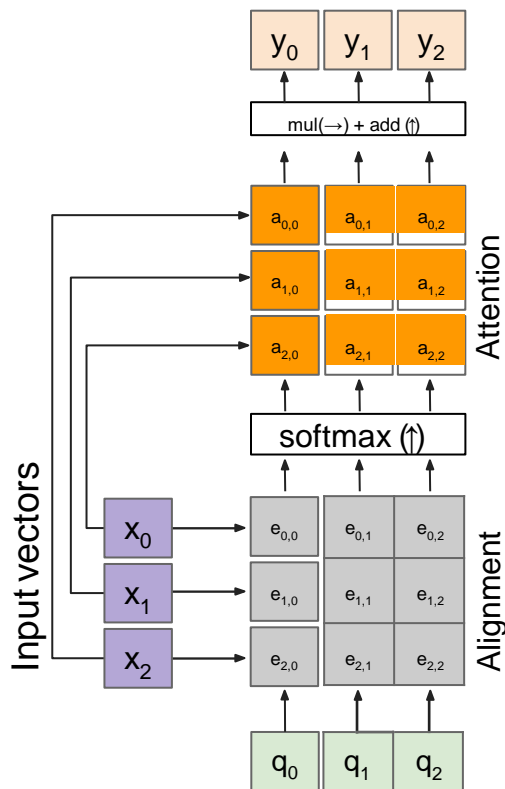
- only works well with key & value transformation trick (will mention in a few slides)

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

Query: \mathbf{h} (shape: D)

General attention layer (alternative slide)



Outputs:

context vectors: \mathbf{y} (shape: D)

Operations:

Alignment: $e_{i,j} = q_j \cdot x_i / \sqrt{D}$

Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

Output: $y_j = \sum_i a_{i,j} x_i$

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

Queries: \mathbf{q} (shape: $M \times D$)

Multiple query vectors

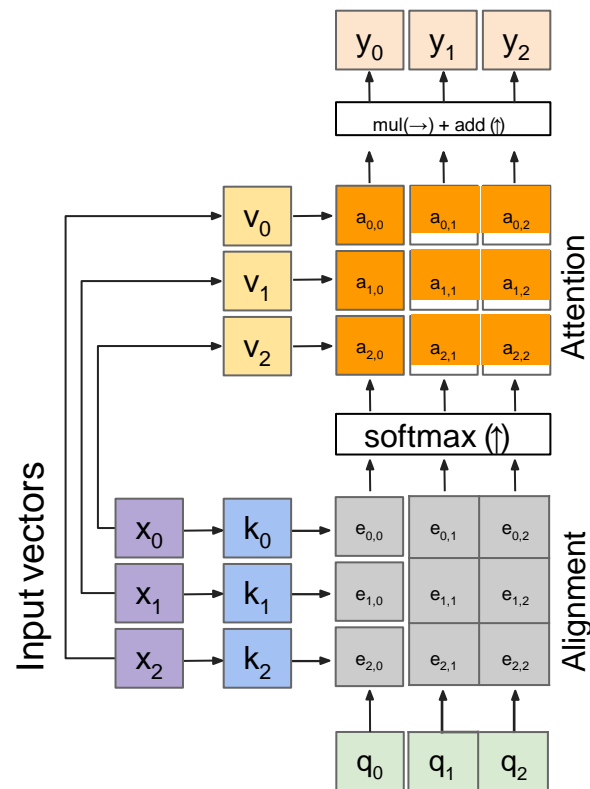
- each query creates a new output context vector

Notice that the input vectors are used for both the alignment as well as the attention calculations.

- We can add more expressivity to the layer by adding a different FC layer before each of the two steps.

Multiple query vectors

General attention layer (alternative slide)



Outputs:

context vectors: \mathbf{y} (shape: D_v)

The input and output dimensions can now change depending on the key and value FC layers

Operations:

Key vectors: $\mathbf{k} = \mathbf{x}\mathbf{W}_k$

Value vectors: $\mathbf{v} = \mathbf{x}\mathbf{W}_v$

Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$

Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

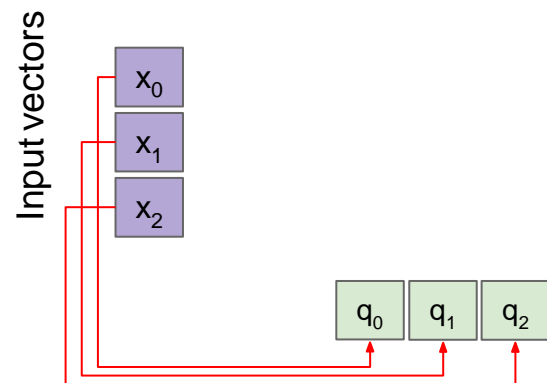
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

Queries: \mathbf{q} (shape: $M \times D_k$)

Self attention layer (alternative slide)



Operations:

Key vectors: $\mathbf{k} = \mathbf{x}\mathbf{W}_k$

Value vectors: $\mathbf{v} = \mathbf{x}\mathbf{W}_v$

Query vectors: $\mathbf{q} = \mathbf{x}\mathbf{W}_q$

Alignment: $e_{i,j} = \mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{D}$

Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

Output: $y_j = \sum_i a_{i,j} \mathbf{v}_i$

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

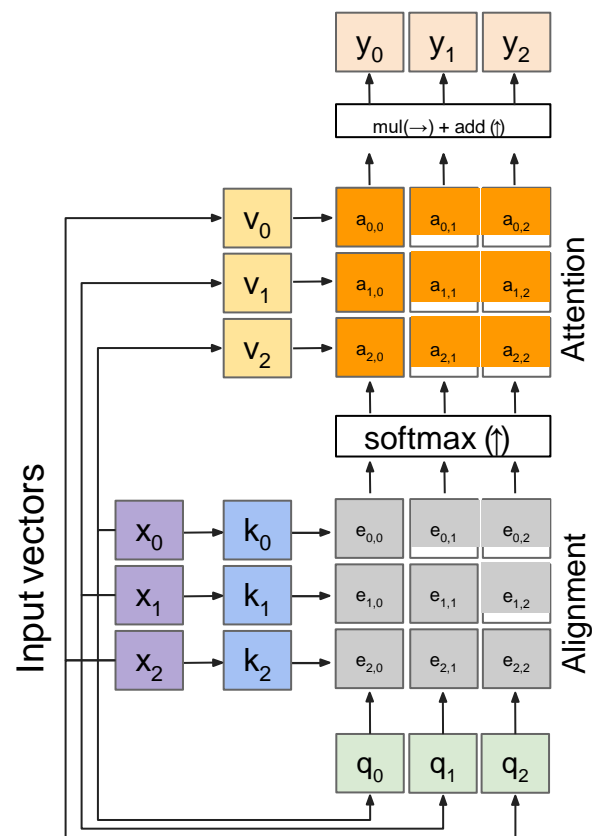
Queries: \mathbf{q} (shape: $M \times D_k$)

We can calculate the query vectors from the input vectors, therefore, defining a "self-attention" layer.

Instead, query vectors are calculated using a FC layer.

No input query vectors anymore

Self attention layer (alternative slide)



Outputs:

context vectors: \mathbf{y} (shape: D_v)

Operations:

Key vectors: $\mathbf{k} = \mathbf{x} \mathbf{W}_k$

Value vectors: $\mathbf{v} = \mathbf{x} \mathbf{W}_v$

Query vectors: $\mathbf{q} = \mathbf{x} \mathbf{W}_q$

Alignment: $e_{i,j} = \mathbf{q}_j \cdot \mathbf{k}_i / \sqrt{D}$

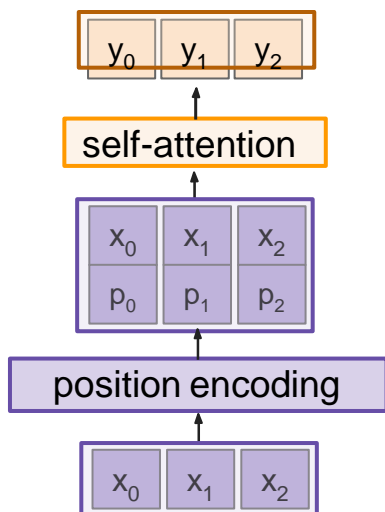
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: \mathbf{x} (shape: $N \times D$)

Positional encoding



Concatenate special positional encoding p_j to each input vector x_j

We use a function $pos: \mathbb{N} \rightarrow \mathbb{R}^d$ to process the position j of the vector into a d -dimensional vector

So, $p_j = pos(j)$

Options for $pos(\cdot)$

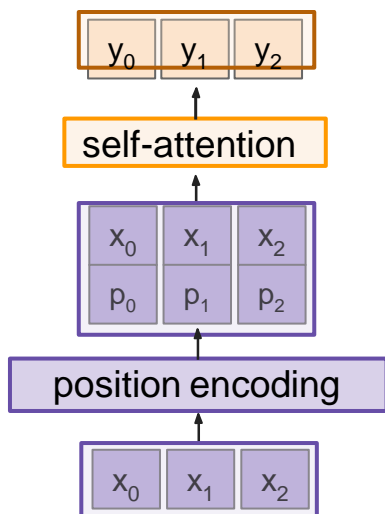
1. Learn a lookup table:
 - Learn parameters to use for $pos(t)$ for $t \in [0, T)$
 - Lookup table contains $T \times d$ parameters.

