Transformers 2

Ling 282/482: Deep Learning for Computational Linguistics

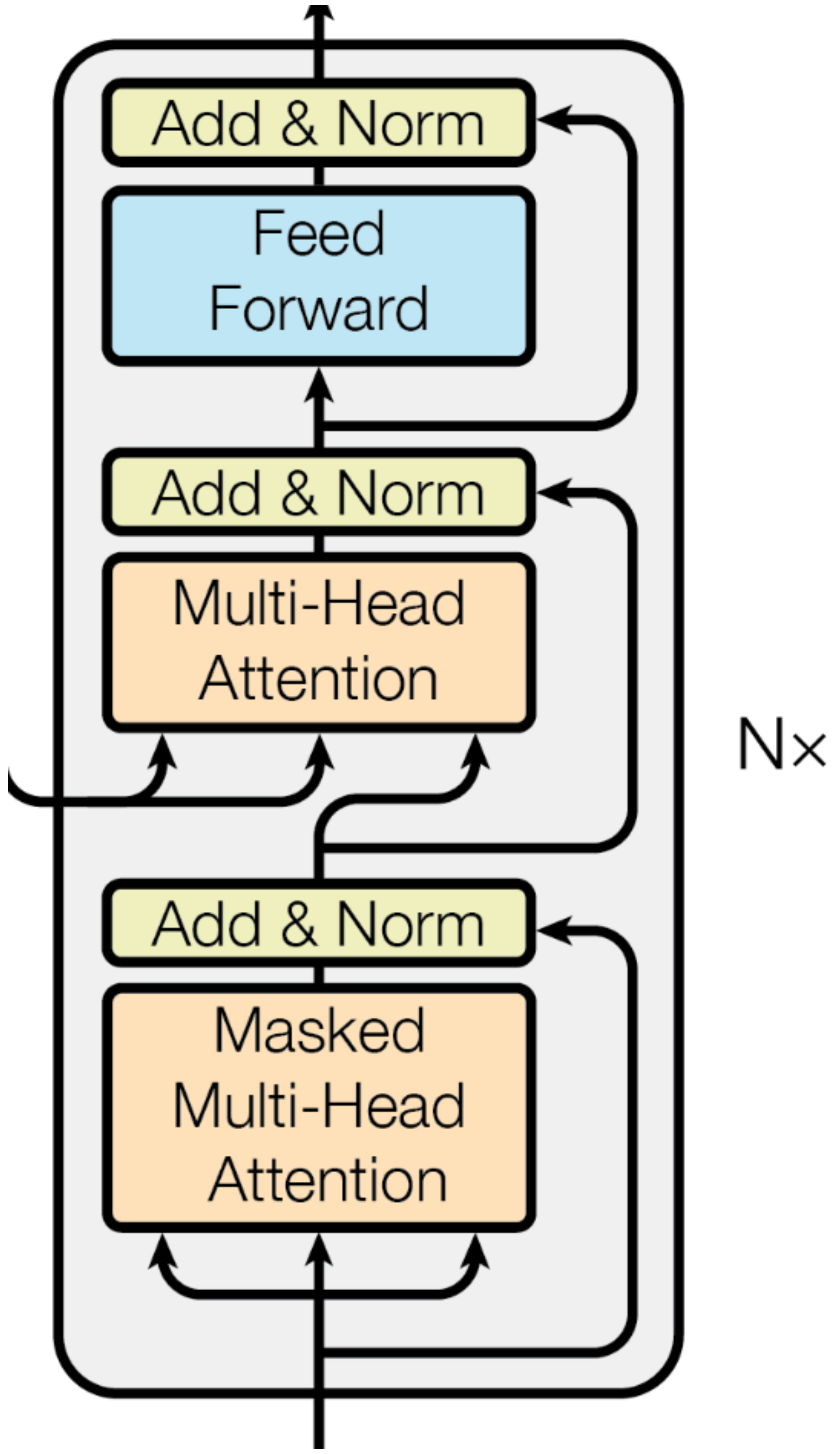
C.M. Downey

Fall 2025

Transformer Decoder

Decoder Block

- Like the encoder, the decoder is many *blocks* stacked vertically
- Two slightly different ingredients:
  - **Masked** self-attention
  - **Cross-attention** (encoder-decoder)

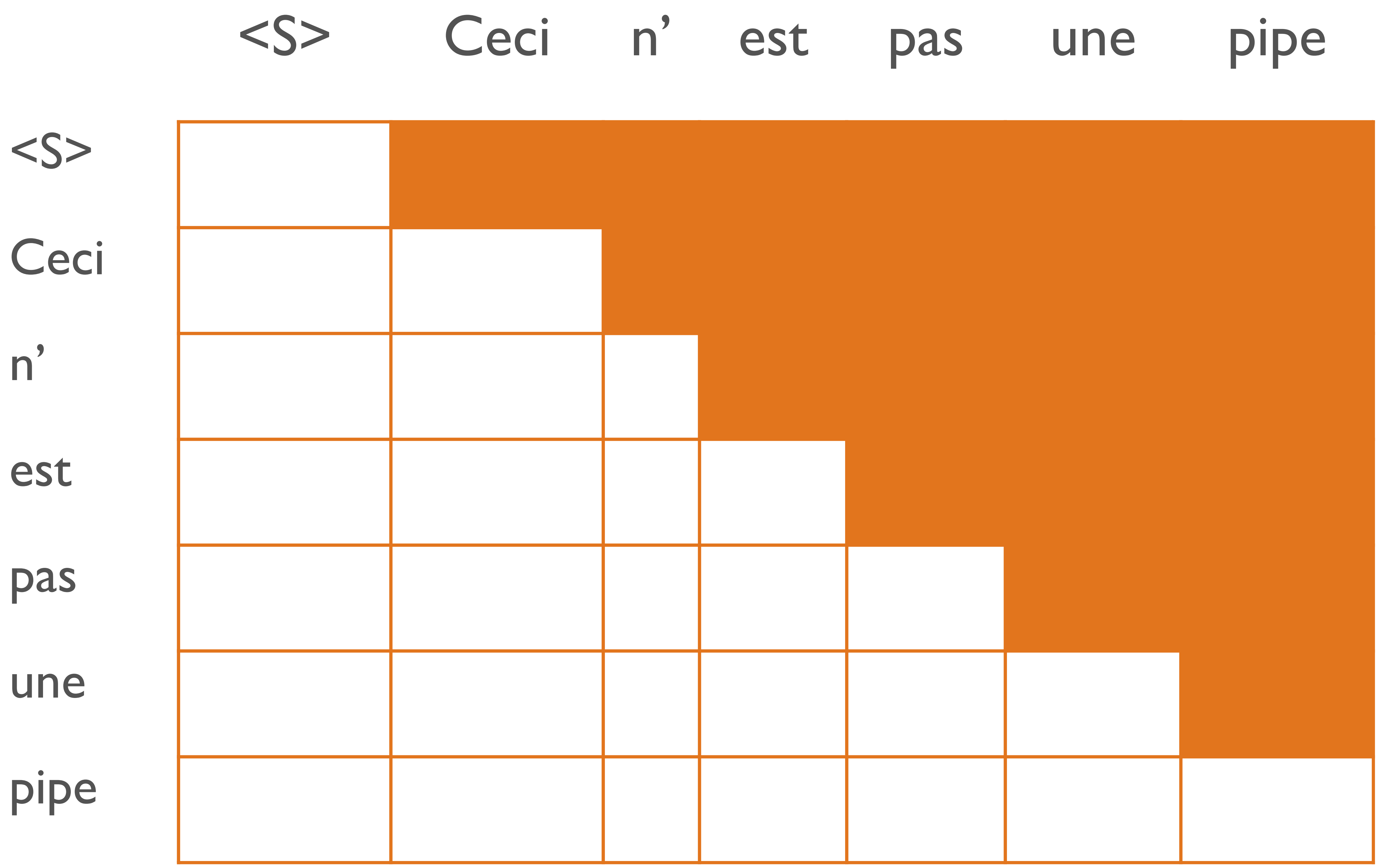


Attention Computation Practice

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 6 & 4 & 2 \\ 5 & 3 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{bmatrix}$$

