Natural Language Processing

AI51701/CSE71001

Lecture 13

10/31/2023

Instructor: Taehwan Kim

Announcements

- We have **no class on Nov. 2** (due to a business trip)
  - Will have a substitute class in the final week (likely on Dec. 12)

- Assignment 2 will be out today
  - Due will be by the end of next week (Nov. 12)

Machine Translation

Beam search decoding

- Core idea: On each step of decoder, keep track of the k most probable partial translations (which we call ***hypotheses***)
  - *K* is the **beam size**(in practice around 5 to 10, in NMT)

- A hypothesis $y_1, \ldots, y_t$ has a score which is its log probability:

$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

  - Scores are all negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step

- Beam search is **not guaranteed** to find optimal solution
- But **much more efficient** than exhaustive search!

Beam search decoding: example

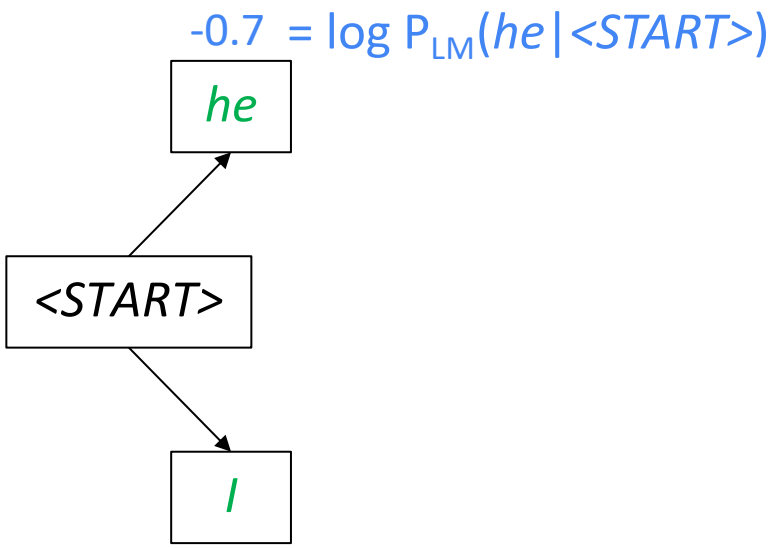
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$

<START>

Calculate prob dist of next word

Beam search decoding: example

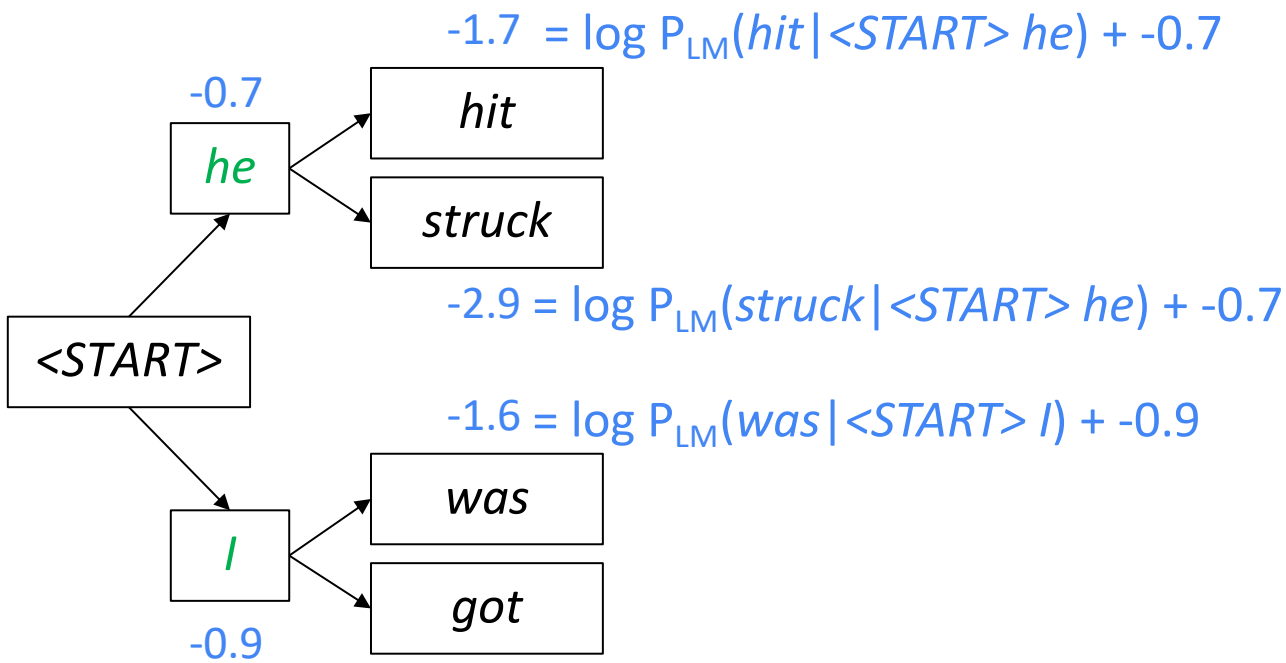
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Take top *k* words and compute scores

Beam search decoding: example

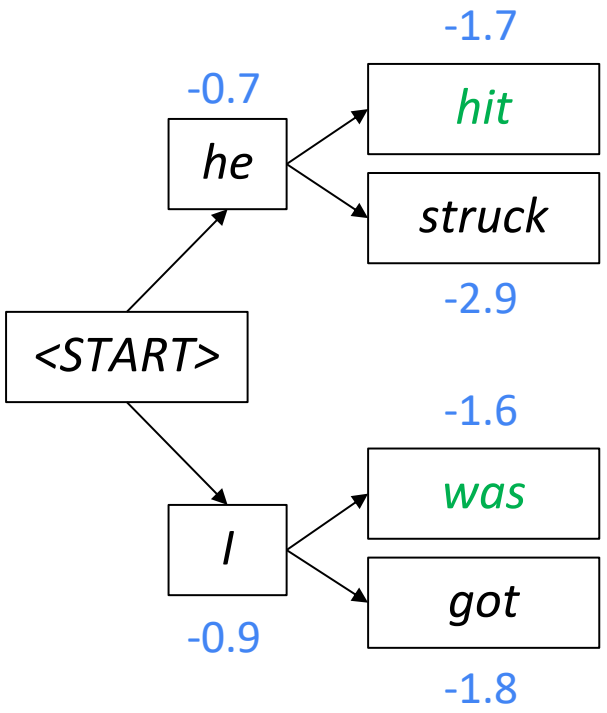
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

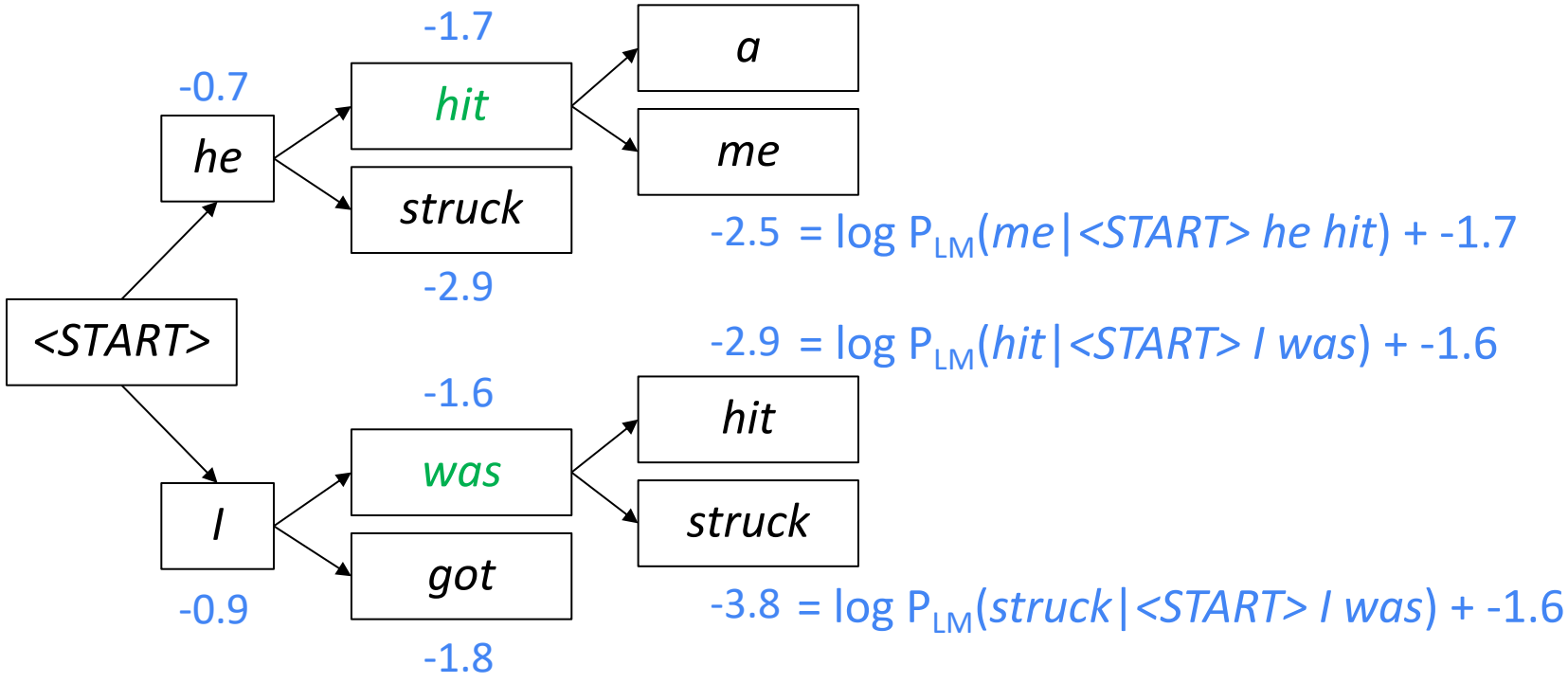
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses,
just keep k with highest scores

Beam search decoding: example

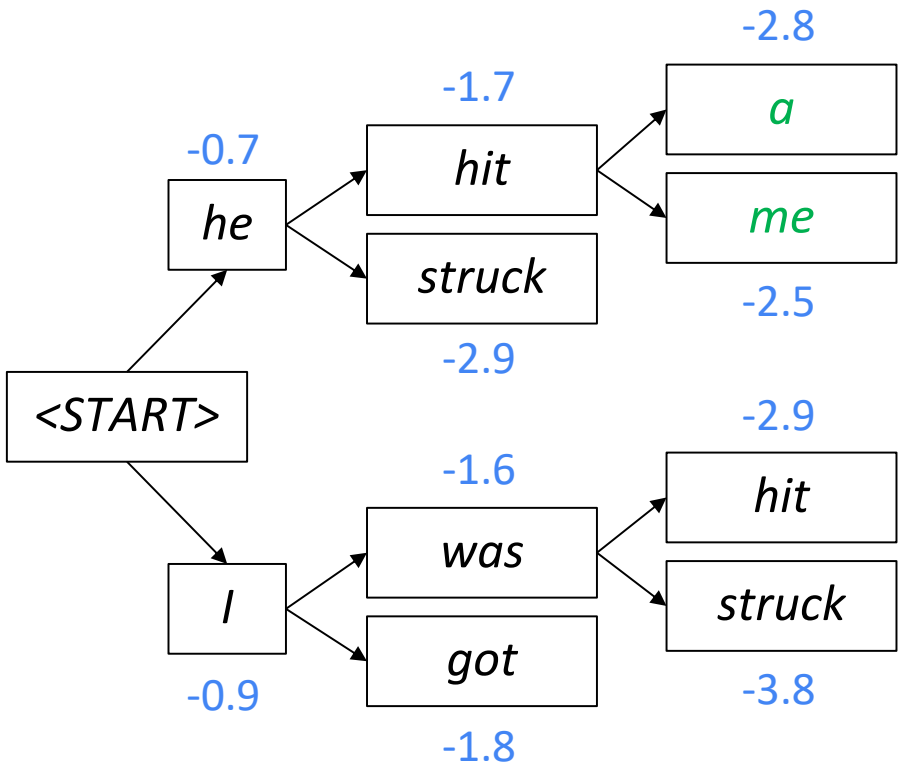
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

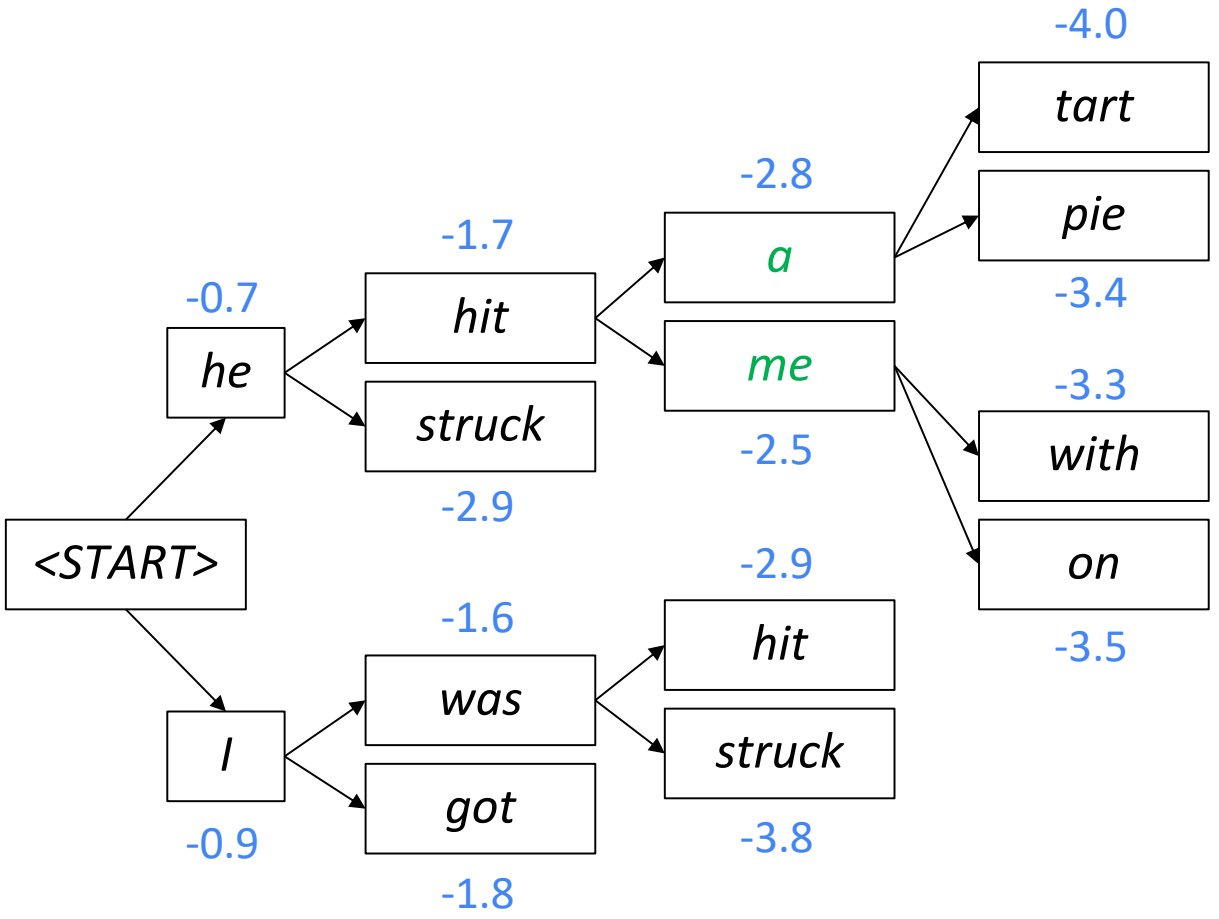
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses,
just keep k with highest scores

Beam search decoding: example

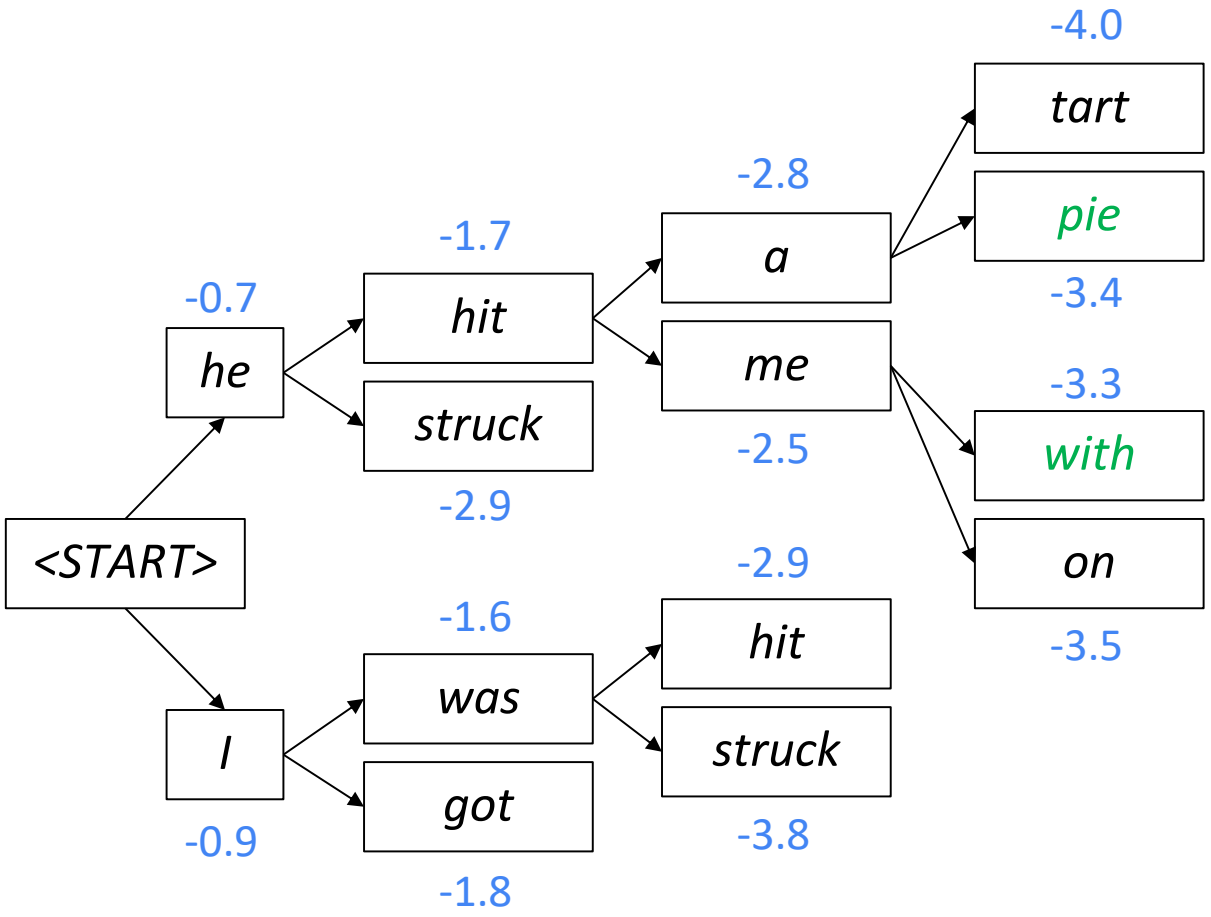
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

