CSCE 5218 & 4930
Deep Learning

Transformers

Slides adapted from Adriana Kovashka

Plan for this lecture

- Background
  – Context prediction, unsupervised learning
- Transformer models
  – Self-attention
  – Adapting self-attention for sequential data
  – The transformer architecture, encoder/decoder
- 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:

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:

$$\text{motel} = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0]$$

$$\text{hotel} = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$

These two vectors are orthogonal.

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

Solution:

- **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***

Word vectors

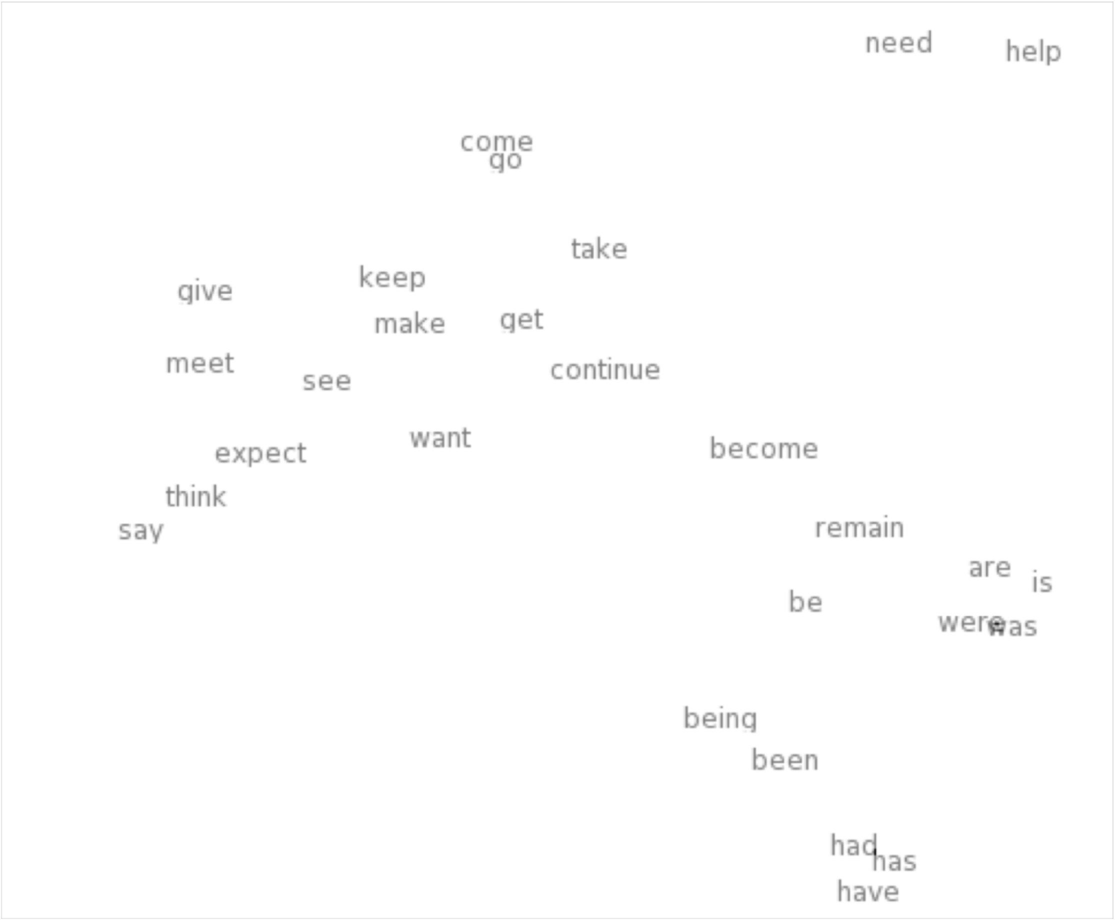
We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts