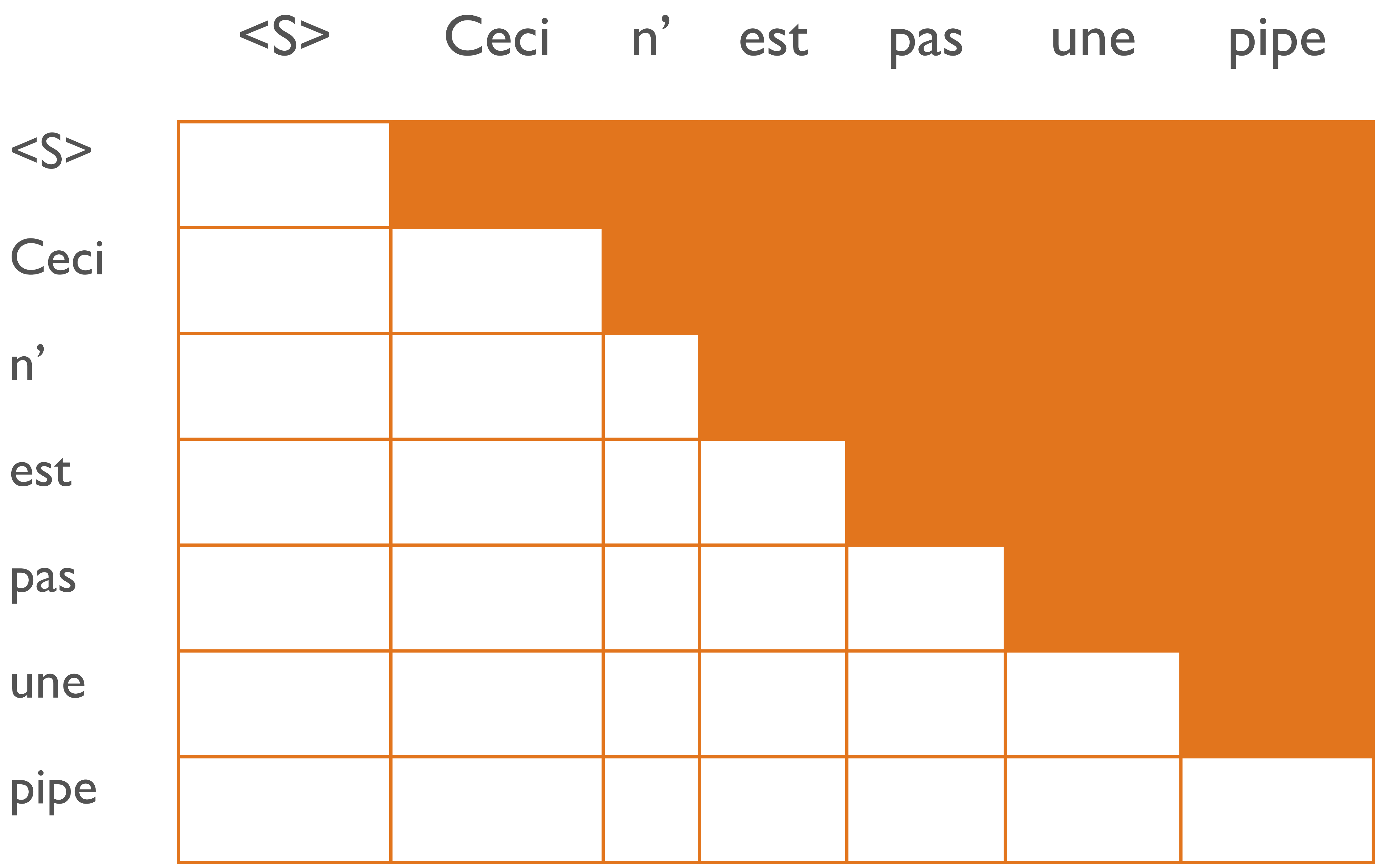
Q K^T V

Masking Out the Future



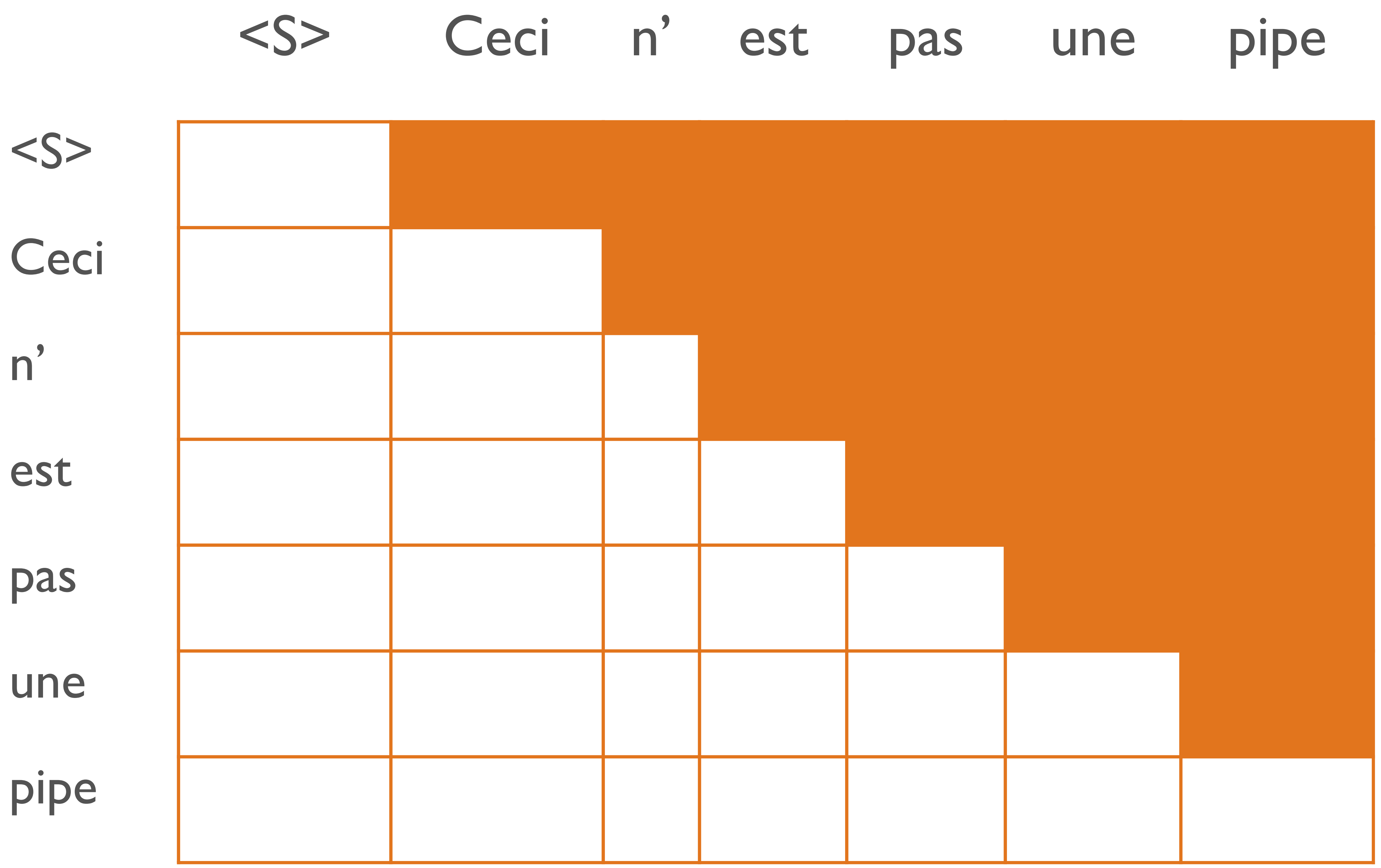
Masking Out the Future

- Key idea: use a **mask** to **block out certain attention scores**



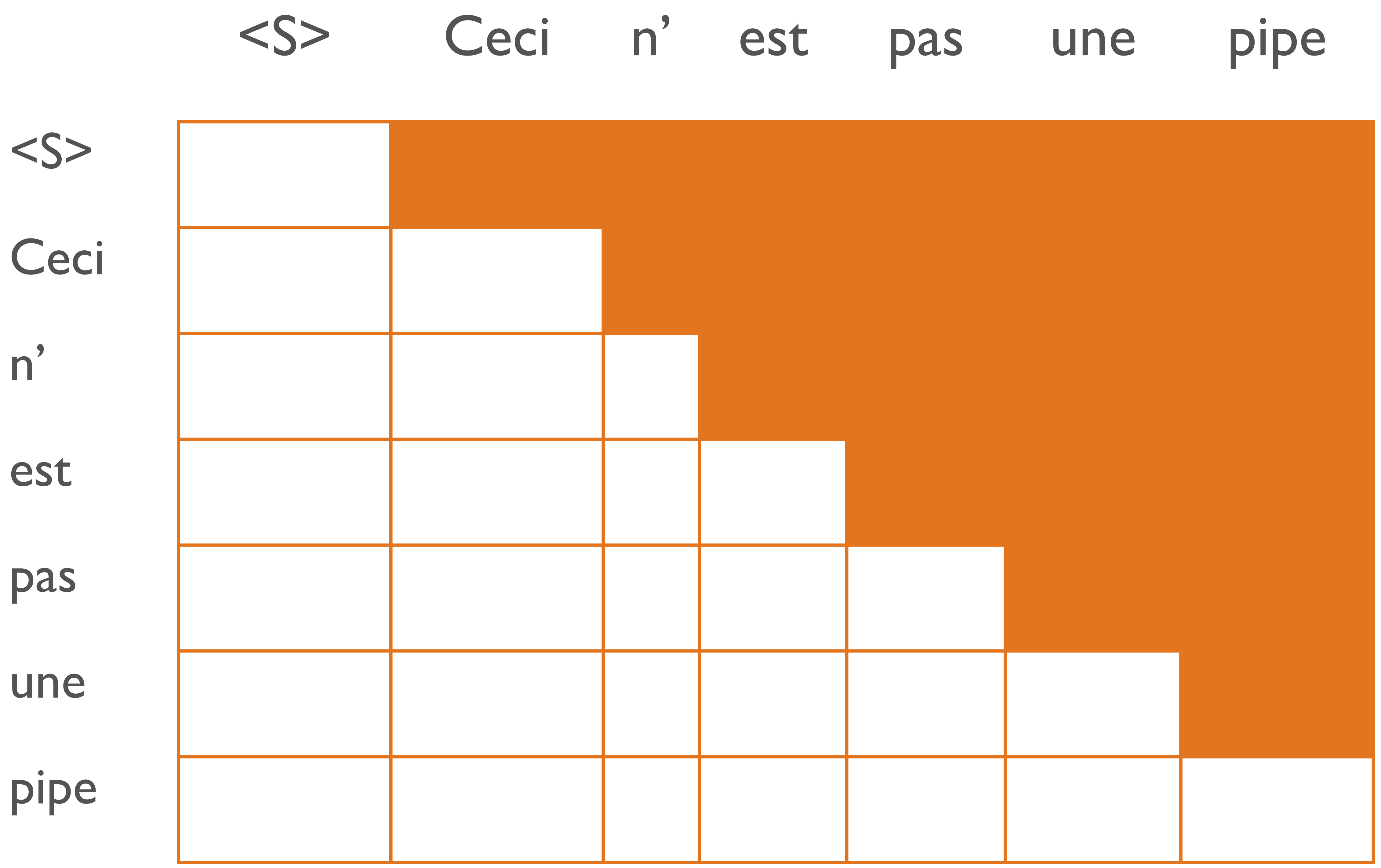
Masking Out the Future

- Key idea: use a **mask** to **block out certain attention scores**
- Tokens in the rows (**queries**) can **not** pay attention to tokens in the columns (**keys**) that are shaded in