Desiderata of $pos(\cdot)$:

1. It should output a **unique** encoding for each time-step (word's position in a sentence)
2. **Distance** between any two time-steps should be consistent across sentences with different lengths.
3. Our model should generalize to **longer** sentences without any efforts. Its values should be bounded.
4. It must be **deterministic**.

Vaswani et al, "Attention is all you need", NeurIPS 2017

Positional encoding



Concatenate special positional encoding p_j to each input vector x_j

We use a function $pos: \mathbb{N} \rightarrow \mathbb{R}^d$ to process the position j of the vector into a d -dimensional vector

So, $p_j = pos(j)$

Options for $pos(\cdot)$

1. Learn a lookup table:
 - Learn parameters to use for $pos(t)$ for $t \in [0, T)$
 - Lookup table contains $T \times d$ parameters.
2. Design a fixed function with the desiderata

$$p(t) = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \vdots \\ \sin(\omega_{d/2} \cdot t) \\ \cos(\omega_{d/2} \cdot t) \end{bmatrix}_d$$

Intuition:

0 : 0 0 0 0	8 : 1 0 0 0
1 : 0 0 0 1	9 : 1 0 0 1
2 : 0 0 1 0	10 : 1 0 1 0
3 : 0 0 1 1	11 : 1 0 1 1
4 : 0 1 0 0	12 : 1 1 0 0
5 : 0 1 0 1	13 : 1 1 0 1
6 : 0 1 1 0	14 : 1 1 1 0
7 : 0 1 1 1	15 : 1 1 1 1

where $\omega_k = \frac{1}{10000^{2k/d}}$

[image source](#)

Vaswani et al, "Attention is all you need", NeurIPS 2017

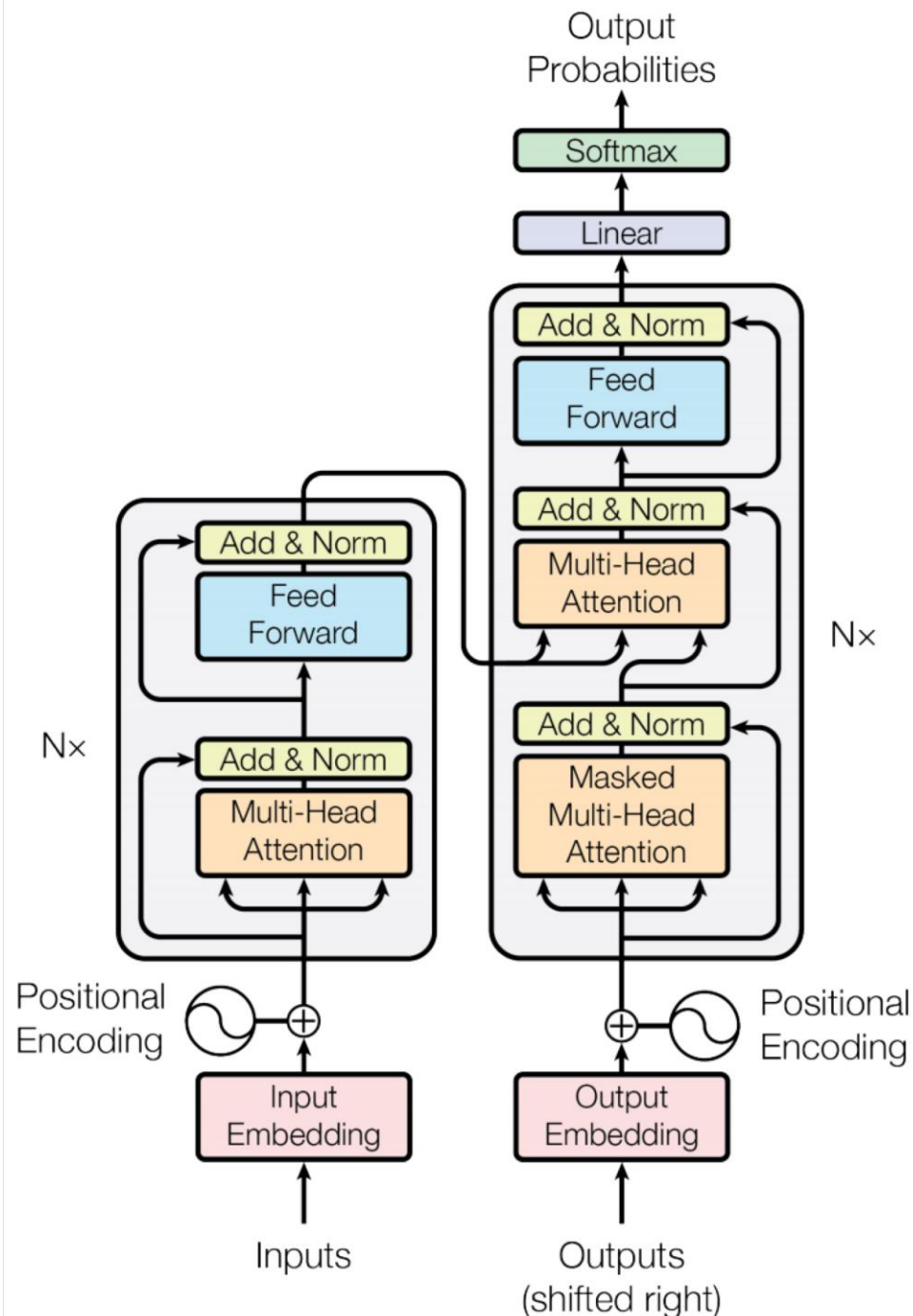
Transformer Overview

Attention is all you need. 2017. Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin

<https://arxiv.org/pdf/1706.03762.pdf>

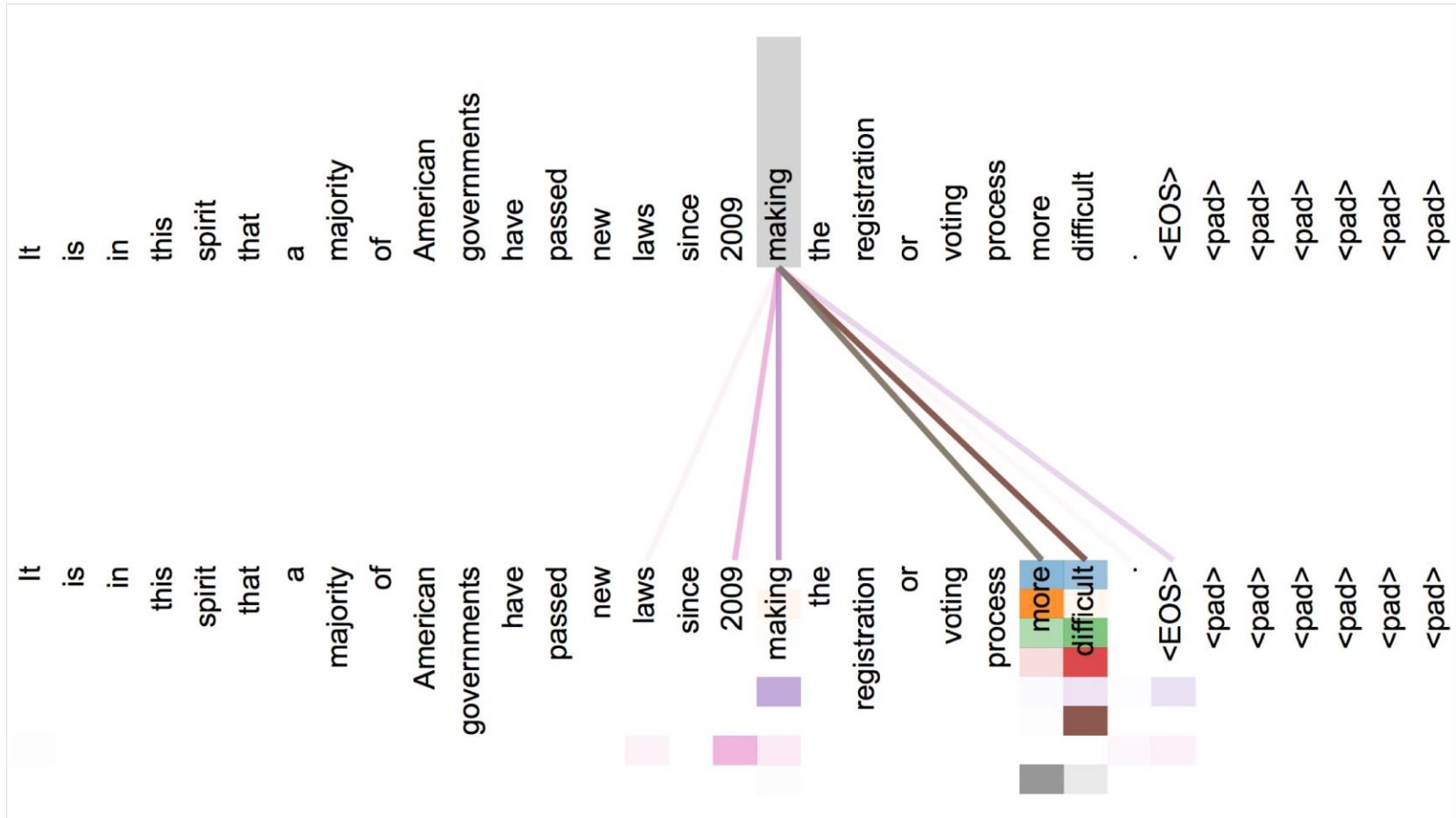
- Non-recurrent sequence-to-sequence encoder-decoder model
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier