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

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

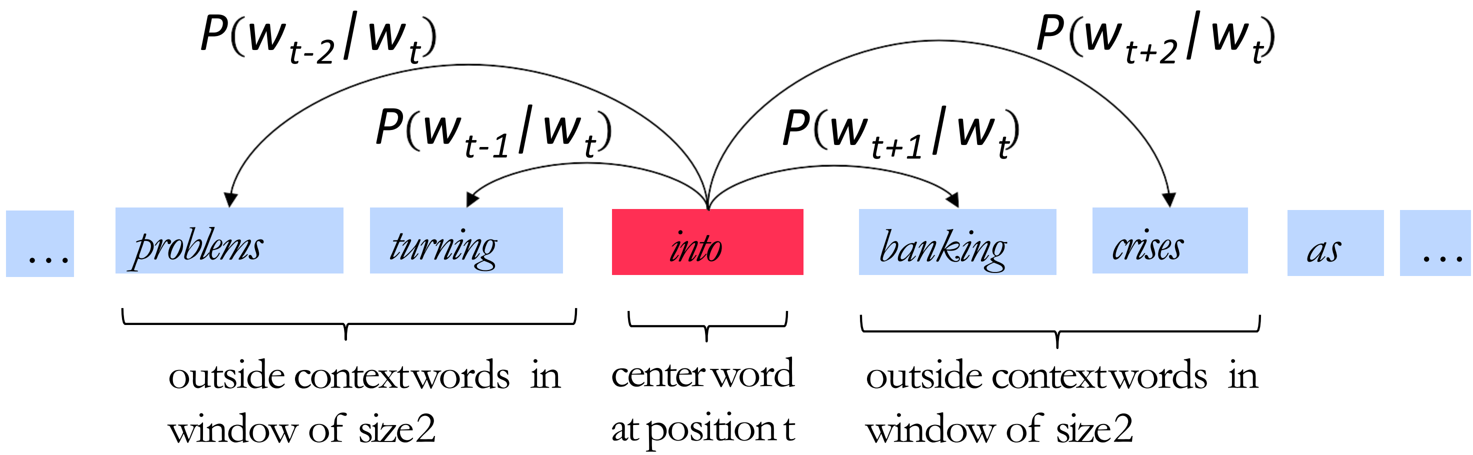
Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

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

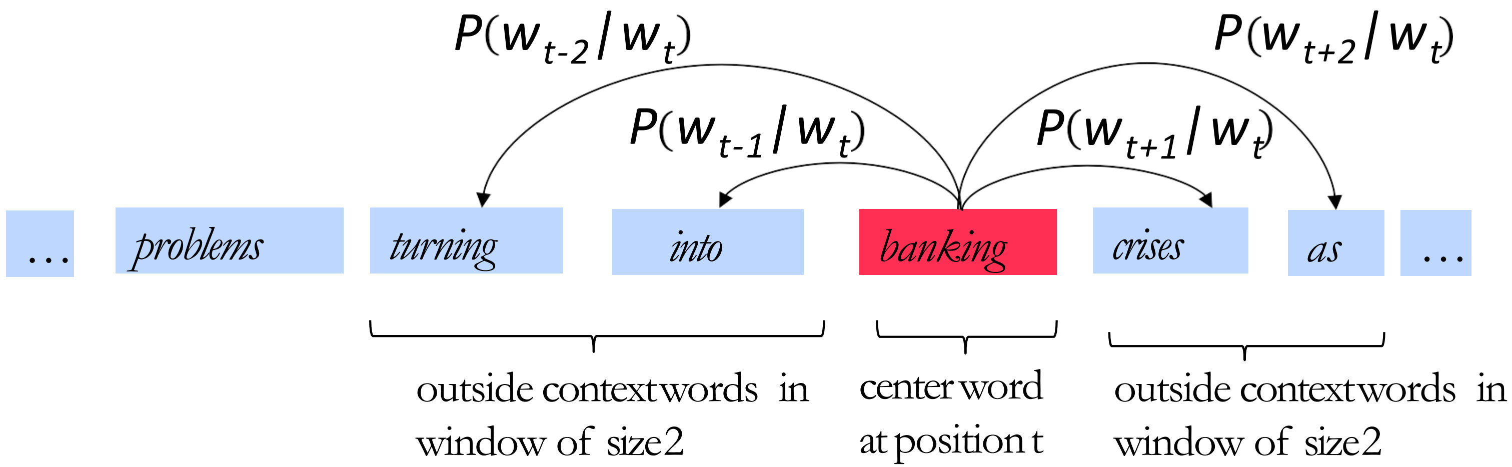
Word2Vec Overview

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

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

θ is all variables to be optimized

Word2Vec: objective function

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

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

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:

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

- <u>Question:</u> How to calculate $P(w_{t+j} | w_t ; \theta)$?

