From Attention to Transformers

Antoine Bosselut

EPFL

nlp lab

Announcements

- **Assignment #1** released last Friday, March 10th
  - **Due Friday, March 24, 2023 @ 11:59 PM**
  - **Please post questions to Ed Discussion Board, so others can benefit from your queries!**

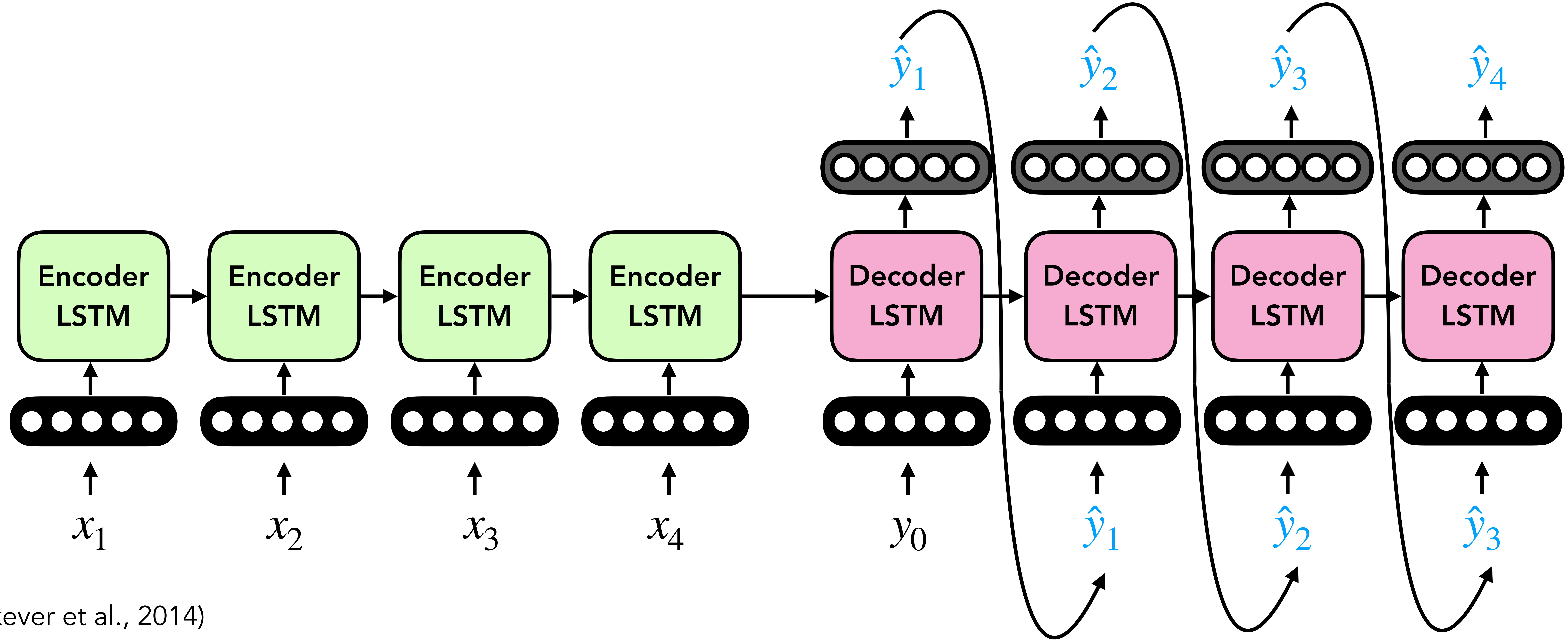
Today's Outline

- **Quick Recap:** Attention
- **Supercharging Attention:** Transformers
- **Generating sequences:** Decoding from sequence to sequence models

Some slides adapted from Danqi Chen, Mohit Iyyer, Emma Strubell

Encoder-Decoder Models

- Encode a sequence fully with one model (**encoder**) and use its representation to seed a second model that decodes another sequence (**decoder**)

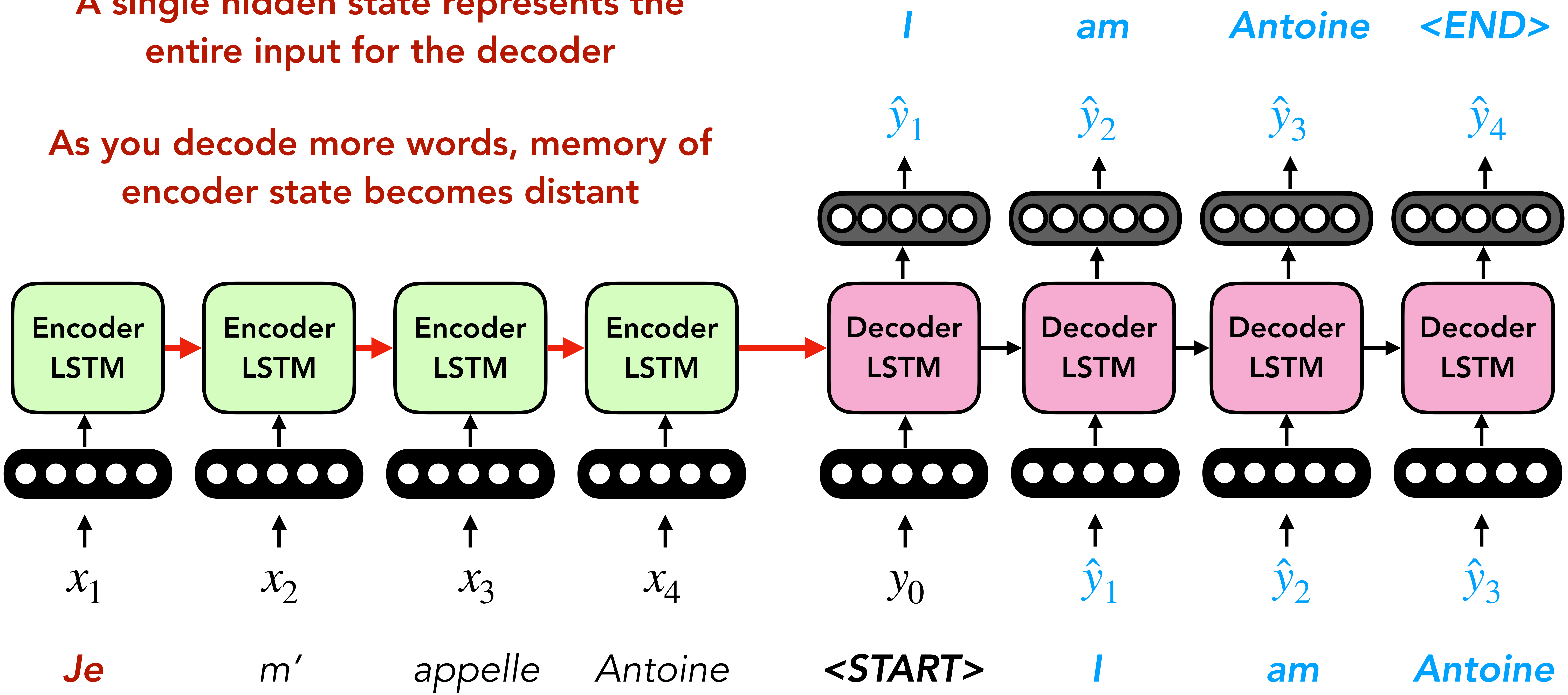


(Sutskever et al., 2014)

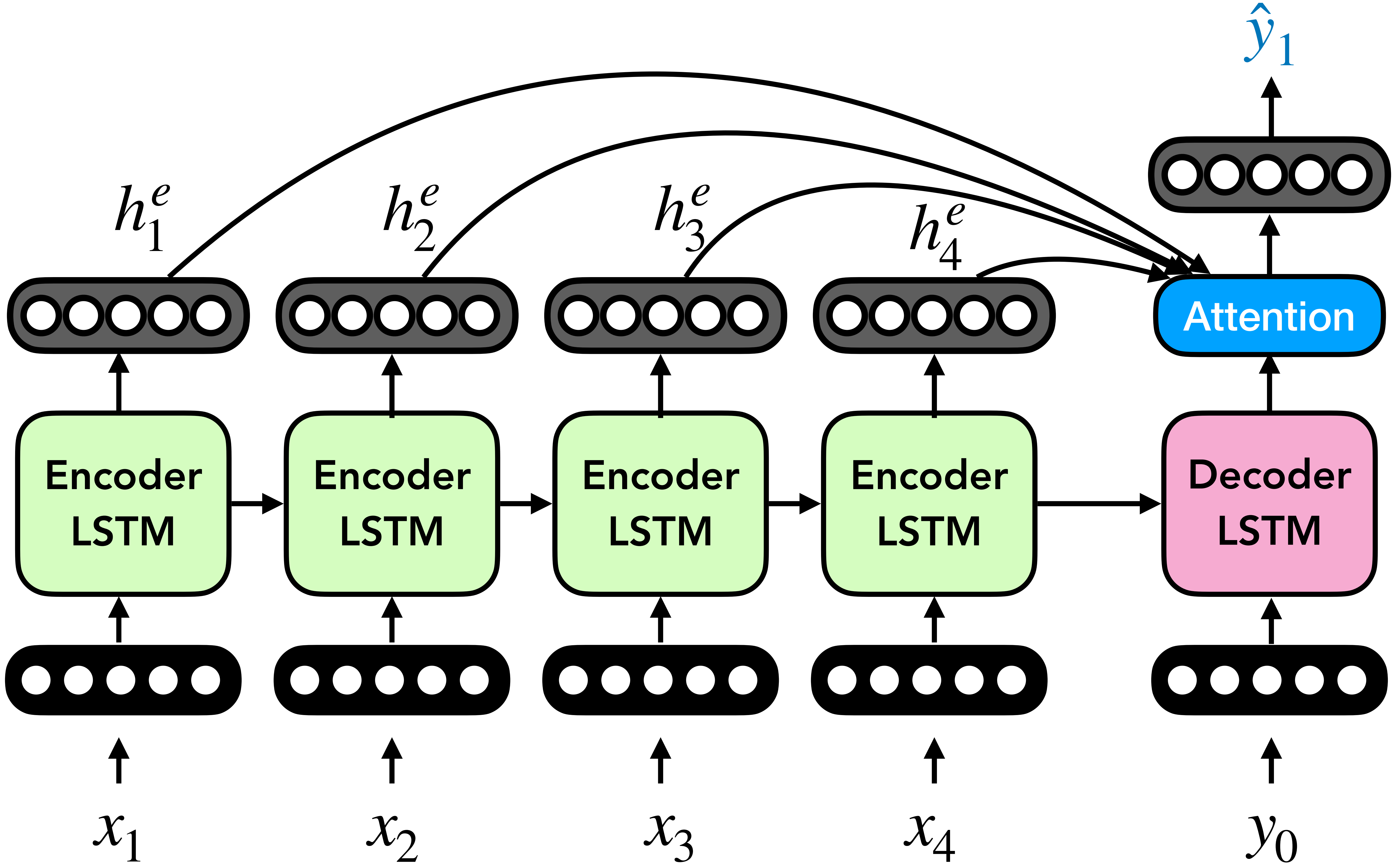
Toy Example

A single hidden state represents the entire input for the decoder

As you decode more words, memory of encoder state becomes distant



Attentive Encoder-Decoder Models



- **Recall:** Attention reduces this temporal bottleneck!
- **Intuition**: focus on different parts of the input at each time step
- **Idea:** Use the output of the Decoder LSTM to compute an **attention** (i.e., a mixture) over all the h_t^e outputs of the encoder LSTM

(Bahdanau et al., 2015)

Attention Function

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")

h_t^e = encoder output hidden states

Also known as a "keys"

h_t^d = decoder output hidden state

Also known as a "query"

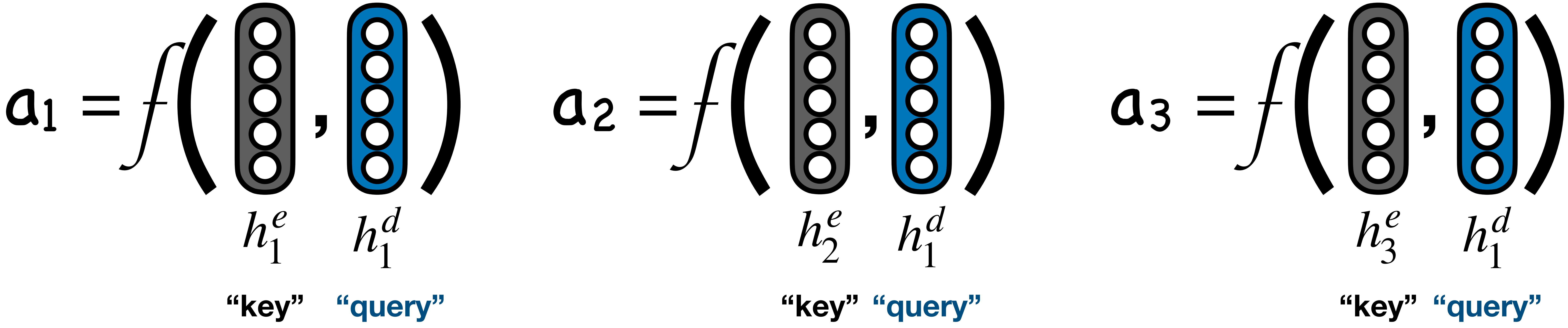
- **We have a single query vector for multiple key vectors**

Attention Function

Attention Function	Formula
Multiplicative	$a = h^e \mathbf{W} h^d$
Linear	$a = v^T \phi(\mathbf{W}[h^e; h^d])$
Scaled Dot Product	$a = \frac{(\mathbf{W}h^e)^T(\mathbf{U}h^d)}{\sqrt{d}}$

Attention Function

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")

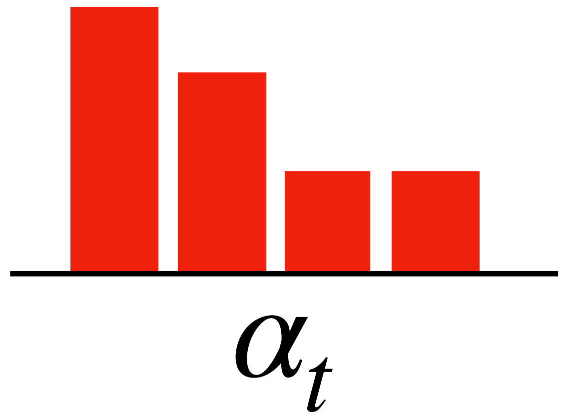


- **Convert** pairwise similarity scores to probability **distribution** (using softmax!) over encoder hidden states and compute weighted average:

Softmax!

$$\boxed{\alpha_t = \frac{e^{a_t}}{\sum_j e^{a_j}}}$$

$\rightarrow$

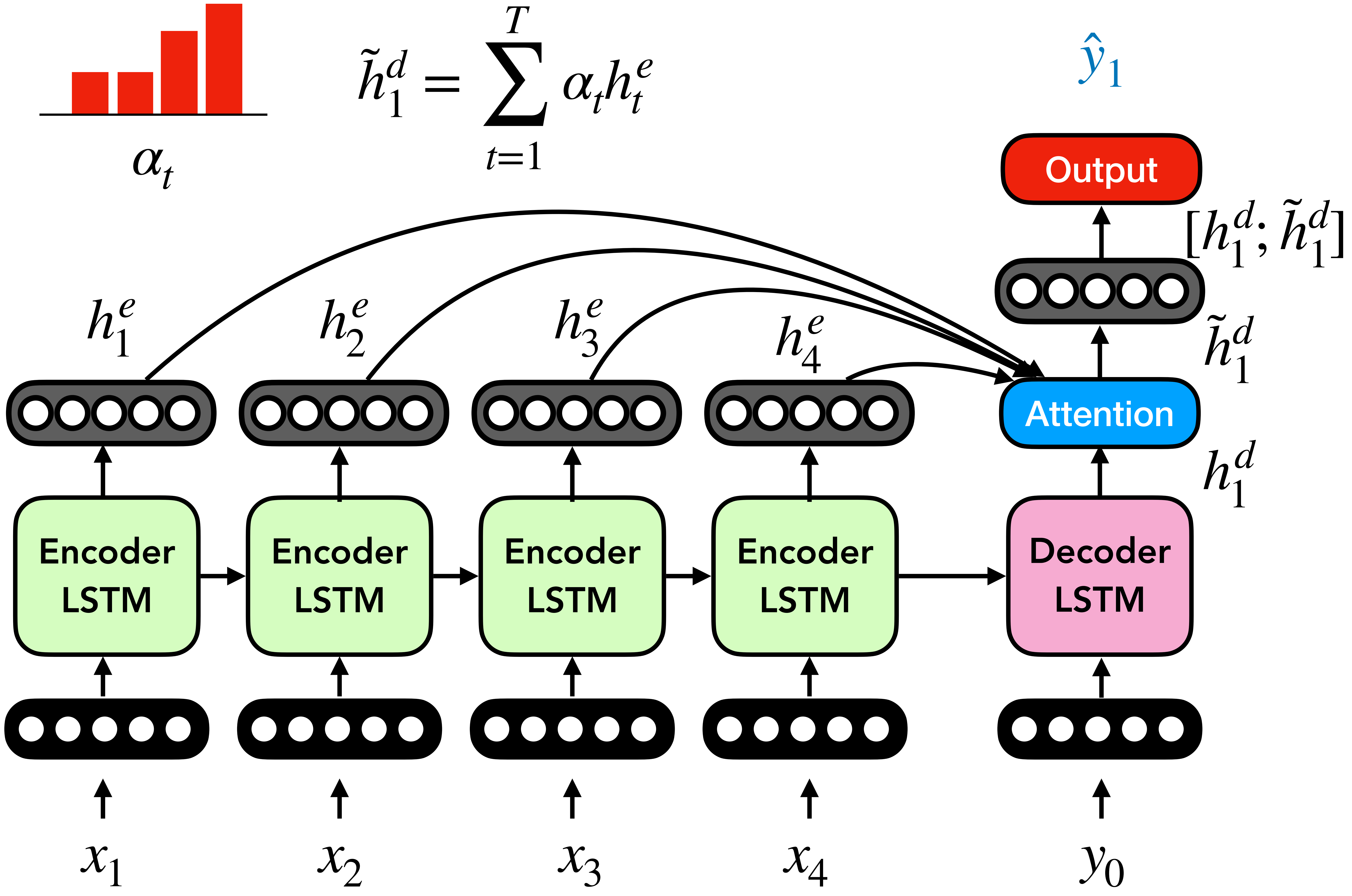


$\rightarrow$

$$\tilde{h}_1^d = \sum_{t=1}^{T} \alpha_t h_t^e$$

Here h_t^e is known as the "value"

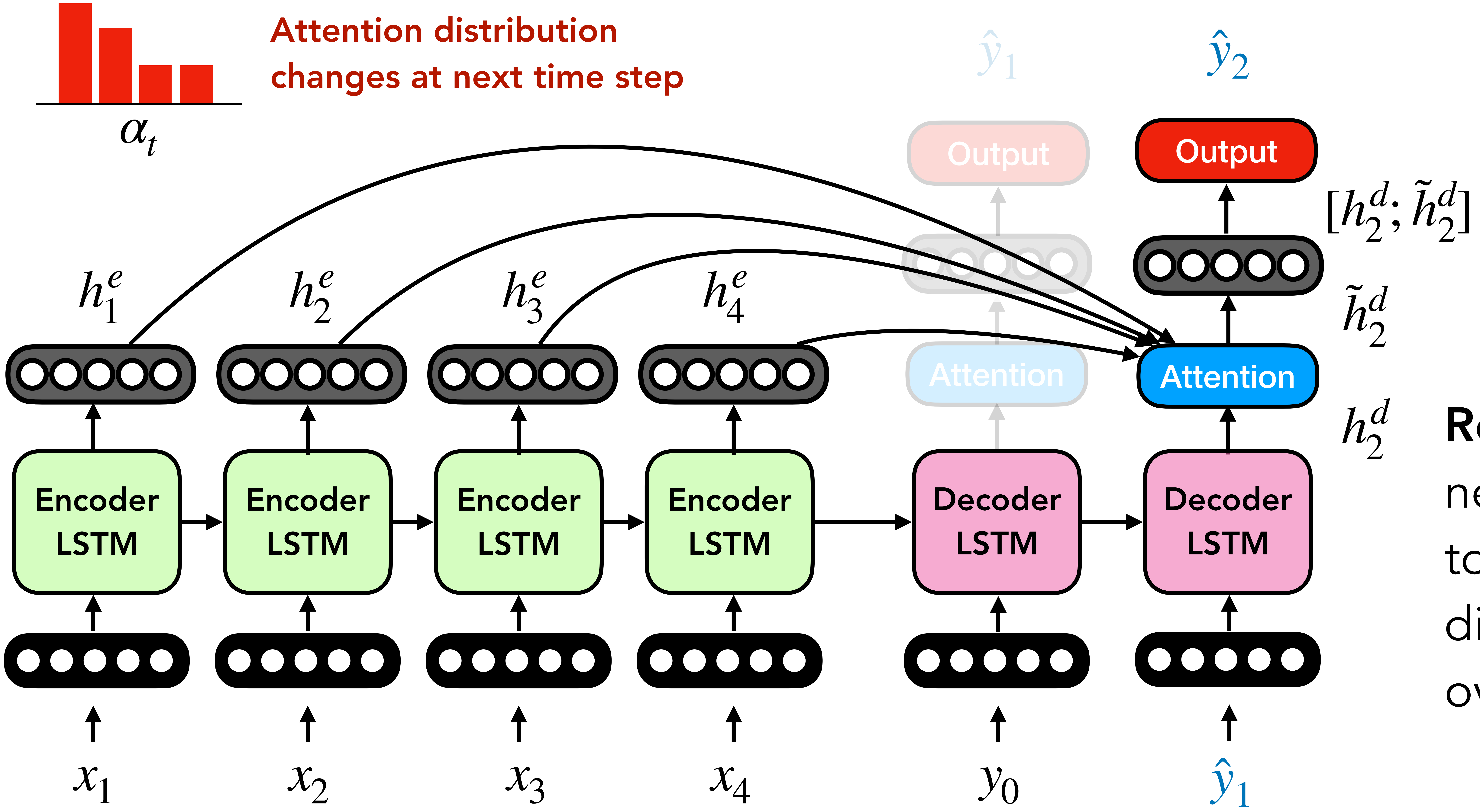
Attentive Encoder-Decoder Models



Intuition: $\tilde{h}_1^d$ contains information about encoder hidden states that got **high** attention from the decoder