Masking Out the Future

- Key idea: use a **mask** to **block out certain attention scores**
- Tokens in the rows (**queries**) can **not** pay attention to tokens in the columns (**keys**) that are shaded in
- Sometimes called a **"causal"** or **"directional"** mask
  - Recall that otherwise Transformers **don't intrinsically model order!**



Masking Out the Future

QK^T: total attention scores

$$\text{mask}_{ij} = \begin{cases} -\infty & j > i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right) V$$

	<S>	Ceci	n’	est	pas	une	pipe
<S>	0	-inf	-inf	-inf	-inf	-inf	-inf
Ceci	0	0	-inf	-inf	-inf	-inf	-inf
n’	0	0	0	-inf	-inf	-inf	-inf
est	0	0	0	0	-inf	-inf	-inf
pas	0	0	0	0	0	-inf	-inf
une	0	0	0	0	0	0	-inf
pipe	0	0	0	0	0	0	0

Masking Out the Future

QK^T: total attention scores

$$\text{mask}_{ij} = \begin{cases} -\infty & j > i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \text{mask}\right)V$$

	<S>	Ceci	n’	est	pas	une	pipe
<S>	0	-inf	-inf	-inf	-inf	-inf	-inf
Ceci	0	0	-inf	-inf	-inf	-inf	-inf
n’	0	0	0	-inf	-inf	-inf	-inf
est	0	0	0	0	-inf	-inf	-inf
pas	0	0	0	0	0	-inf	-inf
une	0	0	0	0	0	0	-inf
pipe	0	0	0	0	0	0	0

Why 0 and -inf?

Cross-Attention

Cross-Attention

- Recall attention's original use: allows a **decoder** to attend to **all the encoder’s representations**, instead of just the final one

Cross-Attention

- Recall attention's original use: allows a **decoder** to attend to **all the encoder's representations**, instead of just the final one
- How can we apply this in Transformer-land?
    - What are the queries, keys, and values?

Cross-Attention

- Recall attention's original use: allows a **decoder** to attend to **all the encoder’s representations**, instead of just the final one
- How can we apply this in Transformer-land?
  - What are the queries, keys, and values?
- **Queries**: learned from **decoder** representations X

Cross-Attention

- Recall attention's original use: allows a **decoder** to attend to **all the encoder’s representations**, instead of just the final one
- How can we apply this in Transformer-land?
    - What are the queries, keys, and values?
- **Queries**: learned from **decoder** representations X
- **Keys** and **Values**: learned from top-layer **encoder** representations Z

Cross-Attention

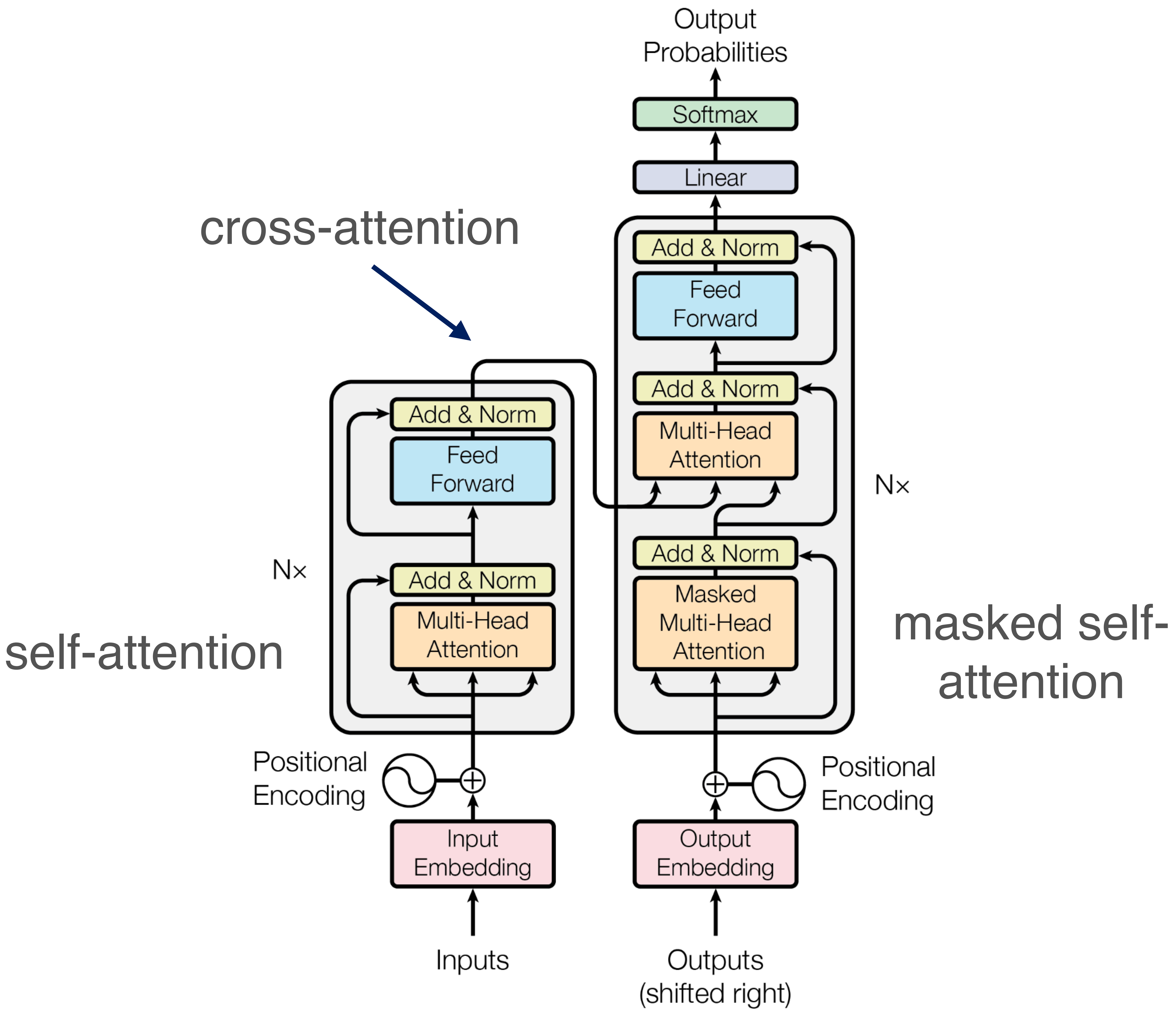
- Recall attention's original use: allows a **decoder** to attend to **all the encoder's representations**, instead of just the final one
- How can we apply this in Transformer-land?
  - What are the queries, keys, and values?
- **Queries**: learned from **decoder** representations X
- **Keys** and **Values**: learned from top-layer **encoder** representations Z
- Learned weight matrices W_q, W_k, W_v as before

Cross-Attention

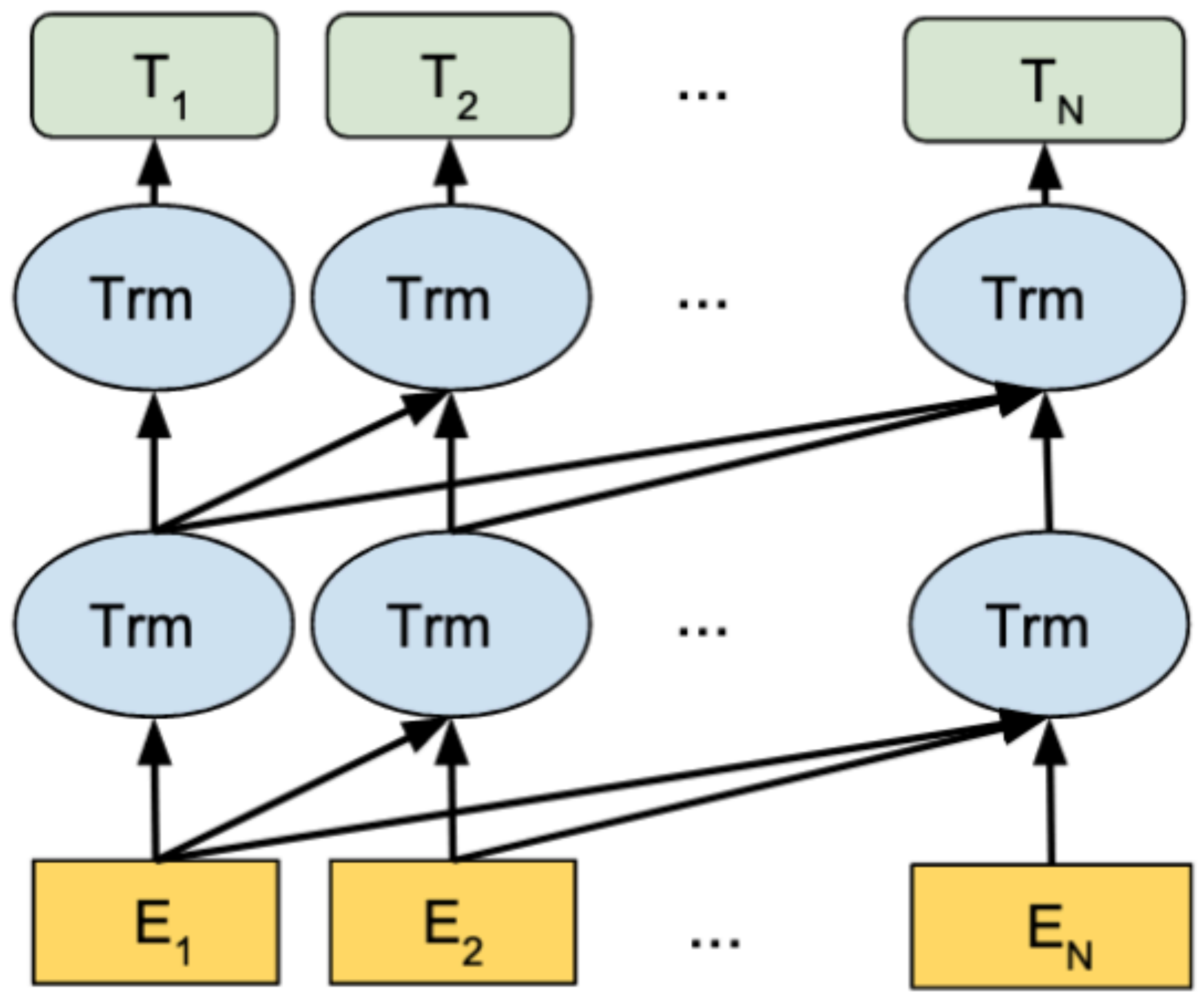
- Recall attention's original use: allows a **decoder** to attend to **all the encoder's representations**, instead of just the final one
- How can we apply this in Transformer-land?
  - What are the queries, keys, and values?
- **Queries**: learned from **decoder** representations X
- **Keys** and **Values**: learned from top-layer **encoder** representations Z
- Learned weight matrices W_q, W_k, W_v as before