Word2Vec: objective function

- We want to minimize the objective function:

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

- <u>Question:</u> How to calculate $P(w_{t+j} | w_t; \theta)$?
- <u>Answer:</u> 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

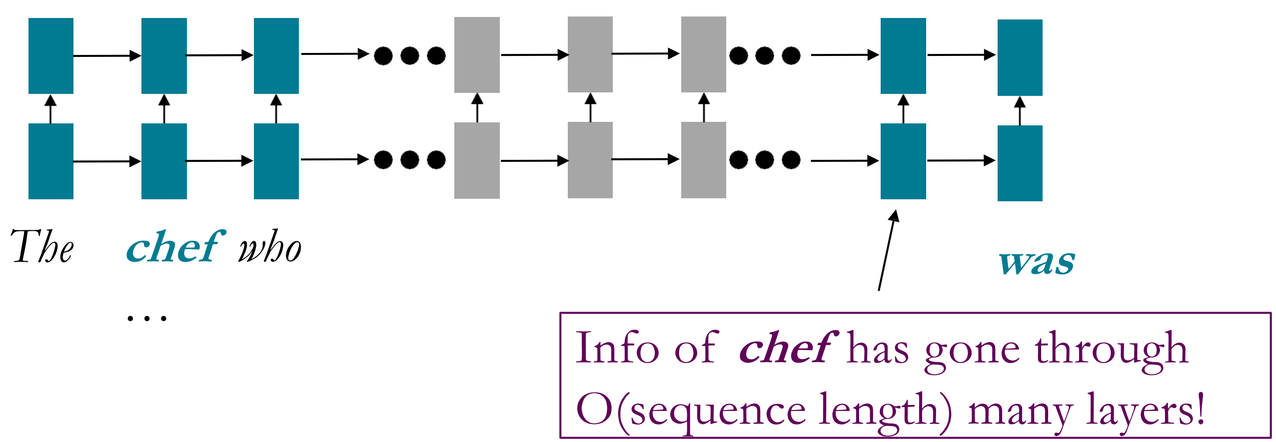
- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow \mathbb{R}^n$

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

- 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

Issues with recurrent models: **Linear interaction distance**

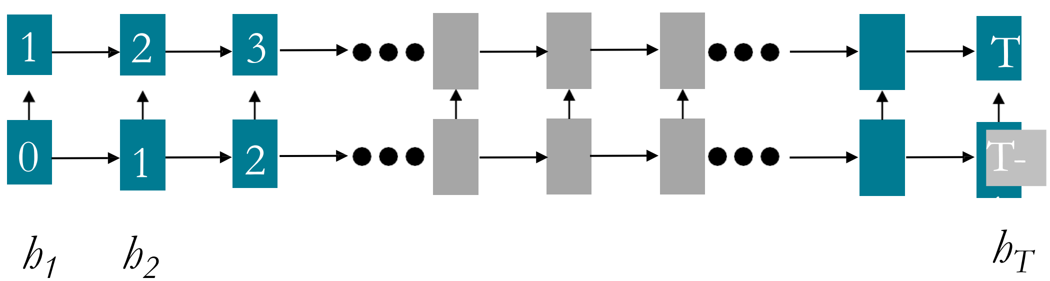
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; not necessarily the right way to think about sentences…



Adapted from John Hewitt

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

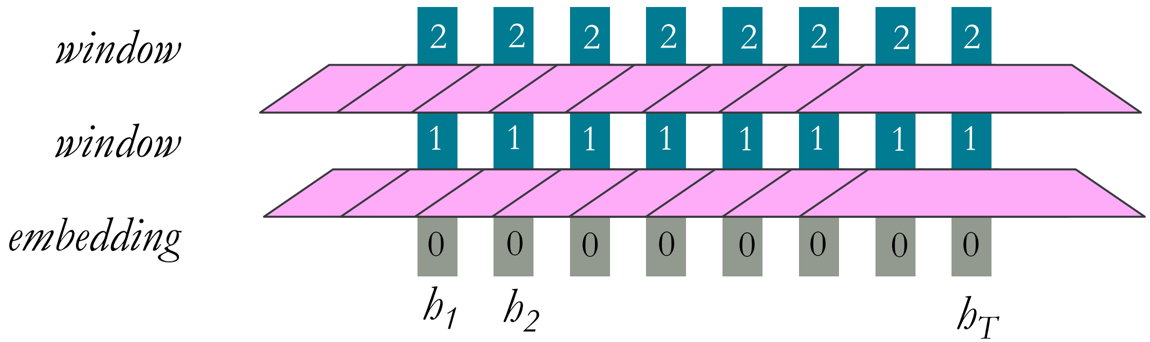
- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



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!