This and related figures from paper ↑

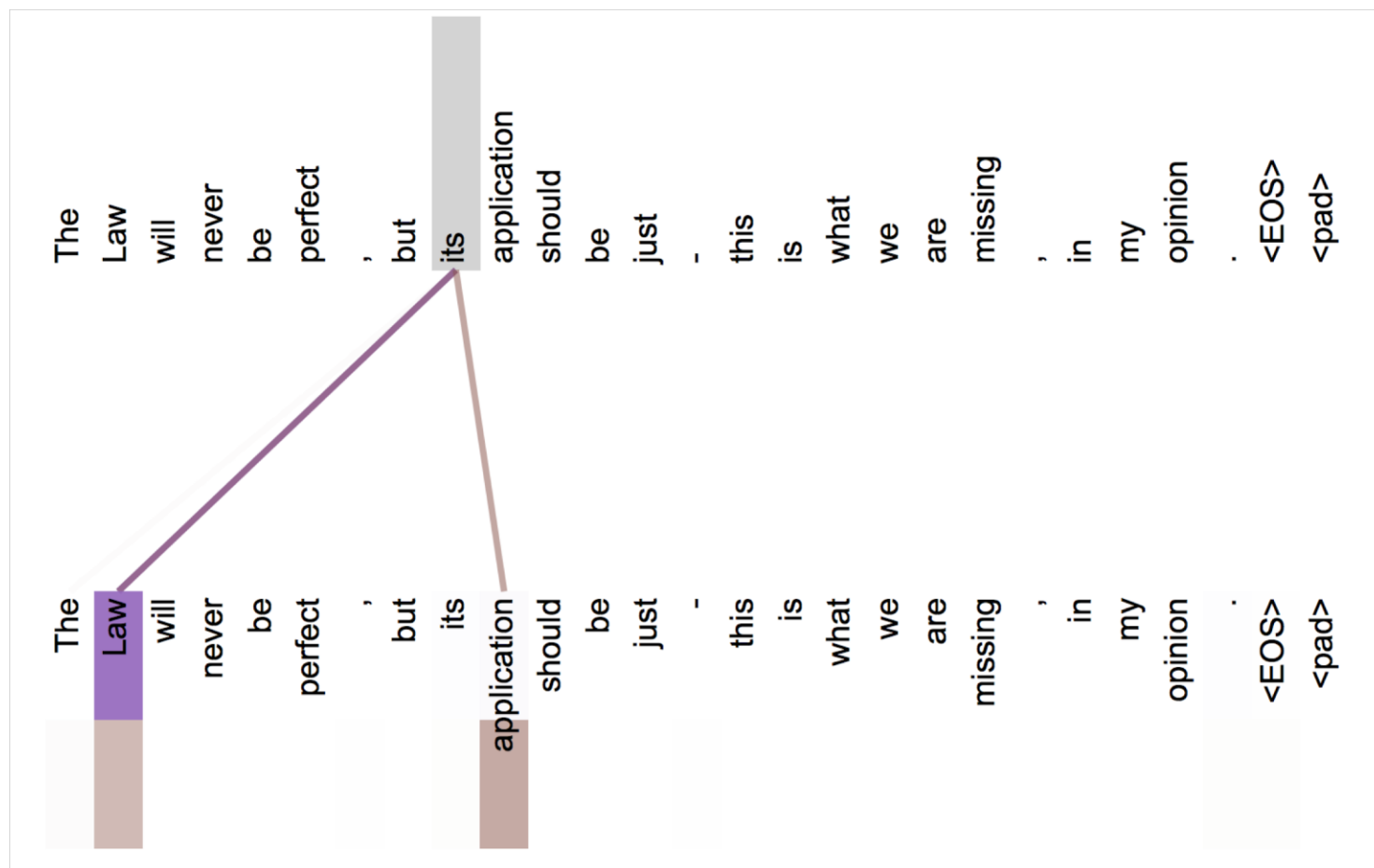


Attention visualization in layer 5

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

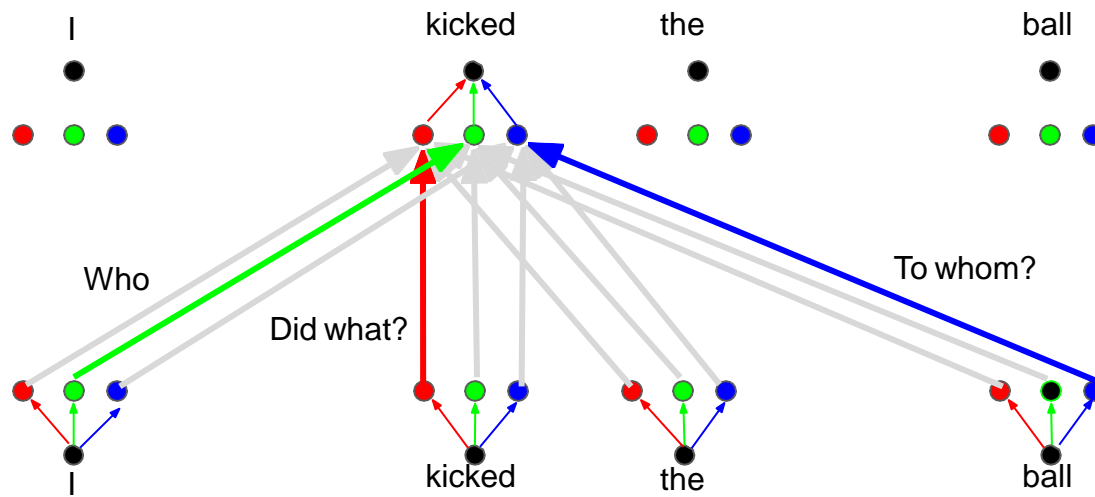


Attention visualization: Implicit anaphora resolution



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.

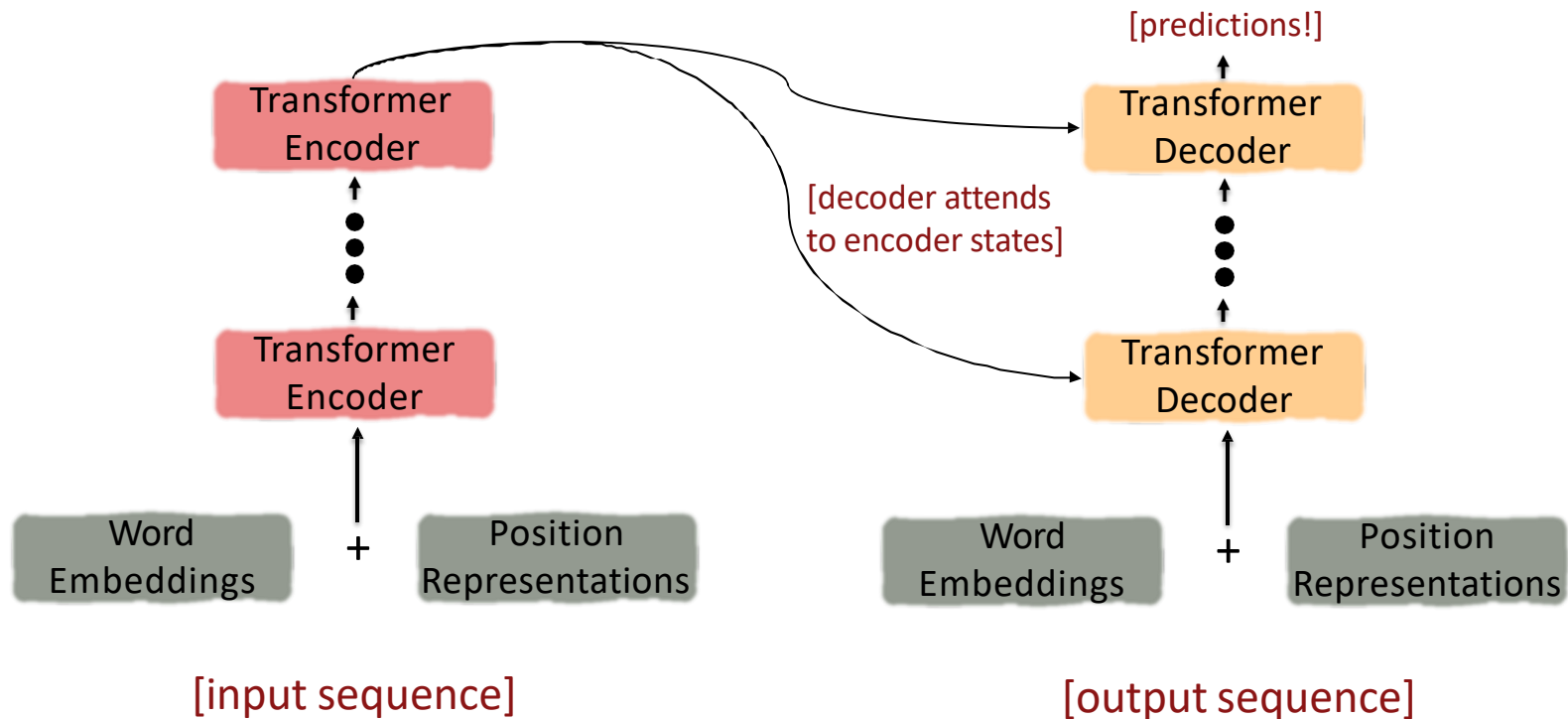
Parallel attention heads



The Transformer Encoder-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]