$$\text{CrossAttention} = \text{Attention}\left(XW_q, ZW_k, ZW_v\right)$$

Full Transformer Encoder-Decoder



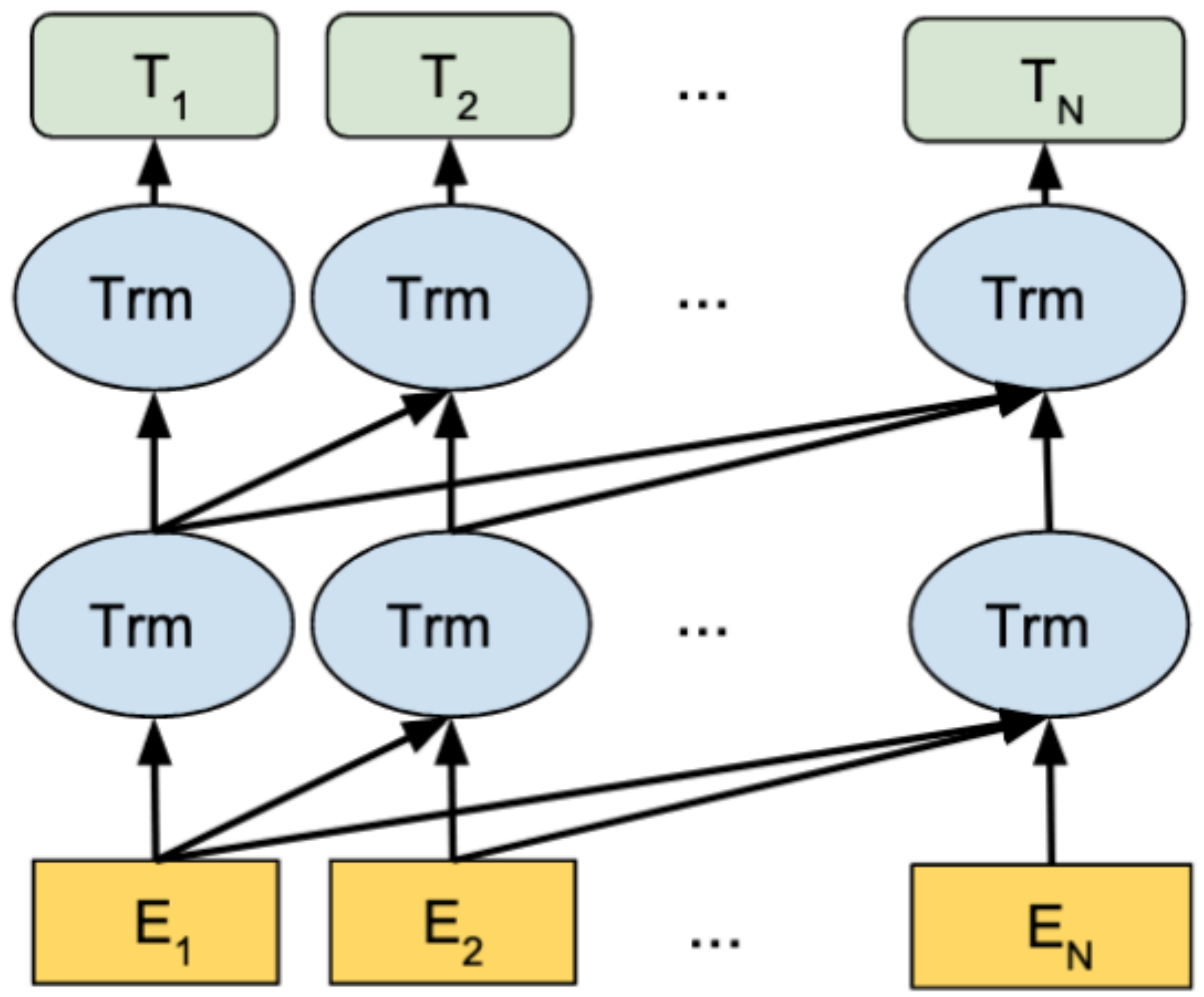
Transformer Decoder



source

Transformer Decoder

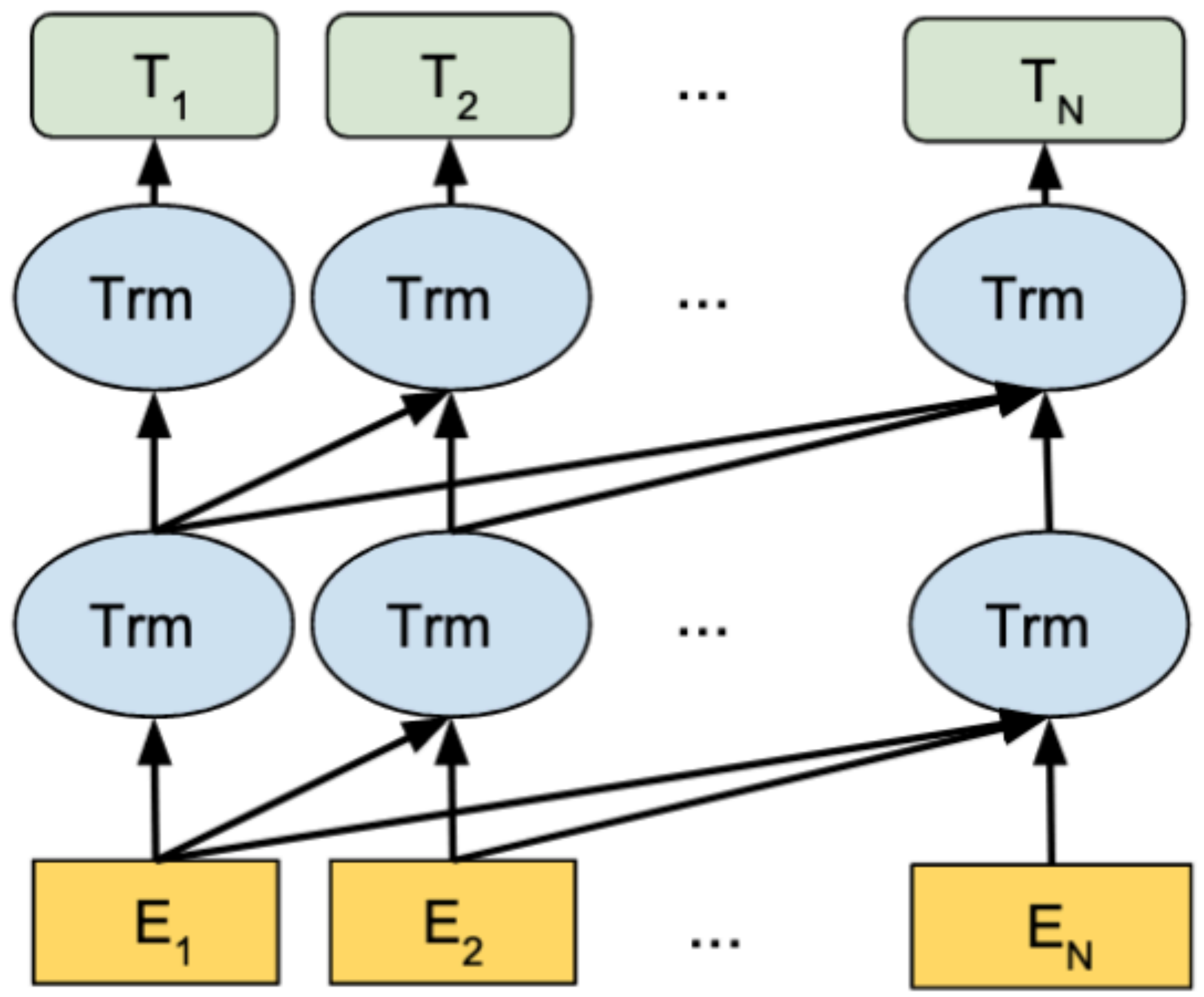
- Can be used any place you would use a decoder