(Bahdanau et al., 2015)

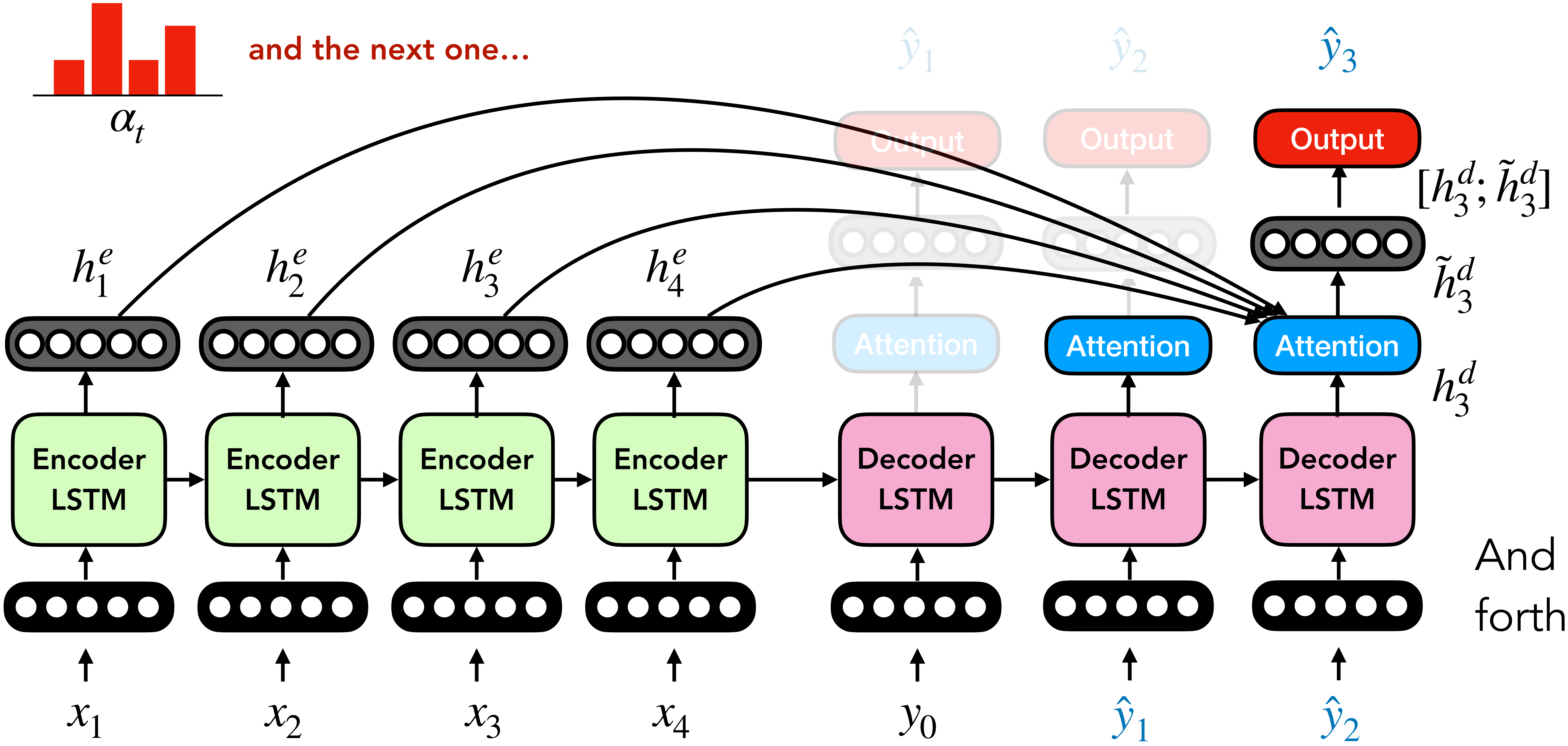
Attentive Encoder-Decoder Models



Repeat in next time step to get new distribution over states

(Bahdanau et al., 2015)

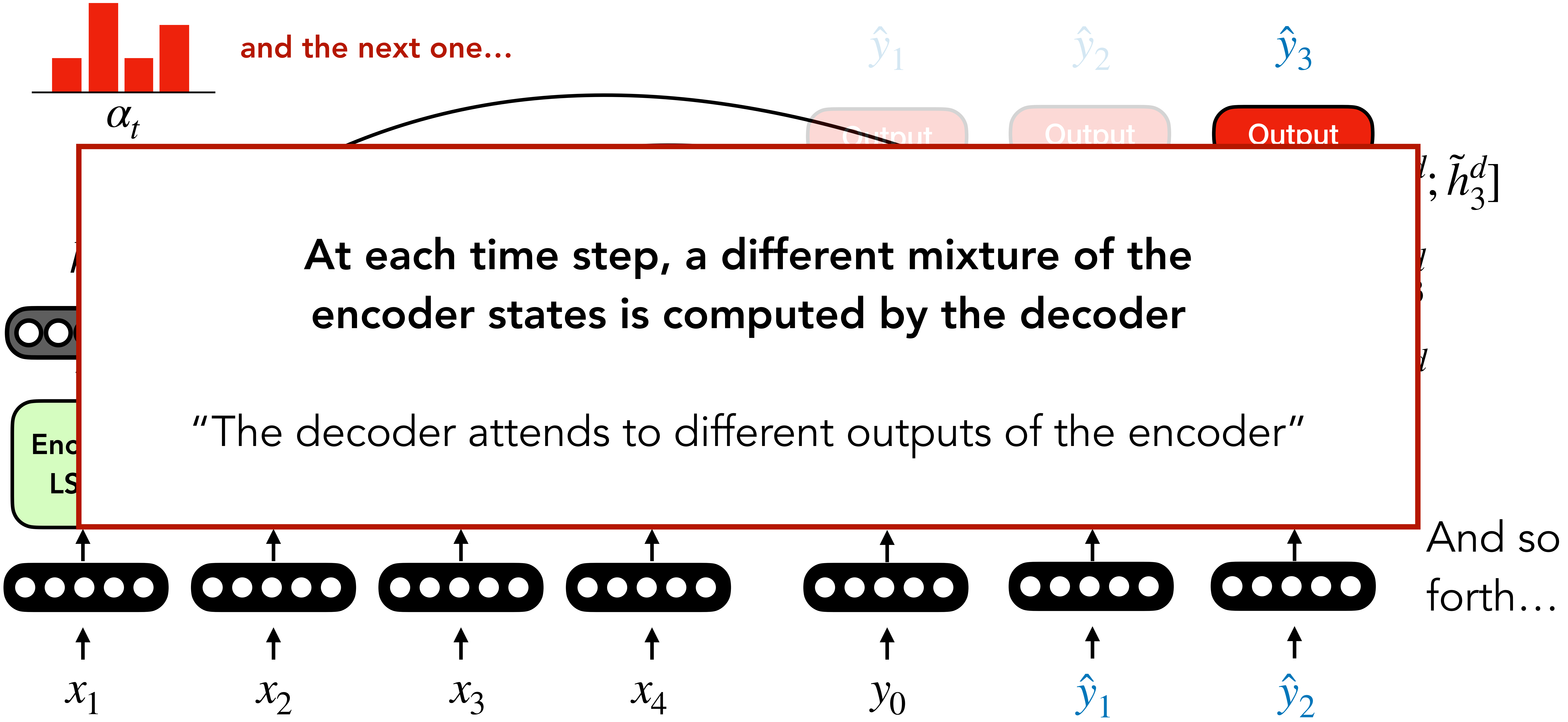
Attentive Encoder-Decoder Models



And so forth…

(Bahdanau et al., 2015)

Attentive Encoder-Decoder Models



At each time step, a different mixture of the encoder states is computed by the decoder

"The decoder attends to different outputs of the encoder"

(Bahdanau et al., 2015)

Attention Recap

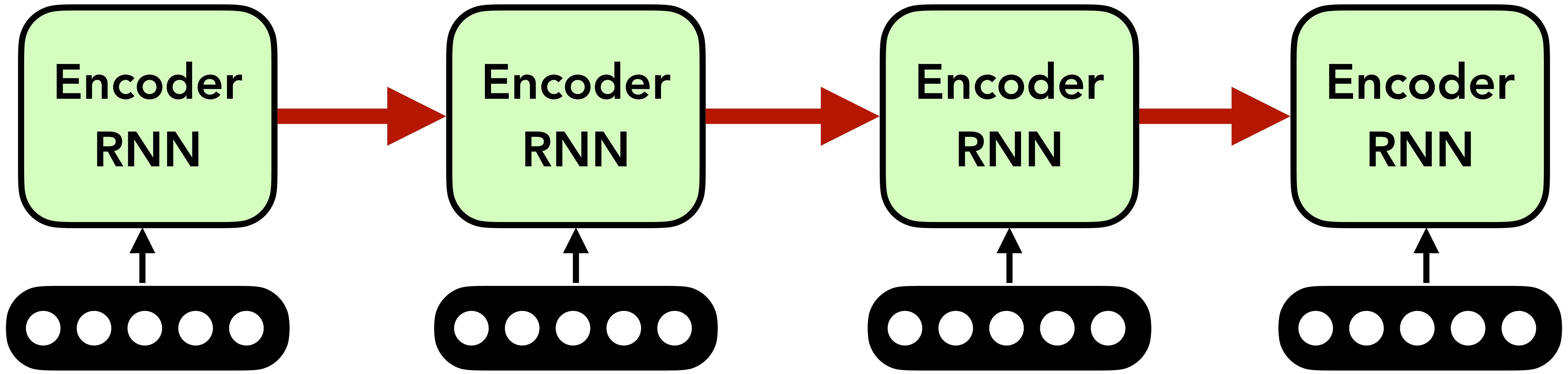
- **Main Idea:** Decoder computes a weighted sum of encoder outputs
  - Compute pairwise score between each encoder hidden state and initial decoder hidden state
- Many possible functions for computing scores (dot product, bilinear, etc.)
- **Temporal Bottleneck Fixed! Direct link** between decoder and encoder states
  - Helps with vanishing gradients and modelling long-term dependencies!
- Attention is **agnostic** to the type of RNN used in the encoder and decoder!

Question

Do any other inefficiencies remain in our sequence to sequence pipelines?

Encoder is still Recurrent

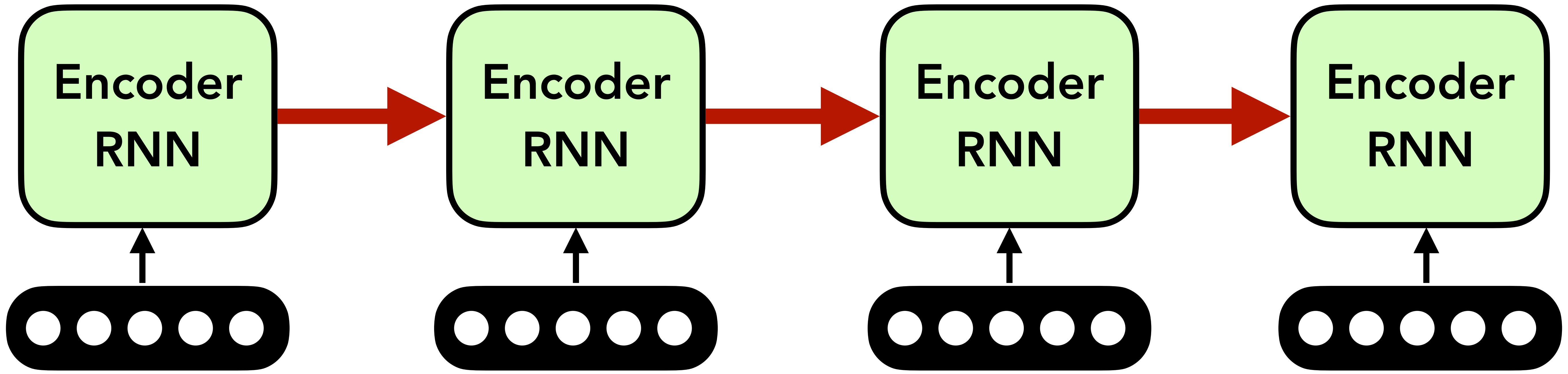
- **Encoder:** Recurrent functions can't be parallelized because previous state needs to be computed to encode next one



- **Problem: Encoder hidden states must still be computed in series**

Encoder is still Recurrent

- **Encoder:** Recurrent functions can't be parallelized because previous state needs to be computed to encode next one



- **Problem: Encoder hidden states must still be computed in series**

Who can think of a task where this might be a problem?

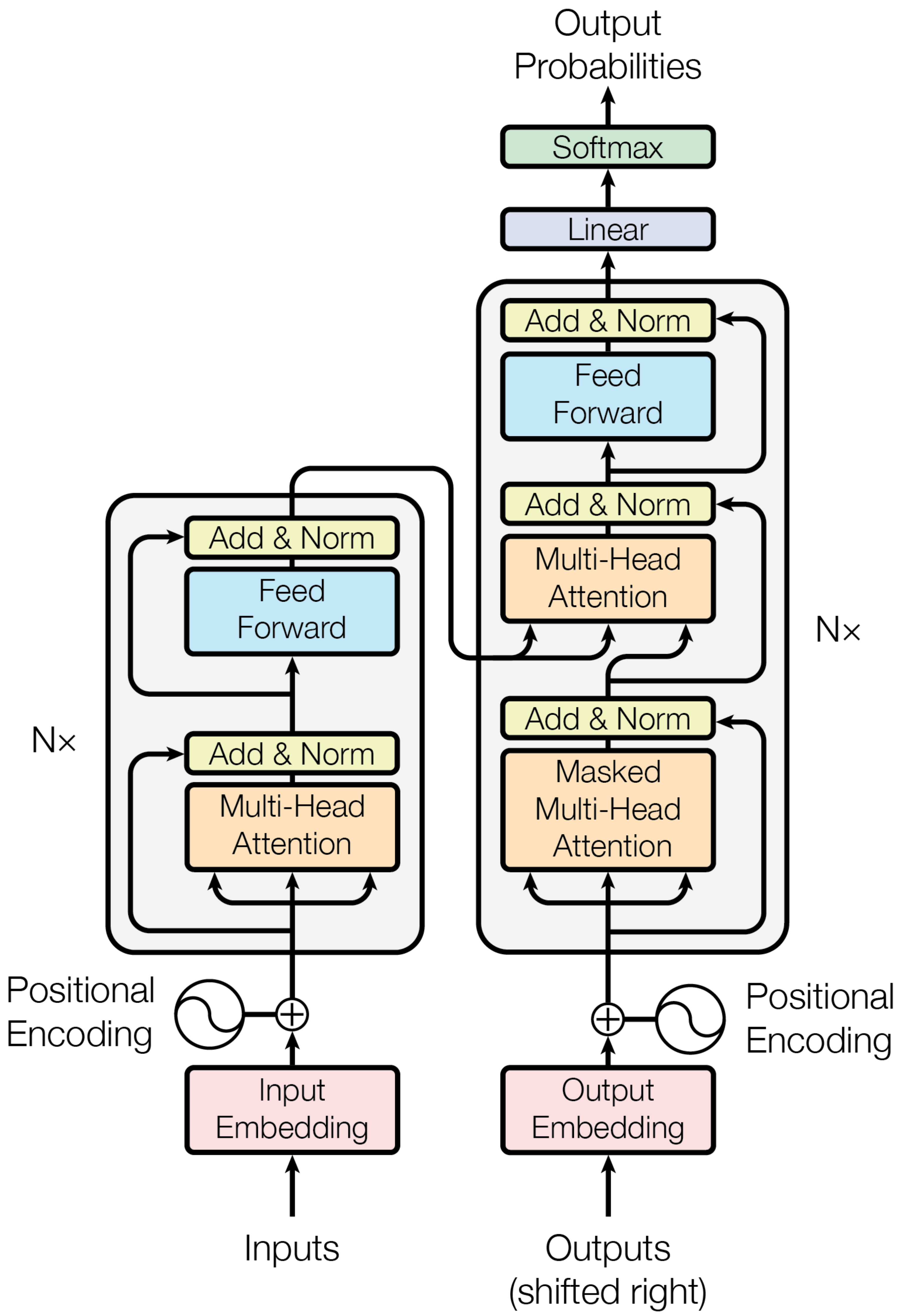
Solution:

Transformers!

Full Transformer

- Made up of encoder and decoder
- Both encoder and decoder made up of multiple cascaded transformer blocks
  - slightly different architecture in encoder and decoder transformer blocks
- Blocks generally made up **multi-headed attention** layers (self-attention) and **feedforward** layers
- No recurrent computations!

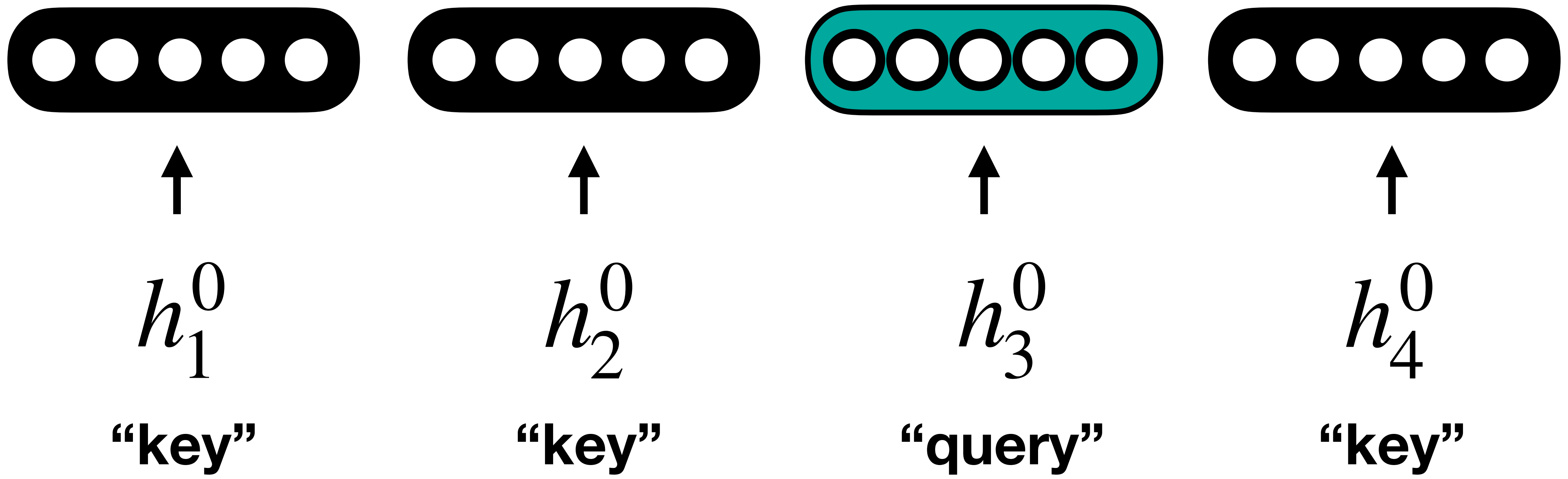
Encode sequences with self-attention



(Vaswani et al., 2017)

Self-Attention Toy Example

h_t^ℓ = encoder hidden state at time step t at layer ℓ

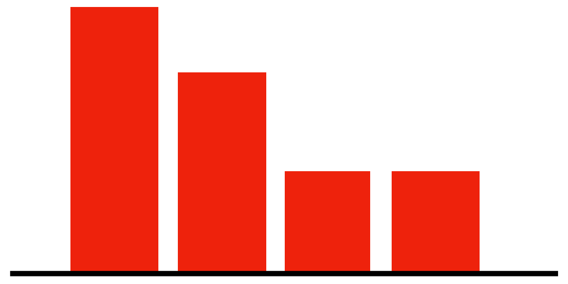


Recap: Attention with RNNs

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")