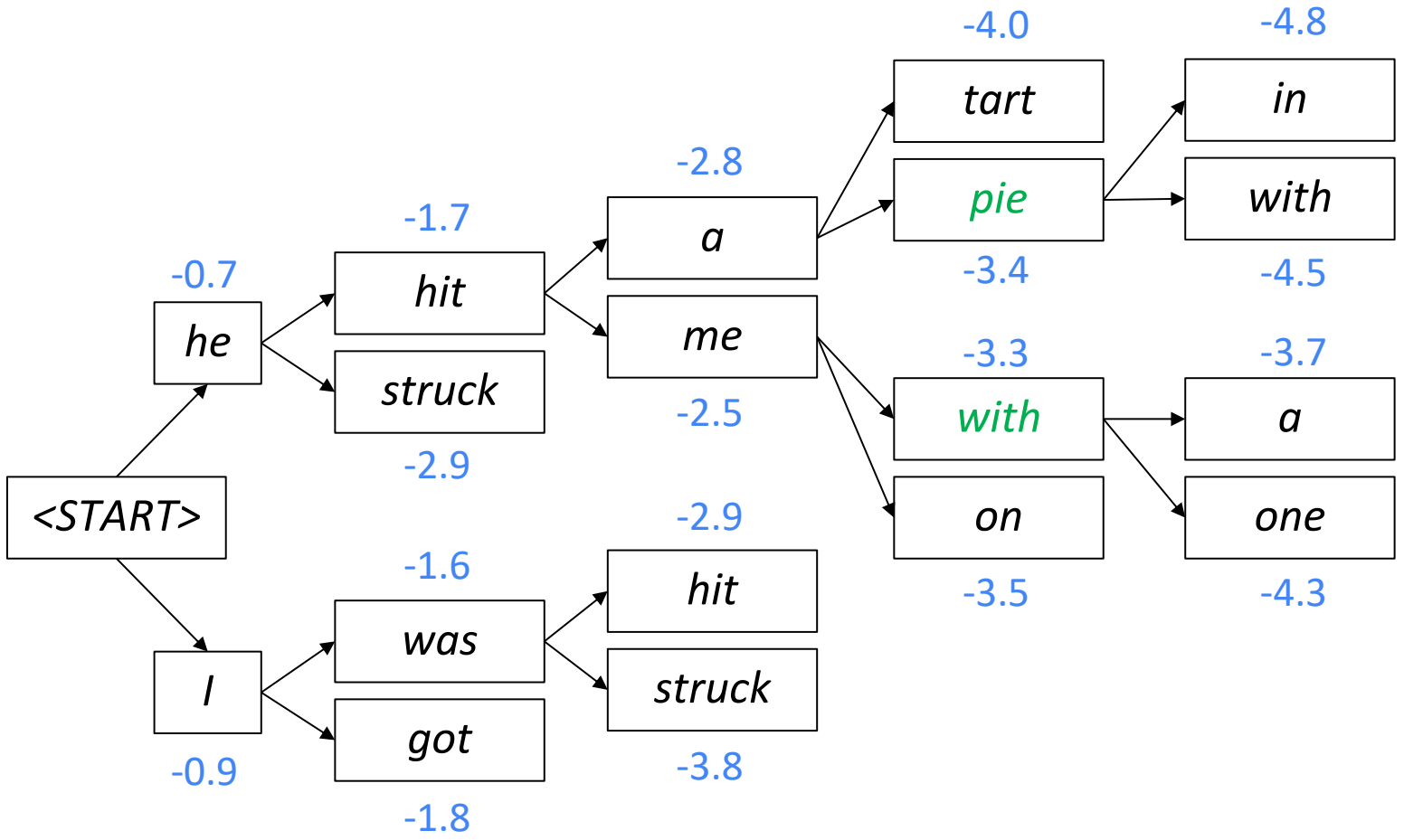
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses,
just keep k with highest scores

Beam search decoding: example

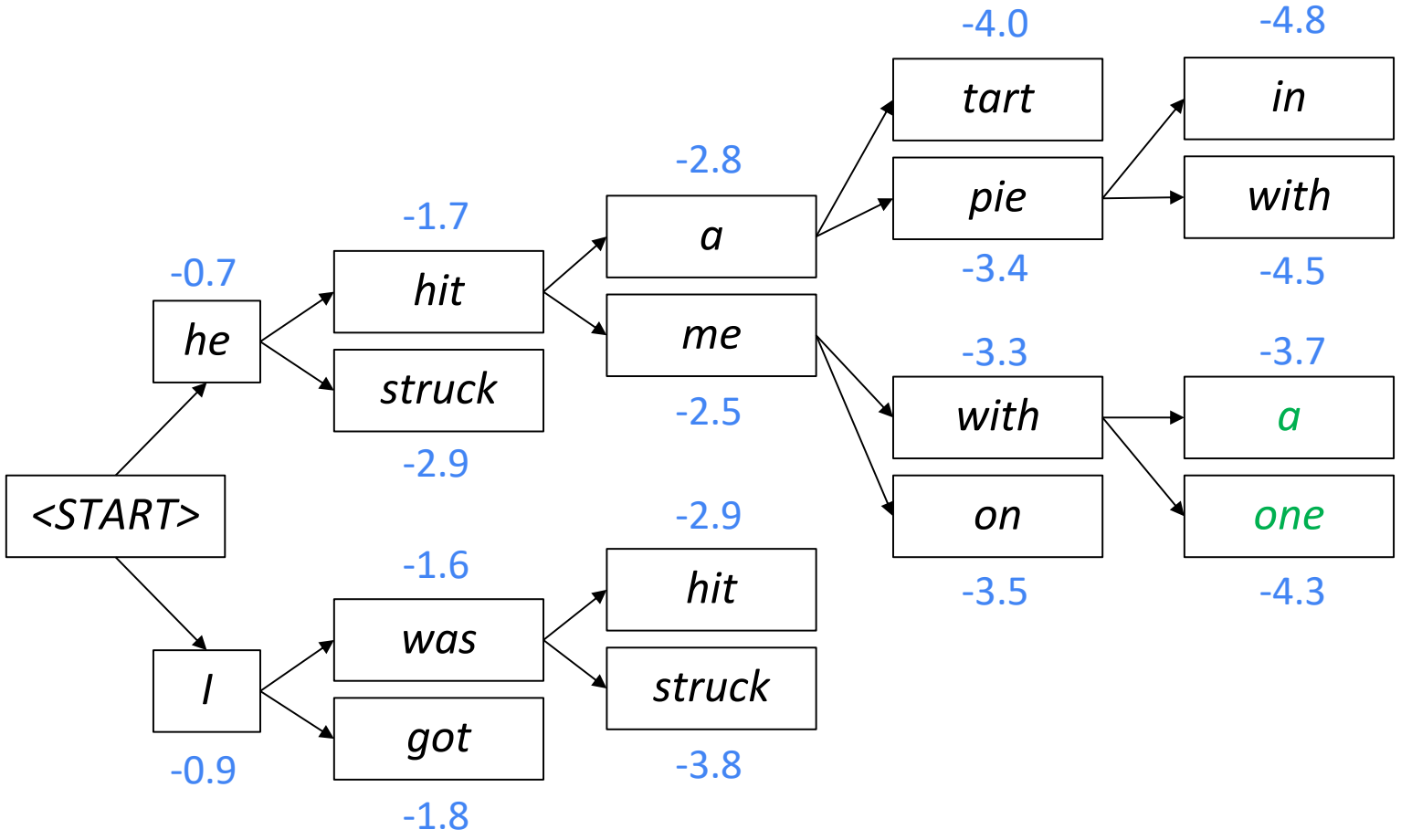
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

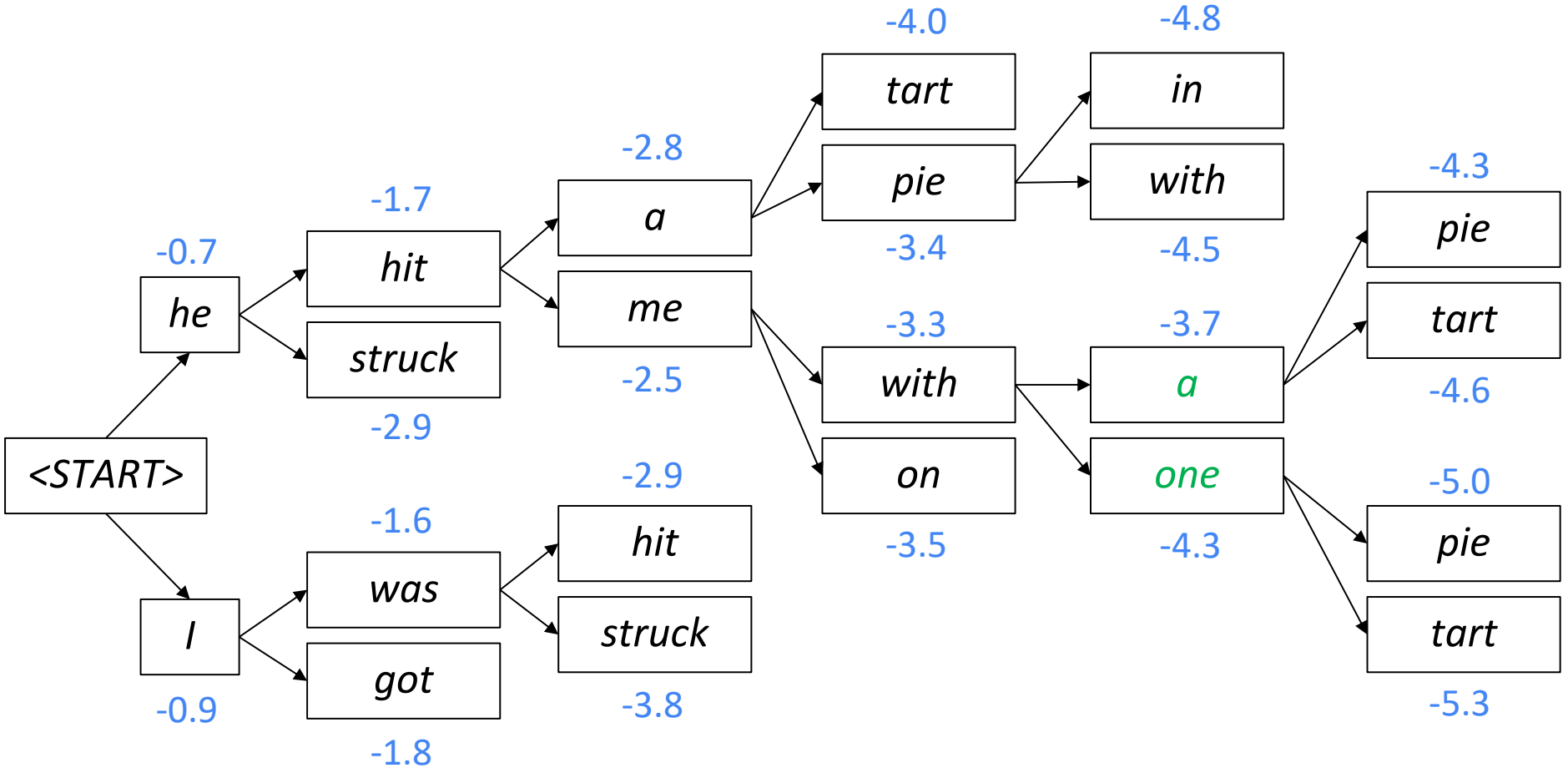
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

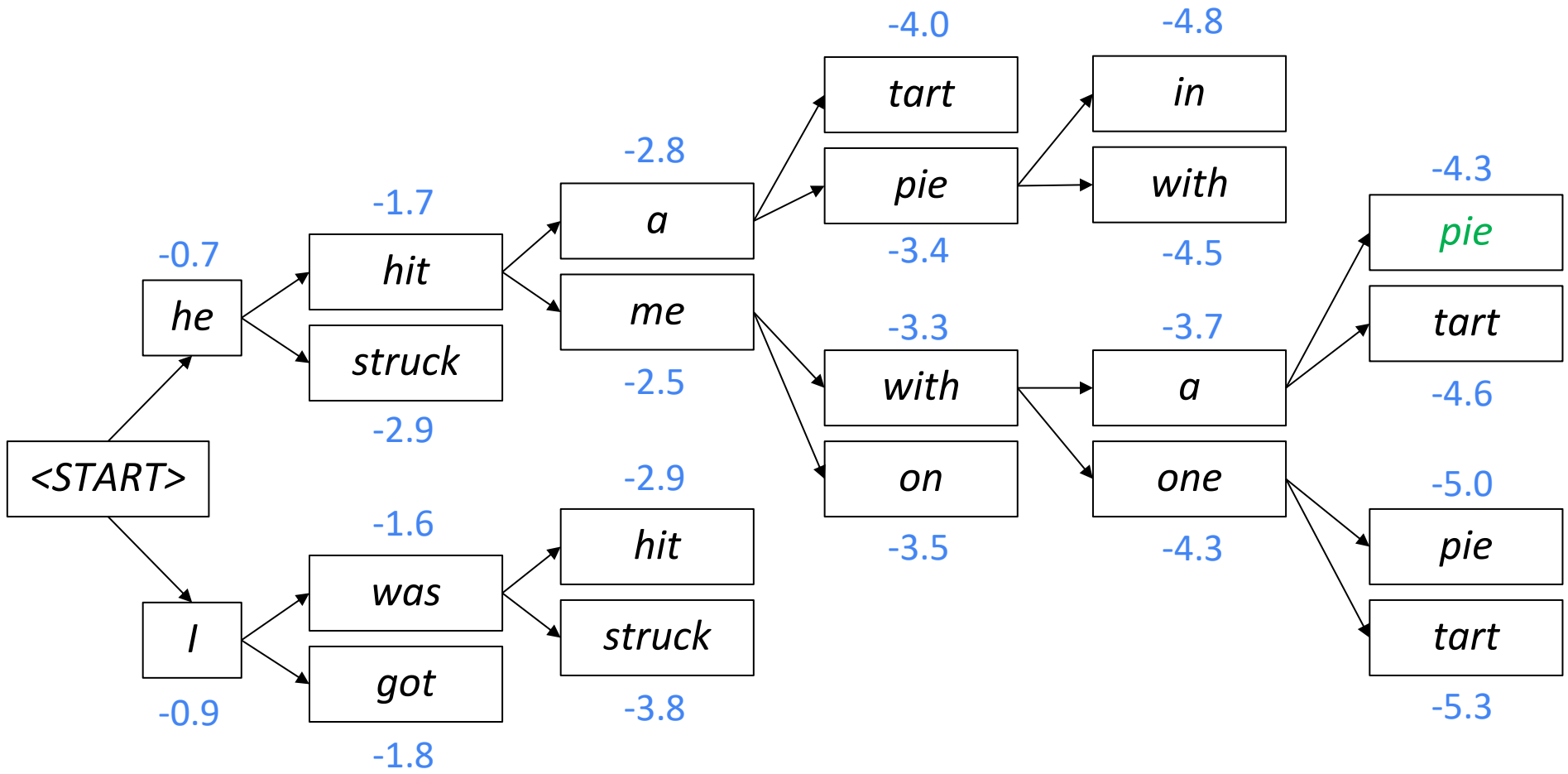
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

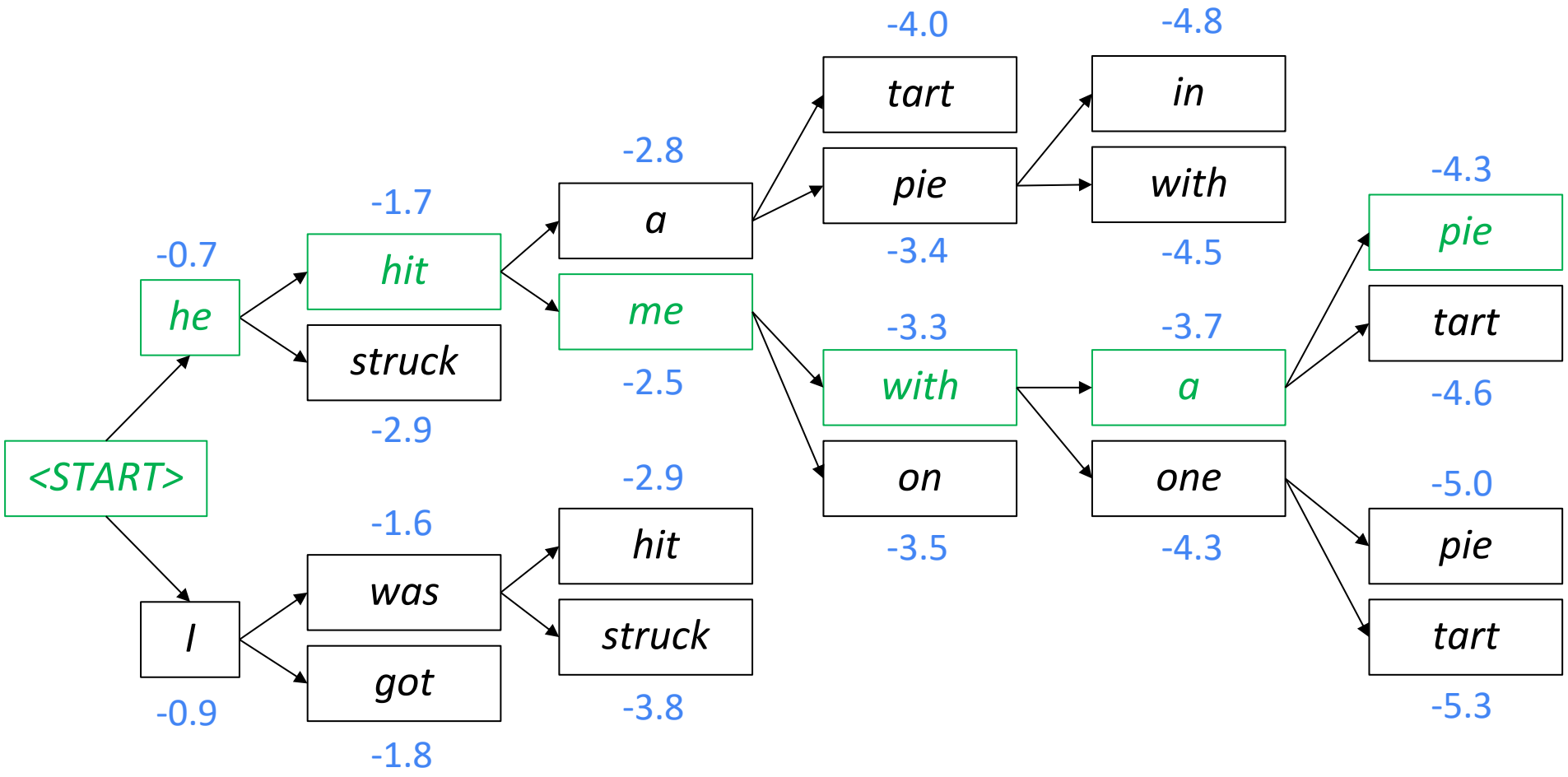
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



This is the top-scoring hypothesis!

Beam search decoding: example

Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Backtrack to obtain the full hypothesis

Advantages of NMT

- Compared to SMT, NMT has many **advantages**:
  - Better **performance**
    - More **fluent**
    - Better use of **context**
    - Better use of **phrase similarities**
  - A **single neural network** to be optimized end-to-end
    - No subcomponents to be individually optimized
  - Requires much **less human engineering effort**
    - No feature engineering
    - Same method for all language pairs

Disadvantages of NMT?

- Compared to SMT:
  - NMT is **less interpretable**
    - Hard to debug
  - NMT is **difficult to control**
    - For example, can't easily specify rules or guidelines for translation
    - Safety concerns!