source

Transformer Decoder

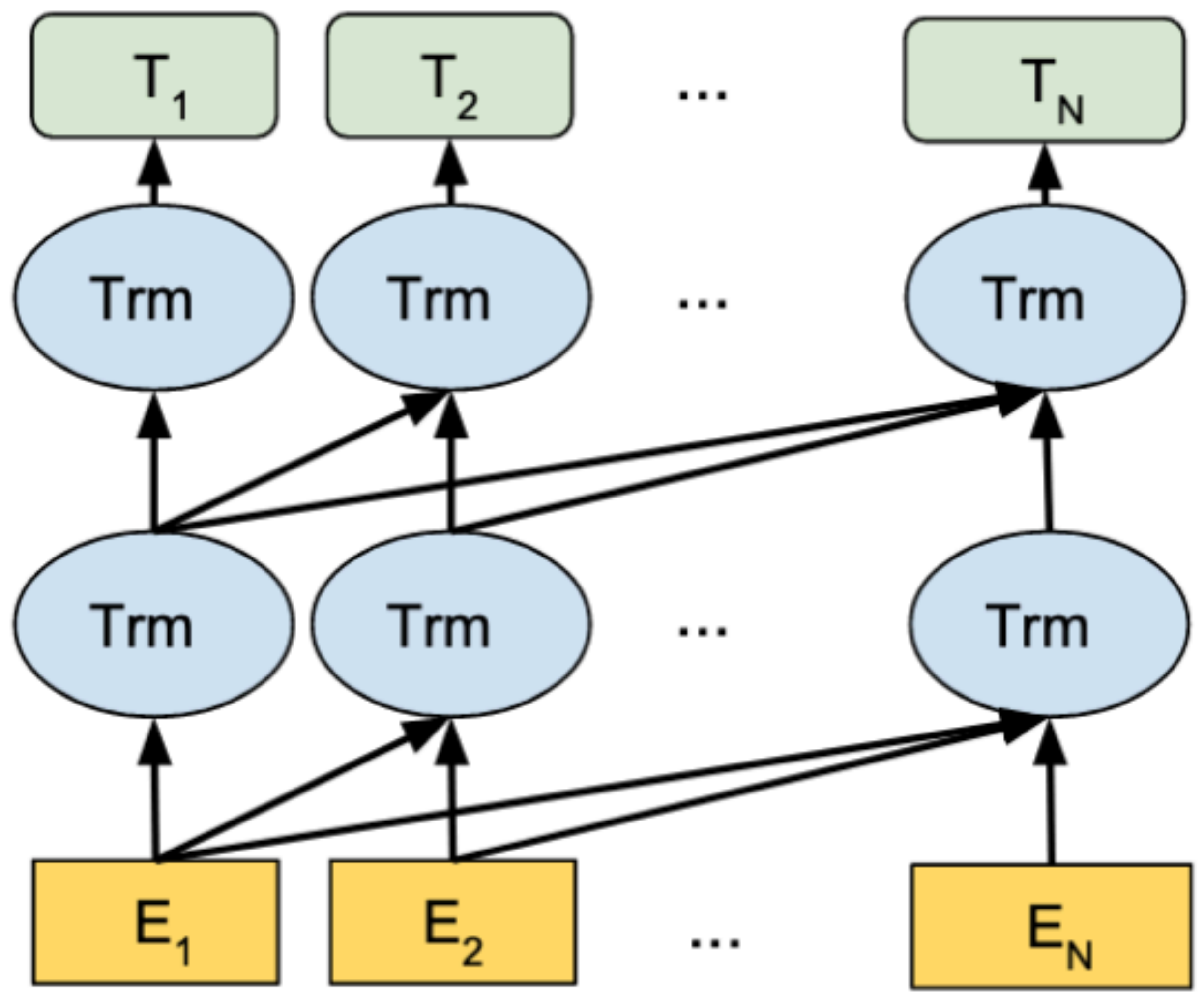
- Can be used any place you would use a decoder
- Masked attention **prevents "peeking into the future"**



source

Transformer Decoder

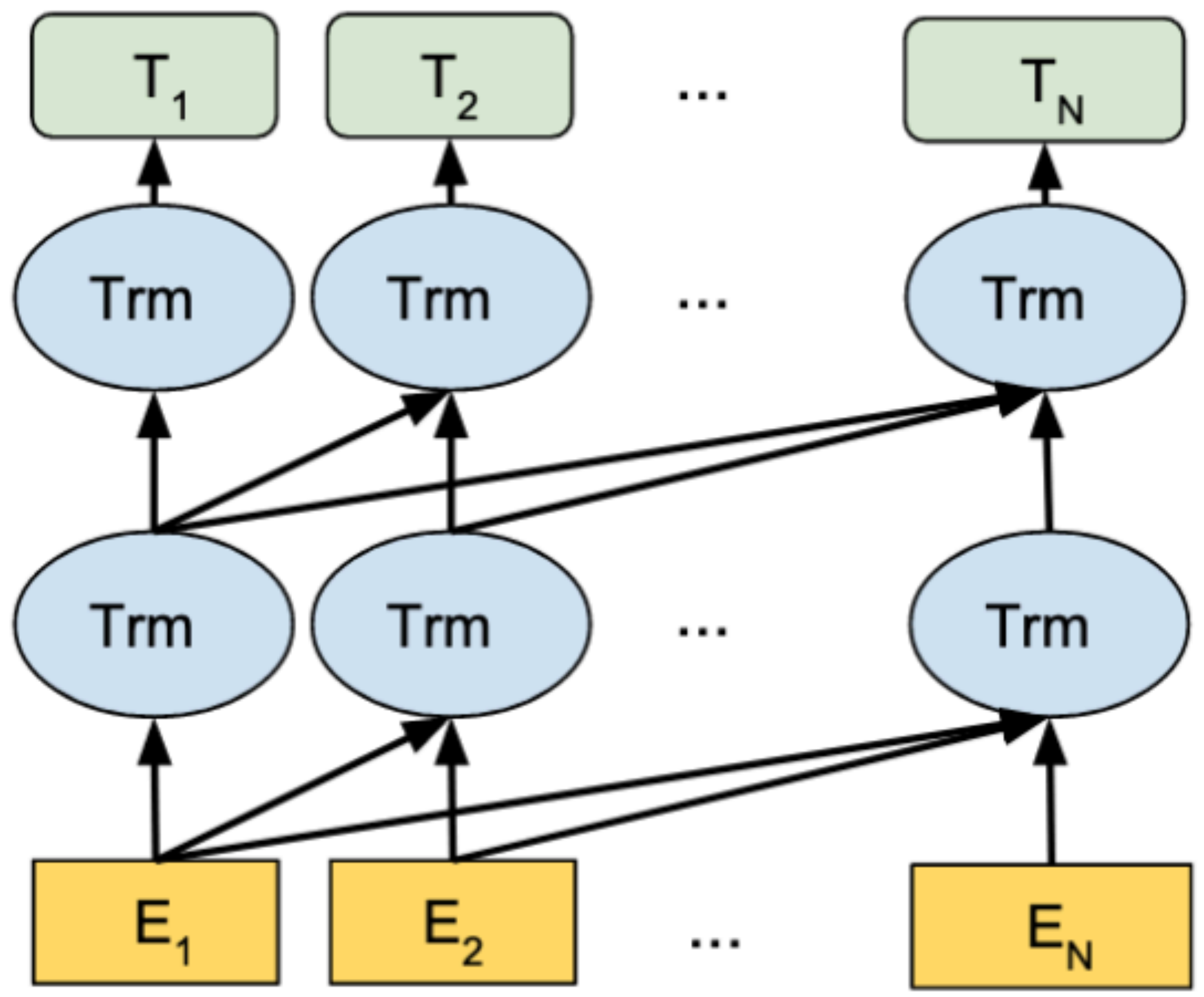
- Can be used any place you would use a decoder
- Masked attention **prevents "peeking into the future"**
- In seq2seq, for conditional language modeling, e.g. translation, summarization