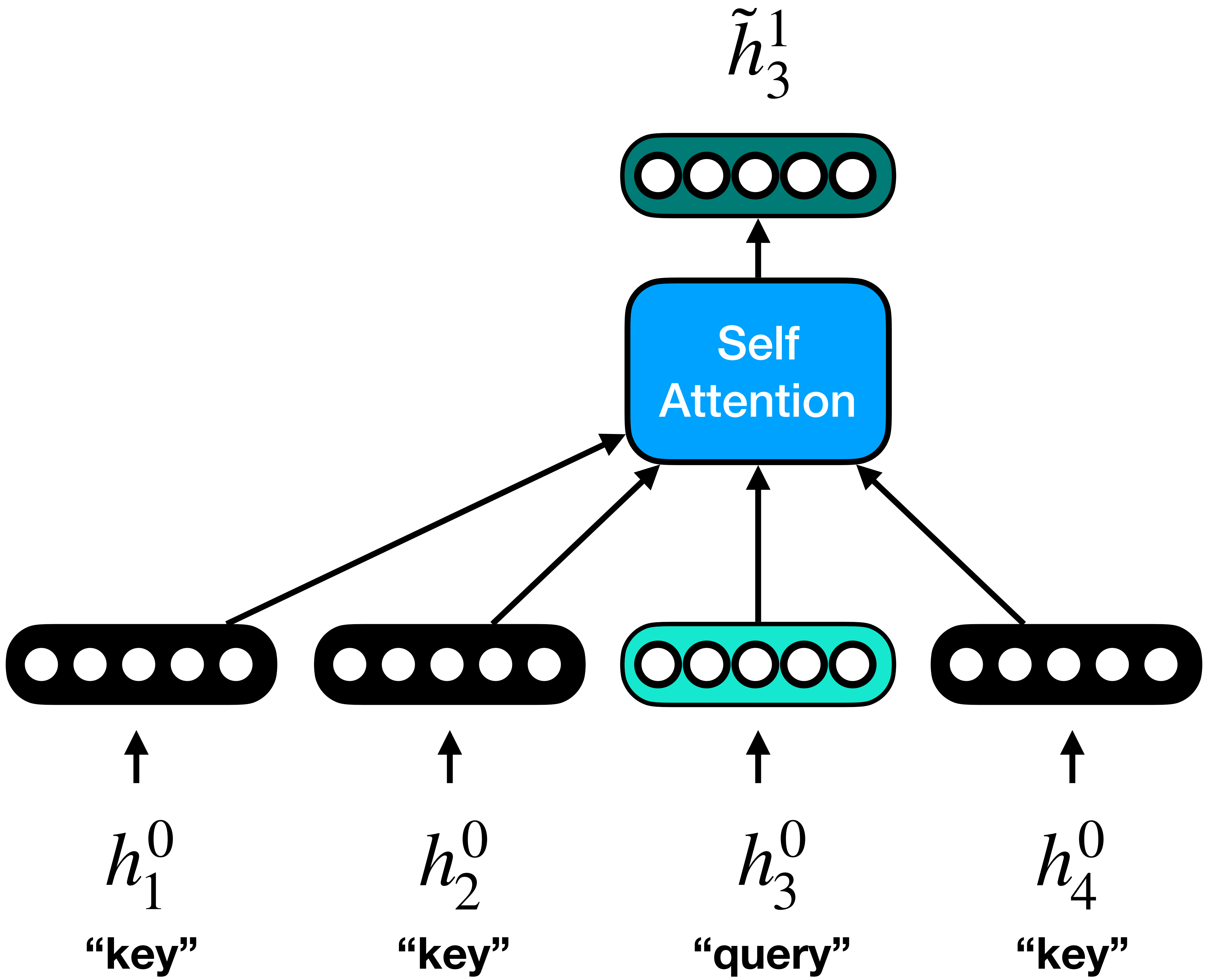
$$a_1 = f\Big(\underbrace{h_1^e}_{\text{"key"}}, \underbrace{h_1^d}_{\text{"query"}}\Big) \qquad a_2 = f\Big(\underbrace{h_2^e}_{\text{"key"}}, \underbrace{h_1^d}_{\text{"query"}}\Big) \qquad a_3 = f\Big(\underbrace{h_3^e}_{\text{"key"}}, \underbrace{h_1^d}_{\text{"query"}}\Big)$$

- **Convert** pairwise similarity scores to probability **distribution** (using softmax!) over encoder hidden states and compute weighted average:

Softmax! $\boxed{\alpha_t = \frac{e^{a_t}}{\sum_j e^{a_j}}}$ $\rightarrow$  $\rightarrow$ $\tilde{h}_1^d = \sum_{t=1}^{T} \alpha_t h_t^e$

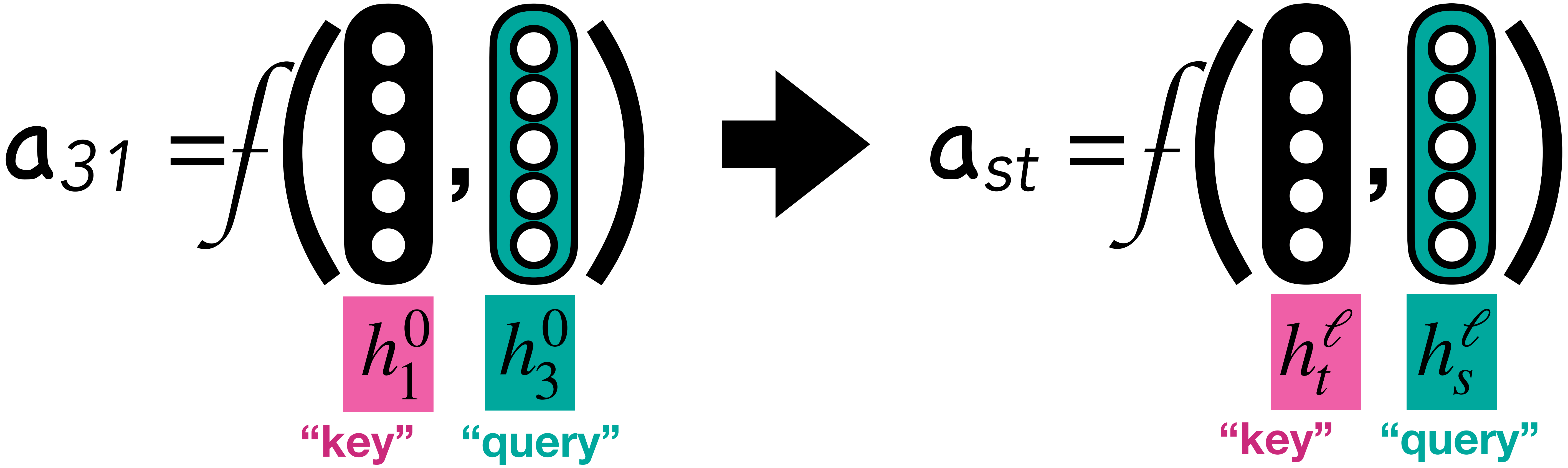
Here h_t^e is known as the "value"

Self-Attention Toy Example



Self-Attention Toy Example

h_t^ℓ = encoder hidden state at time step t at layer ℓ



$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T(\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Compute pairwise scores

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$

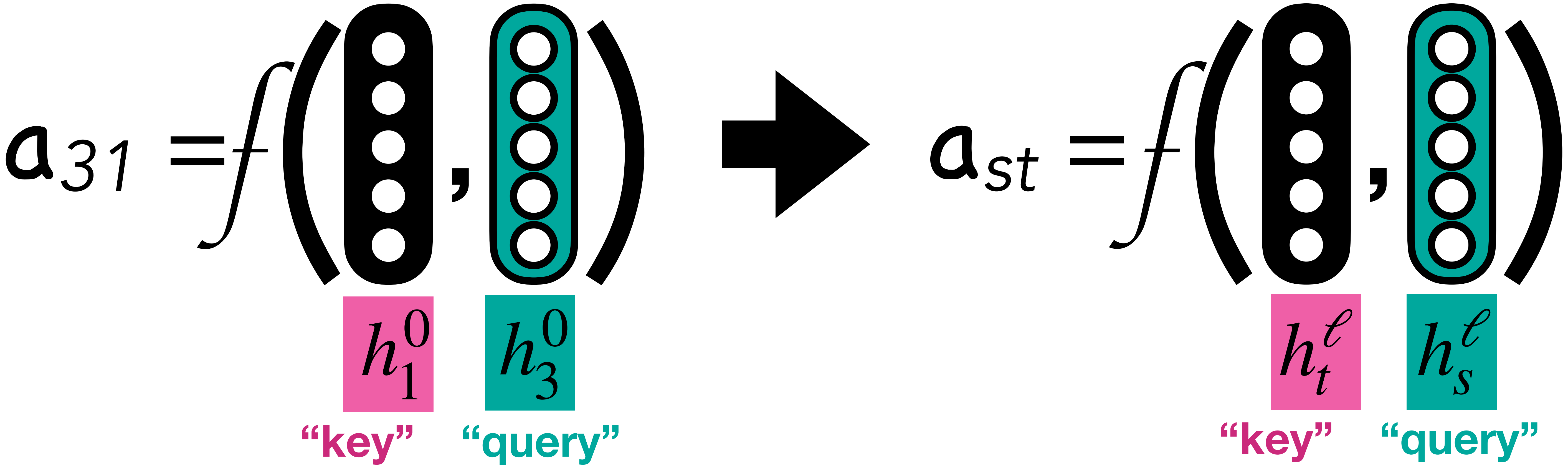
Get attention distribution

$$\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st}(\mathbf{W}^V h_t^\ell)$$

Attend to values to get weighted sum

Self-Attention Toy Example

h_t^ℓ = encoder hidden state at time step t at layer ℓ



{1, …, *t*, …, T} includes *s*!

$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T (\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Compute pairwise scores

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$

Get attention distribution

$$\boxed{\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st} \left(\mathbf{W}^V h_t^\ell \right)}$$

Attend to values to get weighted sum

Self-attention!

Self-Attention Toy Example

Compute pairwise scores

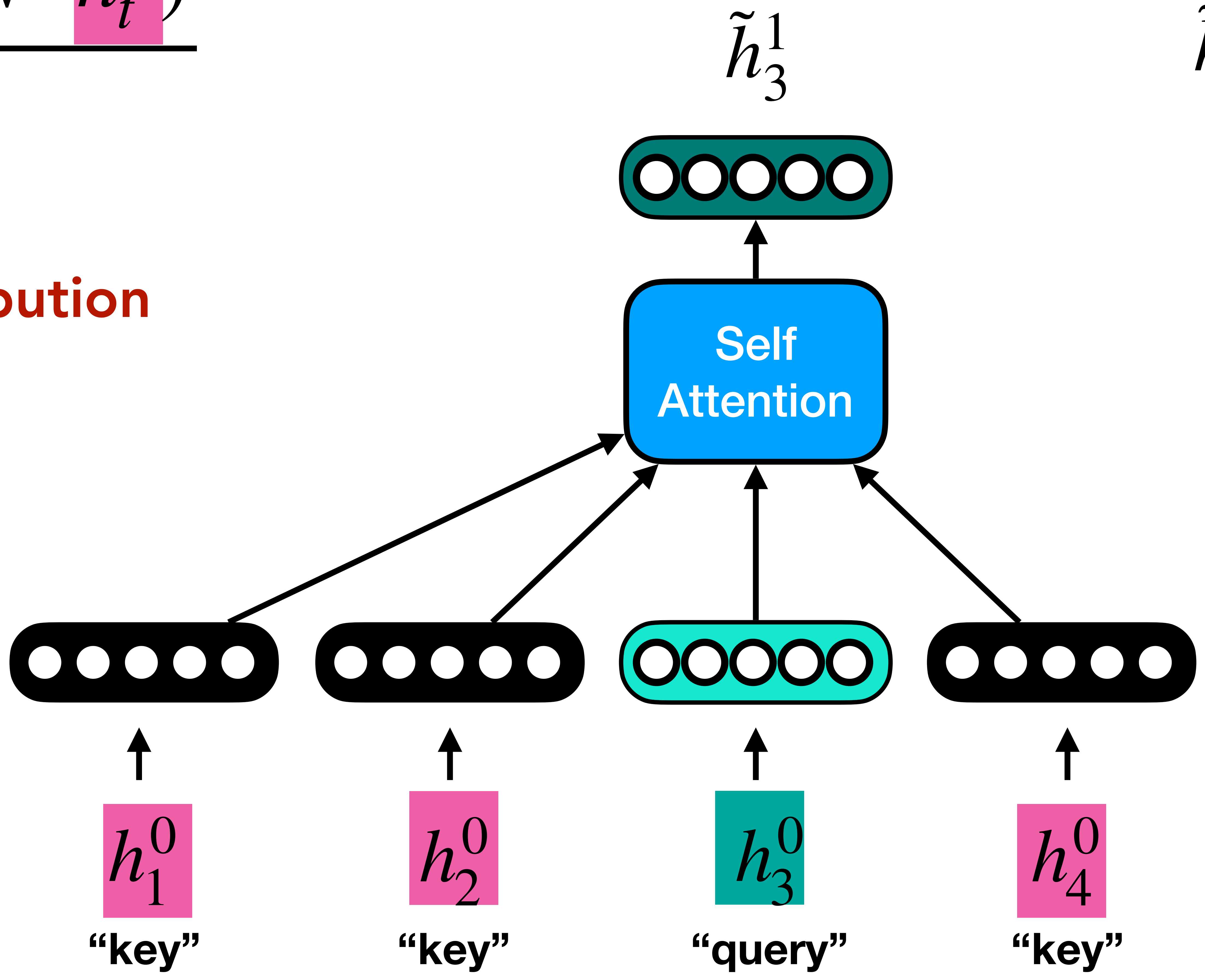
$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T (\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Get attention distribution

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$

Attend to values to get weighted sum

$$\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st} (\mathbf{W}^V h_t^\ell)$$



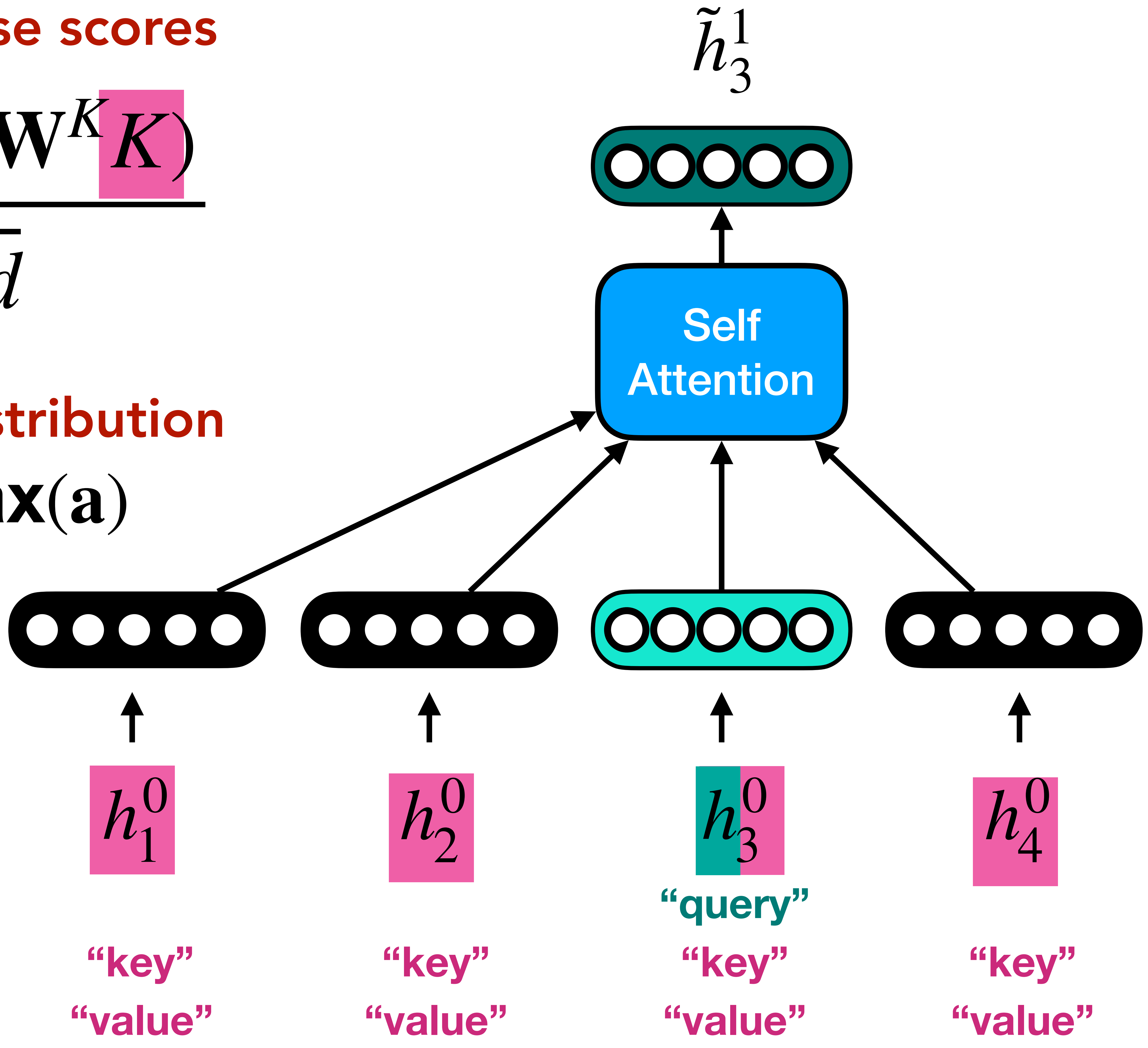
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \textbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha (V \mathbf{W}^V)$$

"query" $q = h_s^\ell$

"values"

$$K = V = \{h_t^\ell\}_{t=0}^T$$

"keys"

For each attention computation, every element is a key and value, and one element is a query

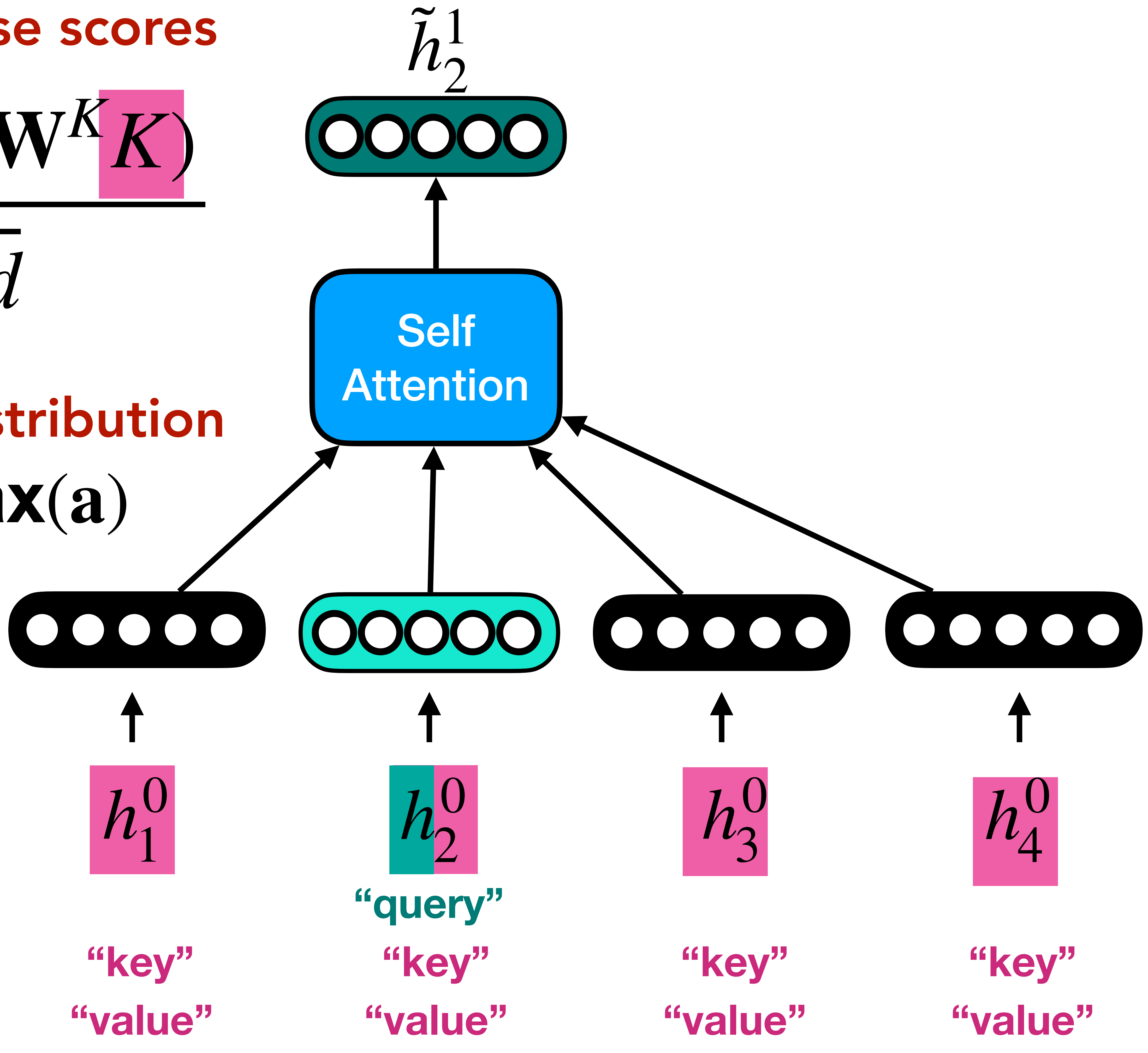
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \textbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha(V\mathbf{W}^V)$$