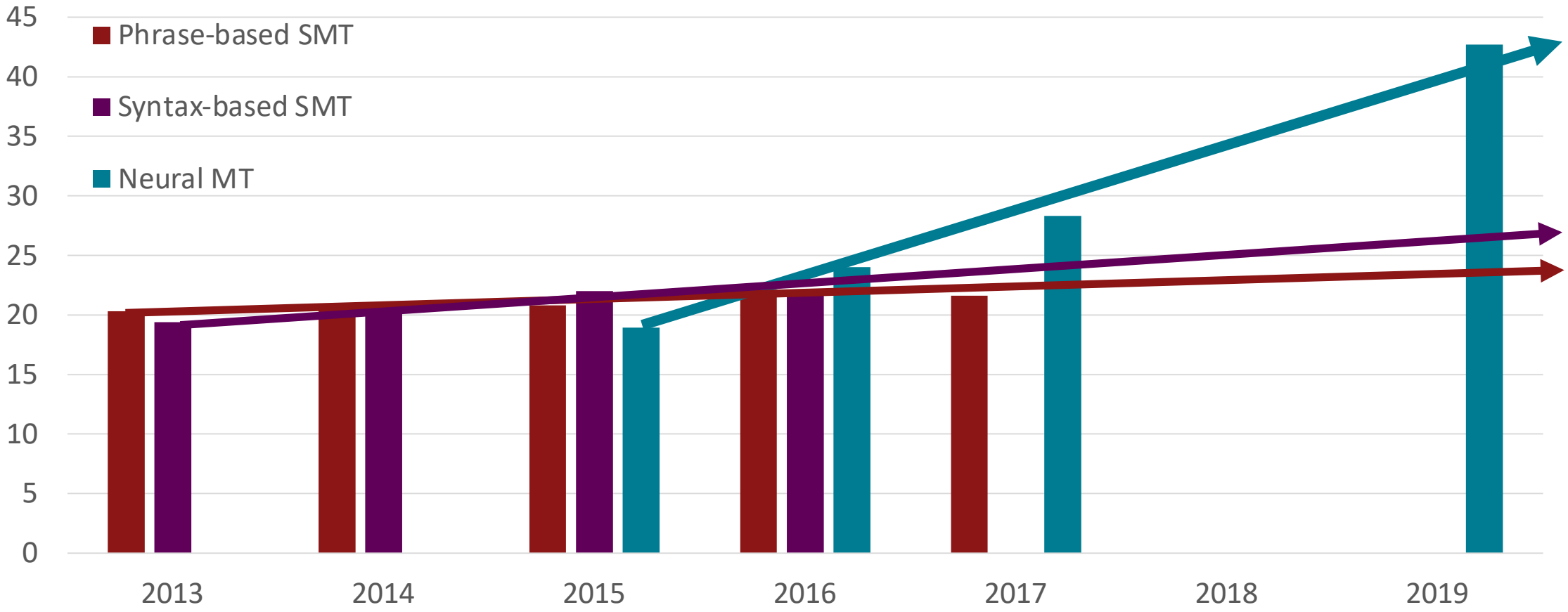
How do we evaluate Machine Translation?

- **BLEU** (**Bil**ingual **E**valuation **U**nderstudy)
  - BLEU compares the **machine-written translation** to one or several **human-written translation(s)**, and computes a **similarity score** based on:
    - **n-gram precision** (usually for 1, 2, 3 and 4-grams)
    - Plus a penalty for too-short system translations
  - BLEU is **useful** but **imperfect**
    - There are many valid ways to translate a sentence
    - So a **good** translation can get a **poor** BLEU score because it has low *n*-gram overlap with the human translation

Source: ”BLEU: a Method for Automatic Evaluation of Machine Translation", Papineni et al, 2002. http://aclweb.org/anthology/P02-1040

MT progress over time

[Edinburgh En-De WMT newstest2013 Cased BLEU; NMT 2015 from U. Montréal; NMT 2019 FAIR on newstest2019]



Sources: http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf & http://matrix.statmt.org/

NMT: perhaps the biggest success story of NLP Deep Learning?

- Neural Machine Translation went from a **fringe research attempt** in 2014 to the **leading standard method** in 2016
  - 2014: First seq2seq paper published
  - 2016: Google Translate switches from SMT to NMT – and by 2018 everyone has

  - This is amazing!
    - SMT systems, built by **hundreds** of engineers over many **years**, outperformed by NMT systems trained by a **small group** of engineers in a few **months**

So, is Machine Translation solved?

- Nope!
- Many difficulties remain:
  - Out-of-vocabulary words
  - Domain mismatch between train and test data
  - Maintaining context over longer text
  - Low-resource language pairs
  - Failures to accurately capture sentence meaning
  - Pronoun (or zero pronoun) resolution errors
  - Morphological agreement errors

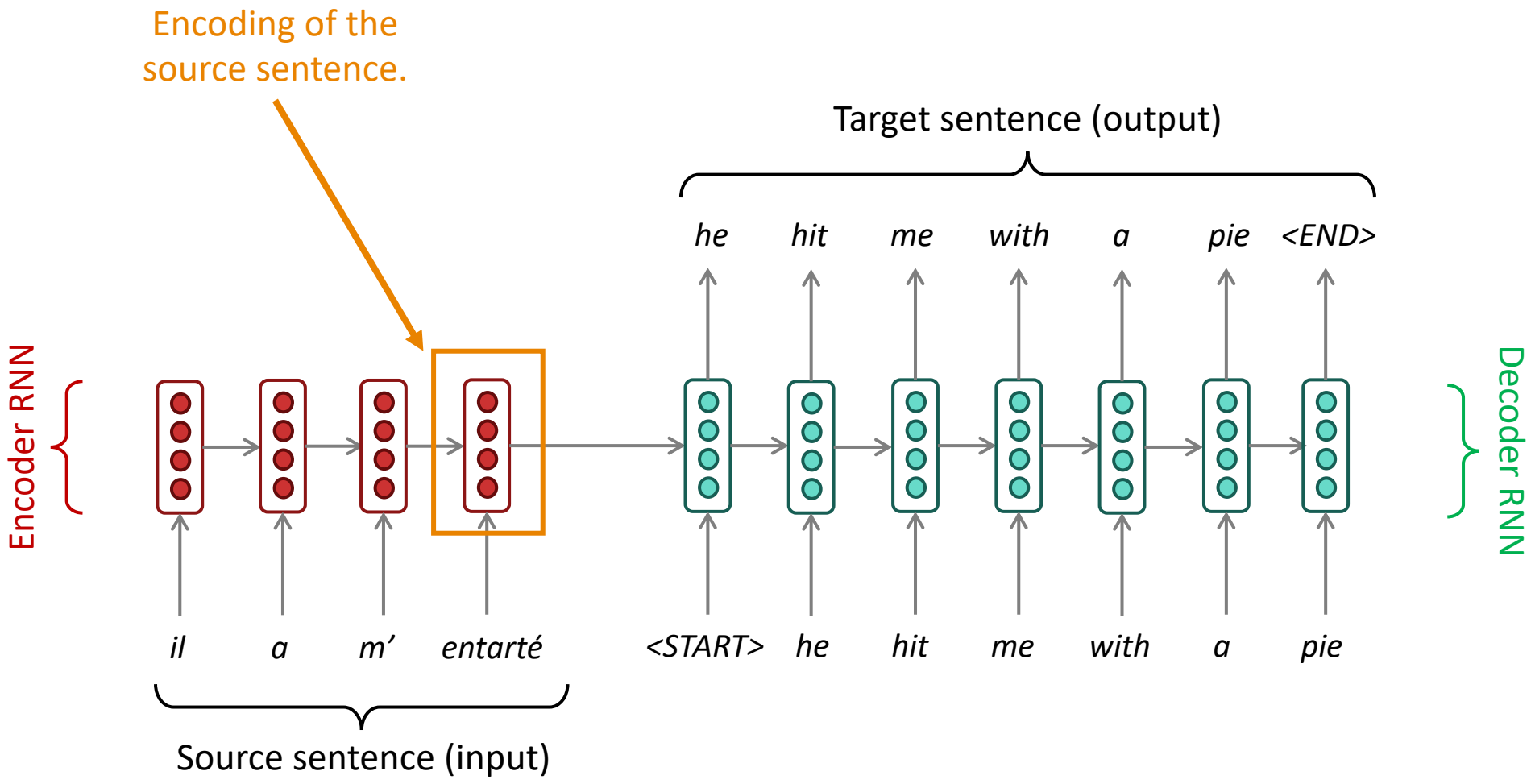
Further reading: *"Has AI surpassed humans at translation? Not even close!"*
https://www.skynettoday.com/editorials/state_of_nmt

NMT research continues

- NMT is a **flagship task** for NLP Deep Learning
  - NMT research has **pioneered** many of the recent **innovations** of NLP Deep Learning
  - NMT research continues to **thrive**
    - Researchers have found **many, many improvements** to the "vanilla" seq2seq NMT system we've just presented
    - But we'll present next **one improvement** so integral that it is the new vanilla...

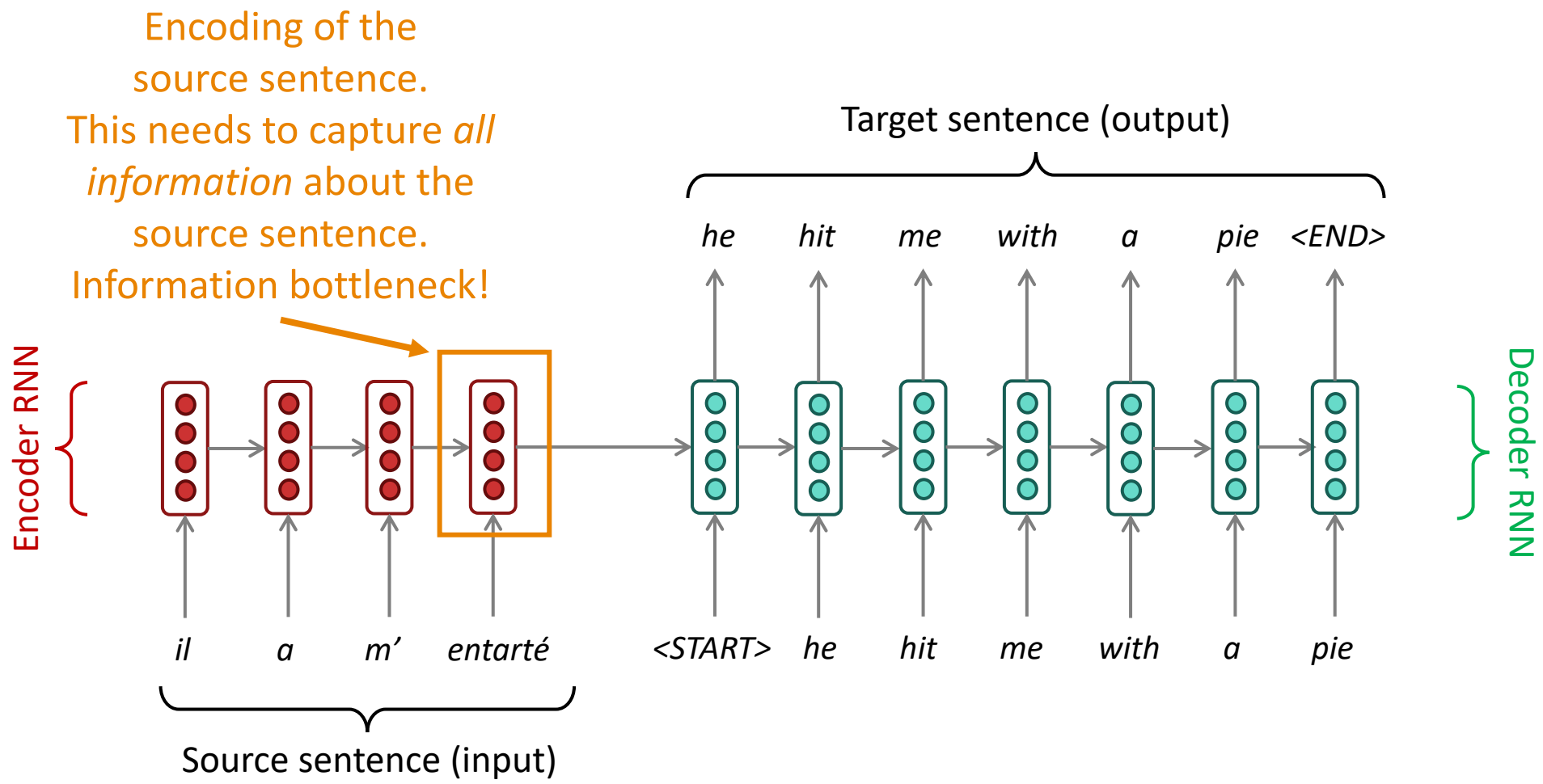
ATTENTION

Sequence-to-sequence: the bottleneck problem



Problems with this architecture?

Sequence-to-sequence: the bottleneck problem



Attention

- **Attention** provides a solution to the bottleneck problem.
- **Core idea**: on each step of the decoder, **use direct connection to the encoder to focus on a particular part** of the source sequence

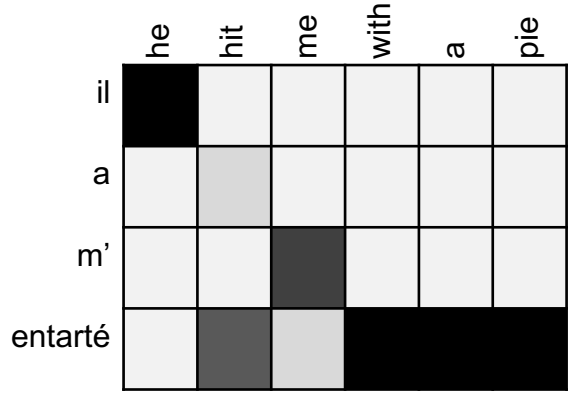
- First, we will show via diagram (no equations), then we will show with equations

Attention is great!

- Attention significantly **improves NMT performance**
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention provides **more "human-like" model** of the MT process
  - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention **solves the bottleneck problem**
  - Attention allows decoder to look directly at source; bypass bottleneck

Attention is great!

- Attention **helps with the vanishing gradient problem**
  - Provides shortcut to far away states
- Attention provides **some interpretability**
  - By inspecting attention distribution, we see what the decoder was focusing on
  - We get (soft) **alignment for free**!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself



There are several attention variants

- We have some ***values*** $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and a *query* $\boldsymbol{s} \in \mathbb{R}^{d_2}$
- Attention always involves:
  - 1. Computing the ***attention scores*** $\boldsymbol{e} \in \mathbb{R}^N$
  - 2. Taking softmax to get ***attention distribution*** α:

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

  - Using attention distribution to take weighted sum of values:

$$\boldsymbol{a} = \sum_{i=1}^{N} \alpha_i \boldsymbol{h}_i \in \mathbb{R}^{d_1}$$

thus obtaining the ***attention output*** *a* (sometimes called the *context vector*)

Attention variants

- There are several ways you can compute $\boldsymbol{e} \in \mathbb{R}^N$ from $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and $\boldsymbol{s} \in \mathbb{R}^{d_2}$
- **Basic dot-product attention** $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
  - Note: this assumes $d_1 = d_2$. This is the version we saw earlier.
- **Multiplicative attention**: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
  - Where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called "bilinear attention"

More information: "Deep Learning for NLP Best Practices", Ruder, 2017. http://ruder.io/deep-learning-nlp-best-practices/index.html#attention
"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017, https://arxiv.org/pdf/1703.03906.pdf

Attention variants

- **Reduced-rank multiplicative attention**: $e_i = s^T(\boldsymbol{U}^T\boldsymbol{V})h_i = (\boldsymbol{U}s)^T(\boldsymbol{V}h_i)$
  - For low rank matrices $\boldsymbol{U} \in \mathbb{R}^{k \times d_2}, \boldsymbol{V} \in \mathbb{R}^{k \times d_1}, k \ll d_1, d_2$
- **Additive attention**: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1\boldsymbol{h}_i + \boldsymbol{W}_2\boldsymbol{s}) \in \mathbb{R}$ [Bahdanau, Cho, and Bengio 2014]
  - Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector.
  - d_3 (the attention dimensionality) is a hyperparameter
  - "Additive" is a weird/bad name. It's really using a feed-forward neural net layer.

More information: "Deep Learning for NLP Best Practices", Ruder, 2017. http://ruder.io/deep-learning-nlp-best-practices/index.html#attention
"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017, https://arxiv.org/pdf/1703.03906.pdf

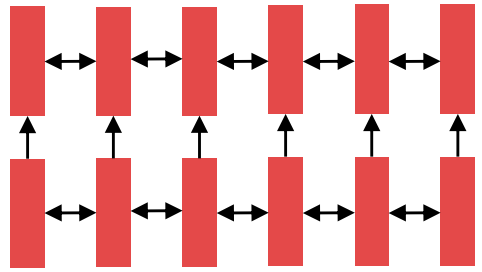
Attention is a *general* Deep Learning technique

- **More general definition of attention**:
  - Given a set of vector **values**, and a vector **query**, **attention** is a technique to compute a weighted sum of the values, dependent on the query.
- **Intuition**:
  - The weighted sum is a ***selective summary*** of the information contained in the values, where the query determines which values to focus on.
  - Attention is a way to obtain a **fixed-size representation of an arbitrary set of representations** (the values), dependent on some other representation (the query).

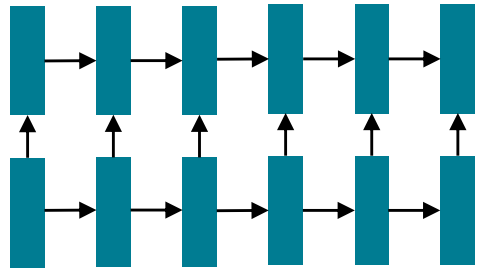
Transformers

As of last week: recurrent models for (most) NLP!

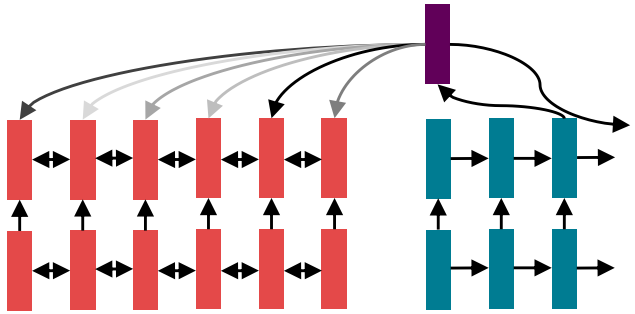
- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM:
  (for example, the source sentence in a translation)



- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.