source

Transformer Decoder

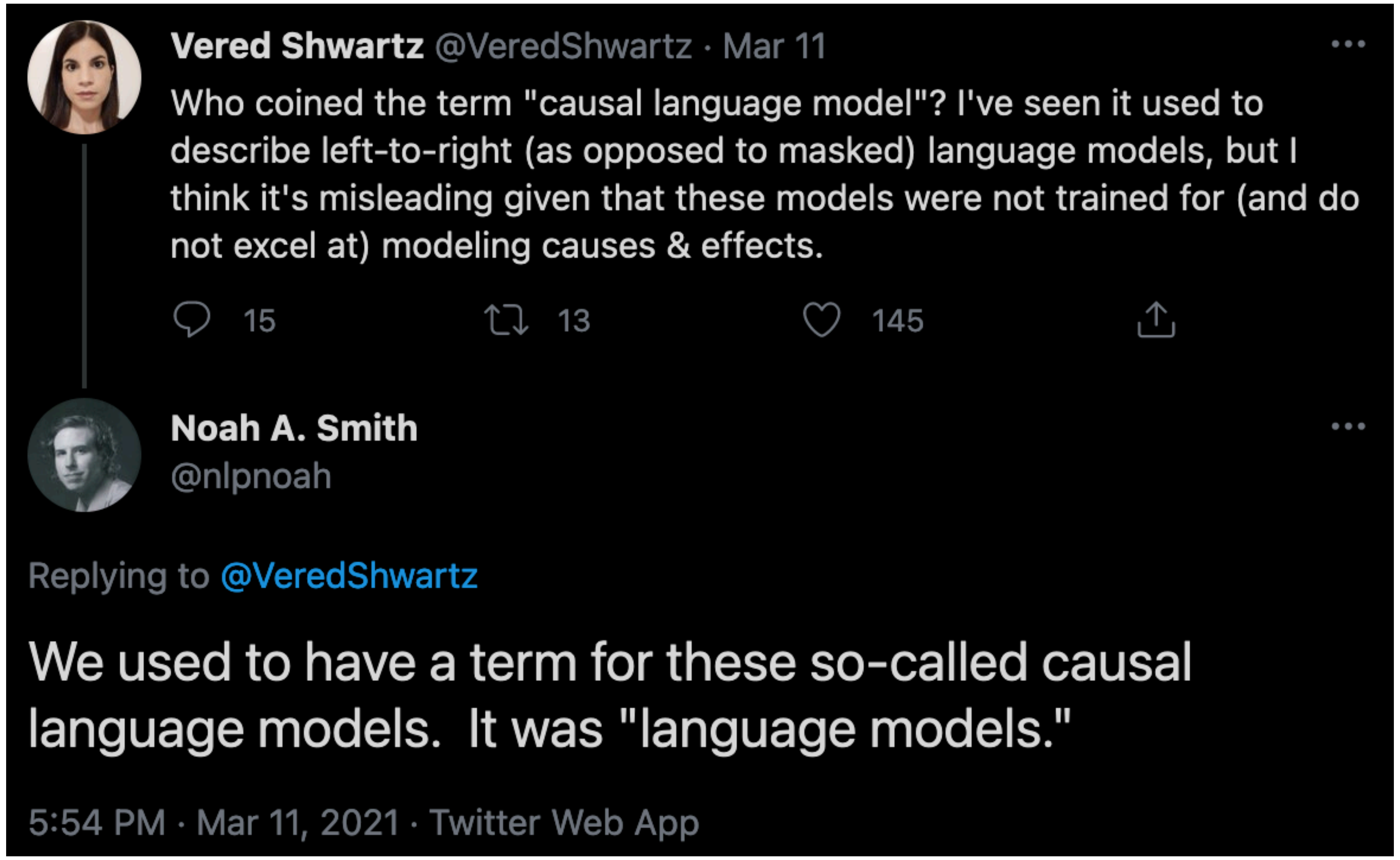
- Can be used any place you would use a decoder
- Masked attention **prevents “peeking into the future”**
- In seq2seq, for conditional language modeling, e.g. translation, summarization
- Can be used **on its own**, as a **“pure” language model**
  - If so, used **without cross-attention**
  - (people now call this **“causal language modeling”** sometimes)



source

Transformer Decoder

- Can be used any place you would use a decoder
- Masked attention **prevents "peeking into the future"**
- In seq2seq, for conditional language modeling, e.g. translation, summarization
- Can be used **on its own**, as a **"pure" language model**
  - If so, used **without cross-attention**
  - (people now call this **"causal language modeling"** sometimes)



source

Transformer Architecture Summary

Transformer Architecture Summary

- Main building block: **attention**
  - Encoder: **self-attention**
  - Decoder: **masked** self-attention
  - Decoder-encoder: **cross-attention**

Transformer Architecture Summary

- Main building block: **attention**
  - Encoder: **self-attention**
  - Decoder: **masked** self-attention
  - Decoder-encoder: **cross-attention**
- **Position embeddings** to "inject" information about **sequence order**

Transformer Architecture Summary

- Main building block: **attention**
  - Encoder: **self-attention**
  - Decoder: **masked** self-attention
  - Decoder-encoder: **cross-attention**
- **Position embeddings** to "inject" information about **sequence order**
- Position-wise feed-forward networks for element-wise nonlinearities

Transformer Architecture Summary

- Main building block: **attention**
  - Encoder: **self-attention**
  - Decoder: **masked** self-attention
  - Decoder-encoder: **cross-attention**
- **Position embeddings** to "inject" information about **sequence order**
- Position-wise feed-forward networks for element-wise nonlinearities
- Residual connections + LayerNorm around every component

Transformers: Limitations

Quadratic Attention

Quadratic Attention

- Attention computes similarity scores between all **pairs of tokens**

Quadratic Attention

- Attention computes similarity scores between all **pairs of tokens**
  - QK^T: [seq_len, seq_len] shape

Quadratic Attention

- Attention computes similarity scores between all **pairs of tokens**
  - QK^T: [seq_len, seq_len] shape
  - In other words, **size of attention is** $O(n^2)$

Quadratic Attention

- Attention computes similarity scores between all **pairs of tokens**
  - QK^T: [seq_len, seq_len] shape
  - In other words, **size of attention is** $O(n^2)$
- Makes **scaling to long sequences difficult**

Quadratic Attention

- Attention computes similarity scores between all **pairs of tokens**
  - QK^T: [seq_len, seq_len] shape
  - In other words, **size of attention is** $O(n^2)$
- Makes **scaling to long sequences difficult**
  - This was a major hurdle for making **chatbots** more useful

Quadratic Attention

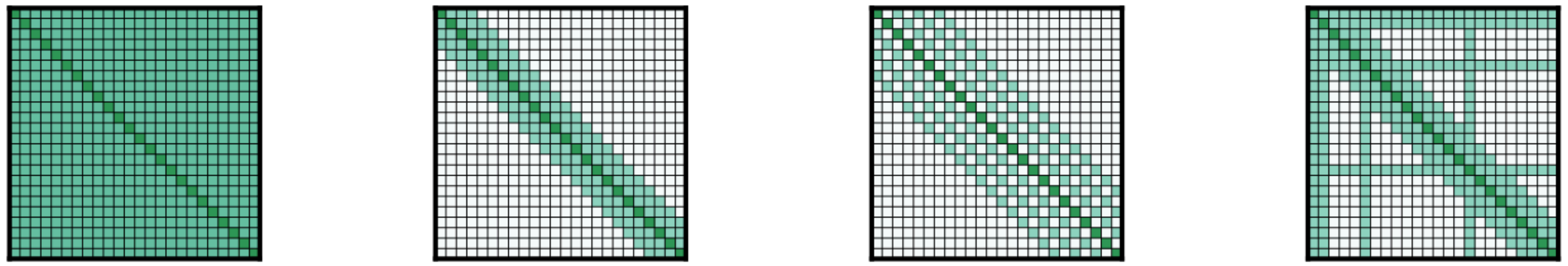
- Attention computes similarity scores between all **pairs of tokens**
    - QK^T: [seq_len, seq_len] shape
    - In other words, **size of attention is** $O(n^2)$
- Makes **scaling to long sequences difficult**
    - This was a major hurdle for making **chatbots** more useful
    - There are **all kinds of tricks** nowadays for allowing very long sequences

Quadratic Attention

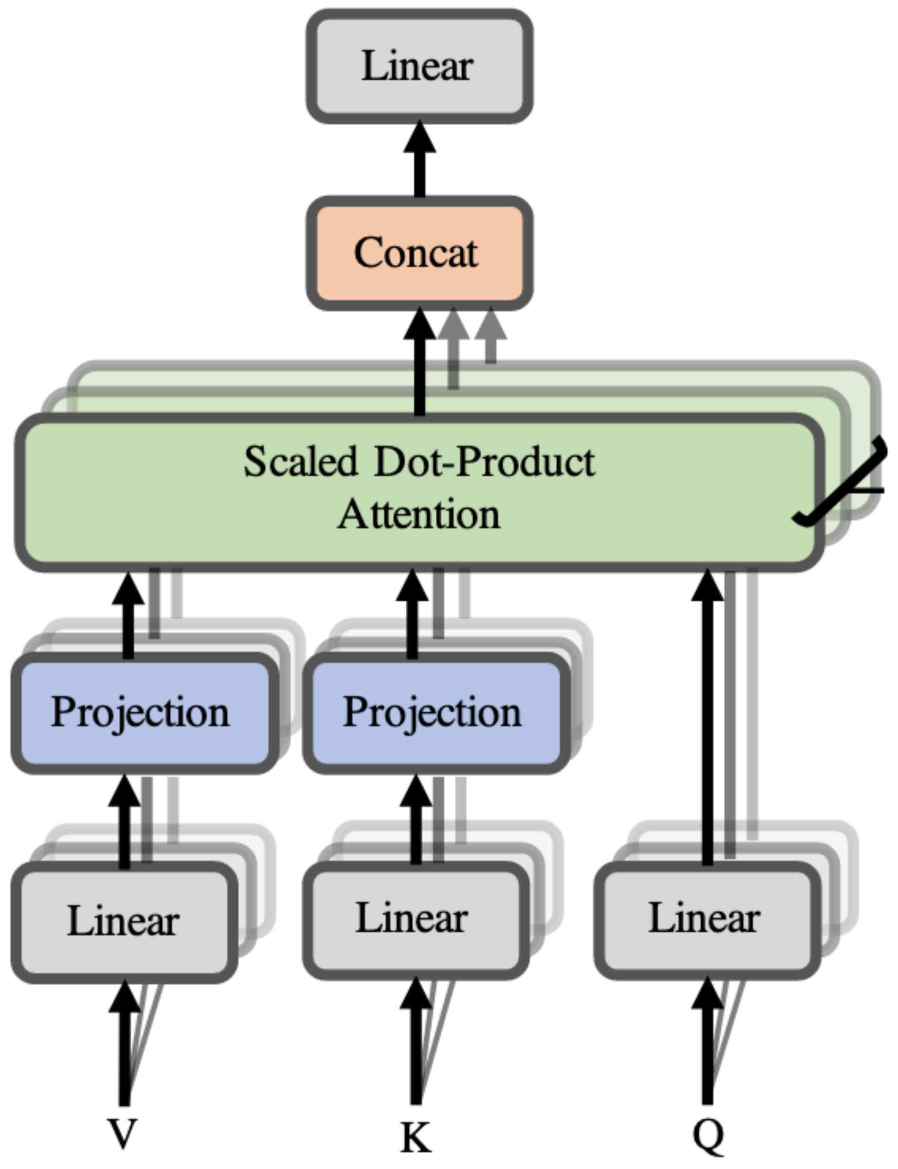
- Attention computes similarity scores between all **pairs of tokens**
  - QK^T: [seq_len, seq_len] shape
  - In other words, **size of attention is** $O(n^2)$
- Makes **scaling to long sequences difficult**
  - This was a major hurdle for making **chatbots** more useful
  - There are **all kinds of tricks** nowadays for allowing very long sequences
  - Some industrial LLMs can handle sequences **tens of thousands of tokens long!**