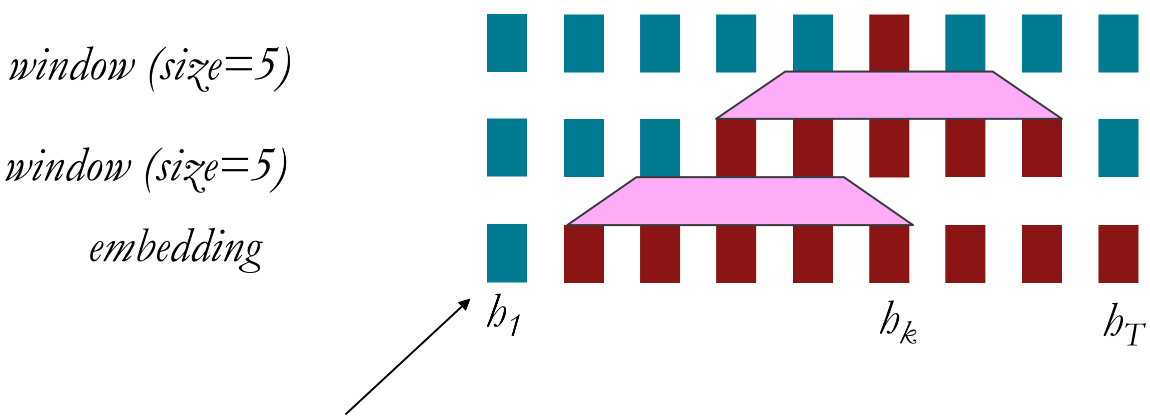


Numbers indicate min # of steps before a state can be computed

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



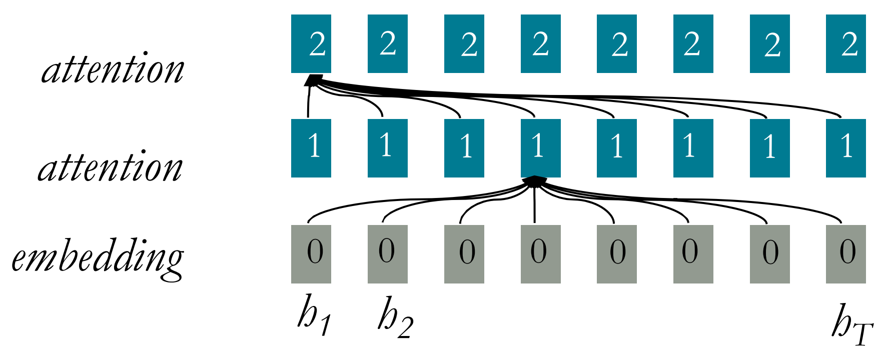
Red states indicate those "visible" to h_k

Too far from h_k to be considered

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** $q_1, q_2, \ldots, q_T$. Each query is $q_i \in \mathbb{R}^d$
  - We have some **keys** $k_1, k_2, \ldots, k_T$. Each key is $k_i \in \mathbb{R}^d$
  - We have some **values** $v_1, v_2, \ldots, v_T$. 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 $x_1, \ldots, x_T$, (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