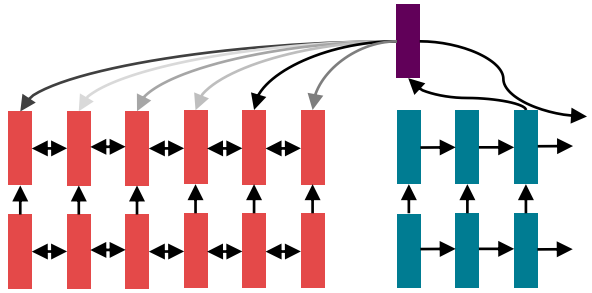


- Use attention to allow flexible access to memory



Today: Same goals, different building blocks

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

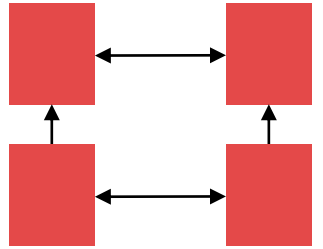


2014-2017ish
Recurrence

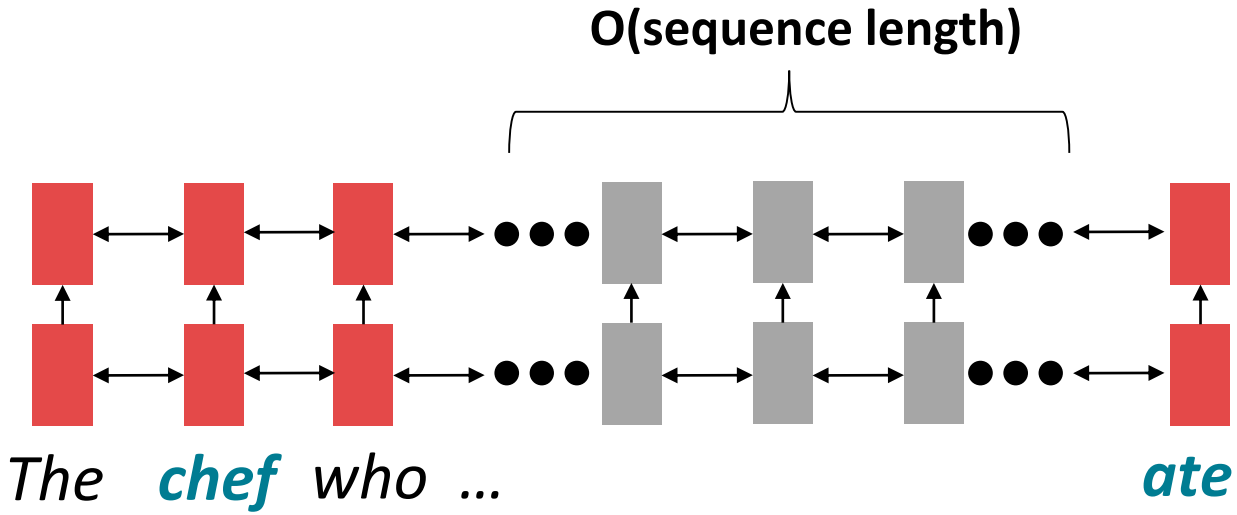
Lots of trial
and error

2021
??????

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

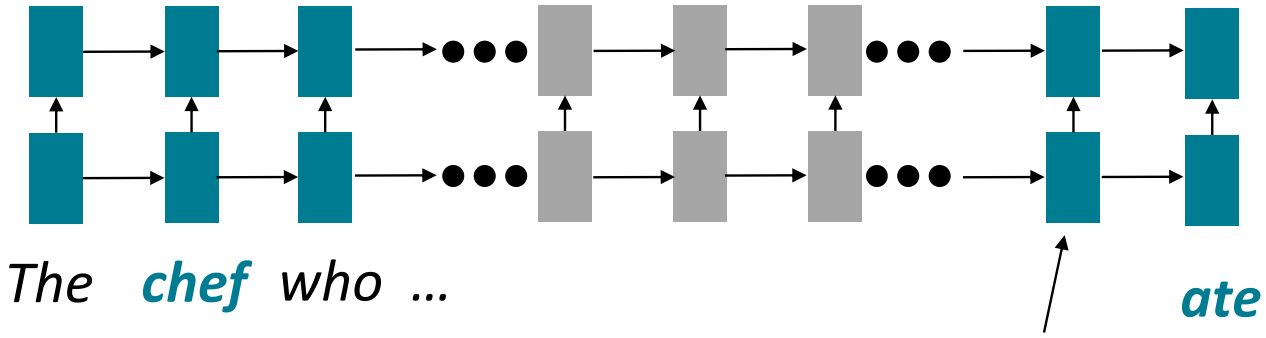


- RNNs are unrolled "left-to-right".
- It encodes linear locality: a useful heuristic!
  - Nearby words often affect each other's meanings
- **Problem:** RNNs take **O(sequence length)** steps for distant word pairs to interact.



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

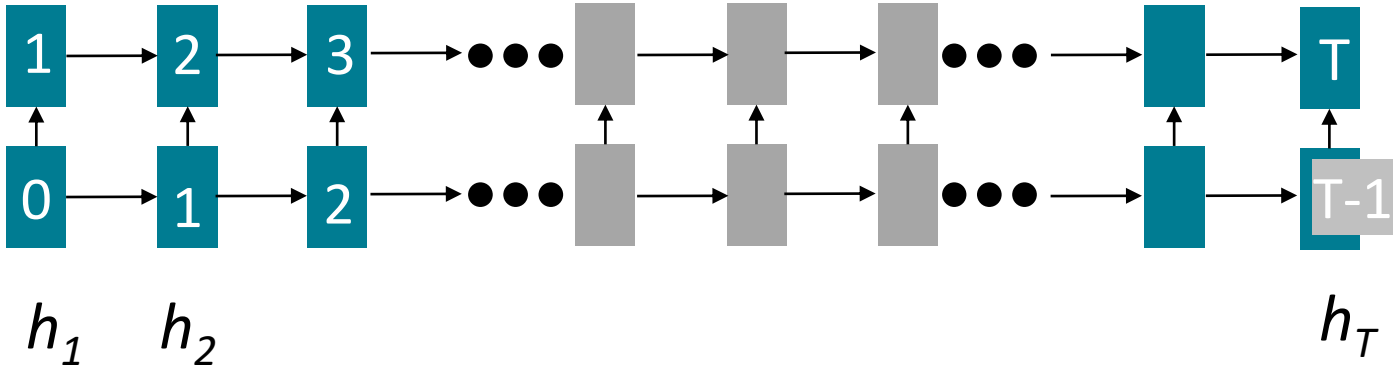
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is “baked in”; we already know sequential structure doesn't tell the whole story...



Info of ***chef*** has gone through O(sequence length) many layers!

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

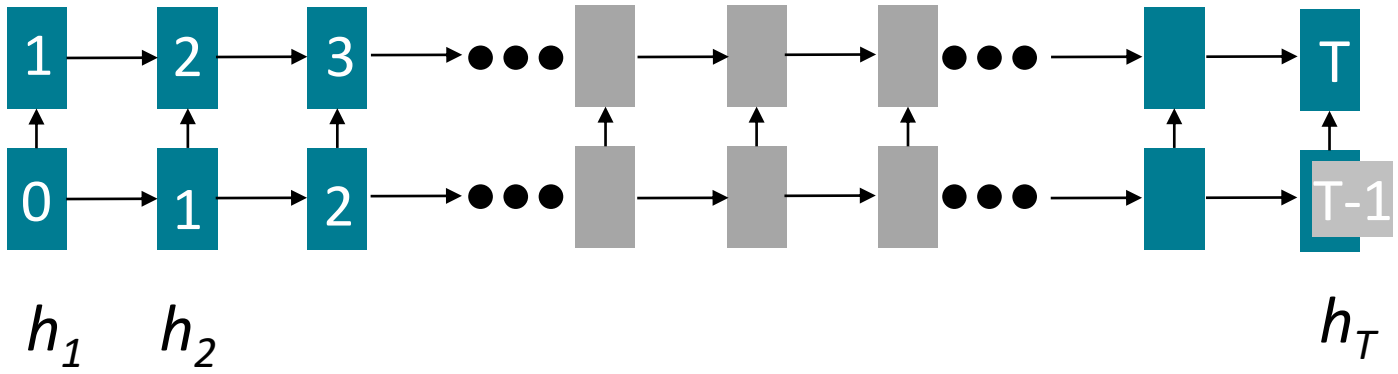
- Forward and backward passes have **O(seq length)** unparallelizable operations
  - GPUs (and TPUs) can perform many independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed



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

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

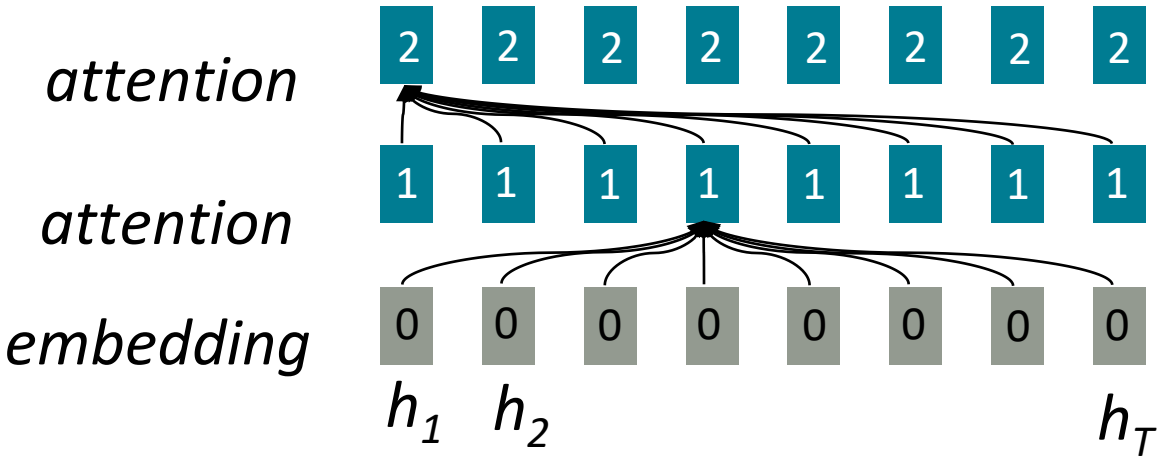
- Forward and backward passes have **O(seq length)** unparallelizable operations
  - Inhibits training on very large datasets!
  - Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations



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

If not recurrence, then what? **How about (self) attention?**

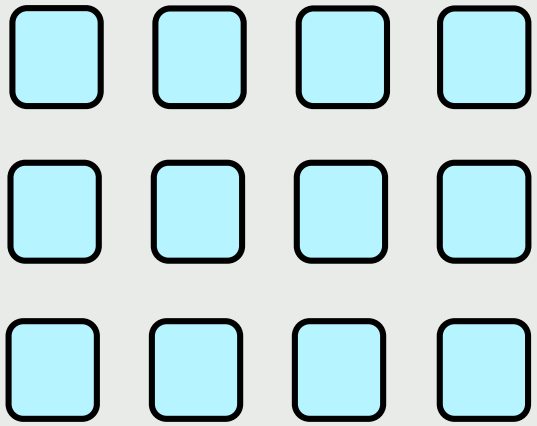
- To recap, **attention** treats each word's representation as a **query** to access and incorporate information from **a set of values**.
  - Previously, we saw attention from the **decoder** to the **encoder**;
  - **Self-attention** is **encoder-encoder** (or **decoder-decoder**) attention where each word attends to each other word **within the input (or output)**.

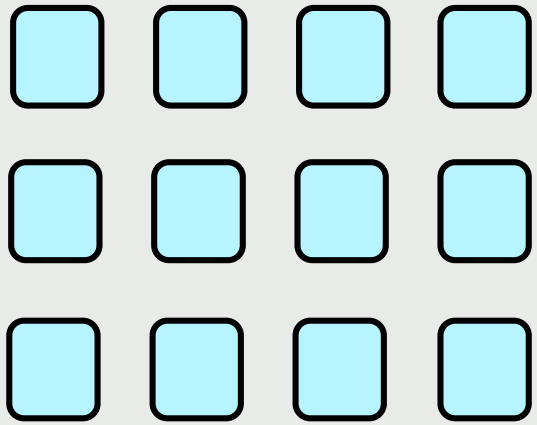


All words attend to all words in previous layer; most arrows here are omitted

Computational Dependencies for Recurrence vs. Attention

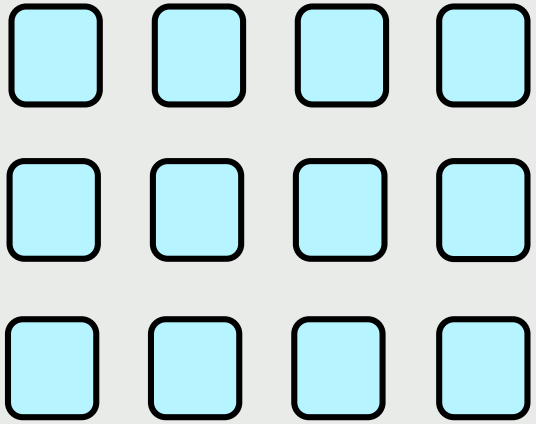
RNN-Based Encoder-Decoder Model with Attention

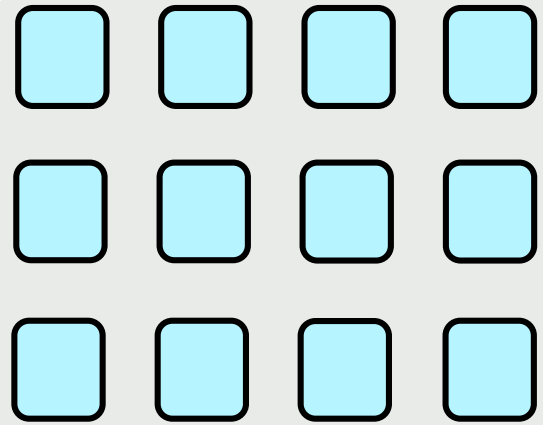




Transformer Advantages:

- Number of unparallelizable operations does not increase with sequence length.
- Each "word" interacts with each other, so maximum interaction distance: O(1).

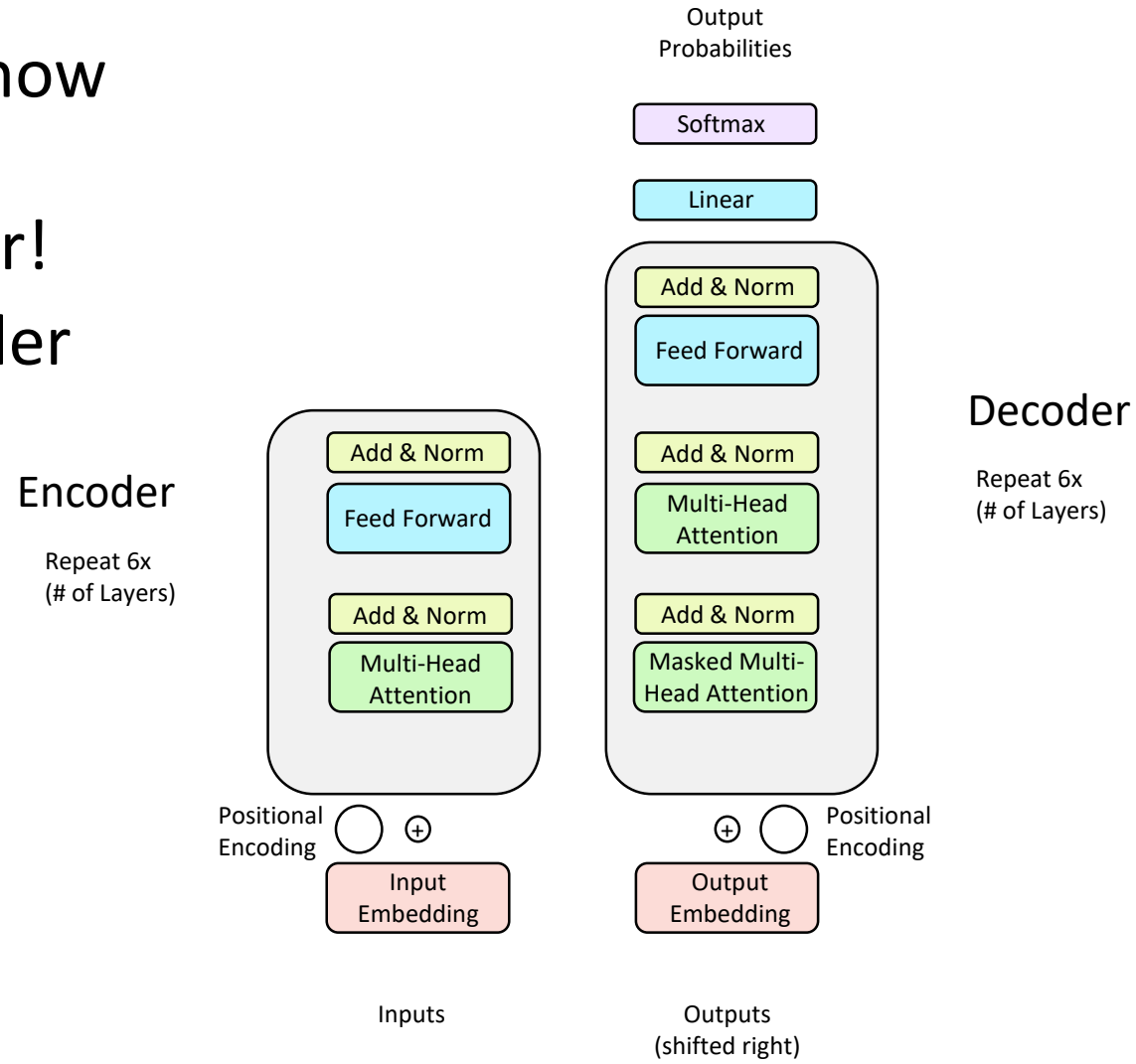




Transformer-Based Encoder-Decoder Model

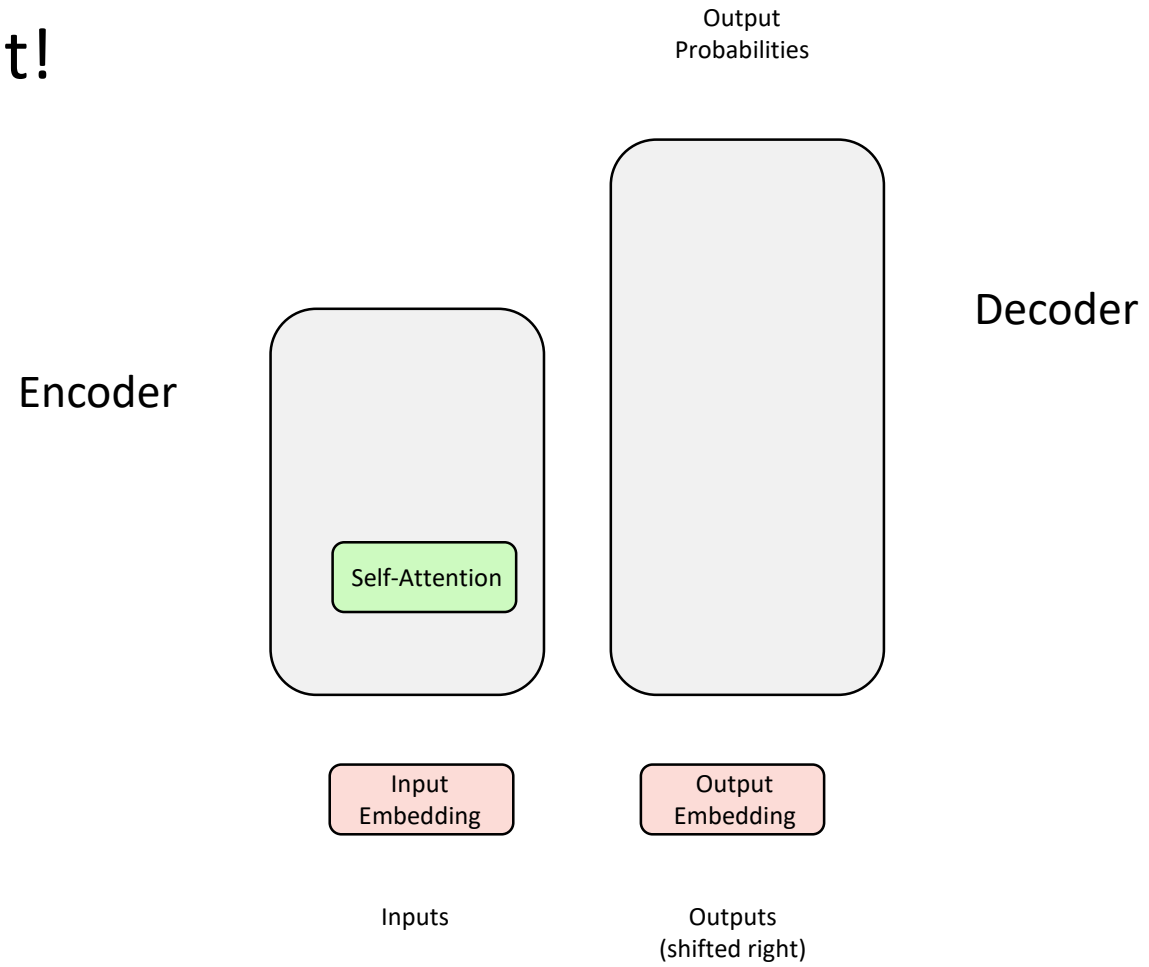
The Transformer Encoder-Decoder [Vaswani et al., 2017]

- In this section, you will learn exactly how the Transformer architecture works:
  - First, we will talk about the Encoder!
  - Next, we will go through the Decoder (which is quite similar)!