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

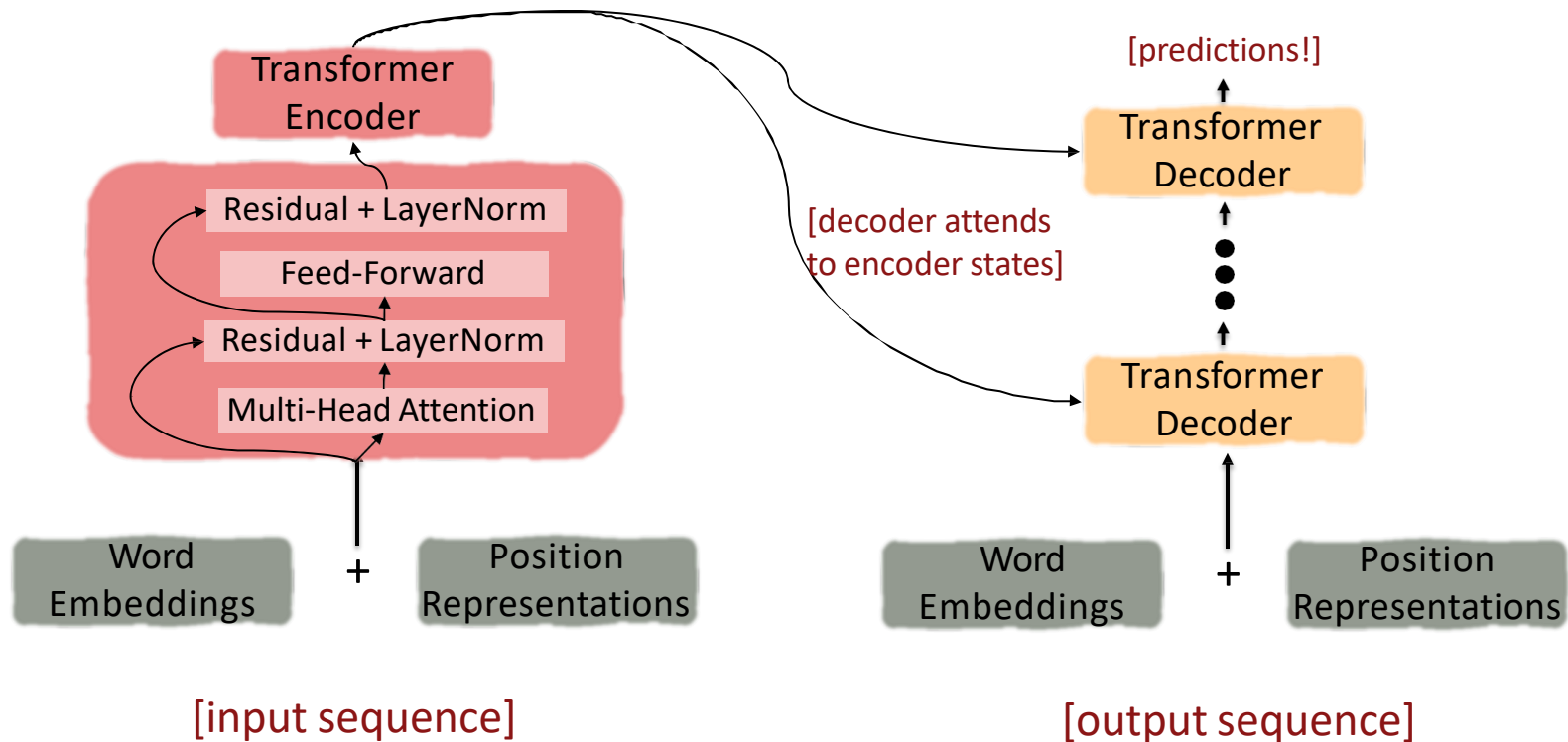
What's in a Transformer Encoder Block?

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! (see hidden slides)**
 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-Decoder

[Vaswani et al., 2017]

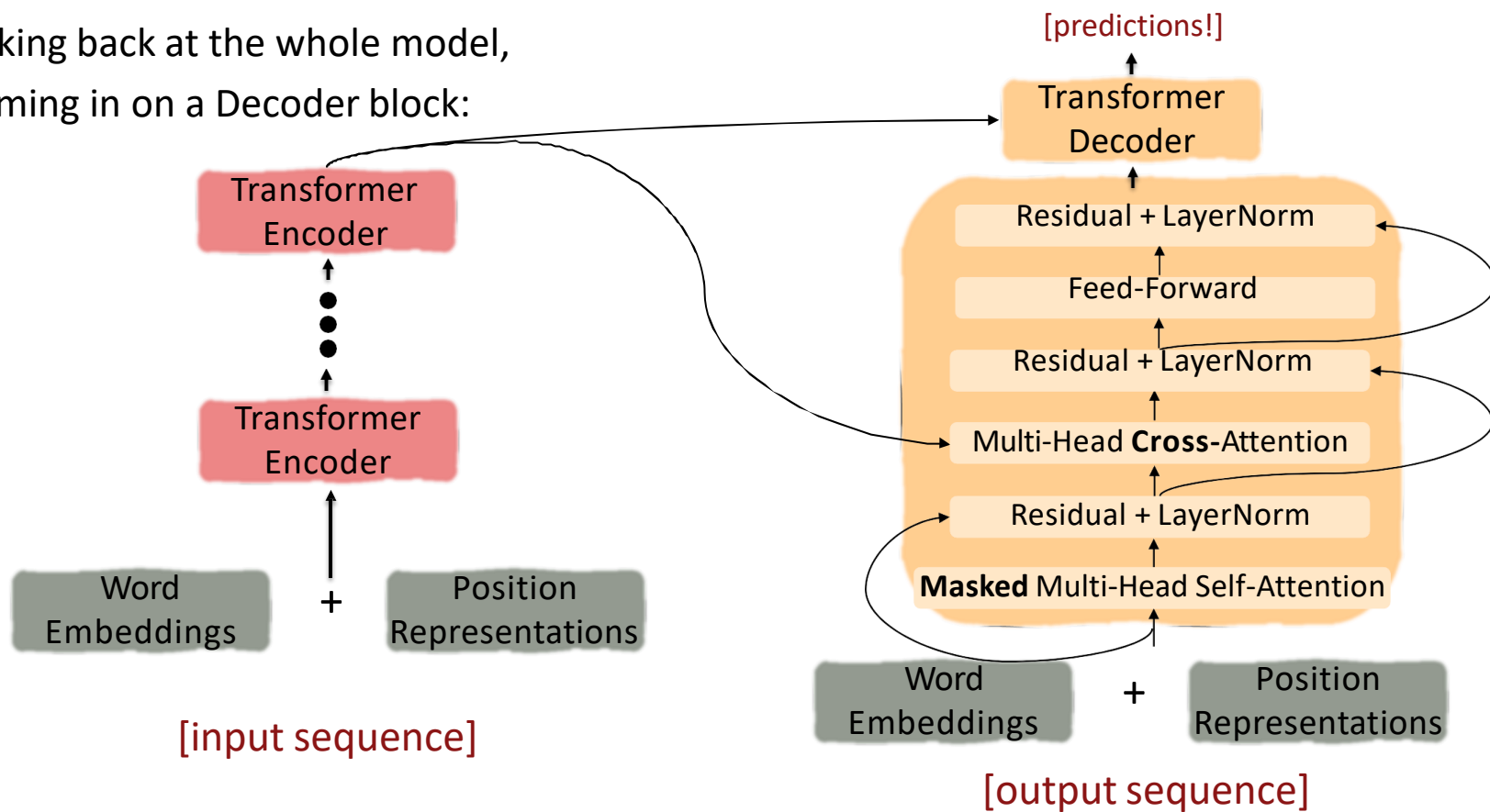
Looking back at the whole model, zooming in on an Encoder block:



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):
 - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.