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

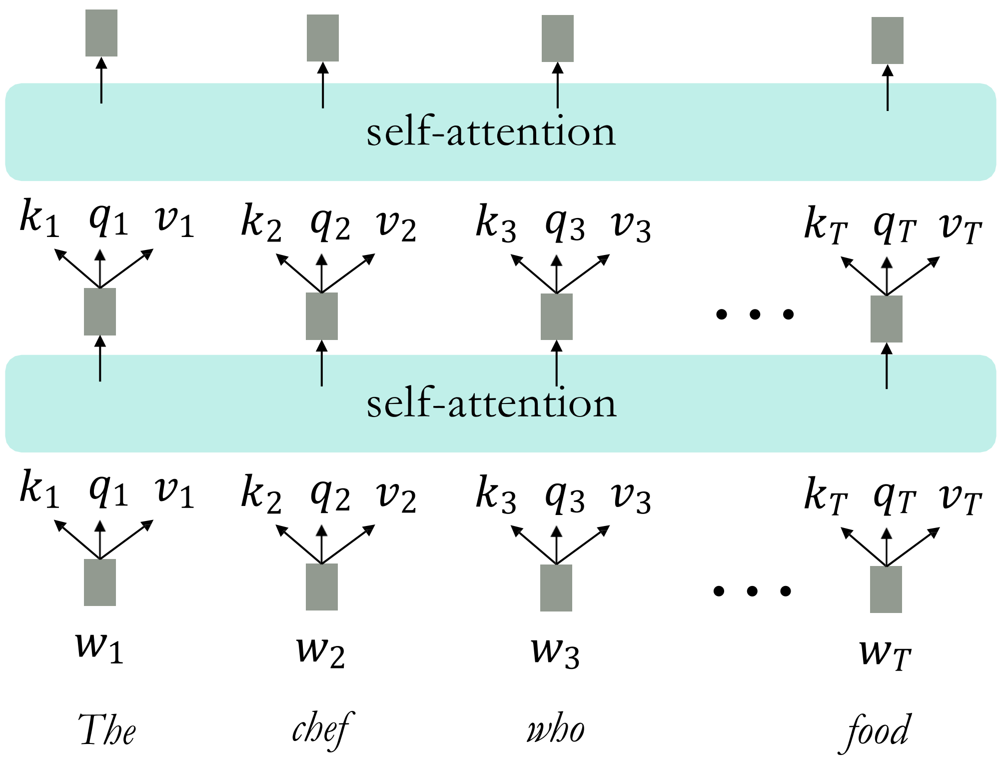
Compute attention weights from affinities (softmax)

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

Compute outputs as weighted sum of **values**

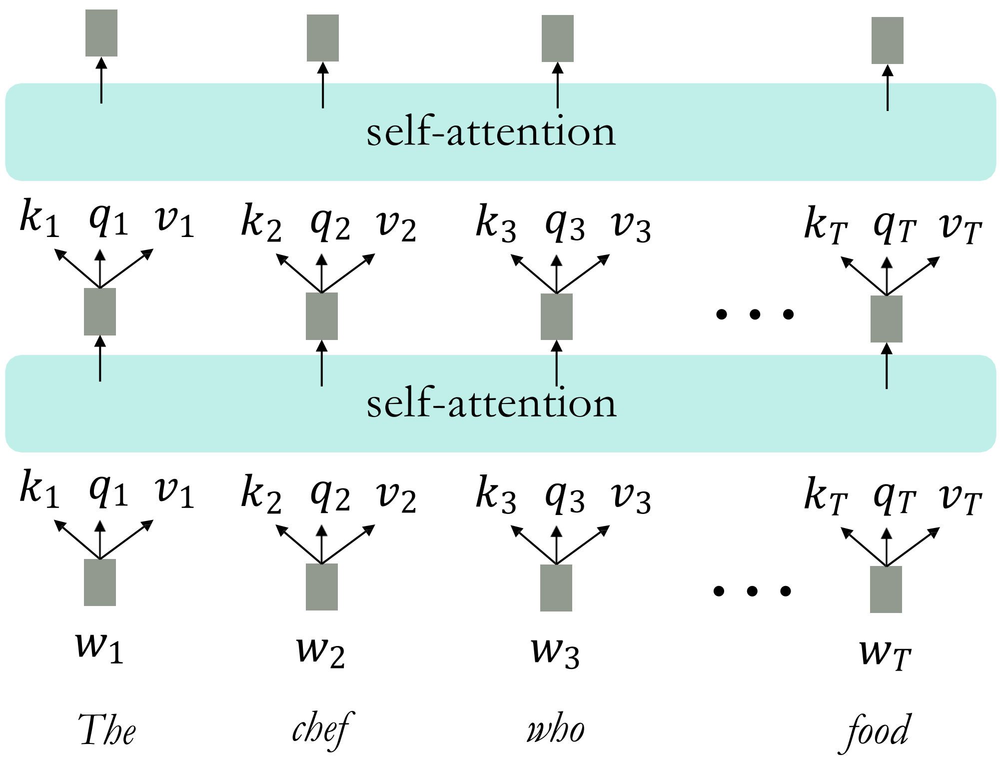
Self-Attention

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.