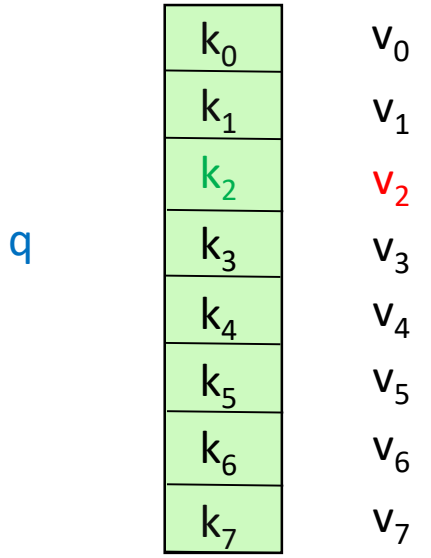
Encoder: Self-Attention

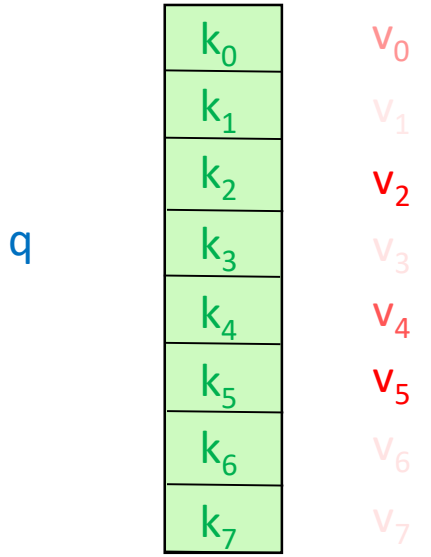
- Self-Attention is the core building block of Transformer, so let's first focus on that!



Intuition for Attention Mechanism

- Let's think of attention as a "fuzzy" or approximate hashtable:
  - To look up a value, we compare a query against keys in a table.
  - In a hashtable (shown on the bottom left):
    - Each **query** (hash) maps to exactly one **key-value** pair.
  - In (self-)attention (shown on the bottom right):
    - Each **query** matches each **key** to varying degrees.
    - We return a sum of **values** weighted by the **query**-**key** match.





Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word x_i, calculate its **query**, **key**, and **value**.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query and keys.**

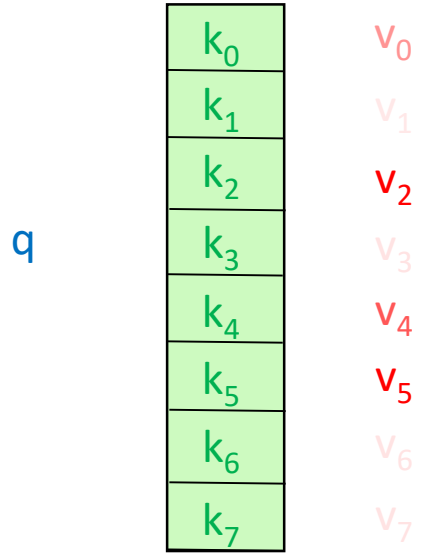
$$e_{ij} = q_i \cdot k_j$$

- Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_k exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**.

$$Output_i = \sum_j \alpha_{ij} v_j$$



Recipe for (Vectorized) Self-Attention in the Transformer Encoder

- Step 1: With embeddings stacked in X, calculate queries, keys, and values.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2: Calculate attention score between **query and keys.**

$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

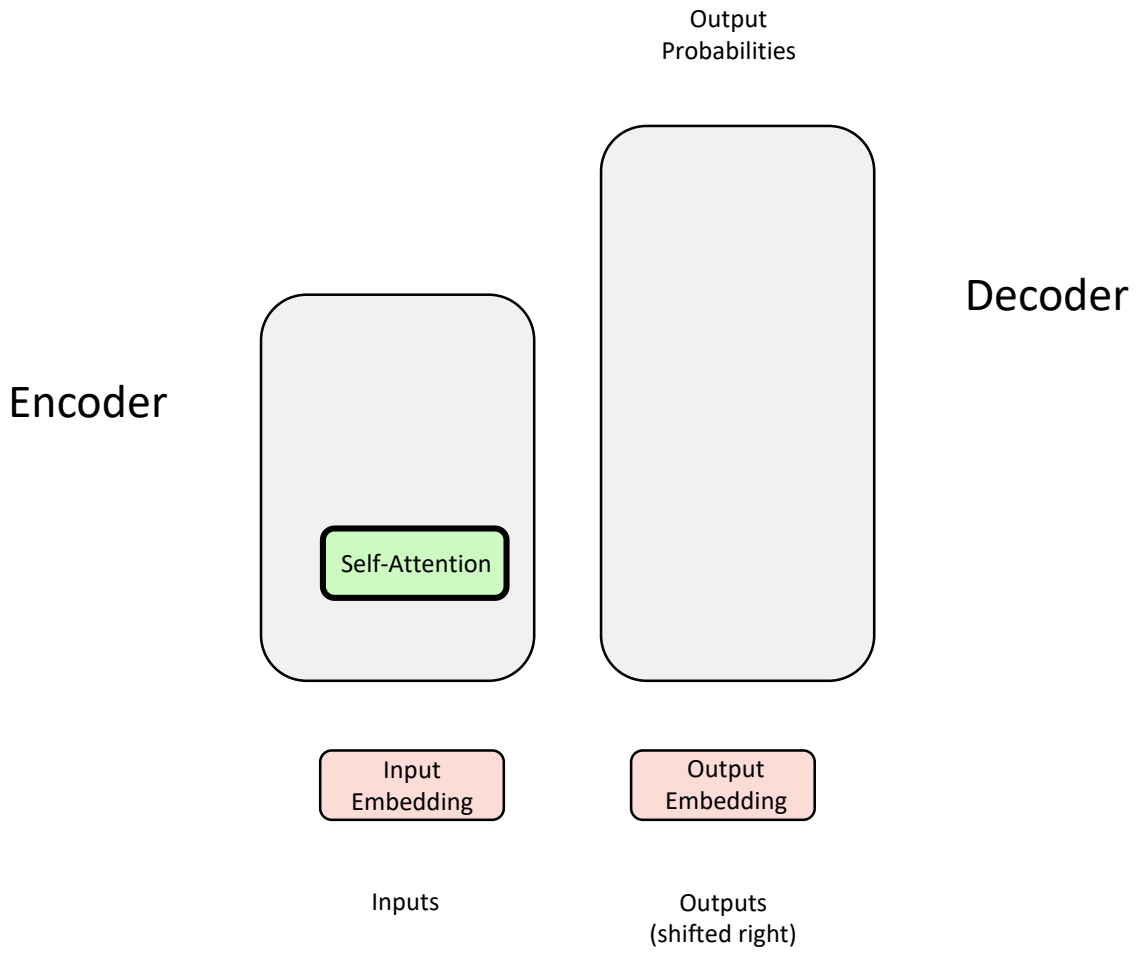
$$A = softmax(E)$$

- Step 4: Take a weighted sum of **values**.

$$Output = AV$$

$$Output = softmax\left(QK^T\right)V$$

What We Have So Far: (Encoder) Self-Attention!

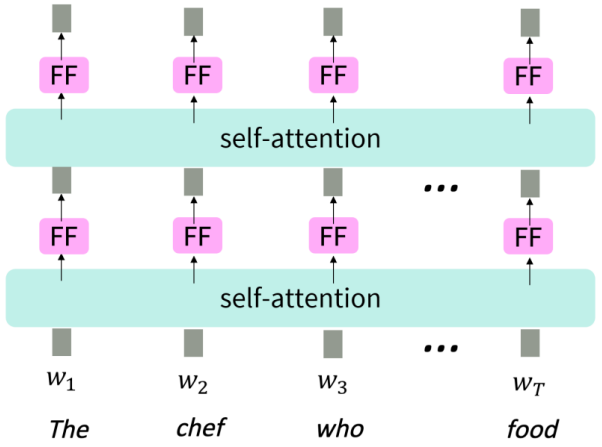


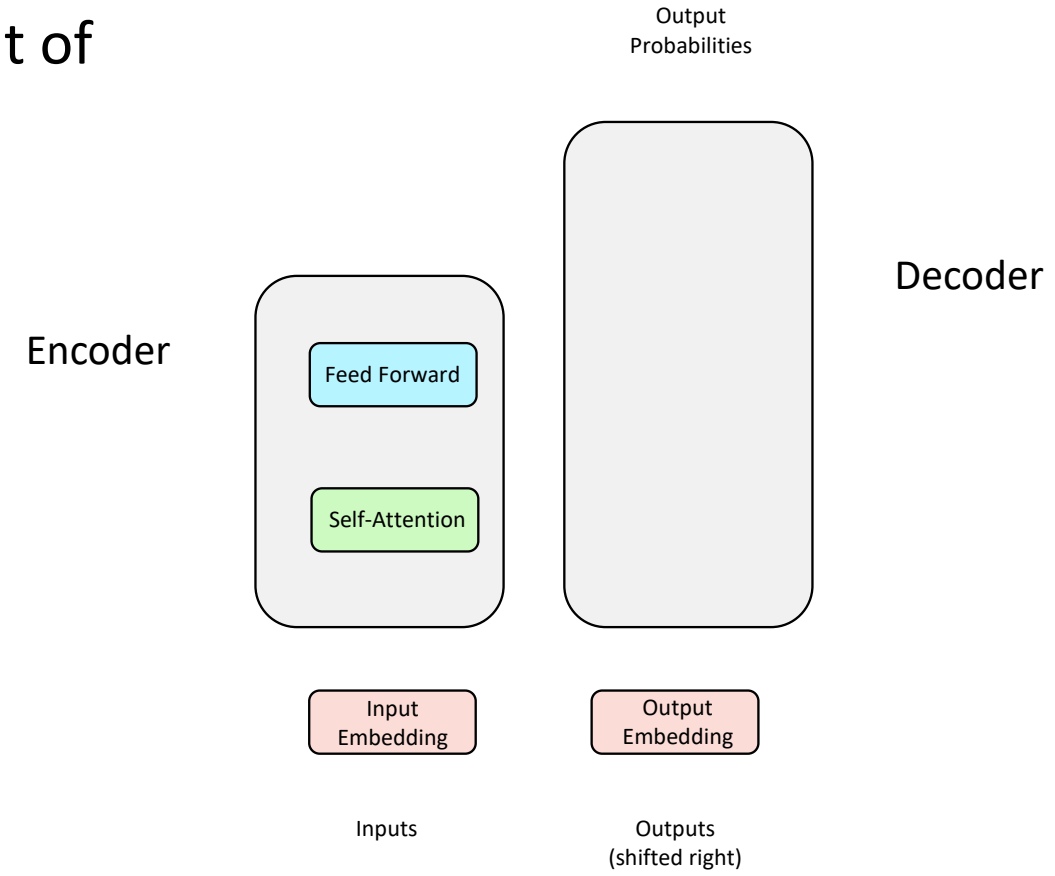
But attention isn't quite all you need!

- **Problem**: Since there are no element-wise non-linearities, self- attention is simply performing a re-averaging of the value vectors.
- **Easy fix:** Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

Equation for Feed Forward Layer

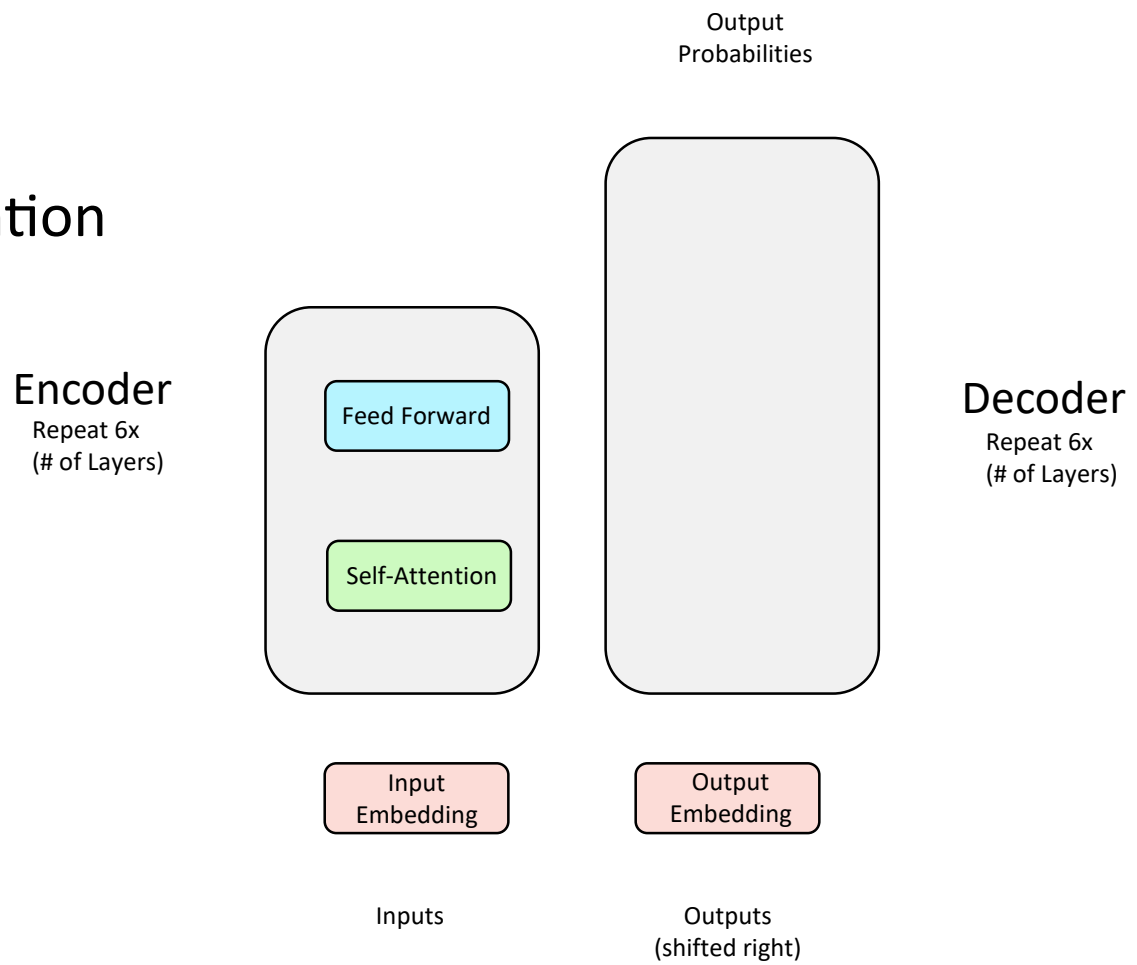
$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$





But how do we make this work for deep networks?

- ❑ Training Trick #1: Residual Connections
- ❑ Training Trick #2: LayerNorm
- ❑ Training Trick #3: Scaled Dot Product Attention

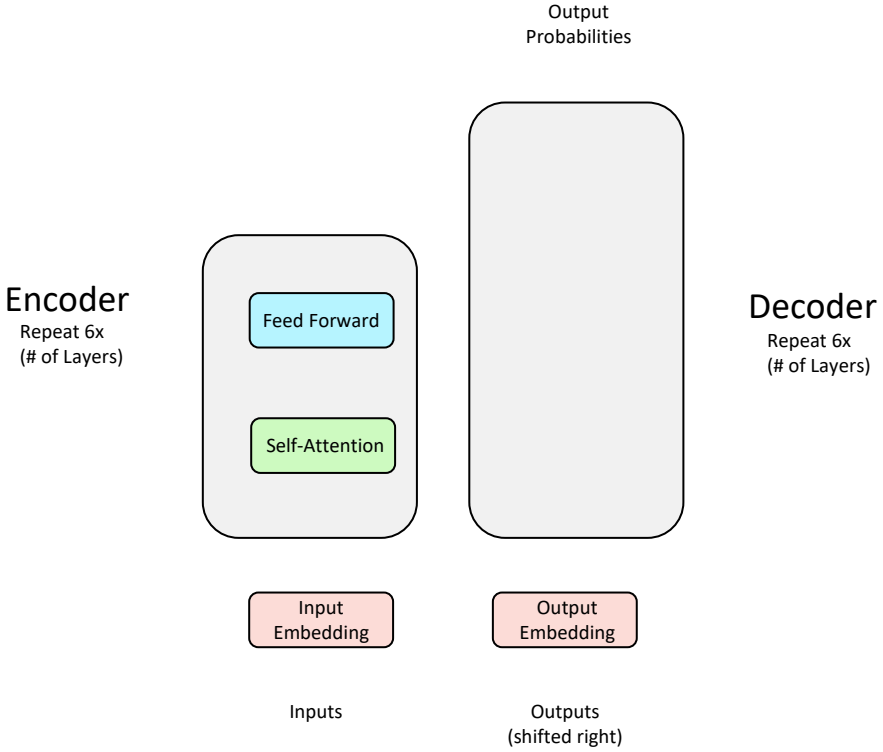


Training Trick #1: Residual Connections [He et al., 2016]

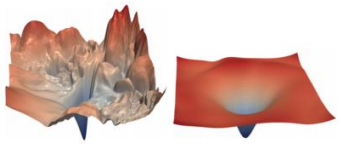
- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!

$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.



Residual connections are also thought to smooth the loss landscape and make training easier!