"query" $q = h_s^\ell$

"values"

$$K = V = \{h_t^\ell\}_{t=0}^T$$

"keys"

For each attention computation, every element is a key and value, and one element is a query

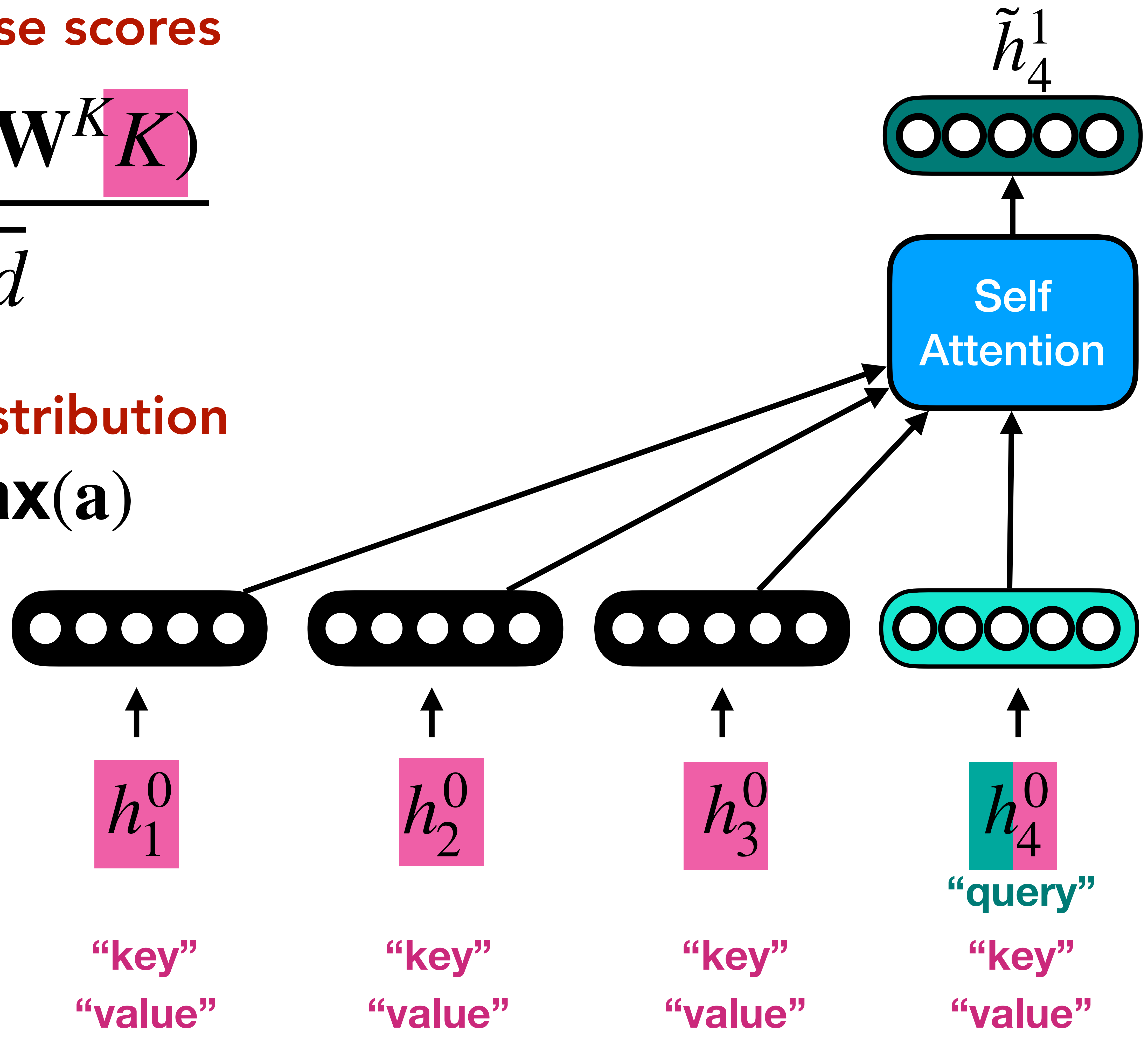
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \mathbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha (V \mathbf{W}^V)$$

"query" $q = h_s^\ell$

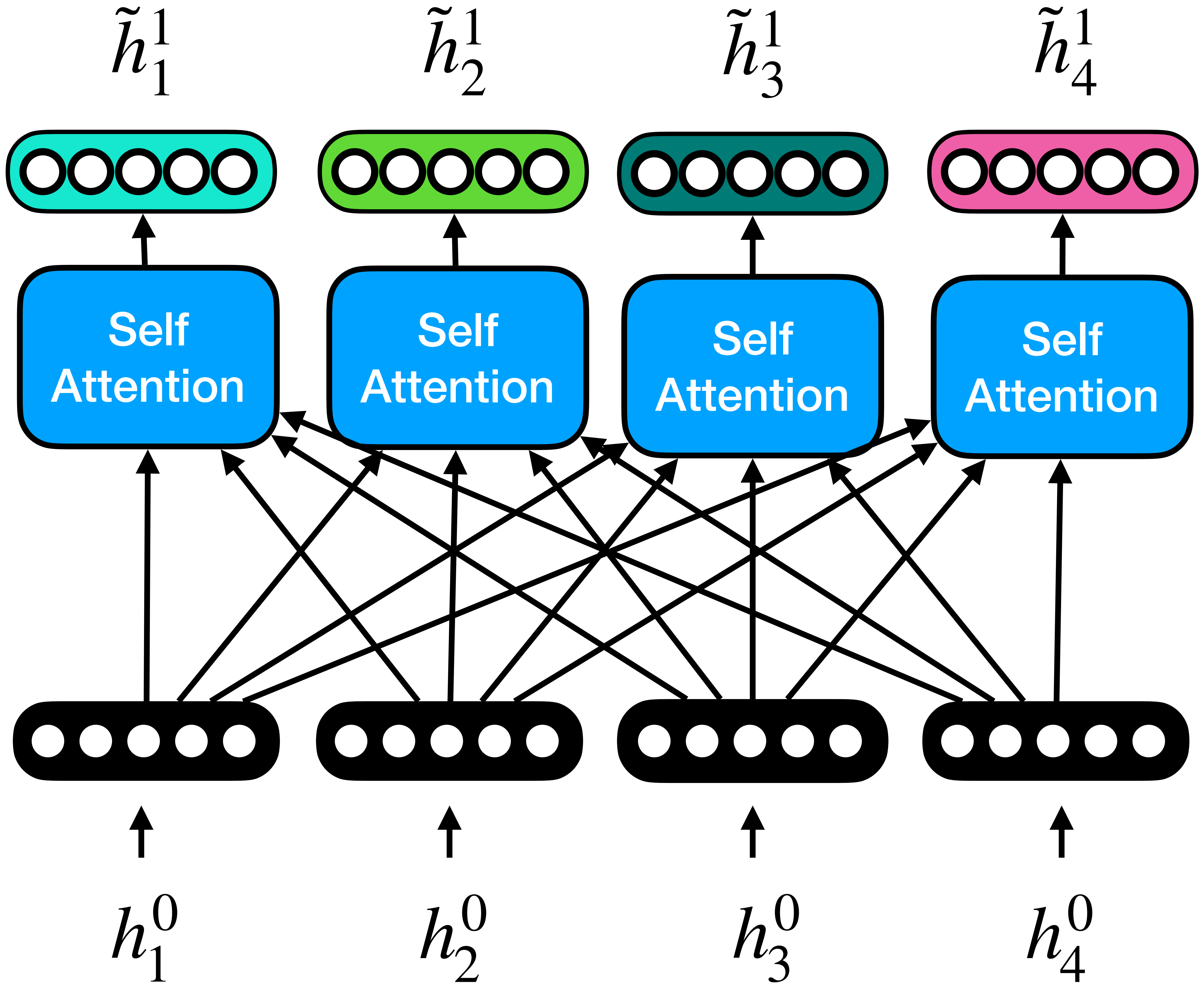
"values"

$$K = V = \{h_t^\ell\}_{t=0}^T$$

"keys"

For each attention computation, every element is a key and value, and one element is a query

Self-Attention Toy Example



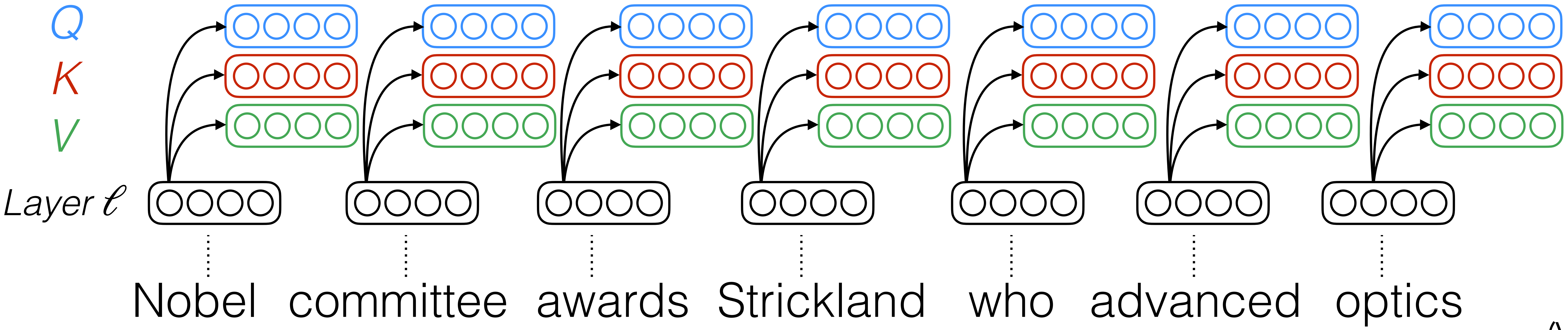
$$\tilde{h}_1^1 = \textbf{Attention}\left(h_1^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

$$\tilde{h}_1^2 = \textbf{Attention}\left(h_2^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

$$\tilde{h}_1^3 = \textbf{Attention}\left(h_3^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

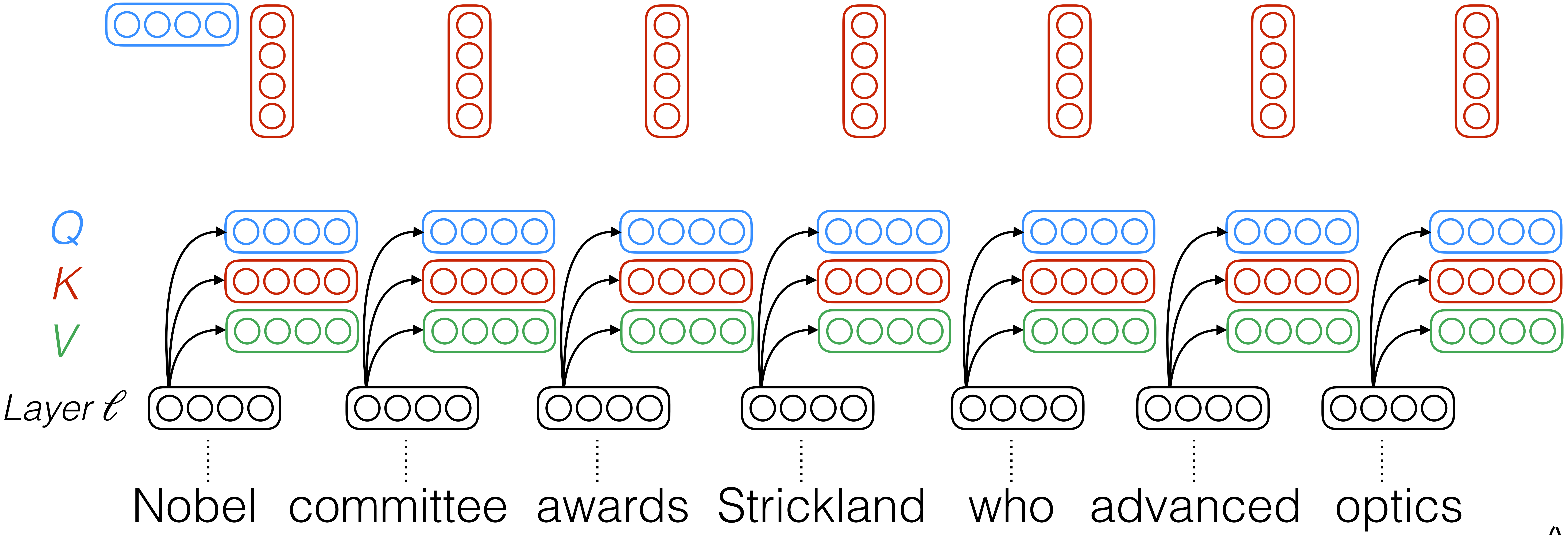
$$\tilde{h}_1^4 = \textbf{Attention}\left(h_4^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

Self-attention (in encoder)



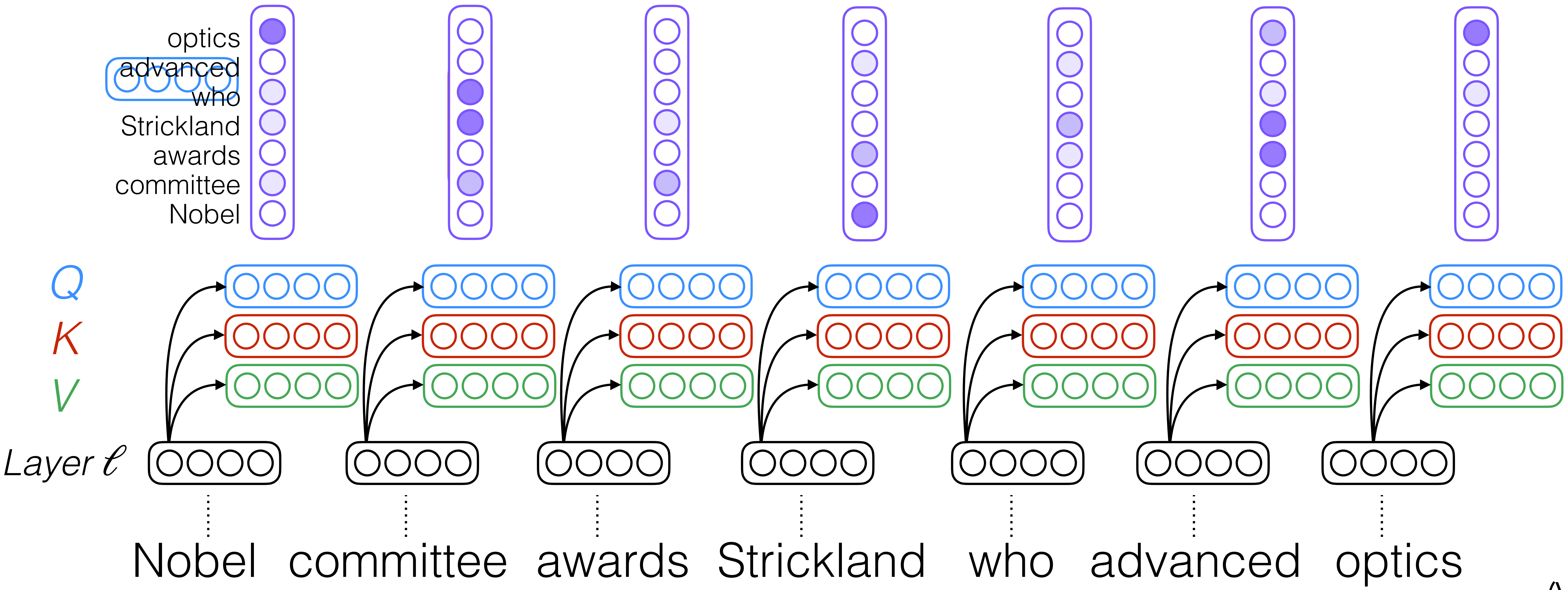
(Vaswani et al., 2017)

Self-attention (in encoder)



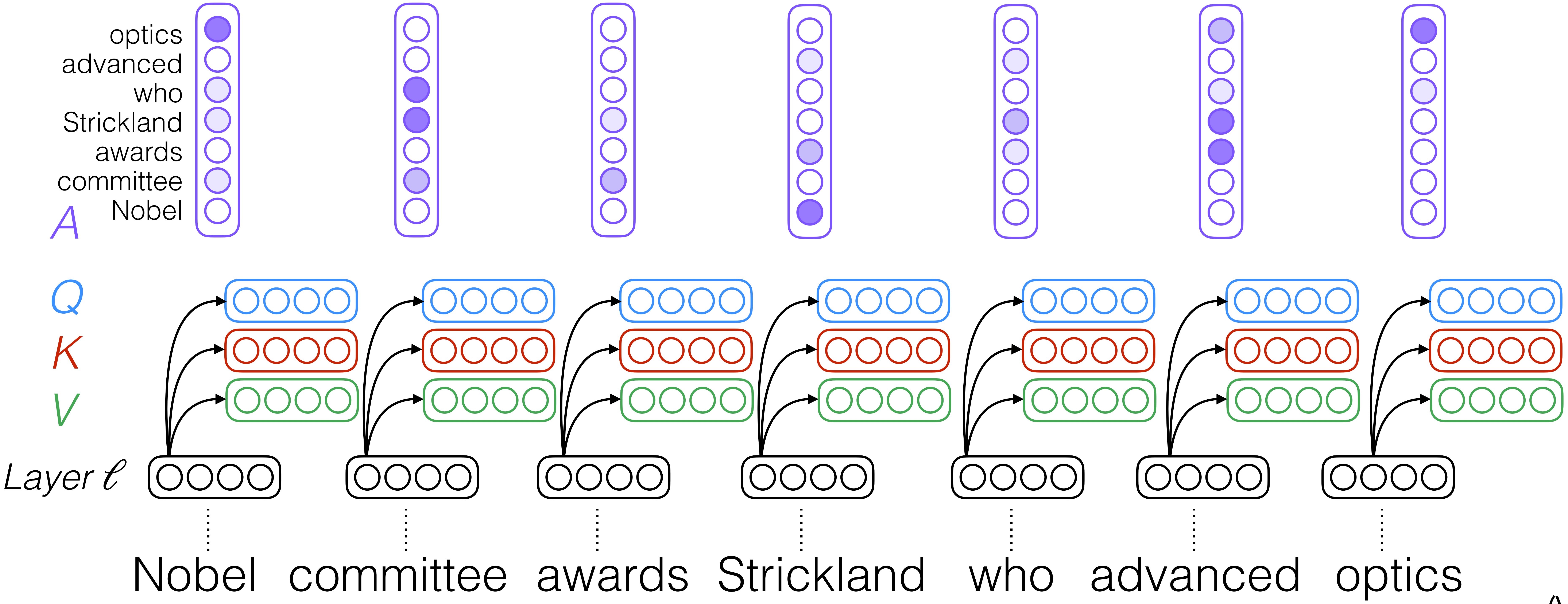
(Vaswani et al., 2017)

Self-attention (in encoder)



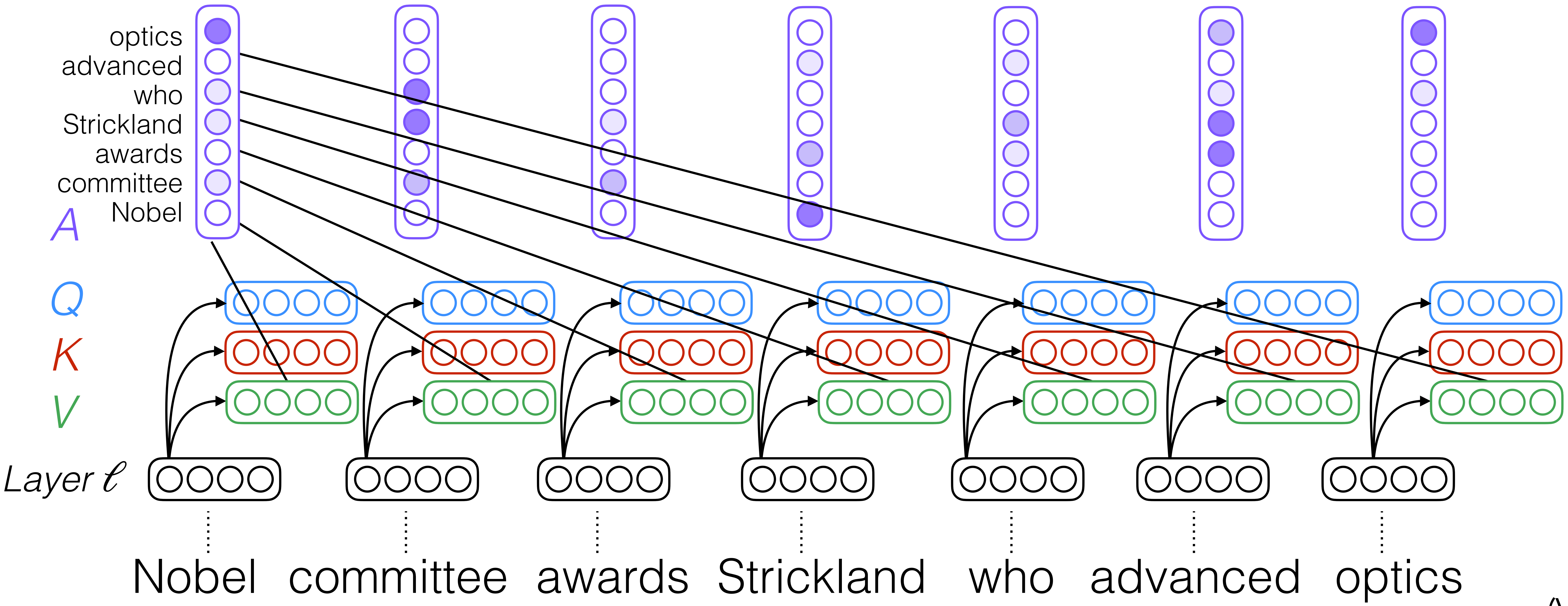
(Vaswani et al., 2017)