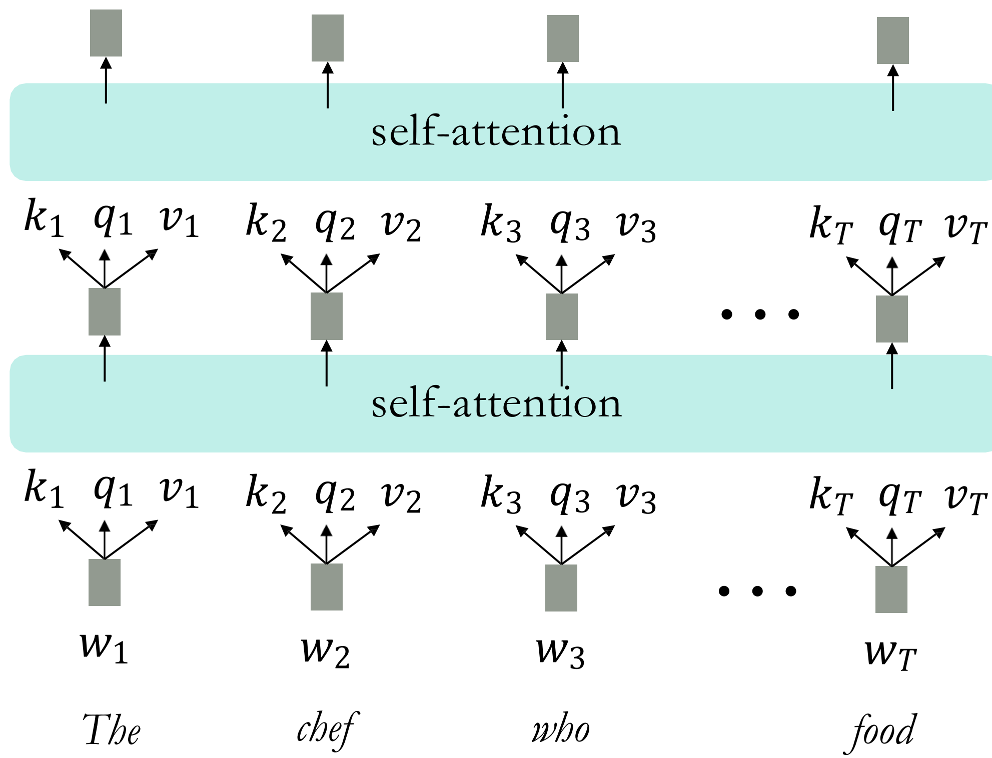
Self-Attention

- 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?



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**

$$p_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,T\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let v_i', k_i', q_i' be our old values, keys, and queries.

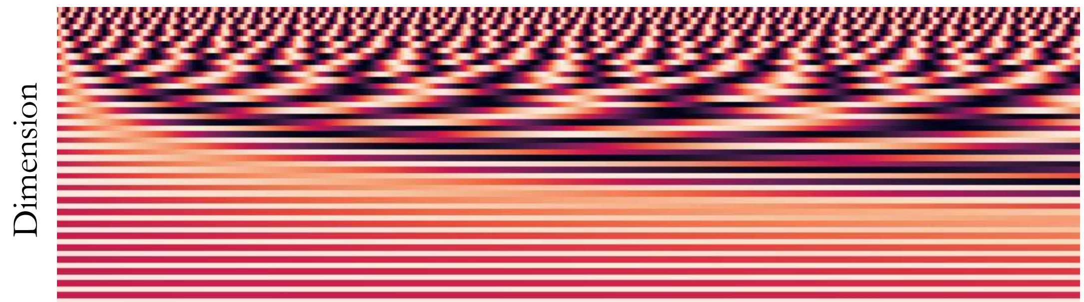
$$v_i = v_i' + p_i$$
$$q_i = q_i' + p_i$$
$$k_i = k_i' + p_i$$