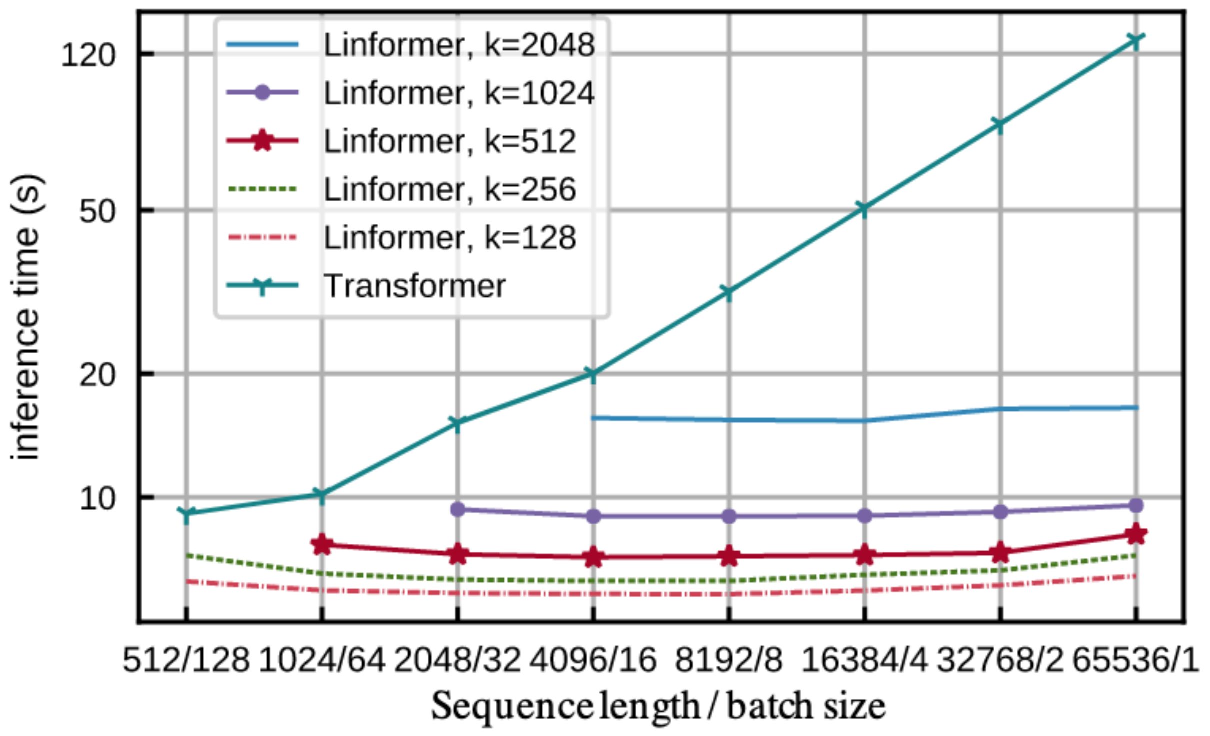
Efficient Attention Examples

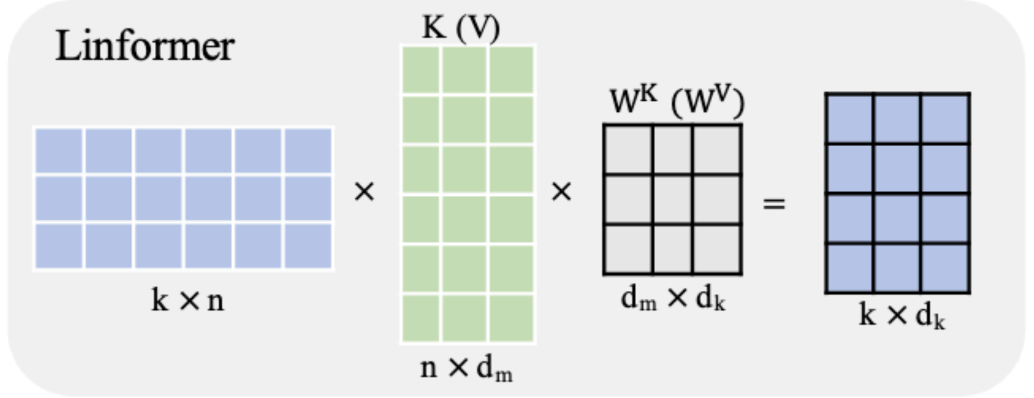
- Longformer:
    - Carefully control positions attended to
- Linformer:
    - Additional projection of Keys/Values to smaller space
    - $O(nk)$, with k a hyper-parameter
- Survey paper



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window







Inference speed does not scale with seq length

Recurrence in Generation

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**
    - **Sample** next token from this distribution

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**
    - **Sample** next token from this distribution
  - **Repeat** until (EOS | max length | other criterion)

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**
    - **Sample** next token from this distribution
  - **Repeat** until (EOS | max length | other criterion)
- This **loop is unavoidable** during generation

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**
    - **Sample** next token from this distribution
  - **Repeat** until (EOS | max length | other criterion)
- This **loop is unavoidable** during generation
  - Transformer's gains on parallelism: work for training, **vanish for generation**

Recurrence in Generation

- Recall the basic method for **generating** from a decoder:
  - Feed **initial token** (or a prompt)
  - Generate **probability over next tokens**
    - **Sample** next token from this distribution
  - **Repeat** until (EOS | max length | other criterion)
- This **loop is unavoidable** during generation
  - Transformer's gains on parallelism: work for training, **vanish for generation**
  - In fact, **RNN decoders** tend to be **much faster** at inference time

KV Caching

KV Caching

- **Simple idea**, very important in practice

KV Caching

- **Simple idea**, very important in practice
- During generation, there's **no need to re-compute** Keys and Values for **previous tokens**

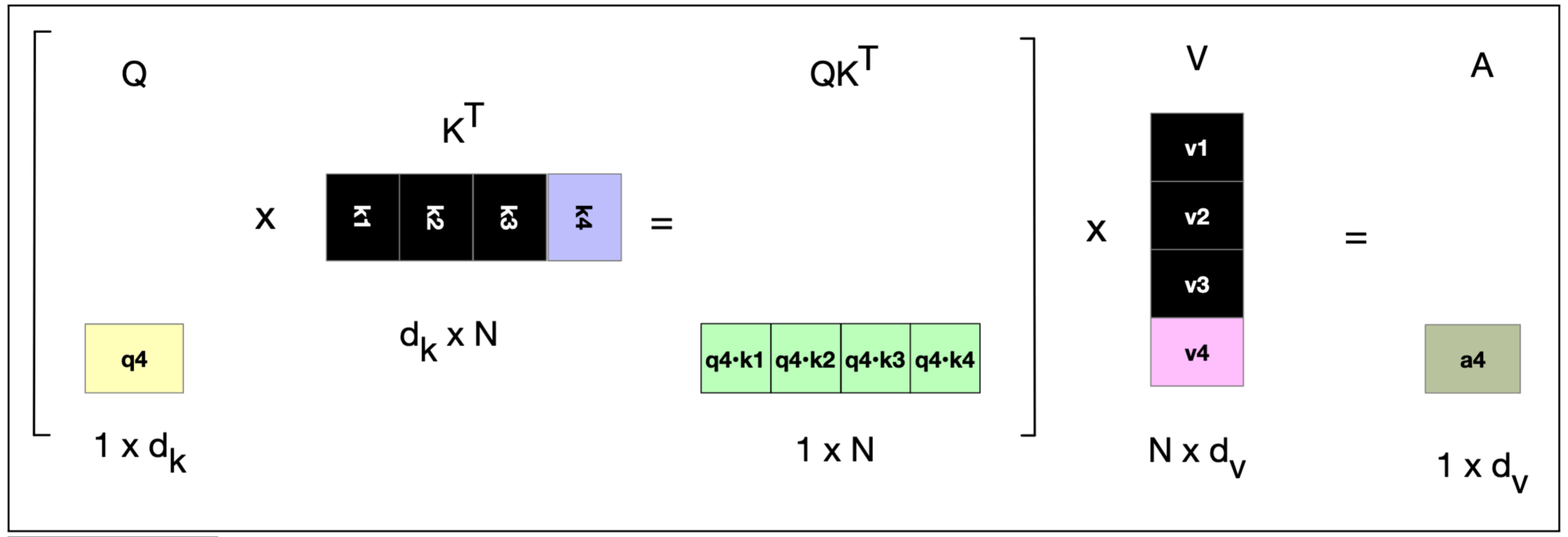


Figure 8.17 Parts of the attention computation (extracted from Fig. 8.10) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

KV Caching

- **Simple idea**, very important in practice
- During generation, there's **no need to re-compute** Keys and Values for **previous tokens**
  - Instead, store them in a **cache** for re-use