[no residuals] [residuals]

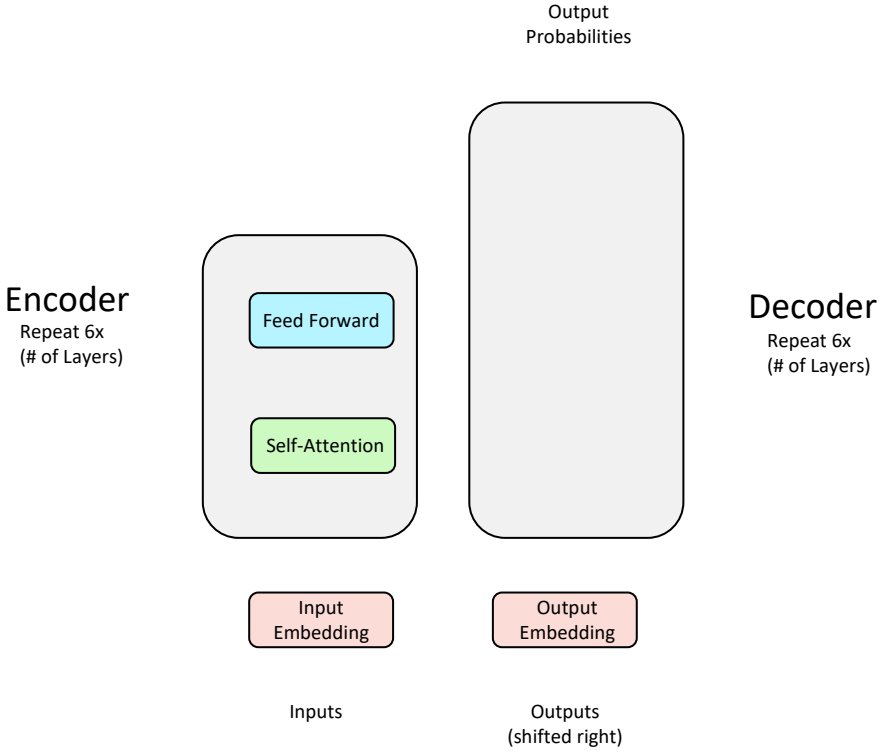
[Loss landscape visualization, Li et al., 2018, on a ResNet]

Training Trick #2: Layer Normalization [Ba et al., 2016]

- **Problem**: Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- **Solution**: Reduce uninformative variation by **normalizing** to zero mean and standard deviation of one within each **layer**.

Mean: $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ Standard Deviation: $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$

$$x^{\ell\prime} = \frac{x^\ell - \mu^\ell}{\sigma^\ell + \epsilon}$$

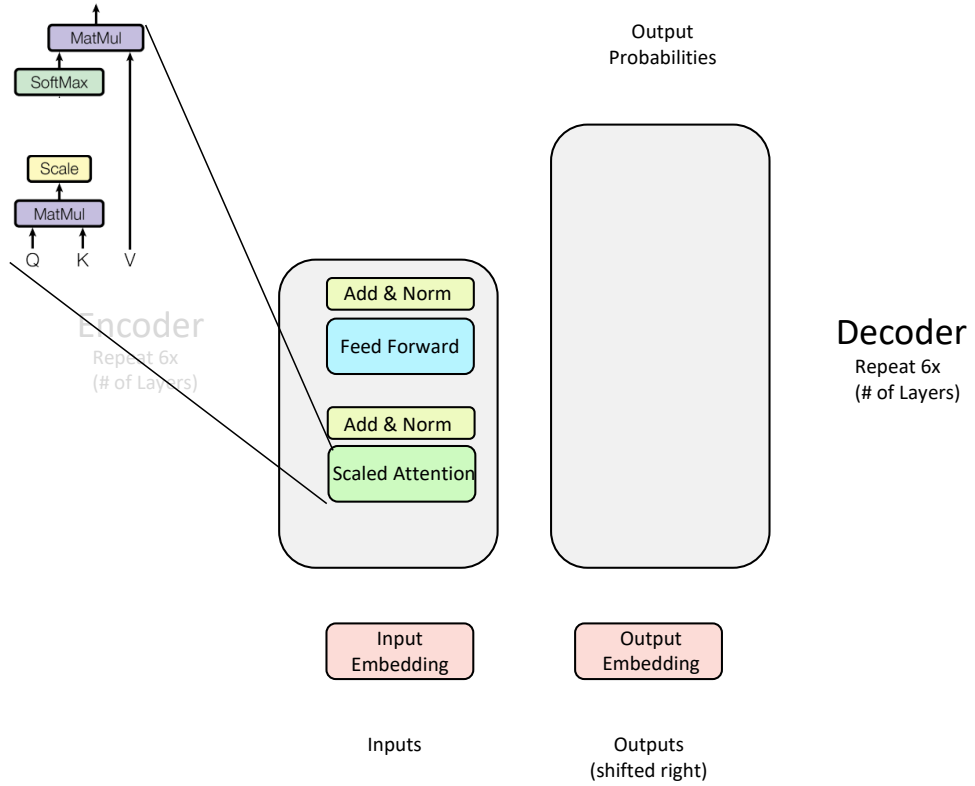


Training Trick #3: Scaled Dot Product Attention

- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!



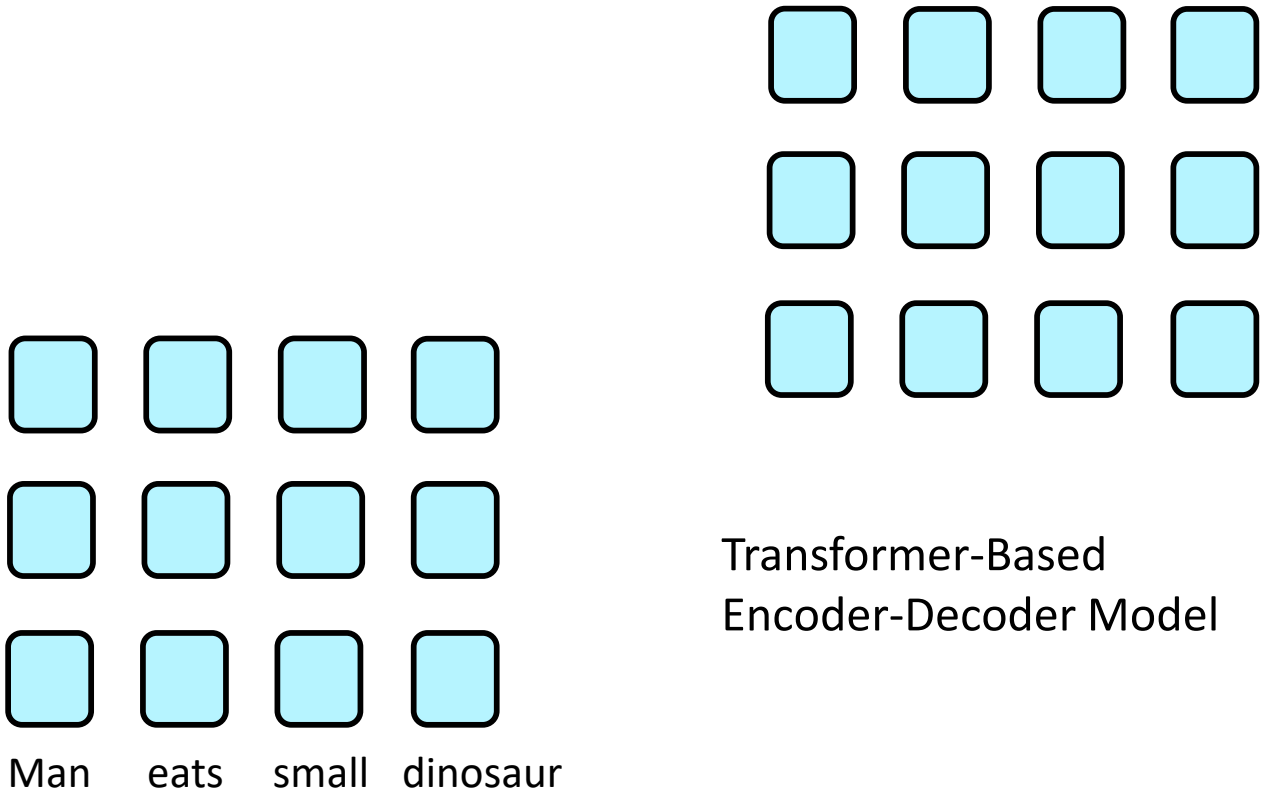
Updated Self-Attention Equation:

$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$

Major issue!

- We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?
- Consider this sentence:
  - "Man eats small dinosaur."

$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$

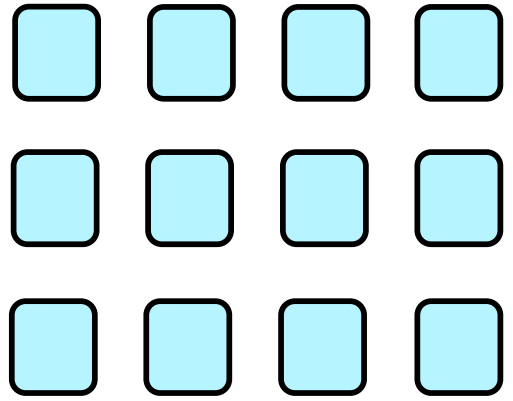


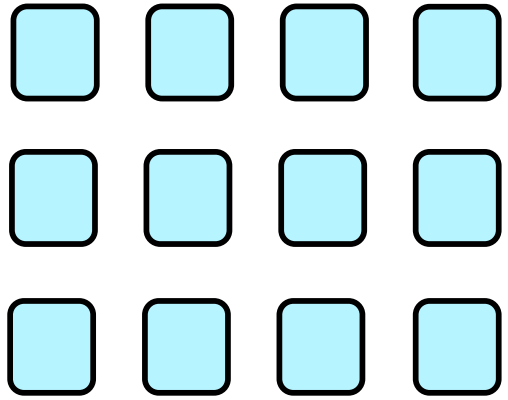
Transformer-Based
Encoder-Decoder Model

Major issue!

- We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?
- Consider this sentence:
  - "Man eats small dinosaur."
- Wait a minute, order doesn't impact the network at all!
- This seems wrong given that word order does have meaning in many languages, including English!

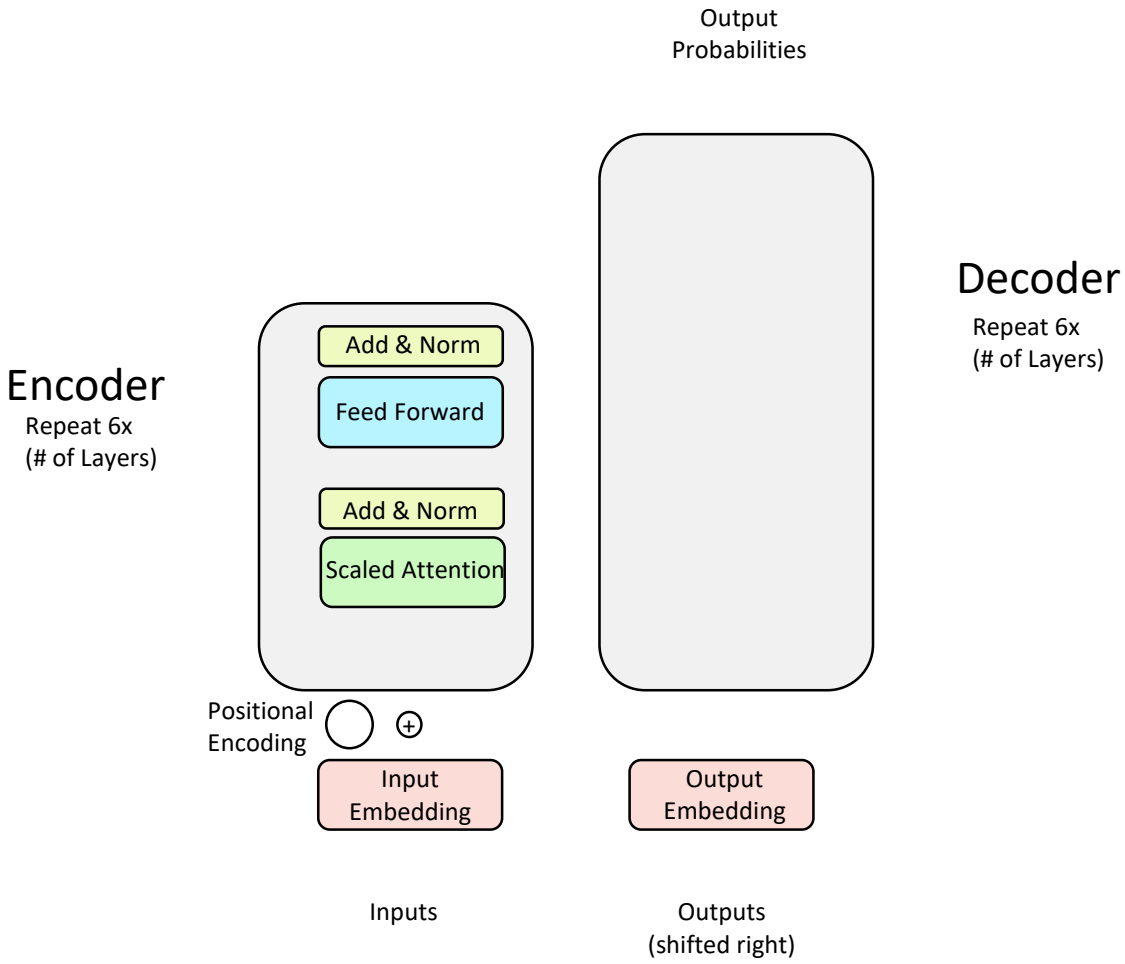
$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$





Transformer-Based
Encoder-Decoder Model

Solution: Inject Order Information through Positional Encodings!



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 $i \in \{1,2, \ldots, T\}$ are position vectors

- Don't worry about what the pi are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let $\tilde{v}_i, \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

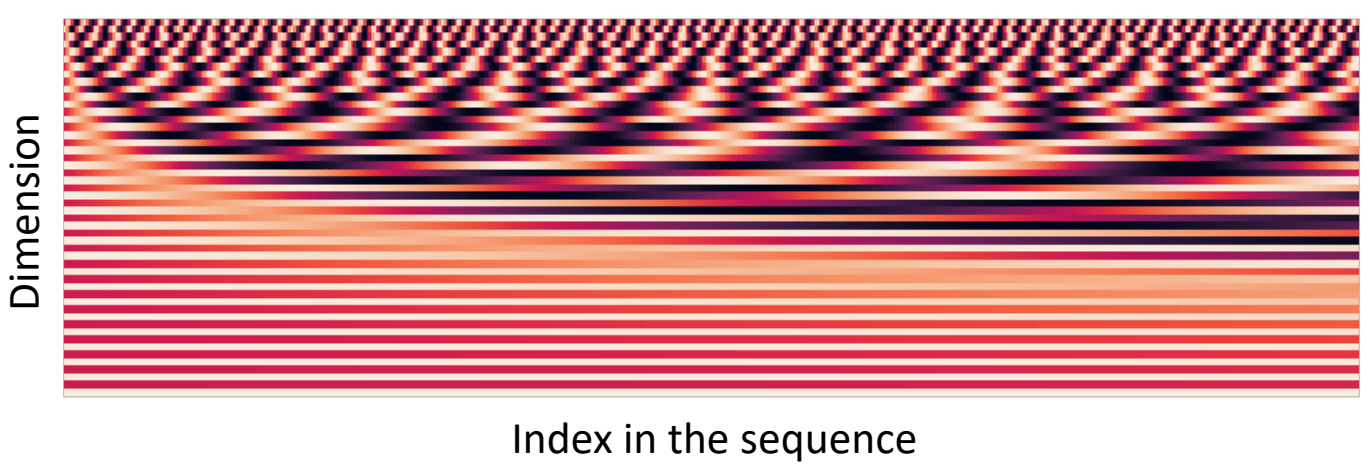
$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

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

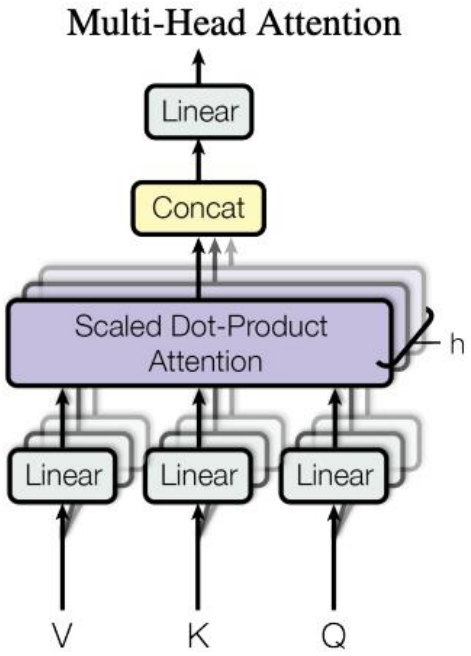
Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Position representation vectors learned from scratch

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

Multi-Headed Self-Attention: k heads are better than 1!

- **High-Level Idea:** Let's perform self-attention multiple times in parallel and combine the results.



[Vaswani et al. 2017]

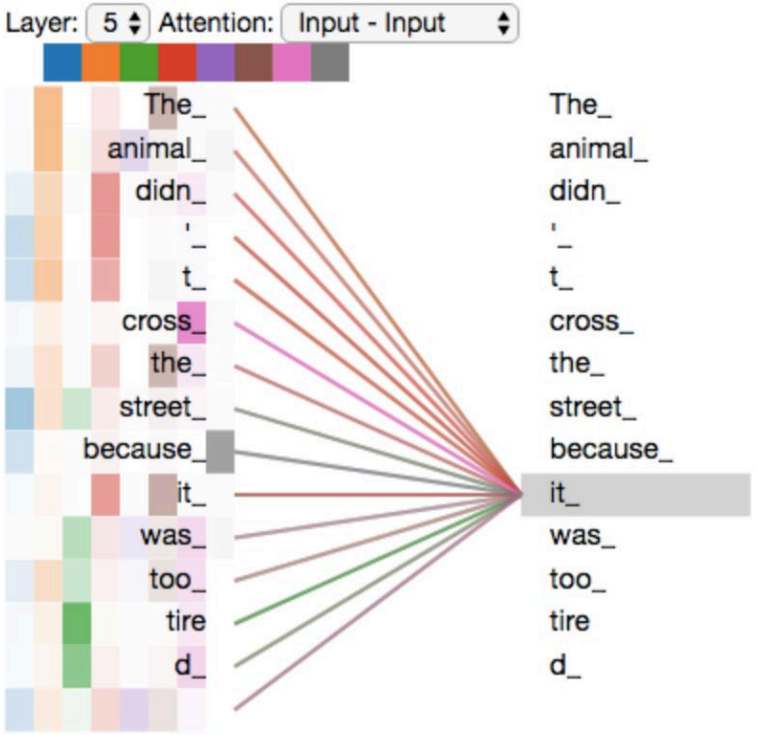
Wizards of the Coast, Artist: Todd Lockwood

The Transformer Encoder: Multi-headed Self-Attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different j for different reasons?