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

$$\begin{matrix} XQ \\ K^T X^T \end{matrix} = XQK^T X^T \in \mathbb{R}^{T \times T}$$

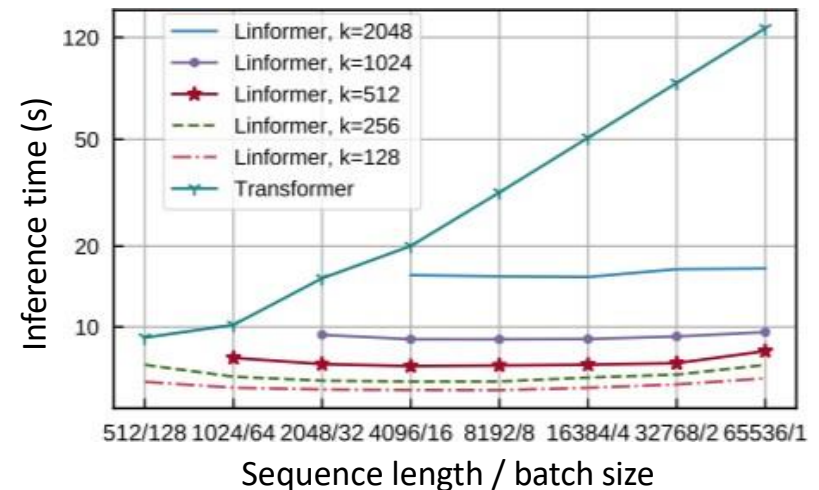
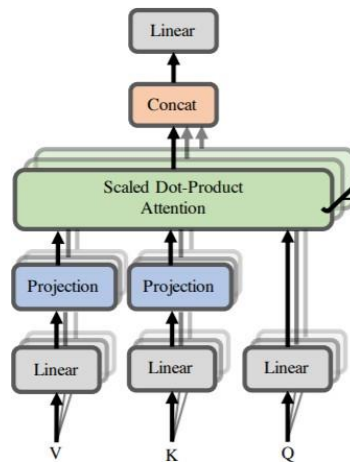
Need to compute all
pairs of interactions!
 $O(T^2d)$

- Think of d as around **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 $T \geq 10,000$?** 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** [\[Wang et al., 2020\]](#)

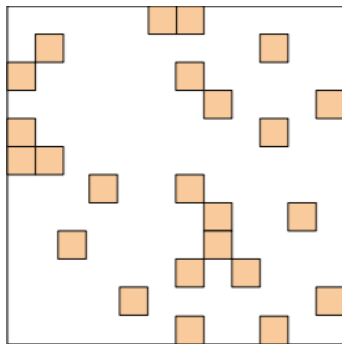
Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



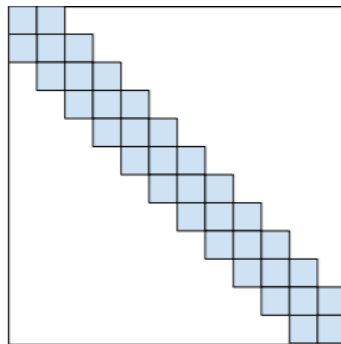
Recent work on improving on quadratic self-attention cost

- 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** [\[Zaheer et al., 2021\]](#)

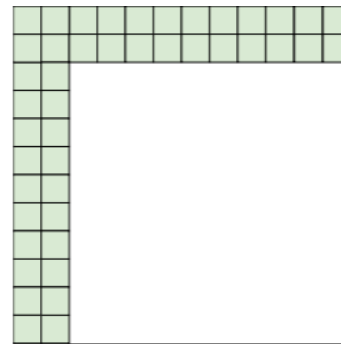
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows, looking at everything**, and **random interactions**.



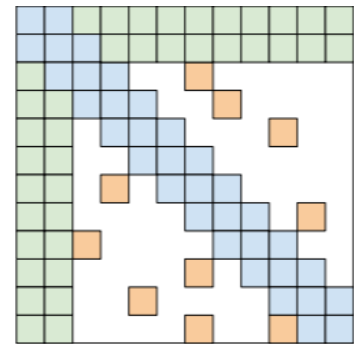
(a) Random attention



(b) Window attention



(c) Global Attention

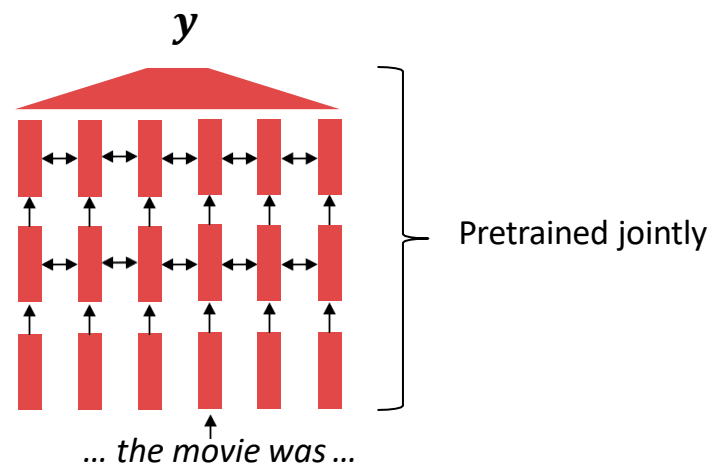


(d) BIGBIRD

Pretraining models

In modern NLP:

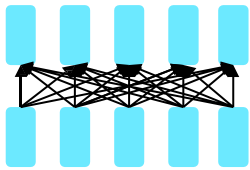
- All (or almost all) parameters in NLP networks are initialized via **pretraining**.
- Pretraining methods hide parts of the input from the model, and train the model to reconstruct those parts.
- This has been exceptionally effective at building strong:
 - **representations of language**
 - **parameter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



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

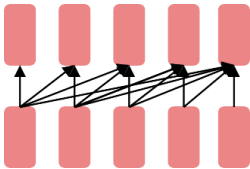
Pretraining for three types of architectures

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



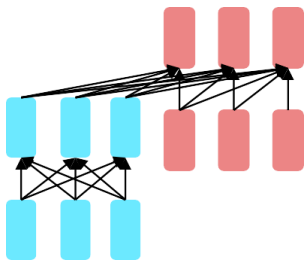
Encoders

- 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?

Pretraining through language modeling

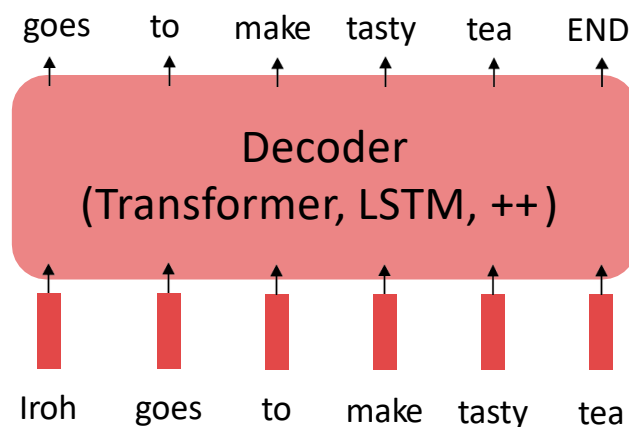
[Dai and Le, 2015]

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.

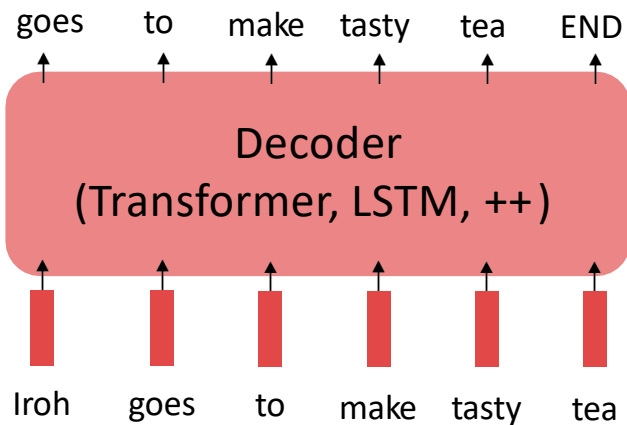


The Pretraining / Finetuning Paradigm

Pretraining can improve NLP applications by serving as parameter initialization.

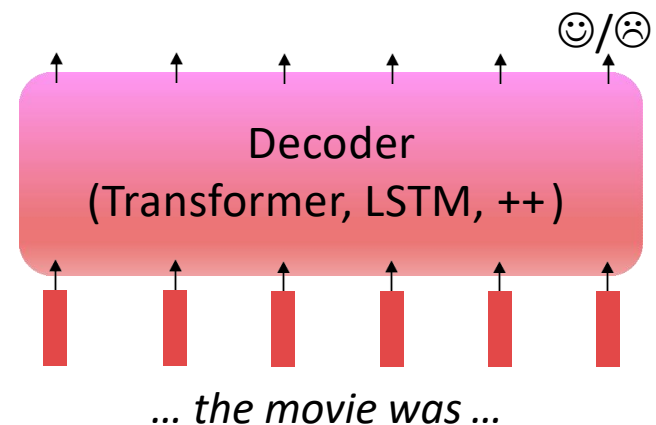
Step 1: Pretrain (on language modeling)

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 Fibonacci sequence]
- Models also learn – and can exacerbate racism, sexism, all manner of bad biases.

Pretraining encoders:

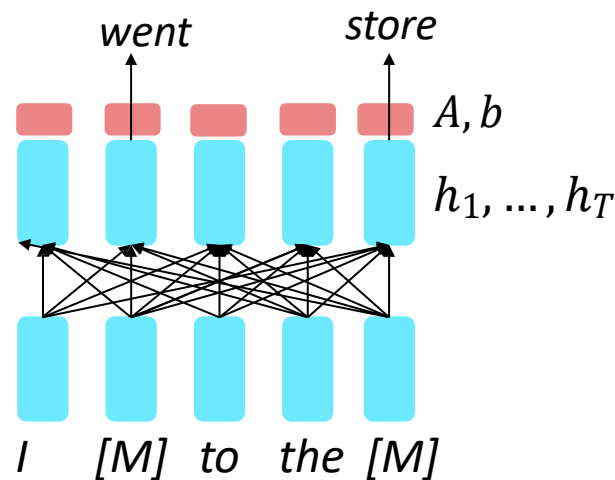
What pretraining objective to use?