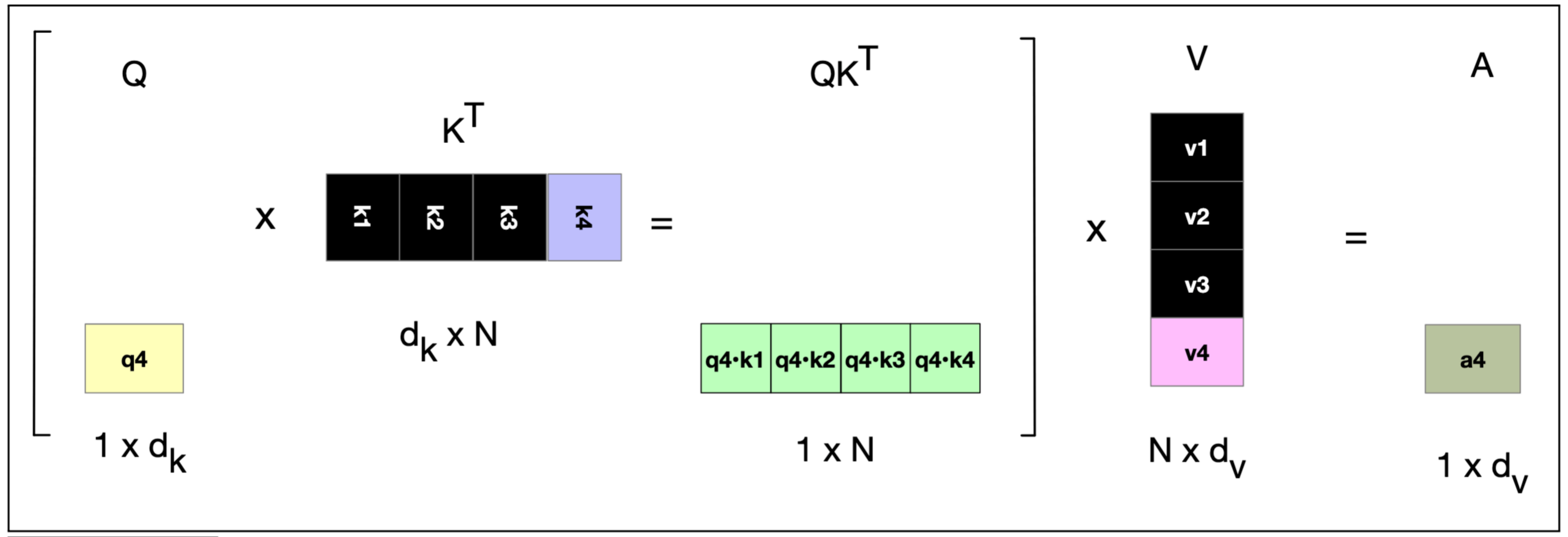


Figure 8.17 Parts of the attention computation (extracted from Fig. 8.10) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

KV Caching

- **Simple idea**, very important in practice
- During generation, there's **no need to re-compute** Keys and Values for **previous tokens**
  - Instead, store them in a **cache** for re-use
  - Only KVs for the **newest token** are computed at each step

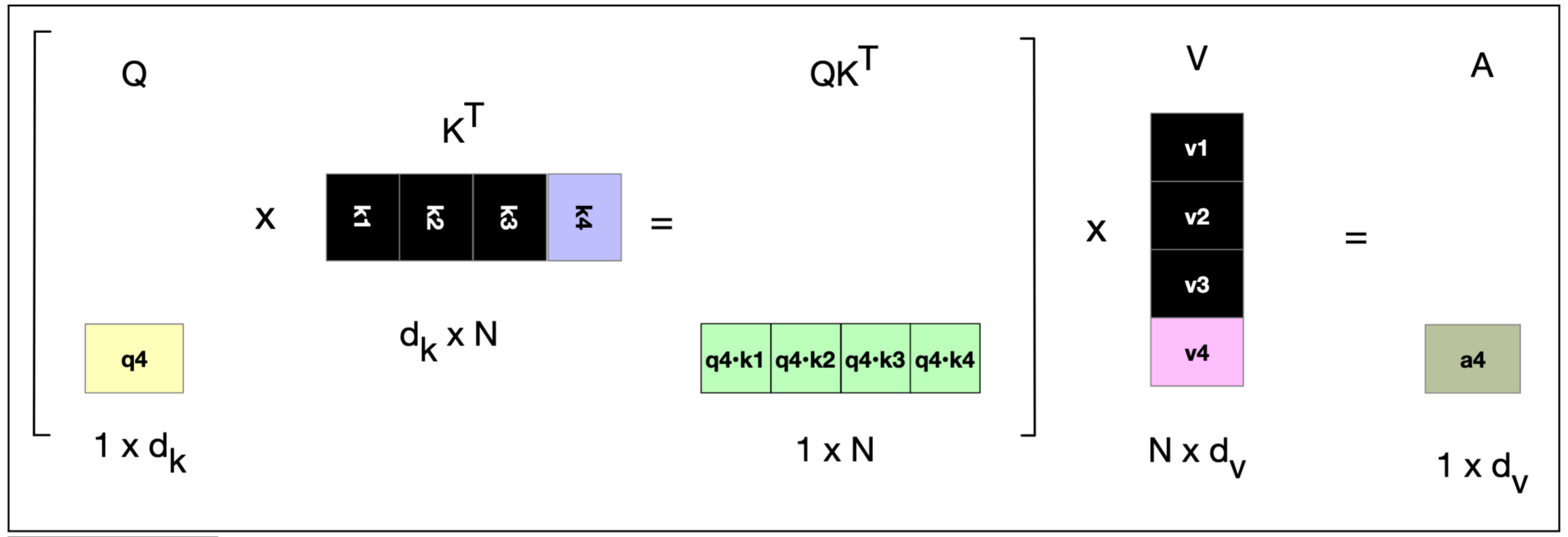


Figure 8.17 Parts of the attention computation (extracted from Fig. 8.10) showing, in black, the vectors that can be stored in the cache rather than recomputed when computing the attention score for the 4th token.

Mixed/Hybrid Architectures

- Encoder-decoder: a general architecture
  - In principle, any model of the right type can be encoder and/or decoder
- "The Best of Both Worlds" for NMT
  - Transformer encoder + RNN decoder
- Google Translate (at one point)
  - "Transformer models have been demonstrated to be generally more effective at machine translation than RNN models, but our work suggested that most of these quality gains were from the transformer *encoder*, and that the transformer *decoder* was not significantly better than the RNN decoder. Since the RNN decoder is much faster at inference time, we applied a variety of optimizations before coupling it with the transformer encoder. The resulting hybrid models are higher-quality, more stable in training, and exhibit lower latency."

Encoder	Decoder	En→Fr Test BLEU
Trans. Big	Trans. Big	40.73 ± 0.19
RNMT+	RNMT+	41.00 ± 0.05
Trans. Big	RNMT+	$\mathbf{41.12} \pm \mathbf{0.16}$
RNMT+	Trans. Big	39.92 ± 0.21

Subword Tokenization

OOV and Vocab Size

OOV and Vocab Size

- Word-level models
  - Tokenize training data
  - Build vocabulary
  - Learn representations

OOV and Vocab Size

- Word-level models
  - Tokenize training data
  - Build vocabulary
  - Learn representations
- Two problems
  - **Cannot generalize** at test time to **OOV (out of vocab) words**
    - (various subtleties, tricks, etc, but generally true)
  - Larger training data —> larger vocabulary
    - Its own problems, e.g. **very expensive softmax over vocab** in decoders
    - (Or put a cap on vocab size, but then miss lower-frequency words entirely)

Finer Representation Levels

Finer Representation Levels

- One solution: **character-level models**
  - Pros:
    - **Small vocabulary size**
    - **No (or very little) OOV**
  - Cons:
    - Much **harder learning problems**; need to learn everything about words, on top of phrases, sentences, etc.

Finer Representation Levels

- One solution: **character-level models**
  - Pros:
    - **Small vocabulary size**
    - **No (or very little) OOV**
  - Cons:
    - Much **harder learning problems**; need to learn everything about words, on top of phrases, sentences, etc.
- In-between solution: **sub-word** tokenization
  - Split words into pieces, but don't go all the way down to character level
  - Many methods: WordPiece, BytePair Encoding (BPE), …

WordPiece Embeddings

WordPiece Embeddings

- Another solution to OOV problem, from NMT context (see Wu et al 2016)

WordPiece Embeddings

- Another solution to OOV problem, from NMT context (see Wu et al 2016)
- Main idea:
  - **Fix vocabulary size** |V| in advance (e.g. 30k for BERT)
  - **Choose |V| wordpieces (subwords)** such that total **number of wordpieces in the corpus is minimized**

WordPiece Embeddings

- Another solution to OOV problem, from NMT context (see Wu et al 2016)
- Main idea:
  - **Fix vocabulary size** |V| in advance (e.g. 30k for BERT)
  - **Choose |V| wordpieces (subwords)** such that total **number of wordpieces in the corpus is minimized**
- Frequent words aren't split, but rarer ones are, e.g.:

WordPiece Embeddings

- Another solution to OOV problem, from NMT context (see Wu et al 2016)
- Main idea:
    - **Fix vocabulary size** |V| in advance (e.g. 30k for BERT)
    - **Choose |V| wordpieces (subwords)** such that total **number of wordpieces in the corpus is minimized**
- Frequent words aren't split, but rarer ones are, e.g.:
- "Backpropagation was confusing at first, but now we grok it."
    - ["Back", "##prop", "##ag", "##ation", "was", "confusing", "at", "first", ",", "but", "now", "we", "gro", "##k", "it", "."]