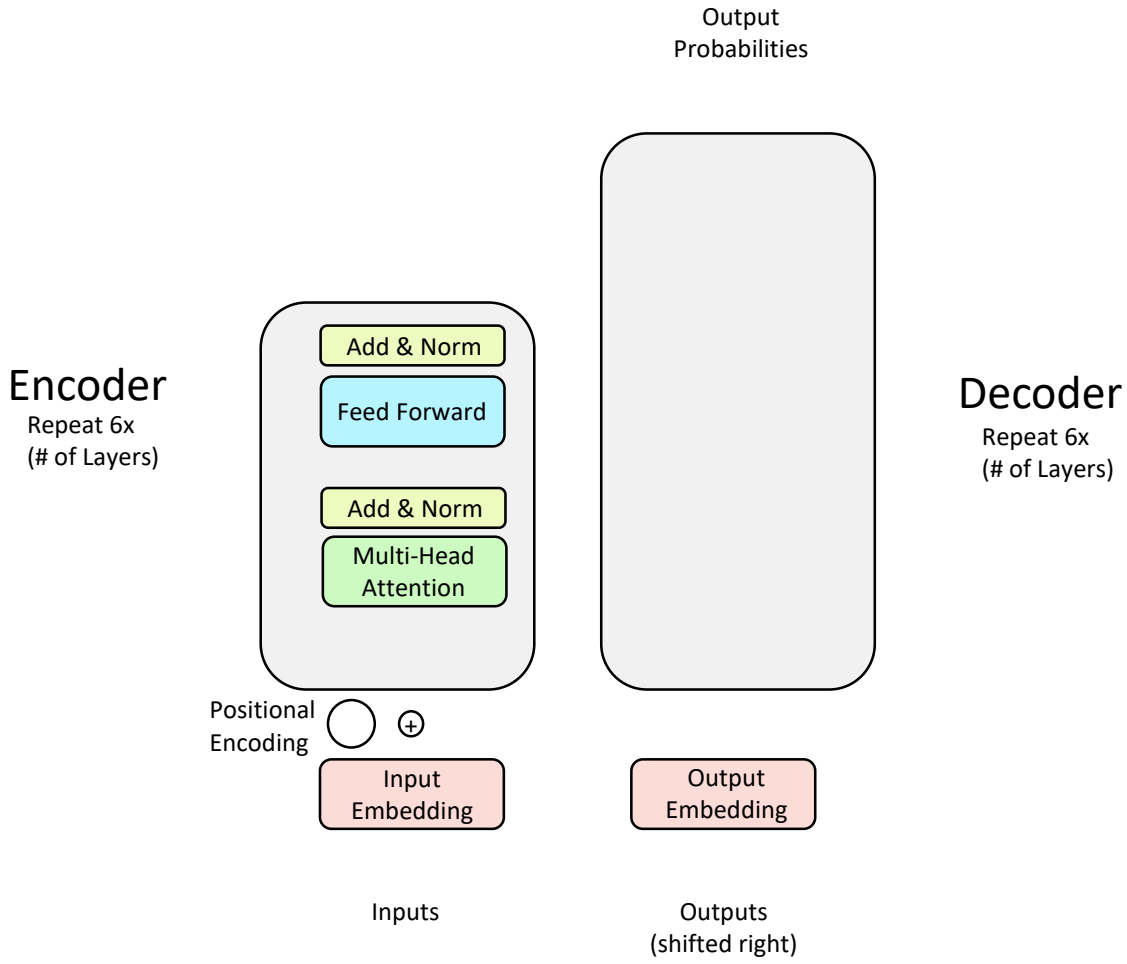
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices.

- Let $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where *h* is the number of attention heads, and ℓ ranges from 1 to h.
- Each attention head performs attention independently:
  - $\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = Y[\text{output}_1; \ldots; \text{output}_h]$, where $Y \in \mathbb{R}^{d \times d}$



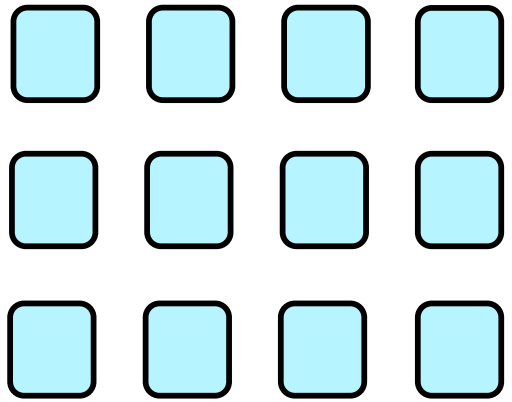
Credit to https://jalammar.github.io/illustrated-transformer/

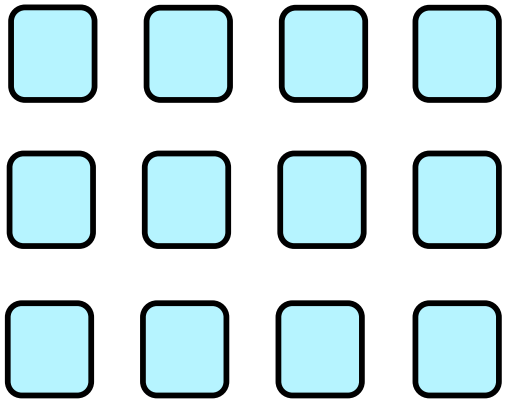
We've completed the Encoder! Time for the Decoder...



Decoder: Masked Multi-Head Self-Attention

- **Problem**: How do we keep the decoder from cheating? If we have a language modeling objective, can't the network just look ahead and "see" the answer?

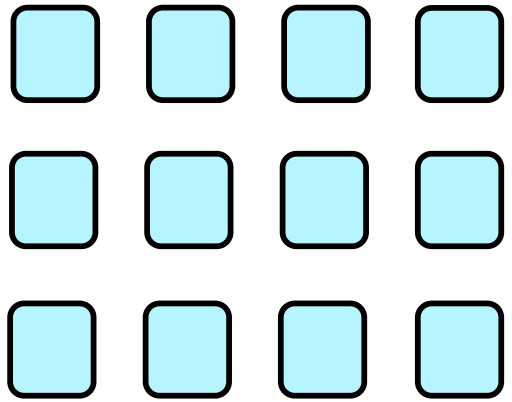


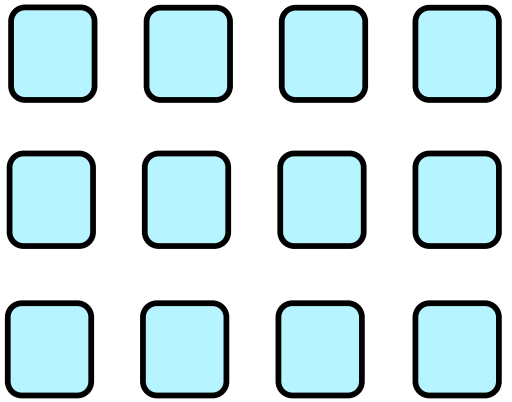


Transformer-Based
Encoder-Decoder Model

Decoder: Masked Multi-Head Self-Attention

- **Problem:** How do we keep the decoder from "cheating"? If we have a language modeling objective, can't the network just look ahead and "see" the answer?
- **Solution:** Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.



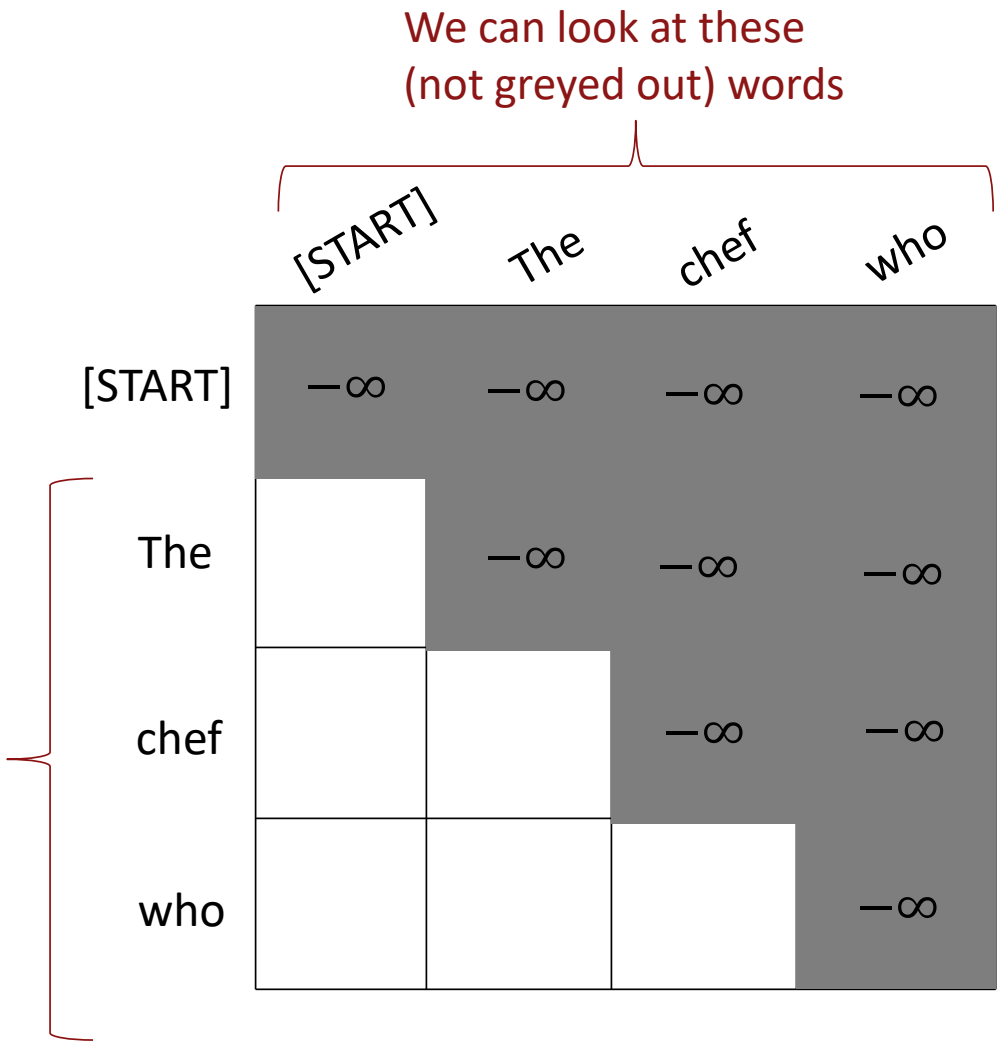


Transformer-Based
Encoder-Decoder Model

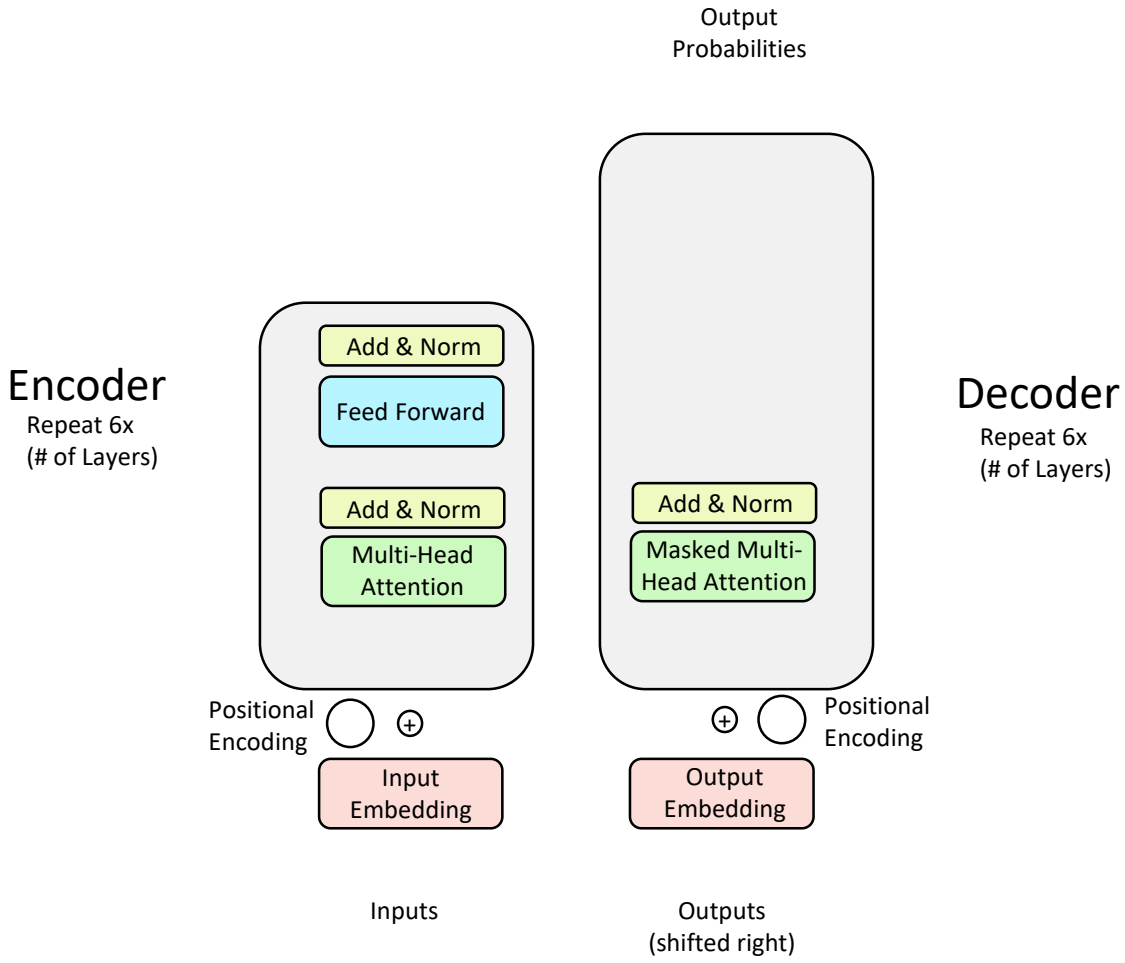
Masking the future in self-attention

- 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 $-\infty$.