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**.



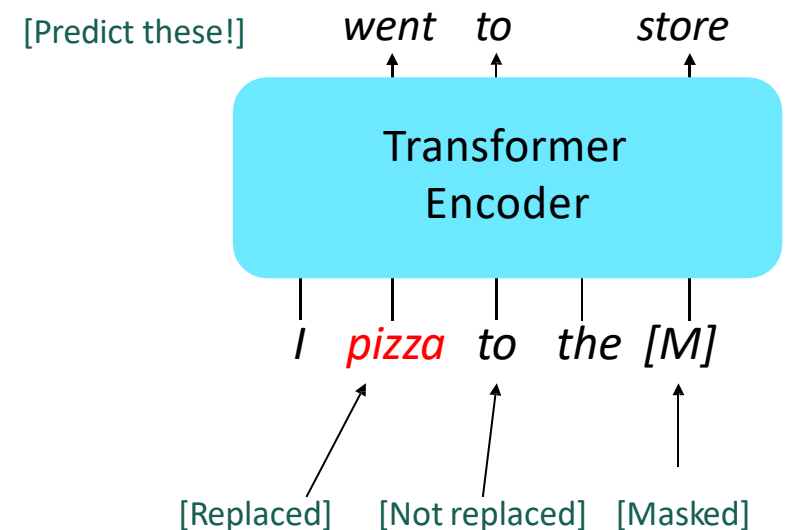
[Devlin et al., 2018]

BERT: Bidirectional Encoder Representations from Transformers

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 Transformers

- 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 Transformers

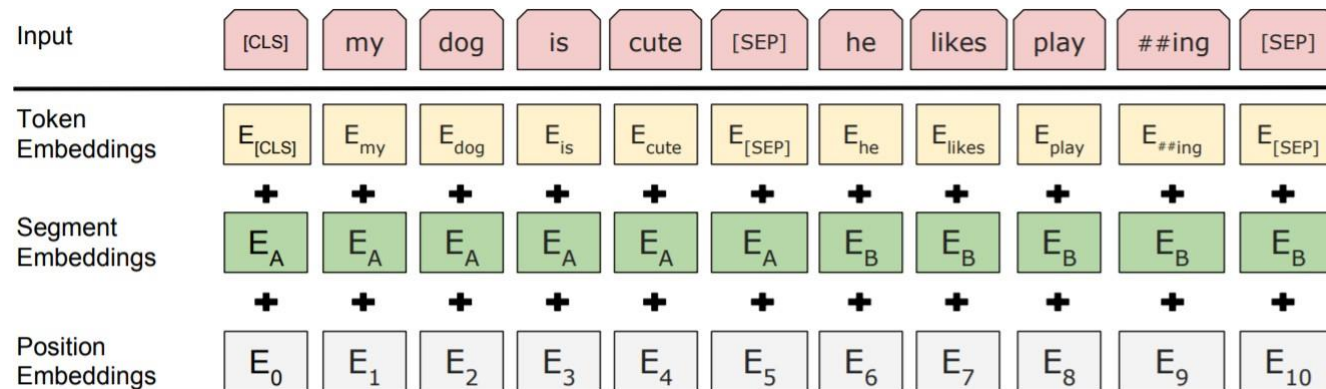
- Additional task: Next sentence prediction
- To learn *relationships* between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

```
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
```

BERT: Bidirectional Encoder Representations from Transformers

- 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]

BERT: Bidirectional Encoder Representations from Transformers

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 Transformers

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) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average -
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
BERT _{BASE}	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT _{LARGE}	86.7/85.9	72.1	92.7	94.9	60.5	86.5	89.3	70.1	82.1

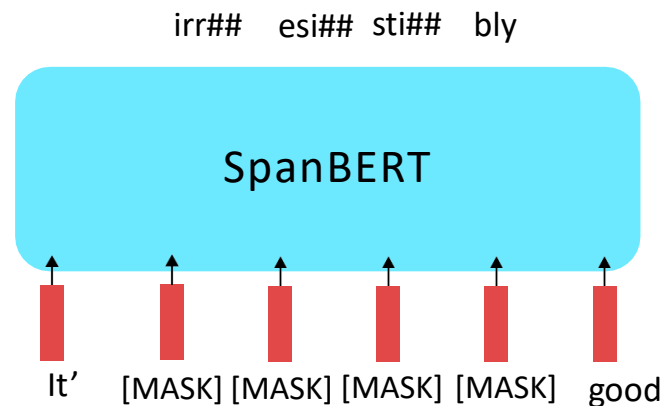
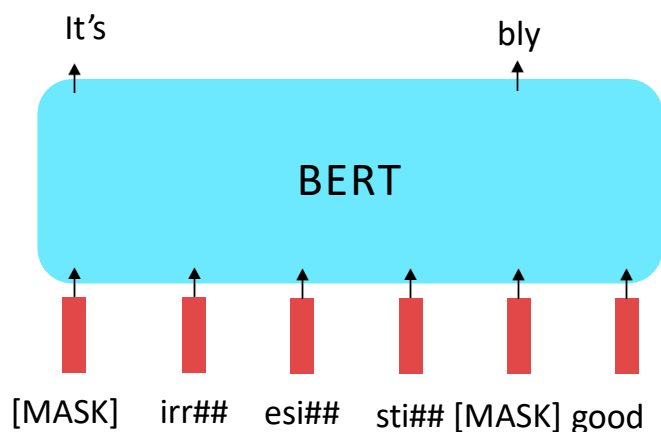
[Devlin et al., 2018]

Extensions of BERT

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

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
BERT _{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

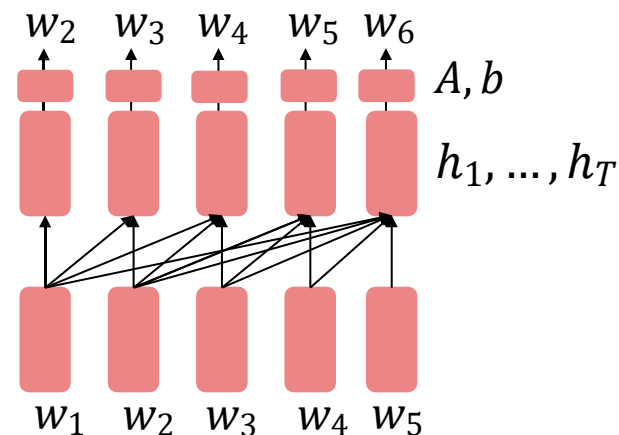
It's natural to pretrain decoders as language models and then use them as generators, finetuning their $p_{\theta}(w_t|w_{1:t-1})$!

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)

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

Where A, b were pretrained in the language model!



[Note how the linear layer has been pretrained.]

Pretraining decoders

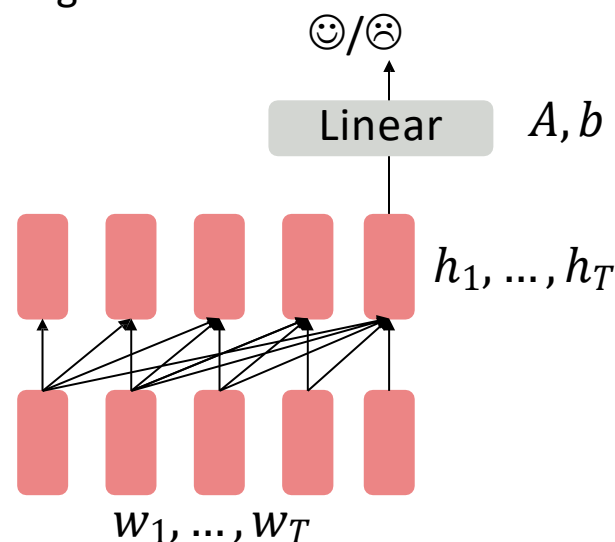
When using language model pretrained decoders, we can ignore that they were trained to model $p(w_t|w_{1:t-1})$.

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.]