Self-attention (in encoder)



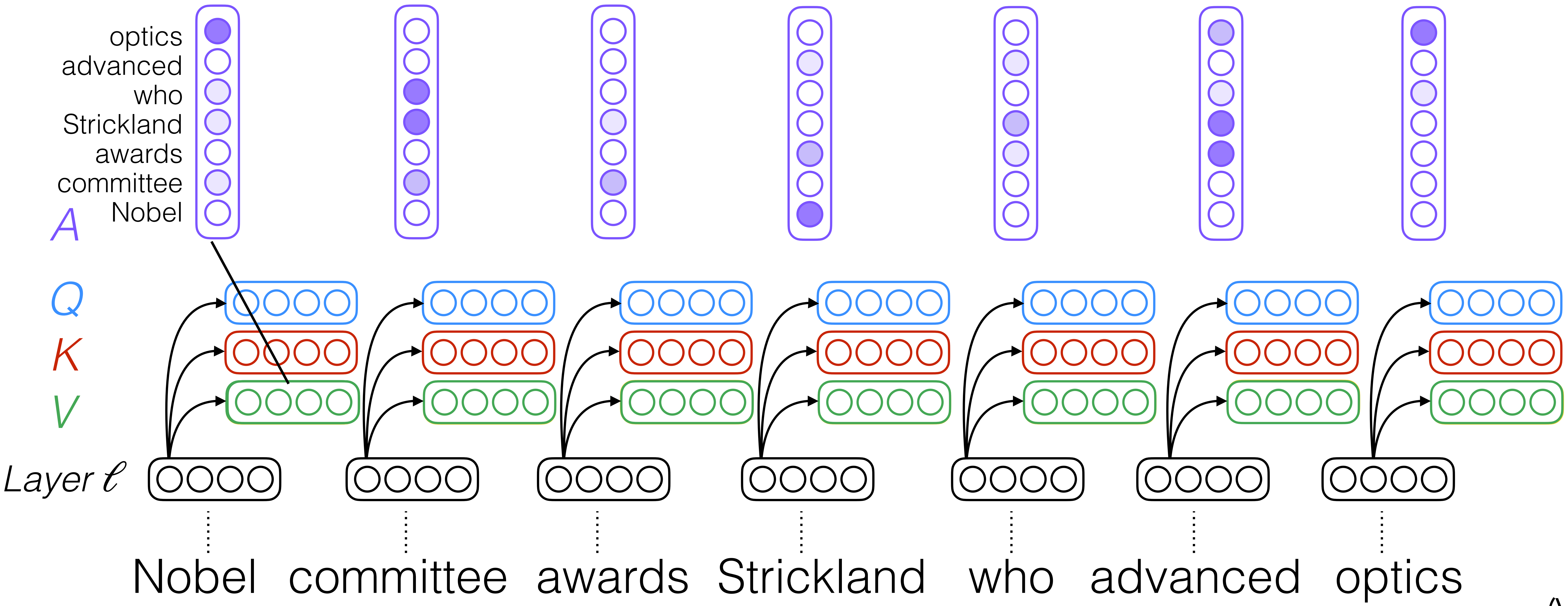
(Vaswani et al., 2017)

Self-attention (in encoder)



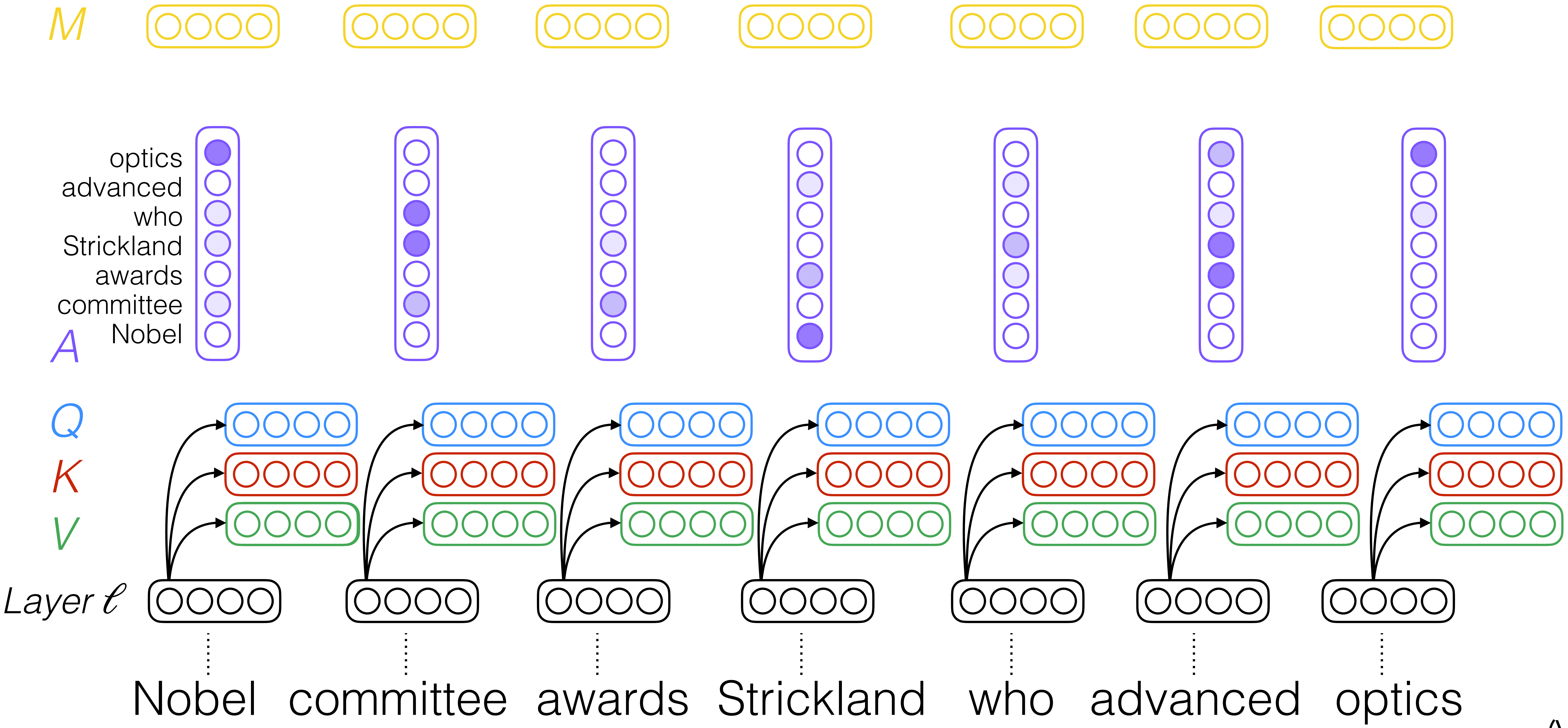
(Vaswani et al., 2017)

Self-attention (in encoder)



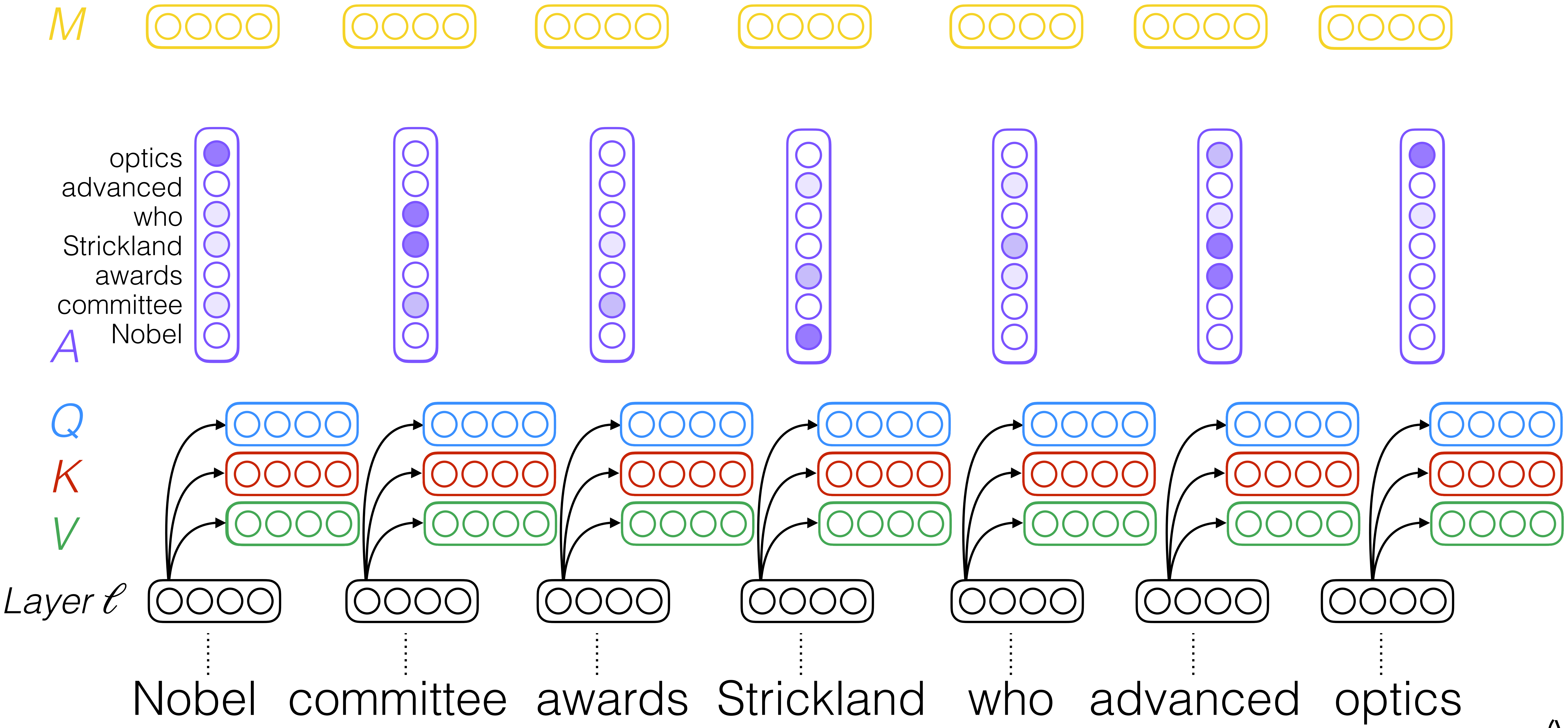
(Vaswani et al., 2017)

Self-attention (in encoder)



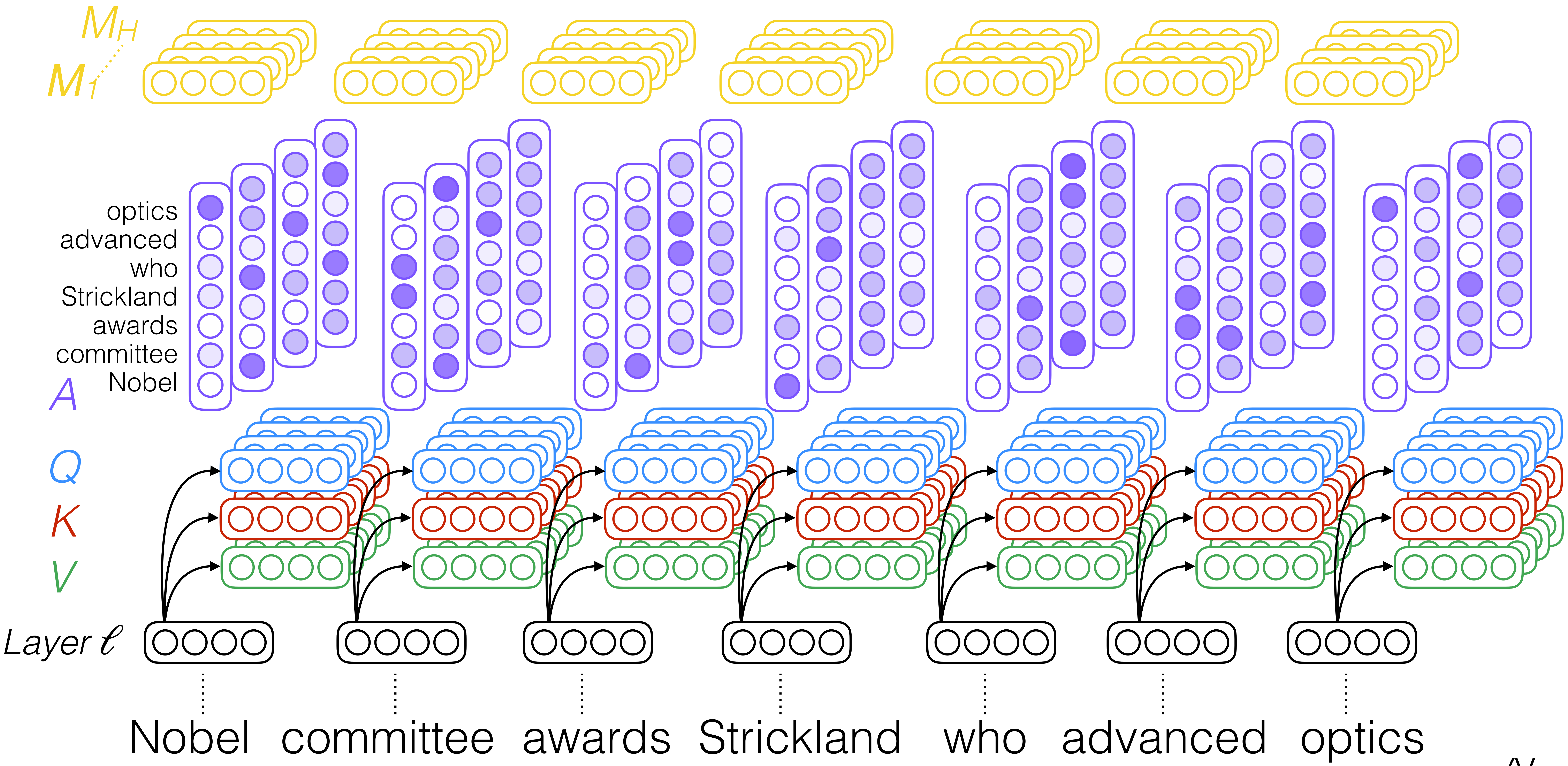
(Vaswani et al., 2017)

Self-attention (in encoder)



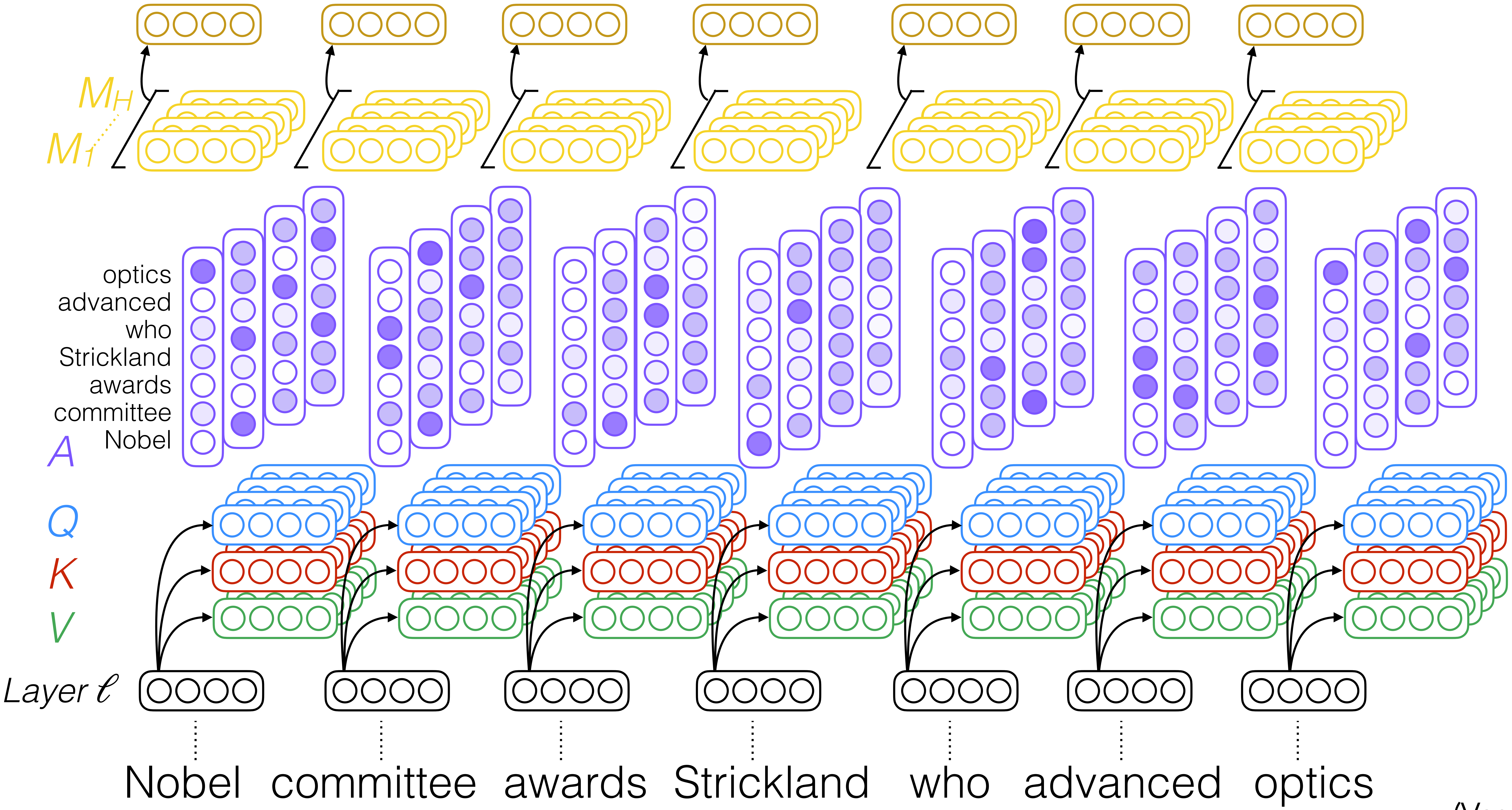
(Vaswani et al., 2017)

Multi-head self-attention



(Vaswani et al., 2017)

Multi-head self-attention



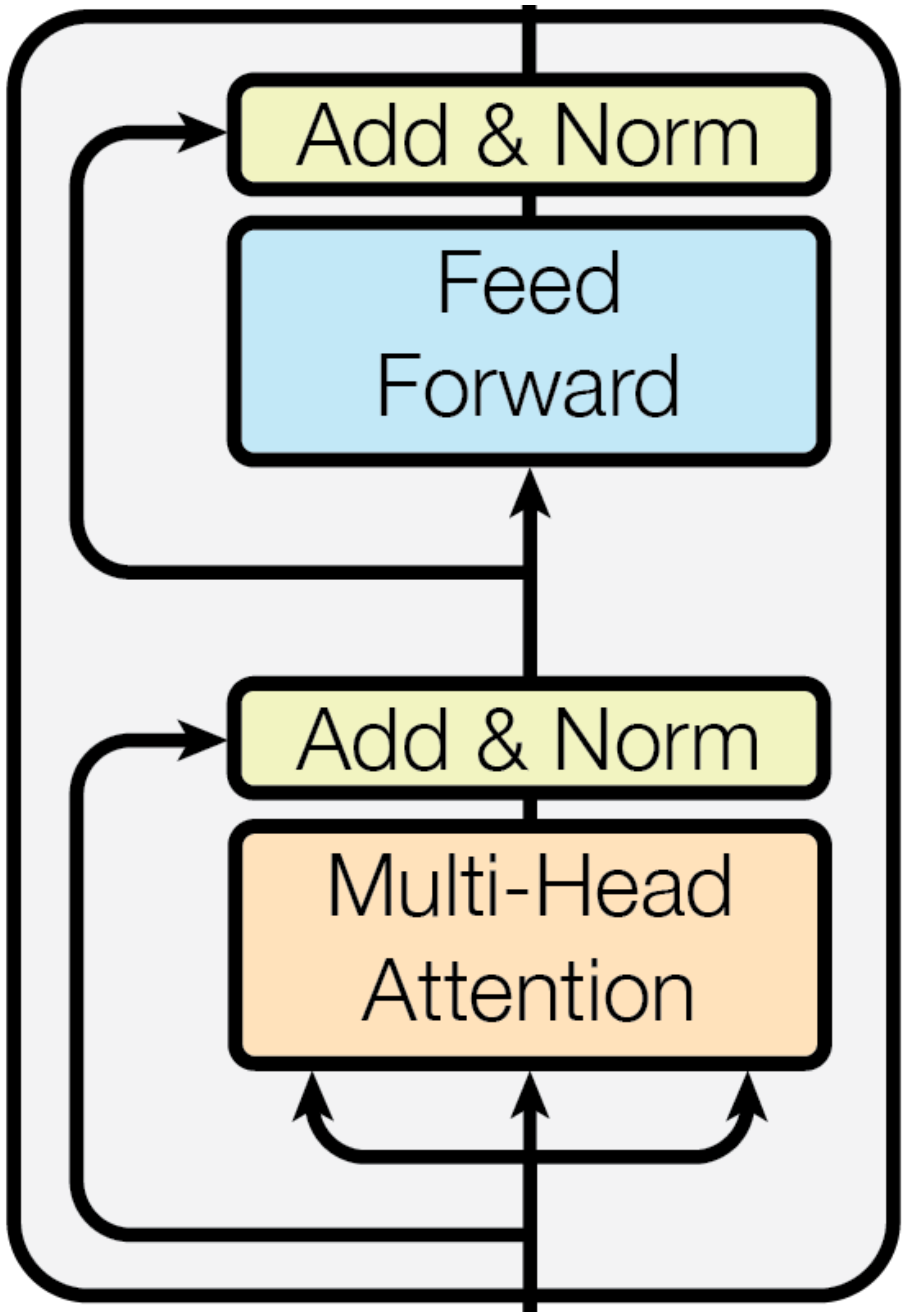
(Vaswani et al., 2017)

Question

What are two advantages of self-attention over recurrent models?