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

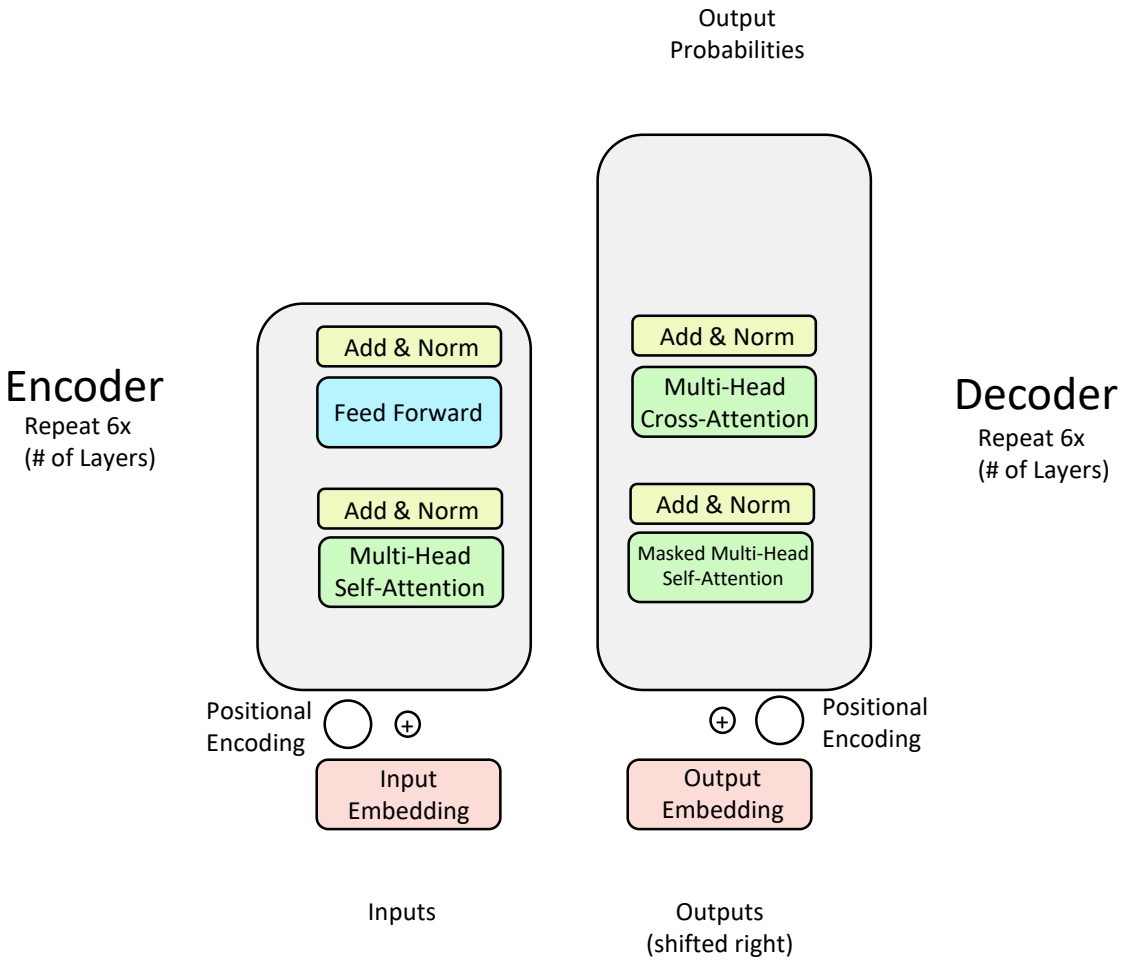


Decoder: Masked Multi-Headed Self-Attention

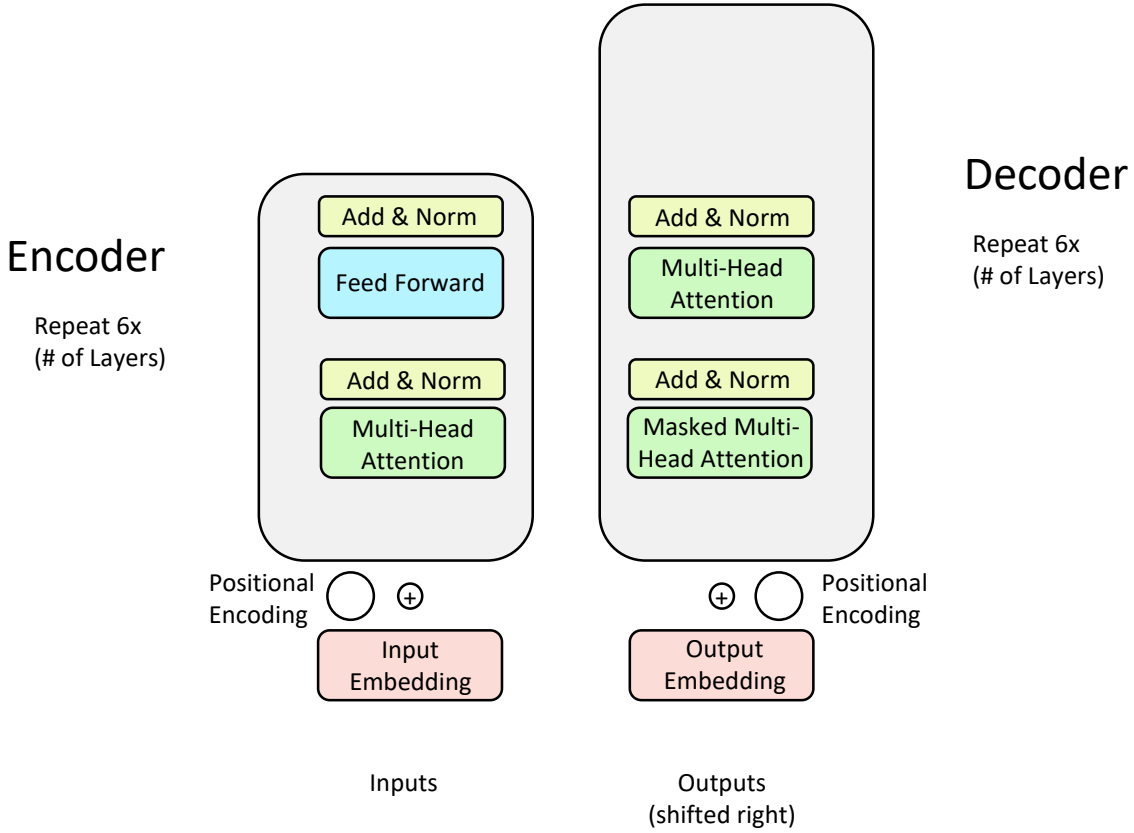


Encoder-Decoder Attention

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

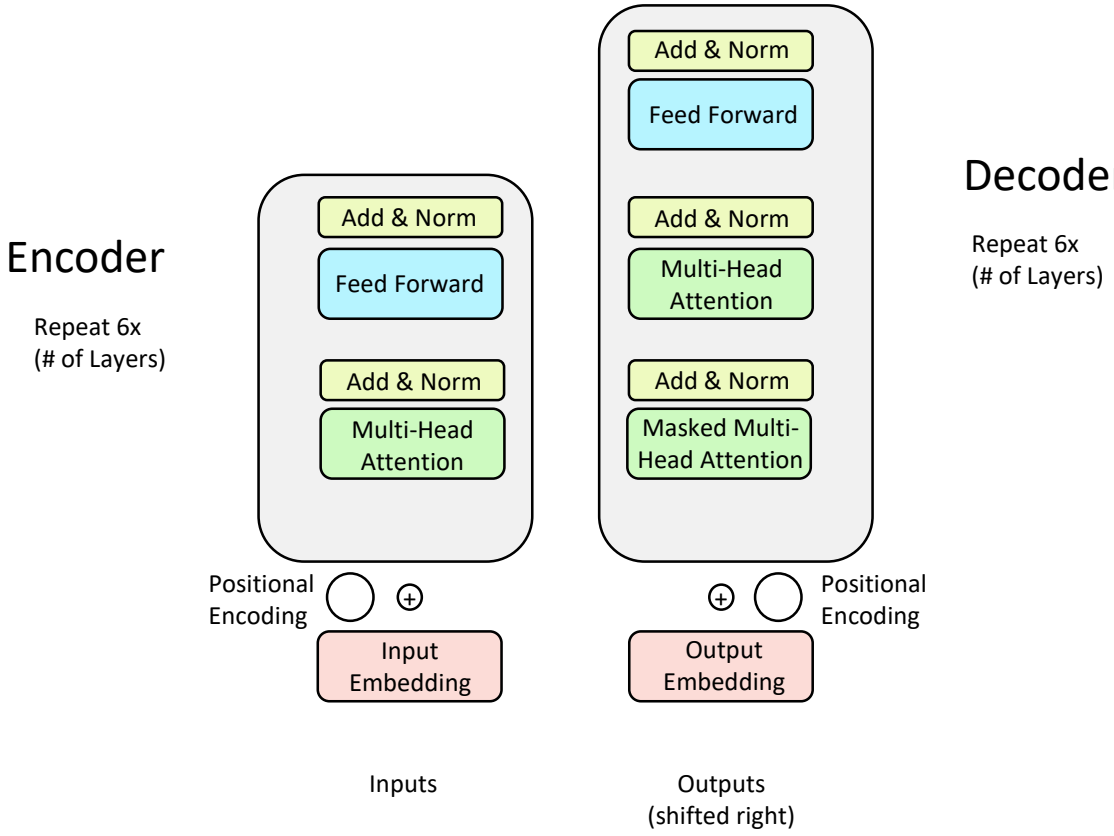


Decoder: Finishing touches!



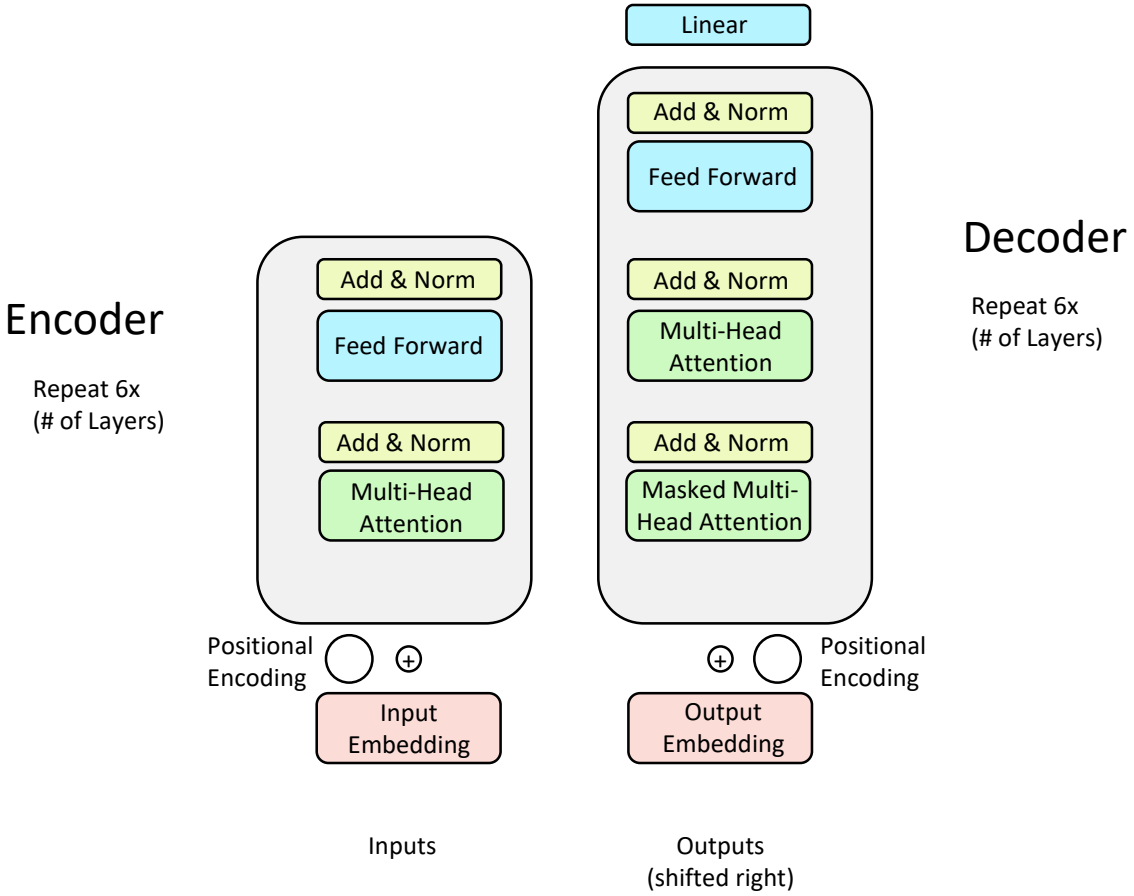
Decoder: Finishing touches!

- Add a feed forward layer (with residual connections and layer norm)



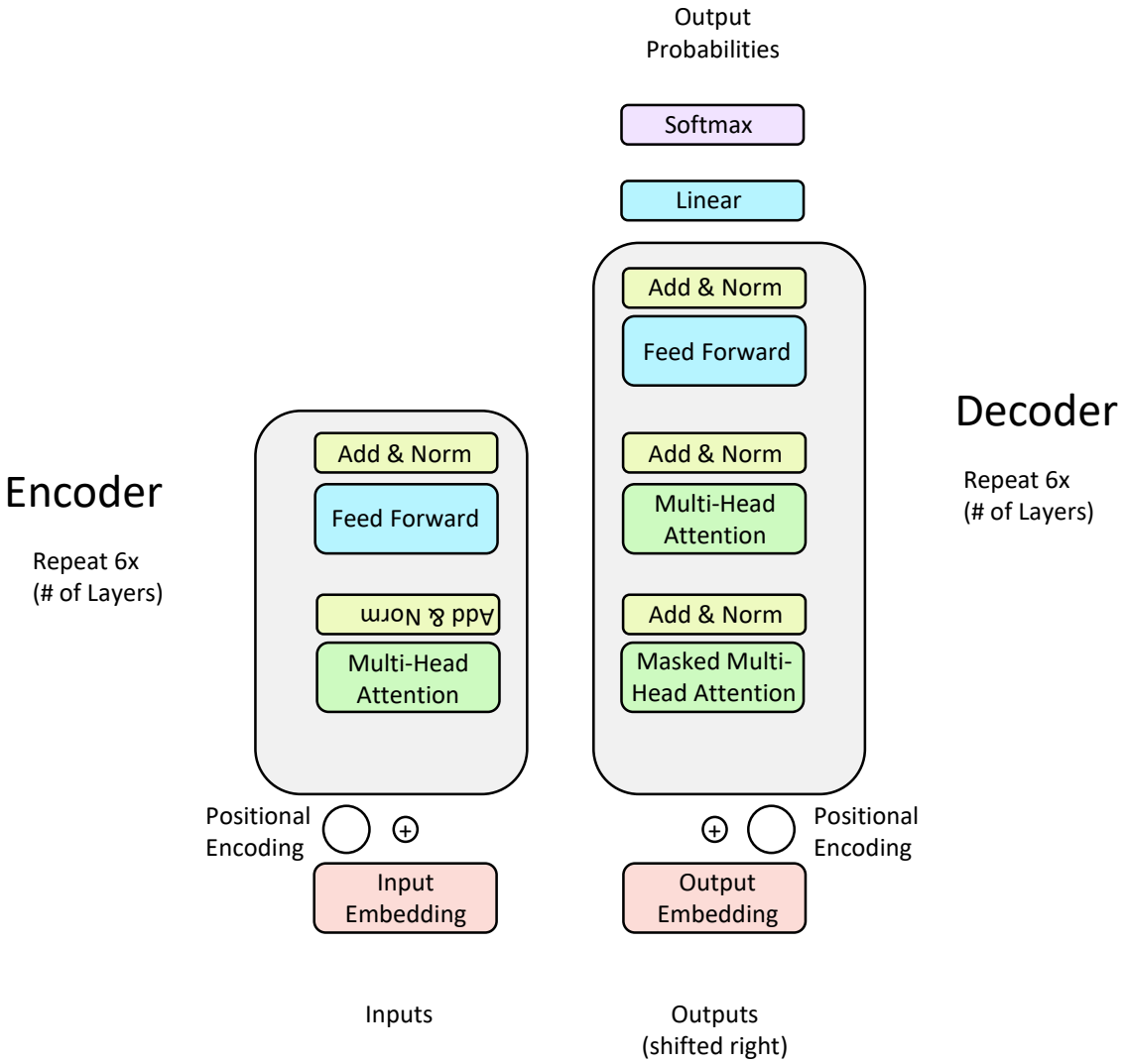
Decoder: Finishing touches!

- ❑ Add a feed forward layer (with residual connections and layer norm)
- ❑ Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)

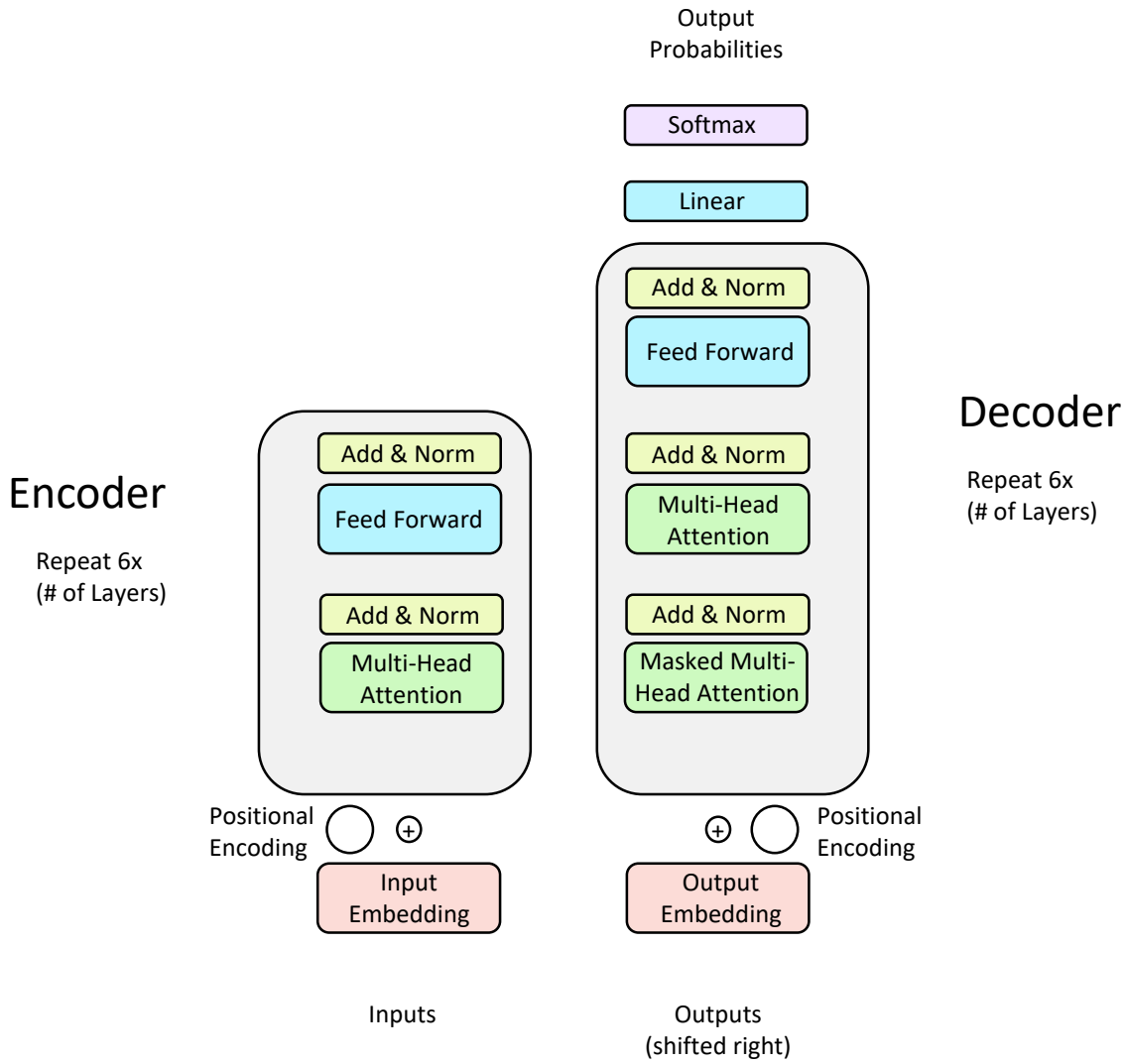


Decoder: Finishing touches!

- ❑ Add a feed forward layer (with residual connections and layer norm)
- ❑ Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- ❑ Add a final softmax to generate a probability distribution of possible next words!



Recap of Transformer Architecture



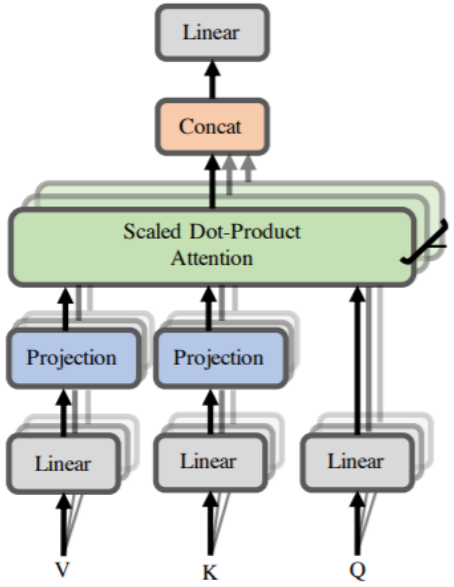
What would we like to fix about the Transformer?

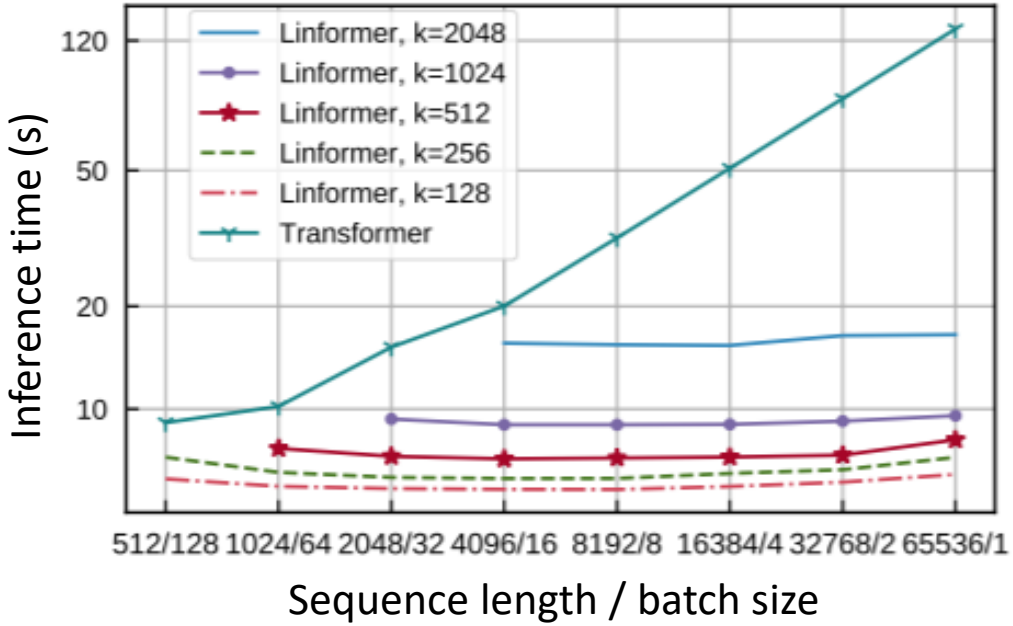
- **Quadratic compute in self-attention (today):**
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!
- **Position representations**:
  - Are simple absolute indices the best we can do to represent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

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

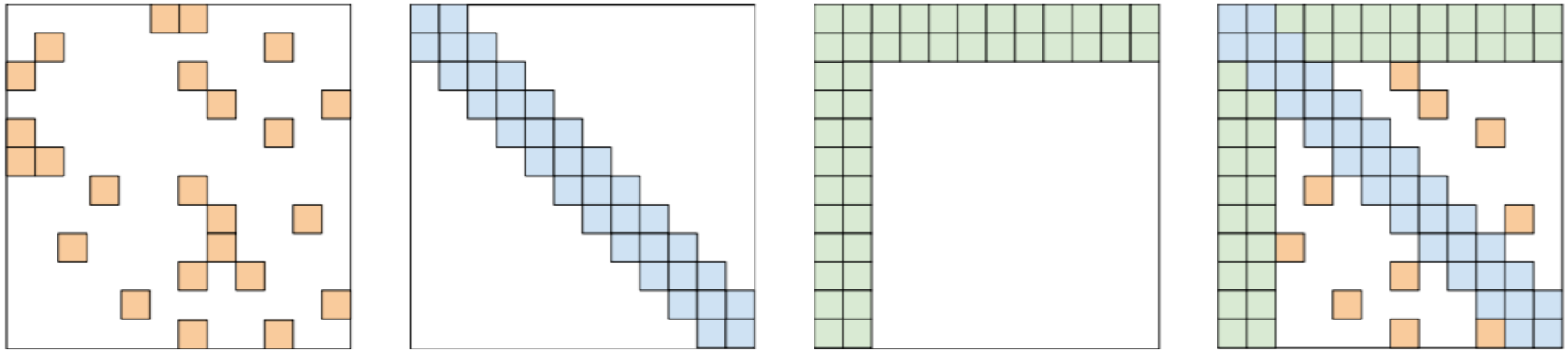




Recent work on improving on quadratic self-attention cost

- 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

Parting remarks

- ❑ You now understand Transformers!
- ❑ Next class, we will see how pre-training can take performance to the next level!