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]	A								
Form										Non-finite forms										Negative									
Infinitive										Positive										kutoambia									
Positive form										Simple finite forms																			
Imperative										Singular																			
Habitual										Plural										ambieni									
										huambia																			
Persons										Persons / Classes										Classes									
Polarity	Sg.	1st	Pl.	Sg.	2nd	Pl.	Sg.	3rd / 1	M-wa	Pl. / 2	3	M-mi	4	5	Ma	6	7	Ki-vi	8	9	N	10	11 / 14	15 / 17	Pa	16	Mu	18	
Positive	niliambia	tulambia	ulambia	mlambia	alambia	walambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia	ilambia
Negative	nilambia	nalambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia
Complex finite forms																													
Past																													
Positive	ninaambia	tunaambia	unaambia	mnaambia	anaambia	wanaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia	inaambia
Negative	nilambia	nalambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia
Present																													
Positive	nitaambia	tutaambia	utaambia	mtaambia	ataambia	wataambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia
Negative	nilambia	nalambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia
Future																													
Positive	nitaambia	tutaambia	utaambia	mtaambia	ataambia	wataambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia	itaambia
Negative	nilambia	nalambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia	hukambia
Subjunctive																													
Positive	niambie	tuambie	uambie	mambie	aambie	waambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie	uambie
Negative	nilambie	nalambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie
Present Conditional																													
Positive	ningeambia	tungeambia	ungeambia	mungeambia	angeambia	wangeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia	ungeambia
Negative	nisingeambia	nalingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia	hukingeambia
Past Conditional																													
Positive	ningaliambia	tungaliambia	ungaliambia	mngaliambia	angaliambia	wangaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia	ungaliambia
Negative	nilingaliambia	nalingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia	hukingaliambia
Conditional Contrary to Fact																													
Positive	ningeliambia	tungeliambia	ungeliambia	mngeliambia	angeliambia	wangeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia	ungeliambia
Negative	nilingeliambia	nalingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia	hukingeliambia
Gnomic																													
Positive	naambia	twaambia	waambia	mwaambia	aambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia	waambia
Negative	nilaambia	nalambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie	hukambie
Perfect																													
Positive	namambia	twamambia	wamambia	mnamambia	amambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia	wamambia
Negative	nilamambia	nalamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia	hukamambia

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

Pretraining encoder-decoders: What pretraining objective to use?

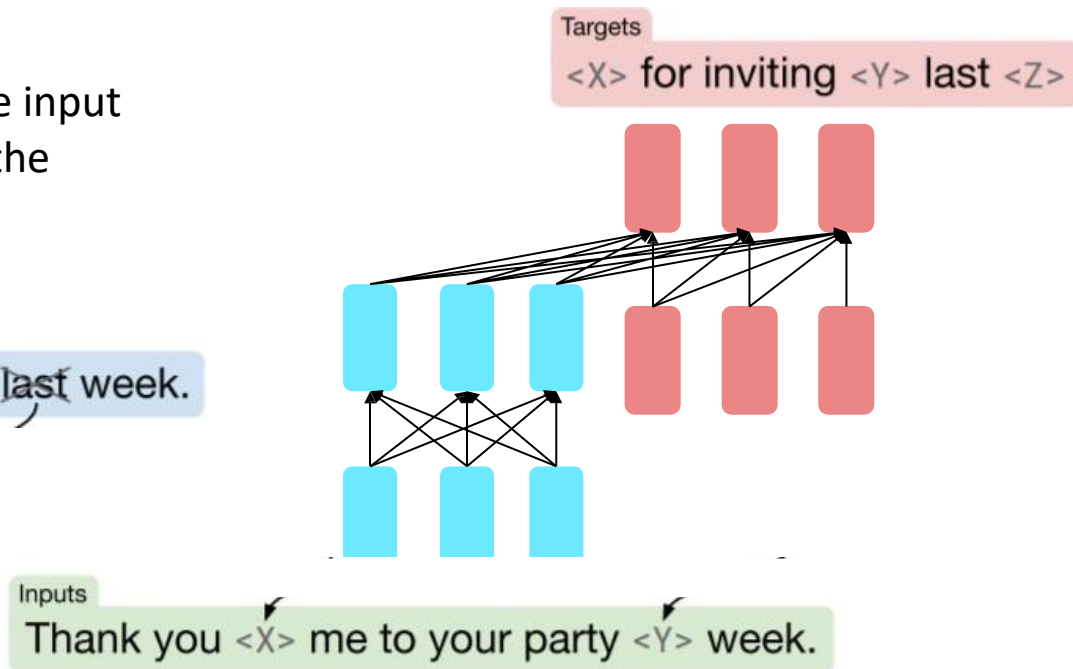
What [Raffel et al., 2018](#) found to work best was **span corruption**. Their model: **T5**.

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

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.

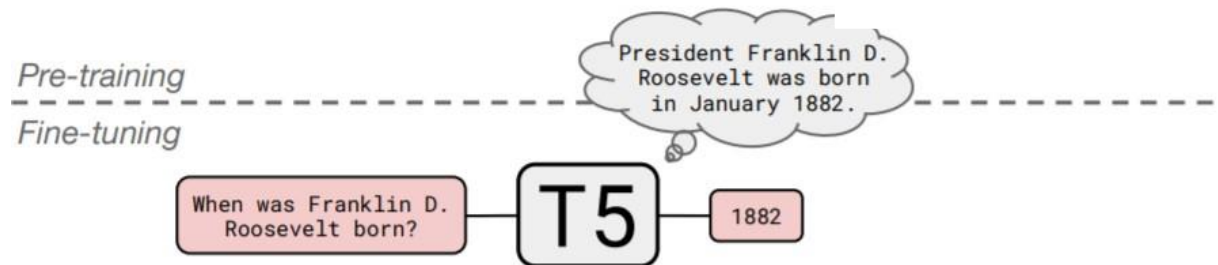
This is implemented in text preprocessing: it's still an objective that looks like **language modeling** at the decoder side.



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	test	
<u>Karpukhin et al. (2020)</u>	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
<u>T5.1.1-XXL + SSM</u>	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 -> ”

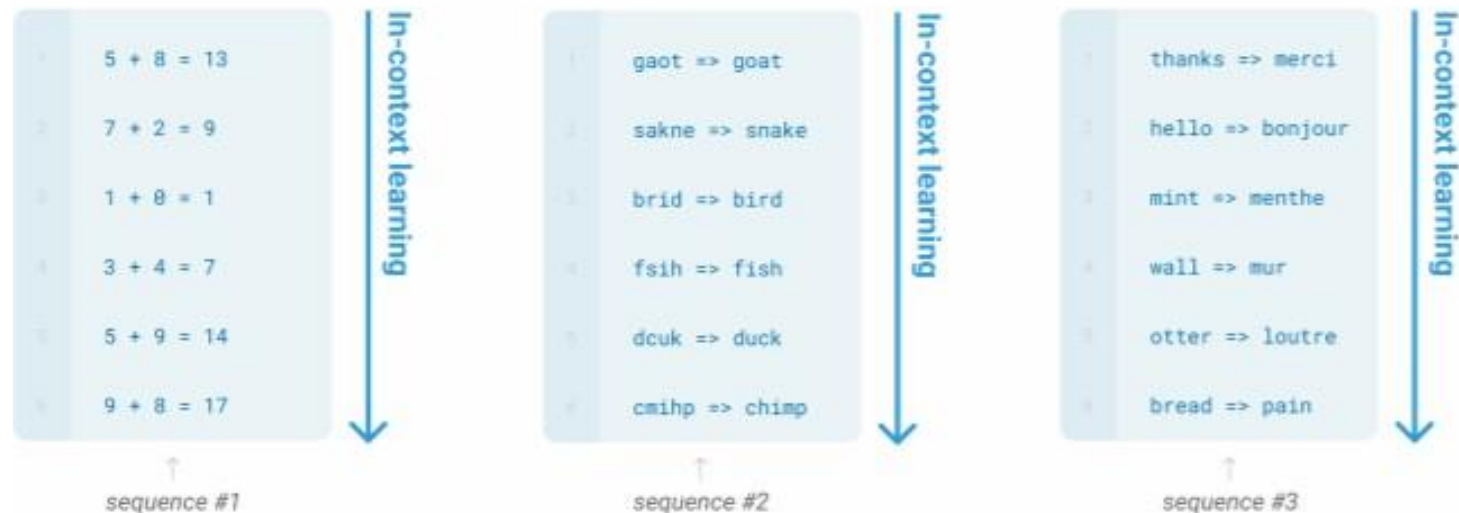
Output (conditional generations):

loutre...”