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

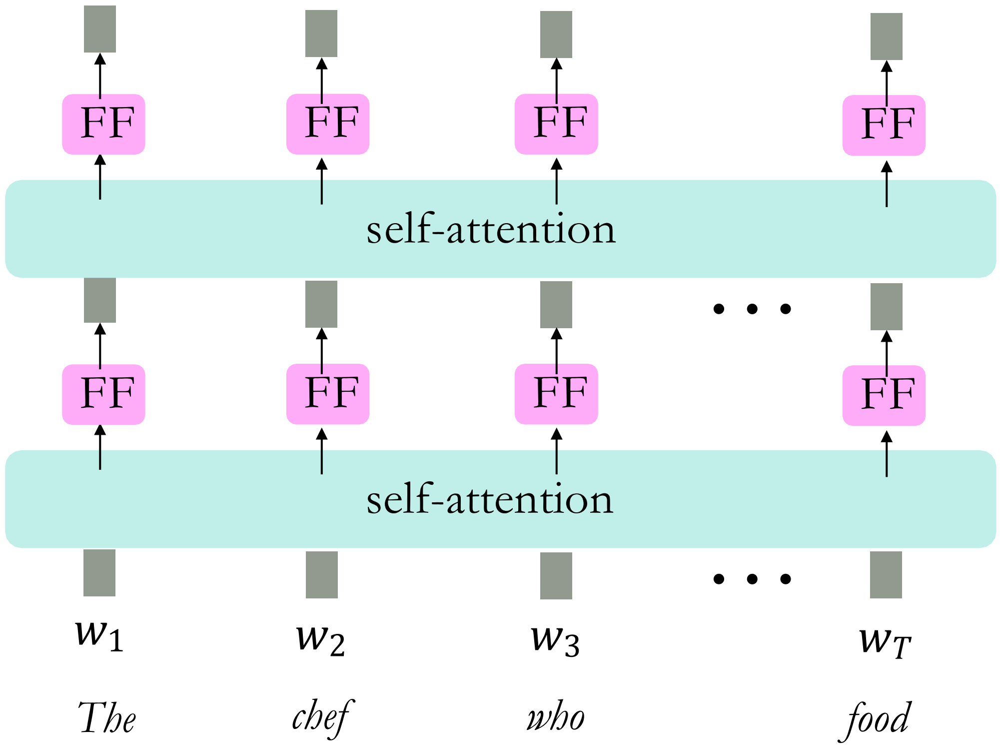
- Add position representations to the inputs

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

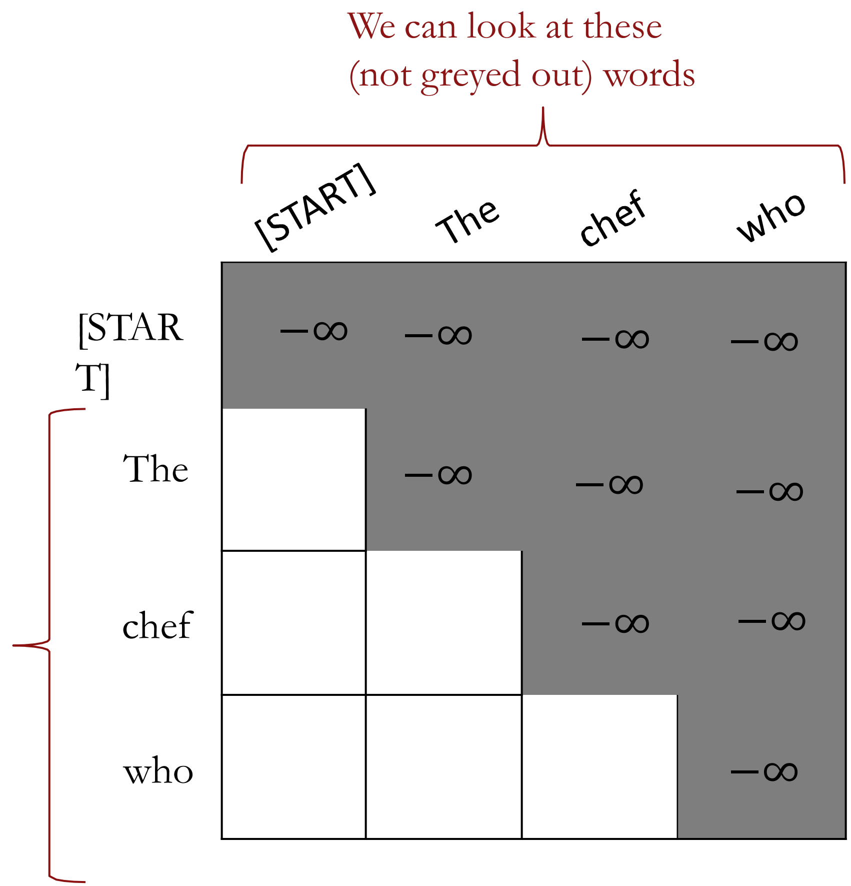
Solutions

- 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 $-\infty$.

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



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.