Transformer Block

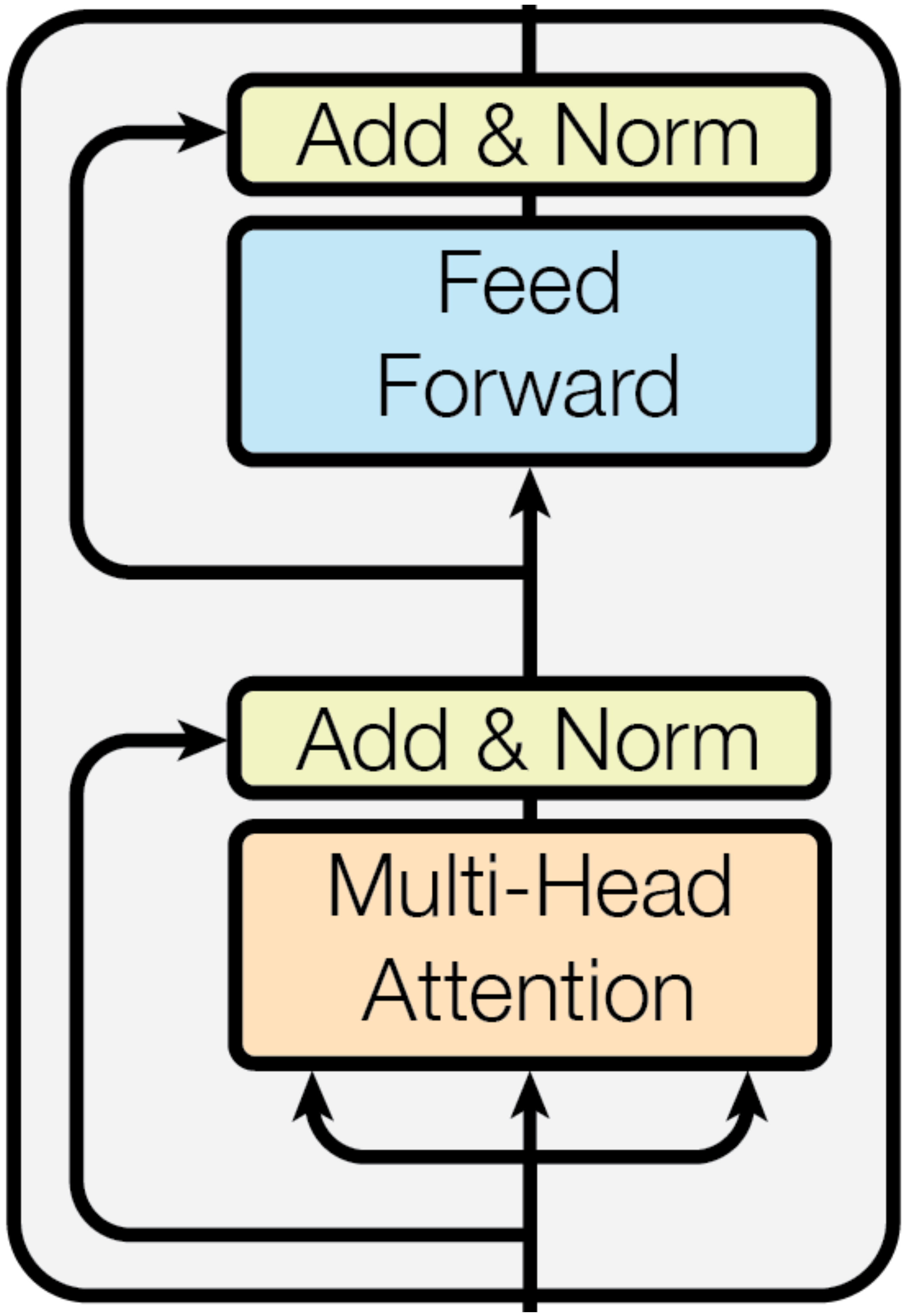
- Multi-headed attention is the main innovation of the transformer model!



Vaswani et al., 2017

Transformer Block

- Multi-headed attention is the main innovation of the transformer model!
- Each block also composed of:
  - a layer normalisations
  - a feedforward network
  - residual connections



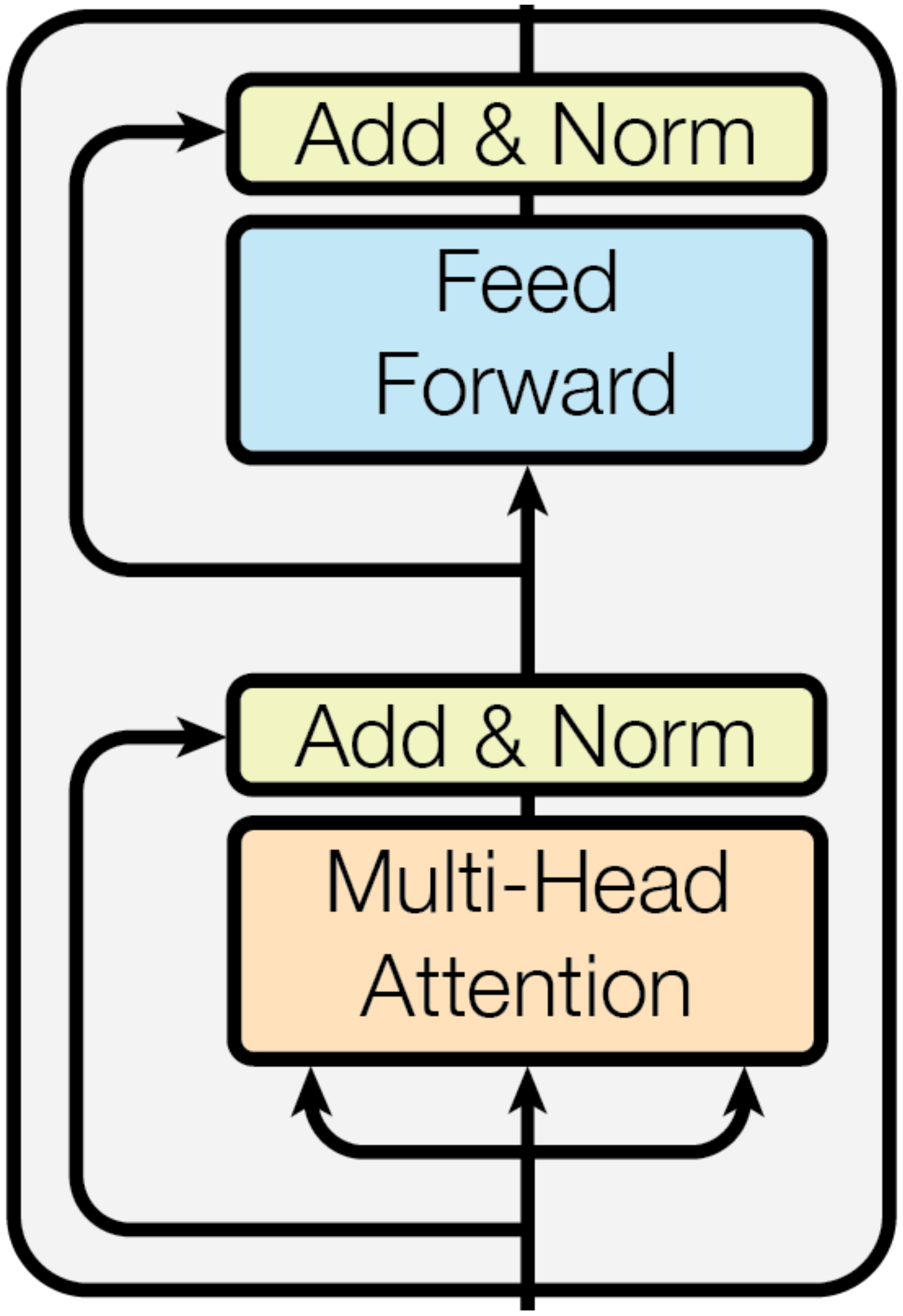
Vaswani et al., 2017

LayerNorm & Residual Connections

- **Layer Normalisation**
  - Normalize the outputs of different modules

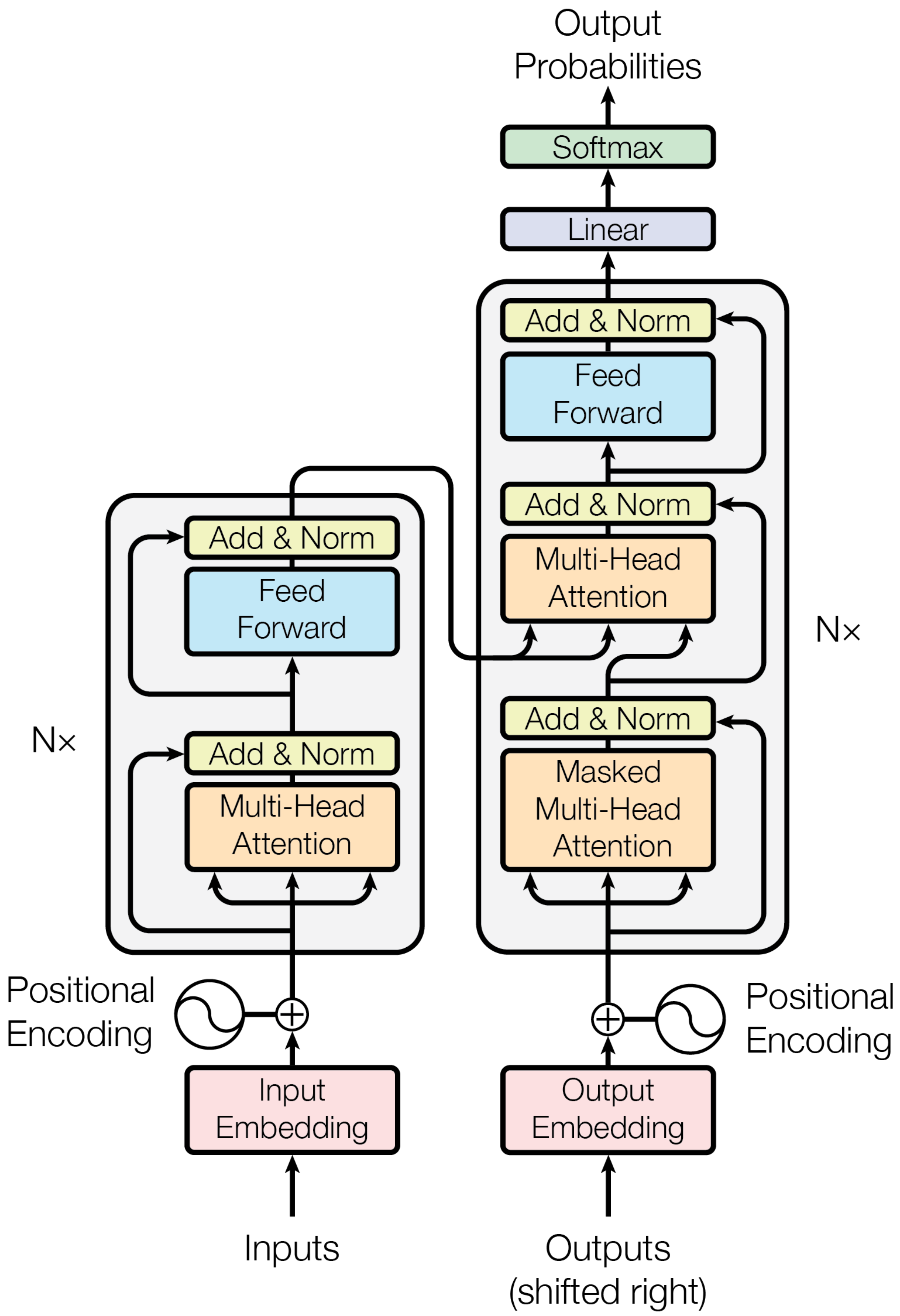
$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- **Residual Connections**
  - Add the input of a module to its output
  - $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$



Full Transformer

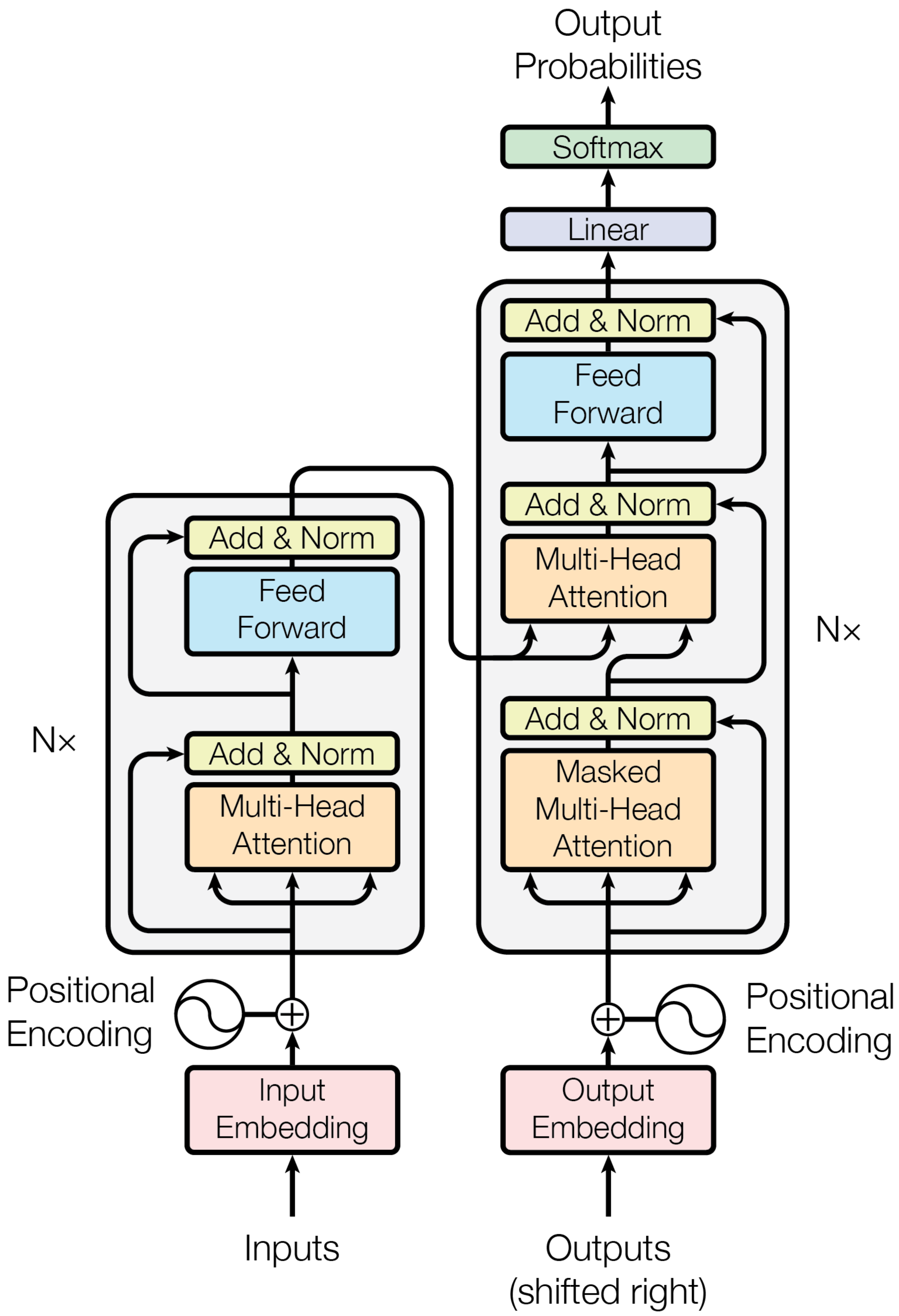
- Full transformer encoder is multiple cascaded transformer blocks
  - **build up compositional representations of inputs**



Vaswani et al., 2017

Full Transformer

- Full transformer encoder is multiple cascaded transformer blocks
  - **build up compositional representations of inputs**
- Transformer decoder (right) similar to encoder
  - First layer of block is **masked** multi-headed attention
  - Second layer is multi-headed attention over *final-layer* encoder outputs **(cross-attention)**
  - Third layer is feed-forward network



Vaswani et al., 2017

Question

What is an issue with self-attention for the decoder?

Masked Multi-headed Attention

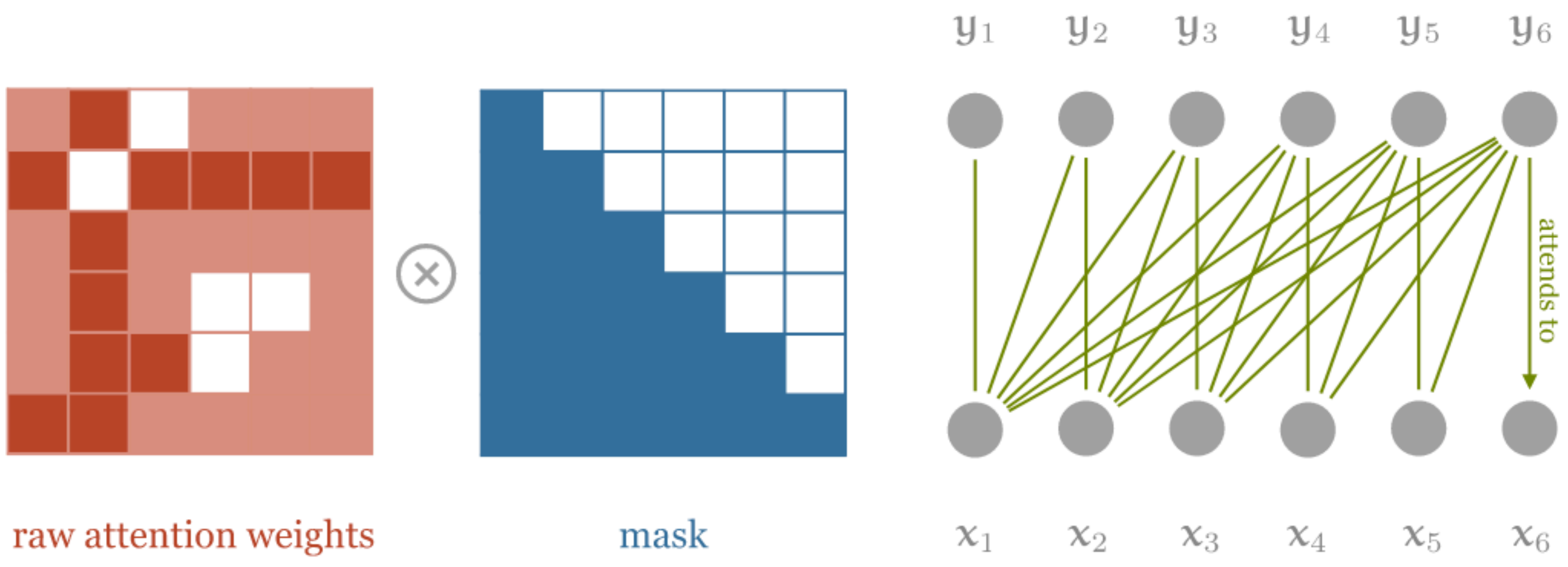
- Self-attention can attend to any token in the sequence

- For the decoder, **you don't want tokens to attend to future tokens**
  - Decoder used to generate text (i.e., machine translation)

Masked Multi-headed Attention

- Self-attention can attend to any token in the sequence
- For the decoder, **you don't want tokens to attend to future tokens**
  - Decoder used to generate text (i.e., machine translation)