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

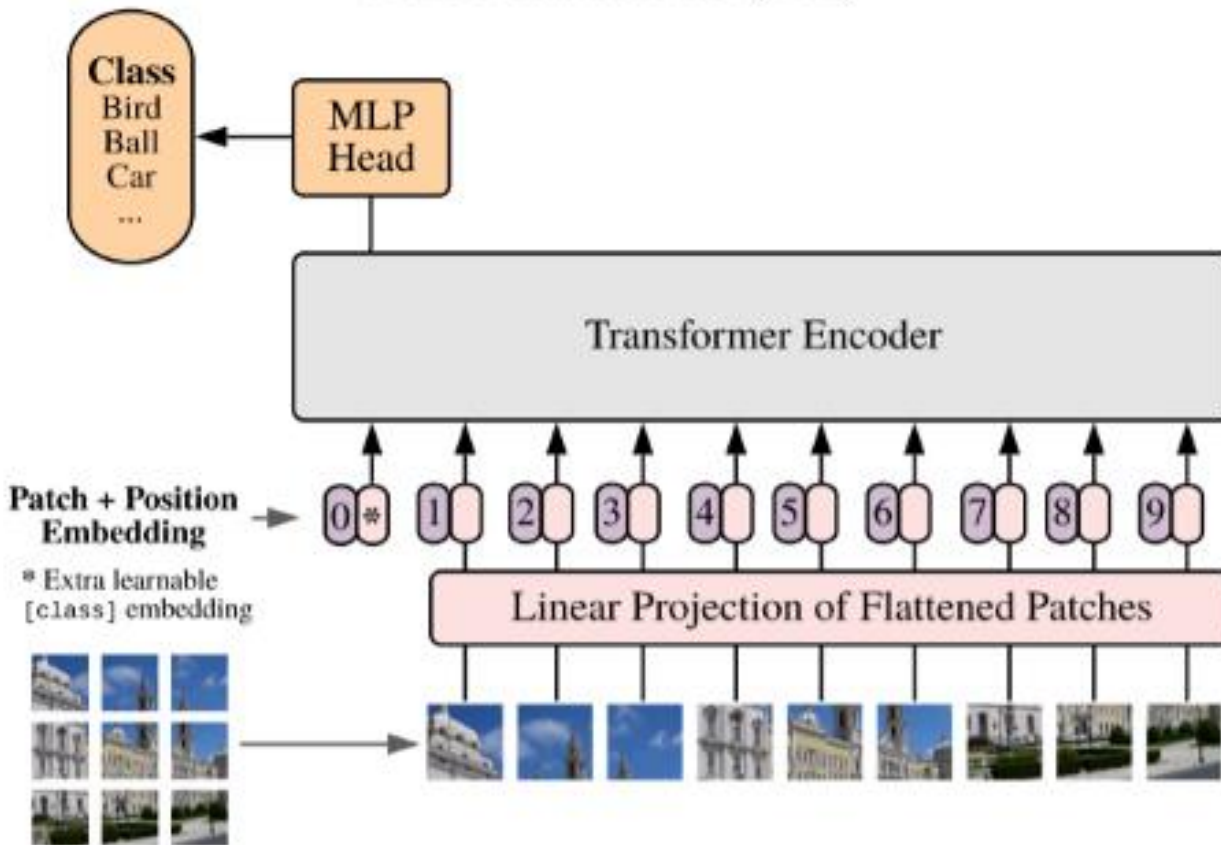
Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.

Learning via SGD during unsupervised pre-training

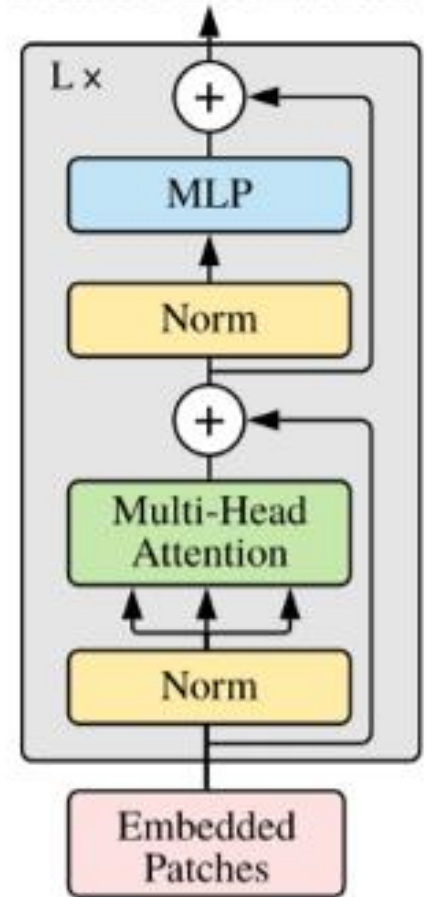


Transformers in vision

Vision Transformer (ViT)



Transformer Encoder



Cross-modal transformers

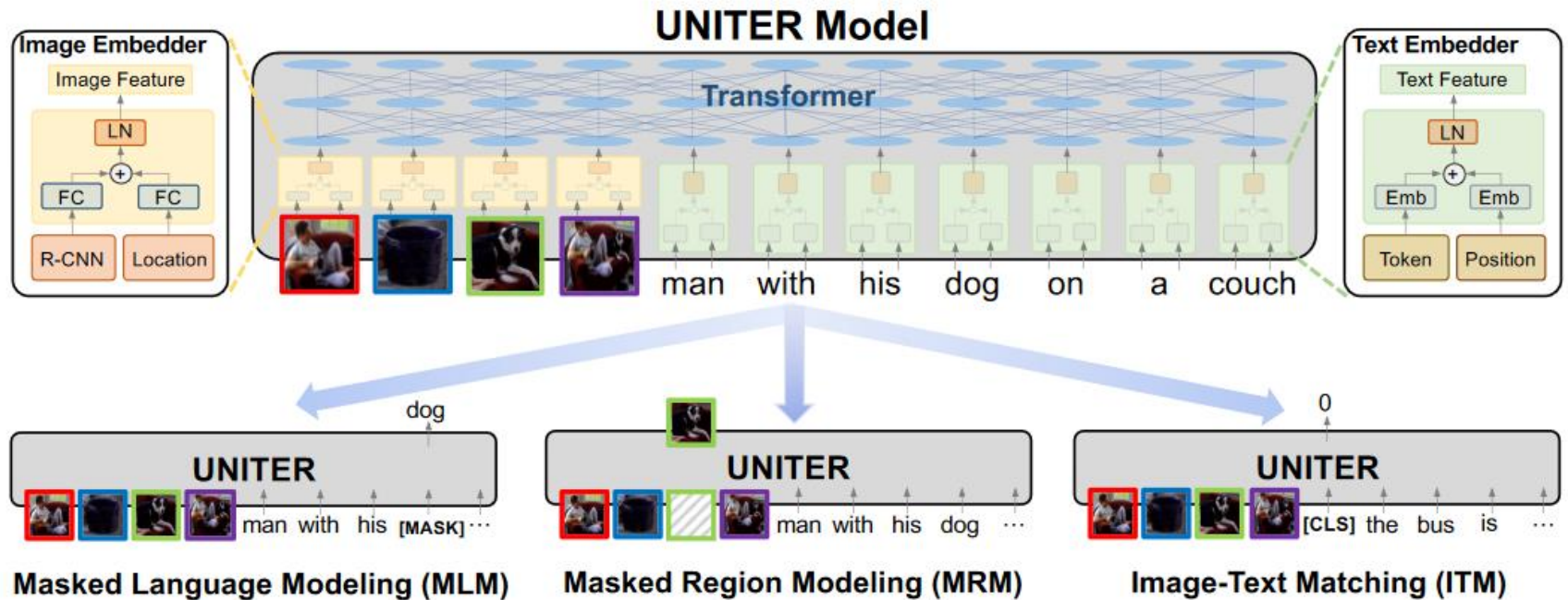


Figure 1: Overview of the proposed UNITER model (best viewed in color), consisting of an Image Embedder, a Text Embedder and a multi-layer self-attention Transformer, learned through three pre-training tasks.

Cross-modal transformers

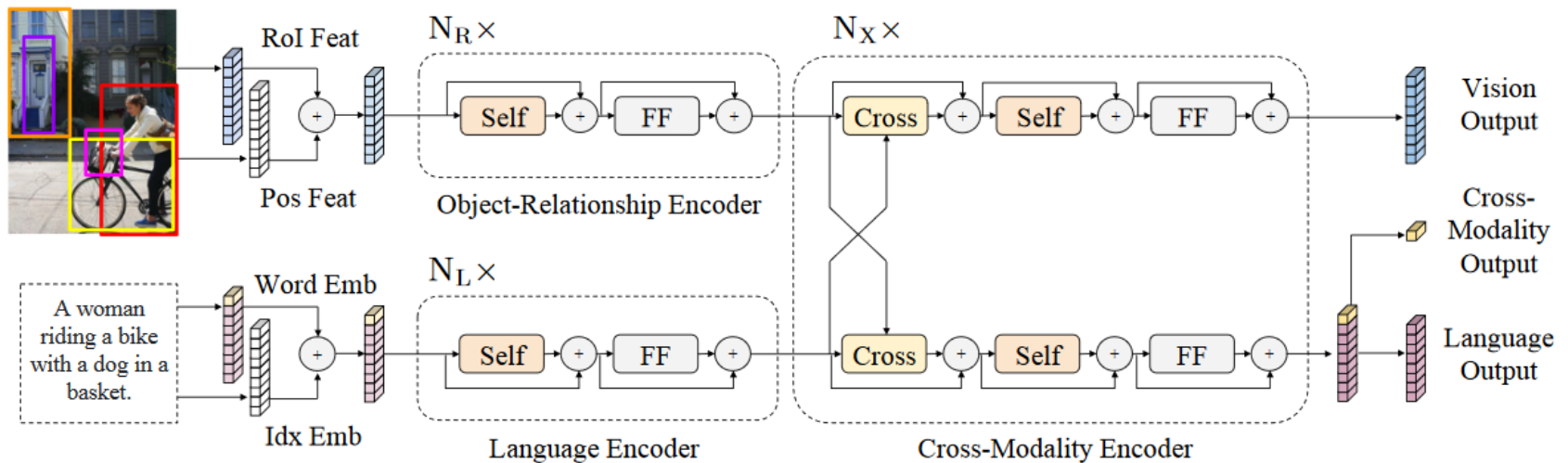


Figure 1: The LXMERT model for learning vision-and-language cross-modality representations. ‘Self’ and ‘Cross’ are abbreviations for self-attention sub-layers and cross-attention sub-layers, respectively. ‘FF’ denotes a feed-forward sub-layer.

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](#)