Mask the attention scores of future tokens so their attention = 0



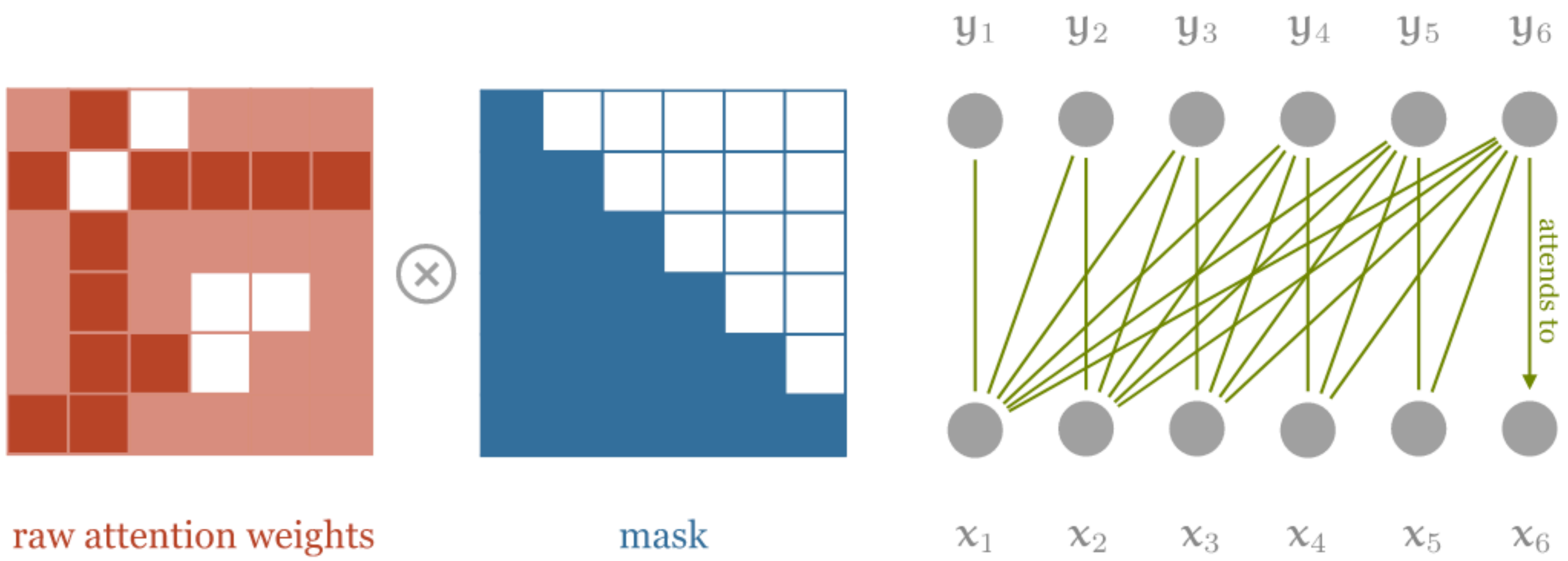
$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T (\mathbf{W}^K h_t^\ell)}{\sqrt{d}} \quad \Rightarrow \quad a_{st} := a_{st} - \infty \; ; \; s < t$$

Masked Multi-headed Attention

- Self-attention can attend to any token in the sequence

- For the decoder, **you don't want tokens to attend to future tokens**
  - Decoder used to generate text (i.e., machine translation)

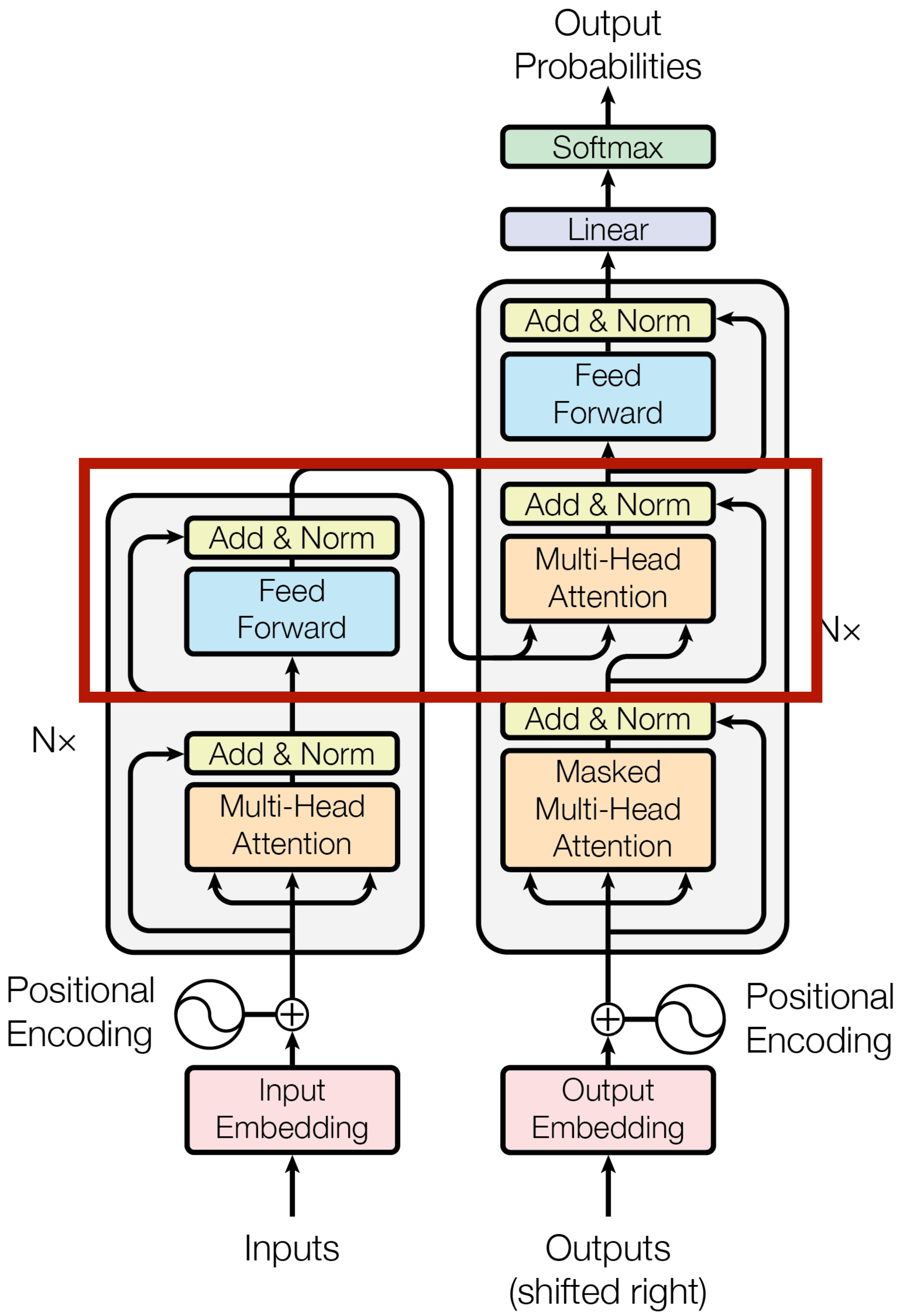
Mask the attention scores of future tokens so their attention = 0



$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T (\mathbf{W}^K h_t^\ell)}{\sqrt{d}} \quad \Rightarrow \quad a_{st} := a_{st} - \infty \ ; \ s < t \quad \Rightarrow \quad \alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}} = 0$$

Cross-attention

- **Cross attention** is the same classical attention as in the RNN encoder-decoder model
- The query to the attention function is the output of the masked multi-headed attention in the decoder (i.e., a decoder state)
- The keys and values are the output of the **final** encoder transformer
- Once again, a representation from the decoder is used to **attend** to the encoder outputs



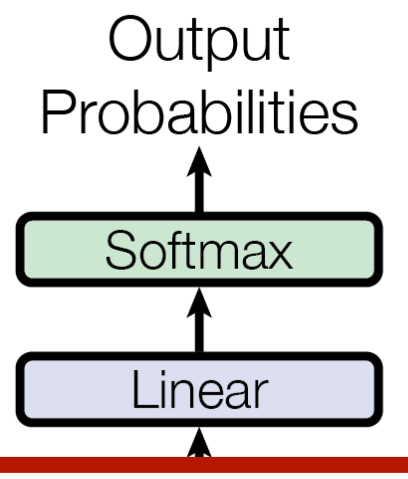
Vaswani et al., 2017

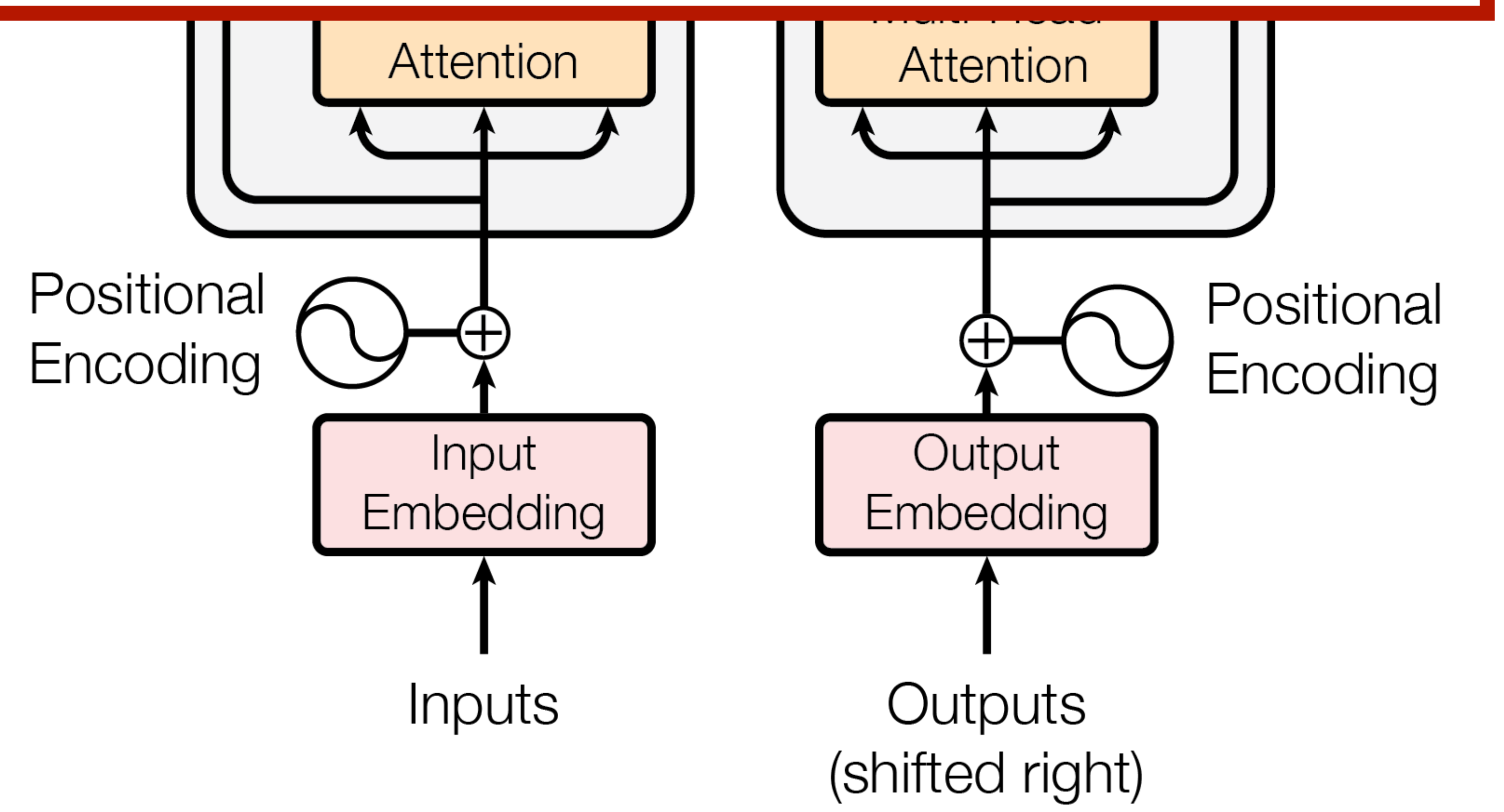
Full Transformer

- Full transformer encoder is multiple cascaded transformer blocks
  -
- Tr
  -
  - Second layer is multi-headed attention over encoder outputs **(cross-attention)**
  - Third layer is feed-forward network

Recurrent models provided word order information

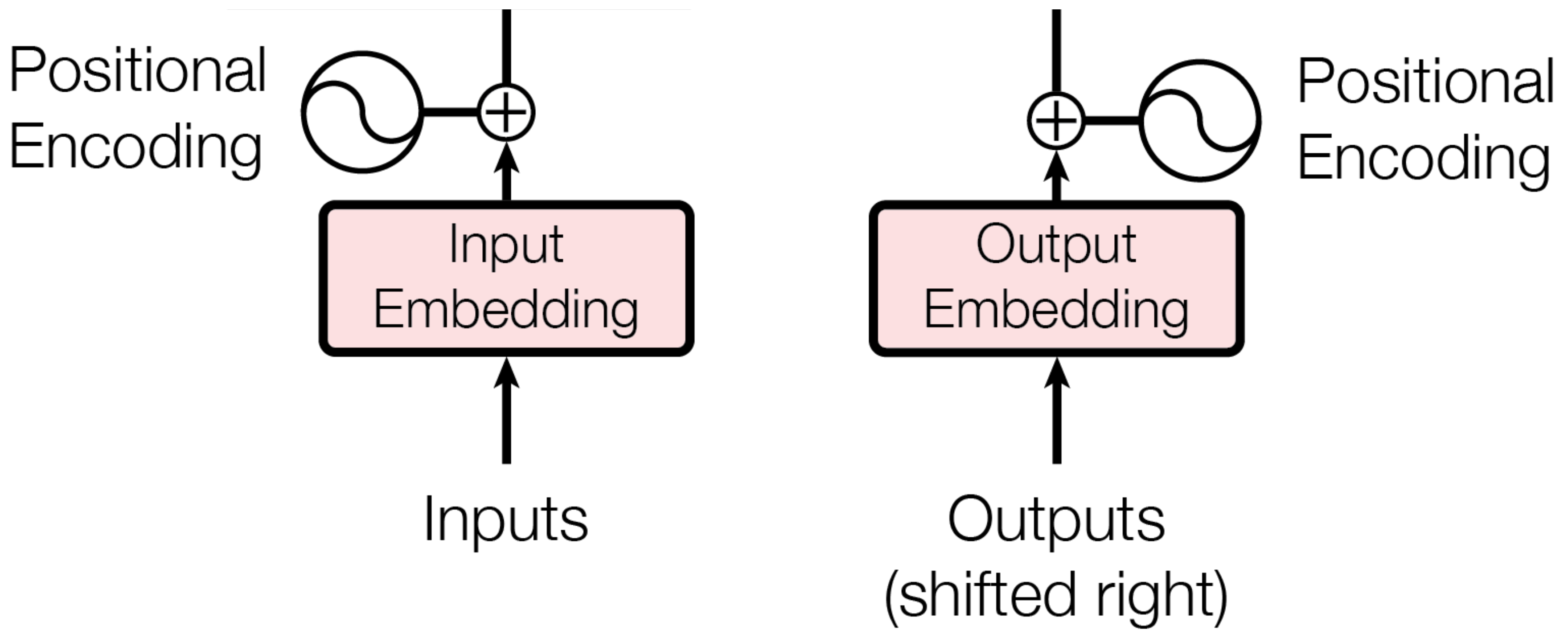
Does self-attention provide word order information?





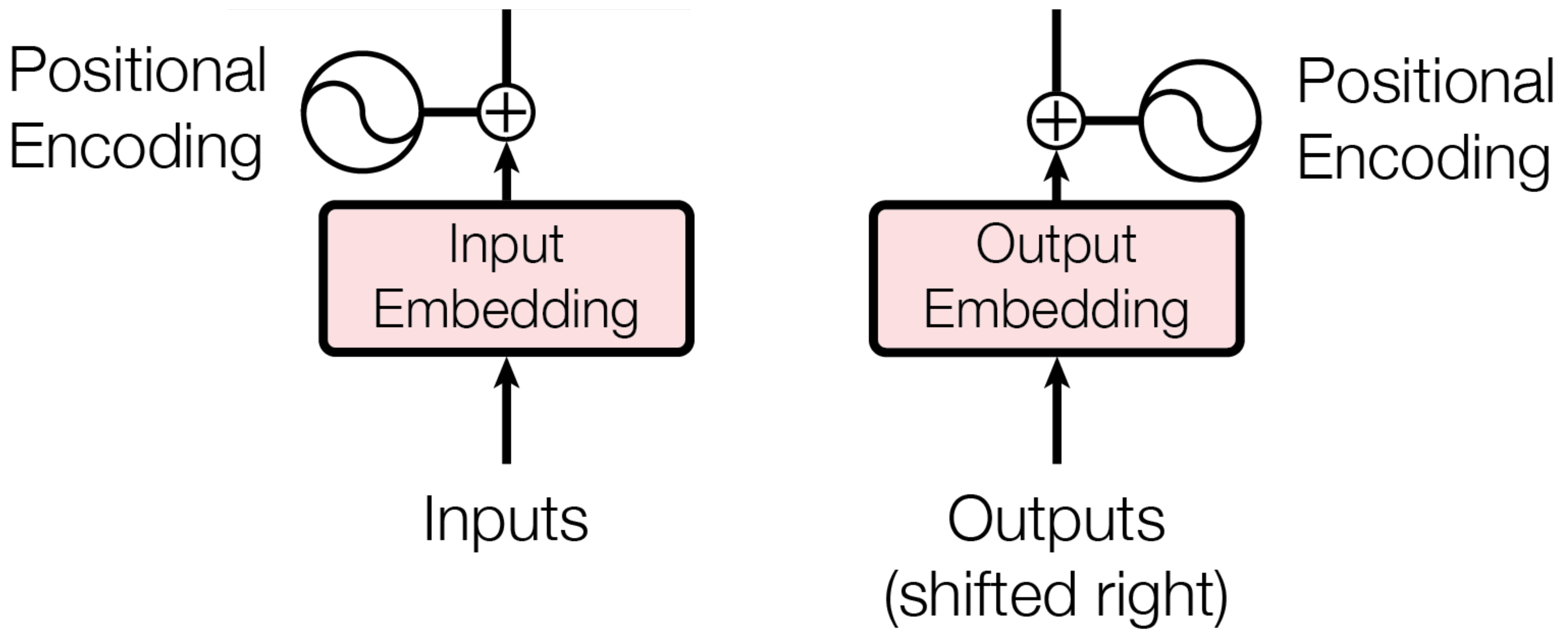
Vaswani et al., 2017

Position Embeddings



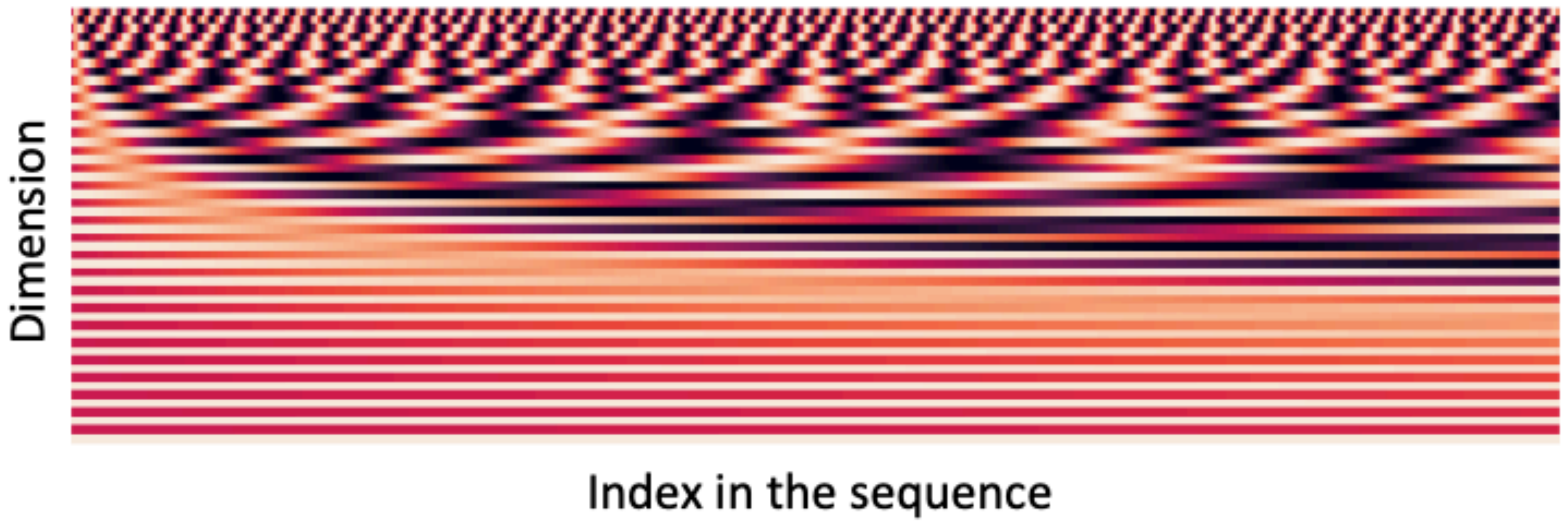
Vaswani et al., 2017

Position Embeddings



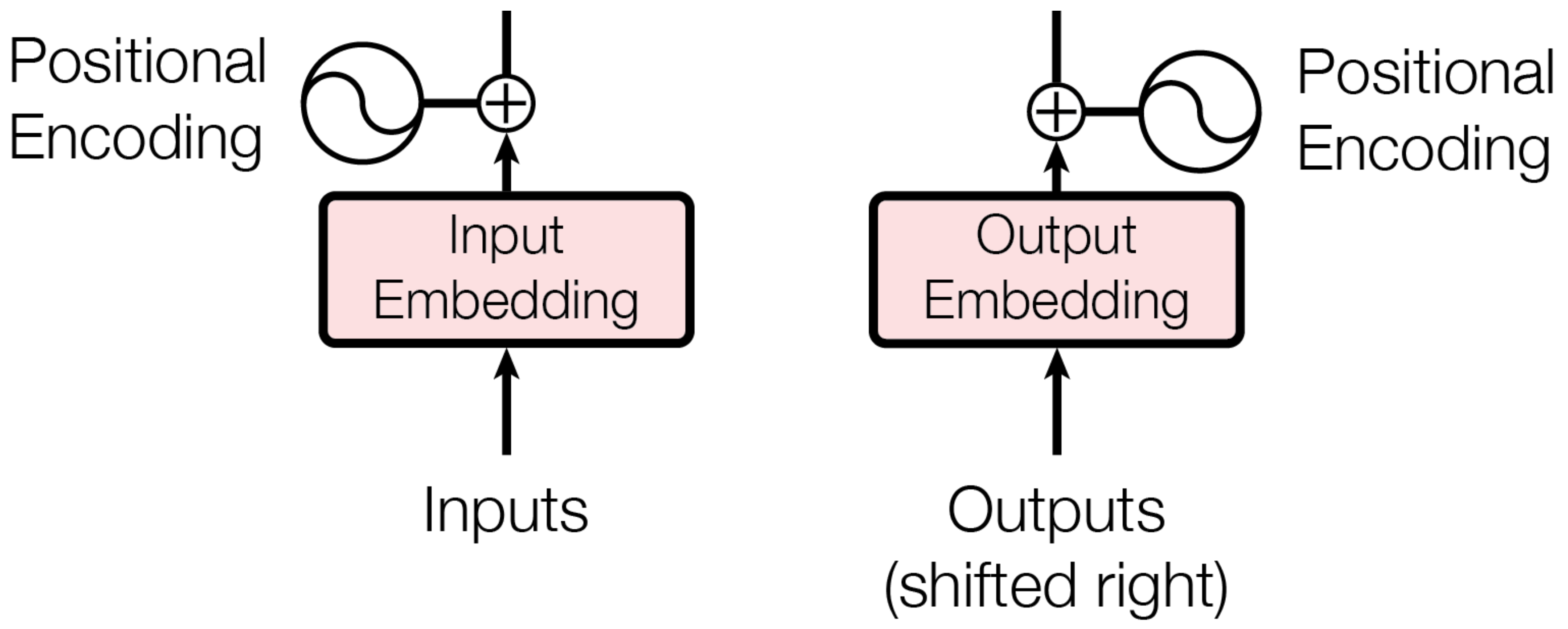
- Early position embeddings encoded a sinusoid function that was offset by a phase shift proportional to sequence position

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



Vaswani et al., 2017

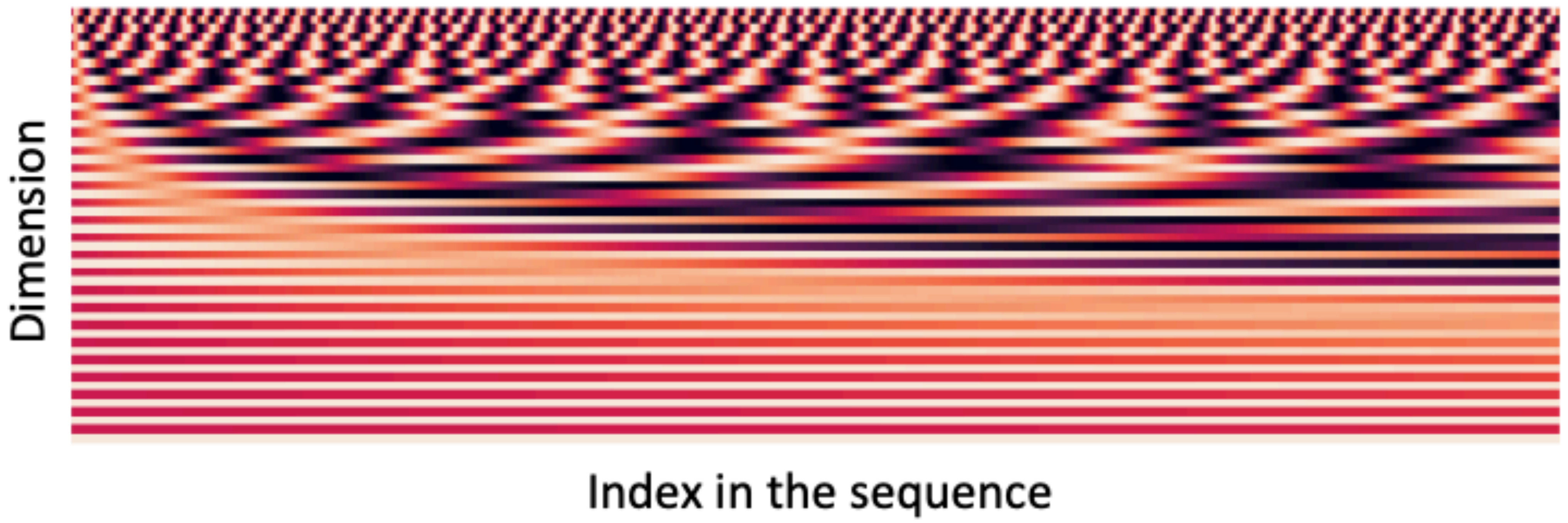
Position Embeddings



- Early position embeddings encoded a sinusoid function that was offset by a phase shift proportional to sequence position

- **In practice, easiest is to learn position embeddings from scratch**

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



Vaswani et al., 2017

Question

What might be a disadvantage of using learned position embeddings?

Poor generalisation to sequences longer than the maximum position embedding you have learned

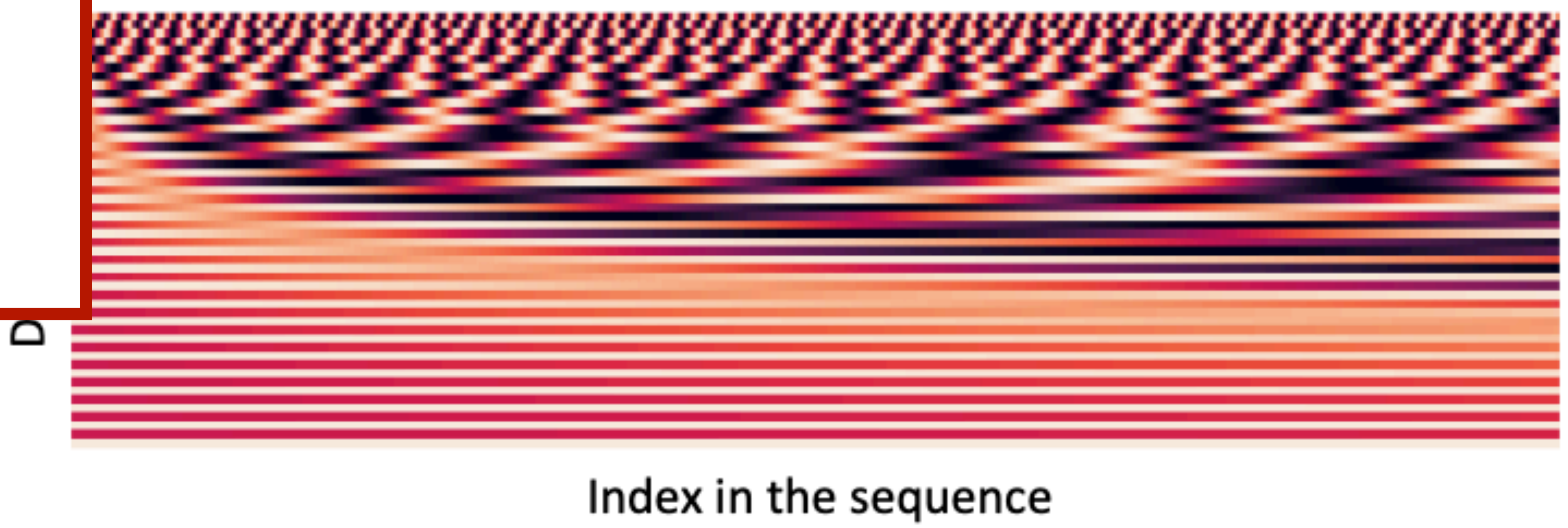
Position Embeddings

Lots of potential for new methods that generalise to longer sequences

Position embeddings remain an active area of research

- Early position embeddings encoded a sinusoid function that was offset by a phase shift proportional to sequence position
- **In practice, easiest is to learn position embeddings from scratch**

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



Vaswani et al., 2017

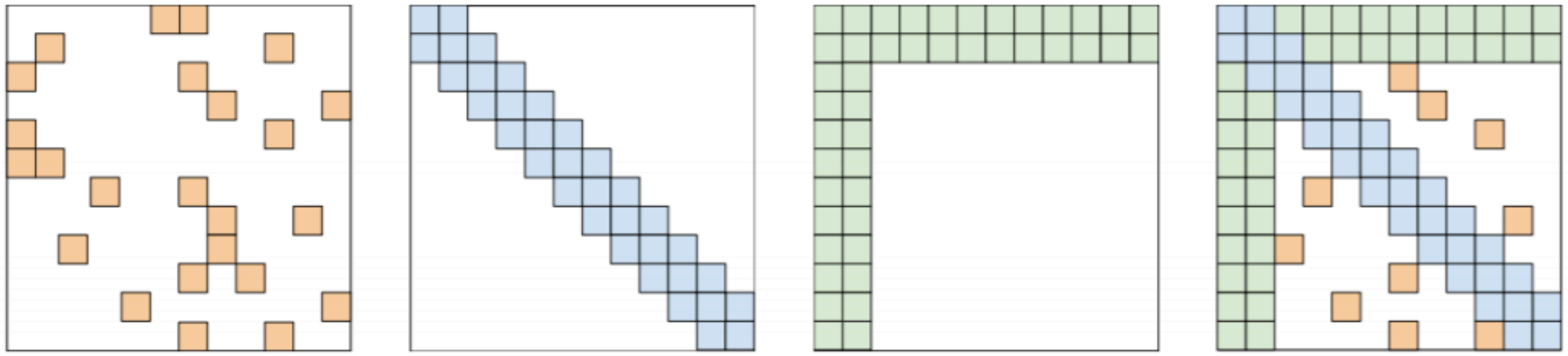
Performance: Machine Translation

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

(Vaswani et al., 2017)

Question

What could be a disadvantage of transformers over RNNs?



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

Other Resources of Interest

- The Annotated Transformer
  - https://nlp.seas.harvard.edu/2018/04/03/attention.html
- The Illustrated Transformer
  - https://jalammar.github.io/illustrated-transformer/
- Only basics presented here today! Many modifications to initial transformers exist

Recap

- **Temporal Bottleneck**: **Vanishing gradients** stop many RNN architectures from learning **long-range dependencies**
- **Parallelisation Bottleneck:** RNN states depend on previous time step hidden state, so must be **computed in series**
- **Attention**: Direct connections between output states and inputs (solves temporal bottleneck)
- **Self-Attention**: Remove recurrence, allowing parallel computation
- Modern **Transformers** use attention, but require position embeddings to capture sequence order

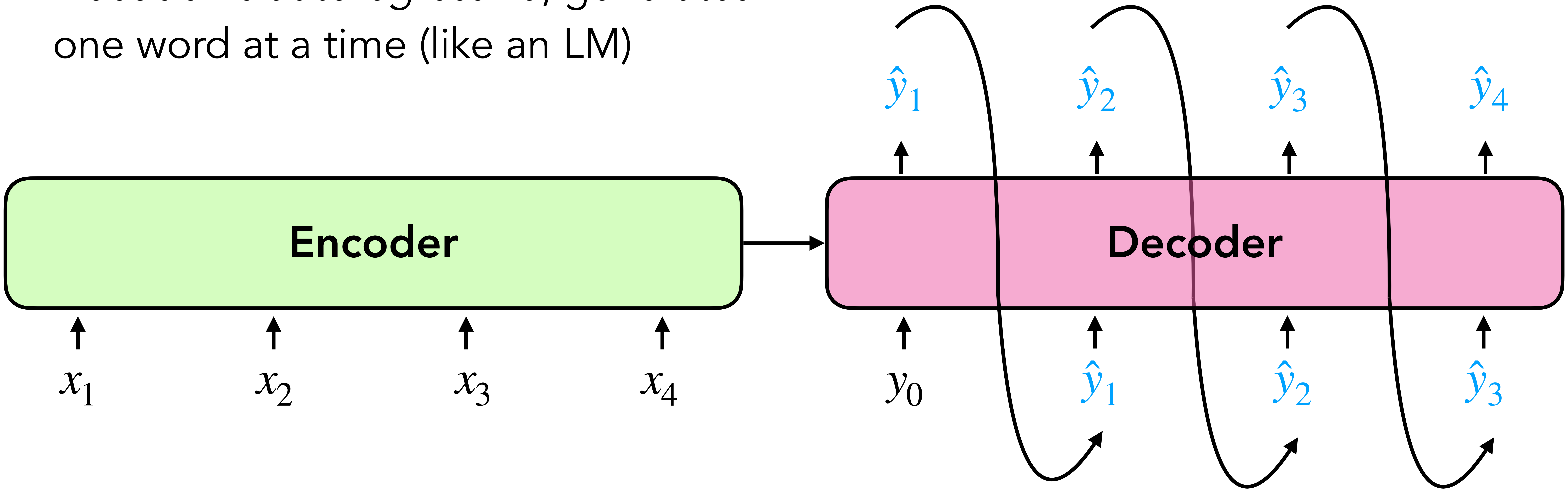
Decoding from Neural Models

Antoine Bosselut

Encoder-Decoder Models

- Encode a sequence fully with one model (**encoder**) and use its representation to seed a second model that decodes another sequence (**decoder**)

- Decoder is *autoregressive*, generates one word at a time (like an LM)



Decoding: Main Idea

- At each time step *t*, our model computes a vector of scores for each token in our vocabulary, $S \in \mathbb{R}^V$:

$$S = f\left(\{y_{<t}\}\right)$$

$f(\,.\,)$ is your decoder

- Then, we compute a probability distribution P over these scores (with a softmax):

$$P\left(y_t = w \,\middle|\, \{y_{<t}\}\right) = \frac{\exp(S_w)}{\sum_{w' \in V} \exp(S_{w'})}$$

- Decoding algorithm defines a function to select a token from this distribution:

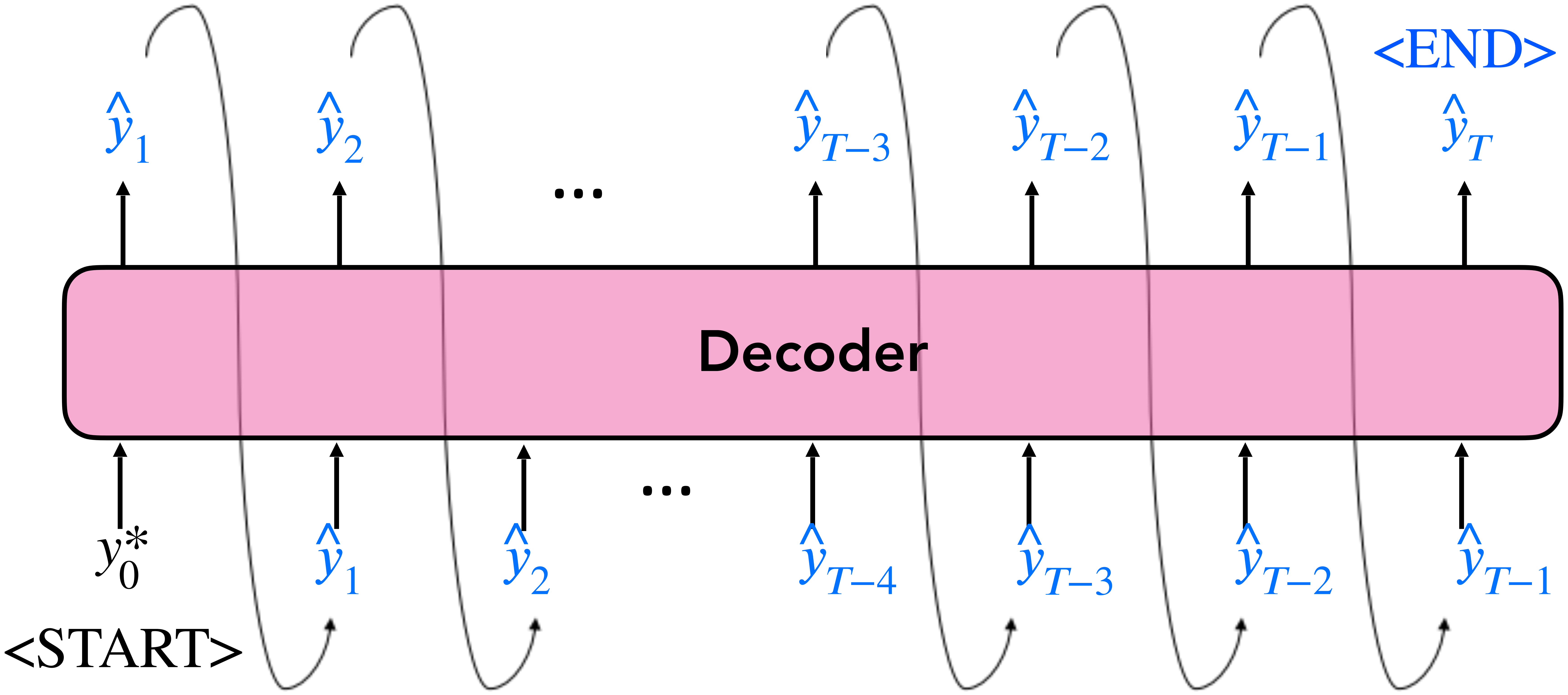
$$\hat{y}_t = g\left(P\left(y_t \,|\, \hat{y}_{<t}\right)\right)$$

$g(\,.\,)$ is your decoding algorithm

Decoding: Main Idea

- Decoding algorithm defines a function to select a token from this distribution

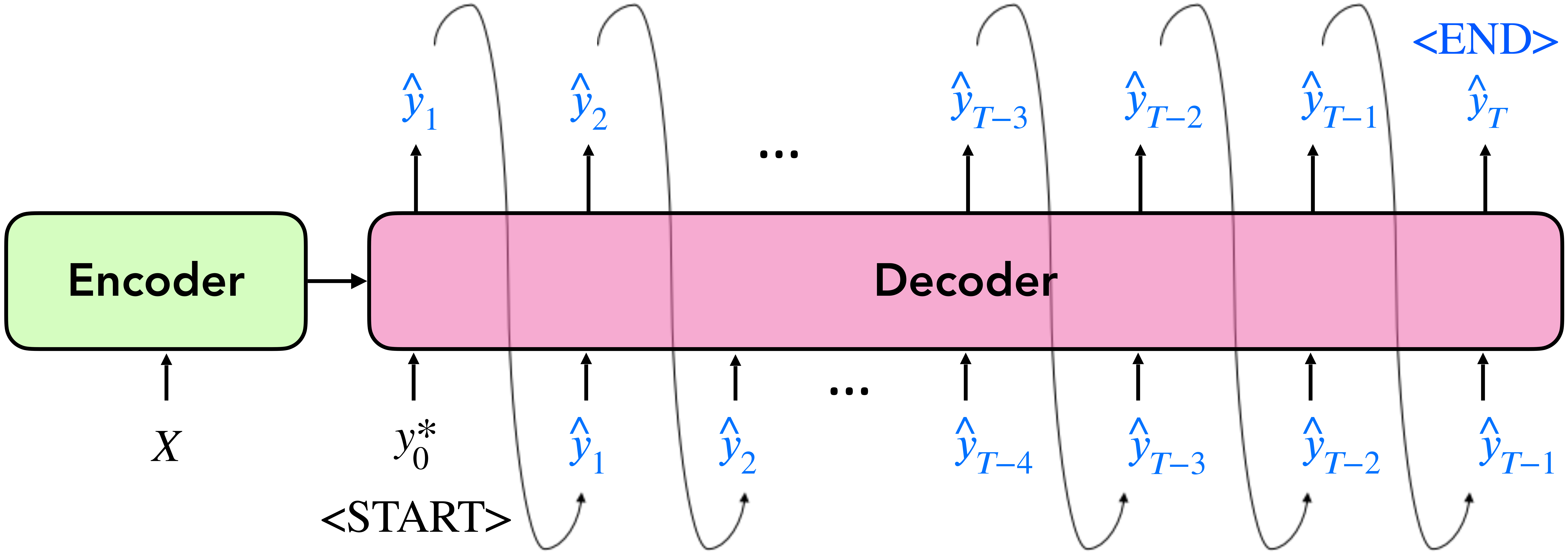
$$\hat{y}_t = g\left(P\left(y_t \mid \hat{y}_{<t}\right)\right)$$



Optional: Encoder Input

- Decoding algorithm defines a function to select a token from this distribution

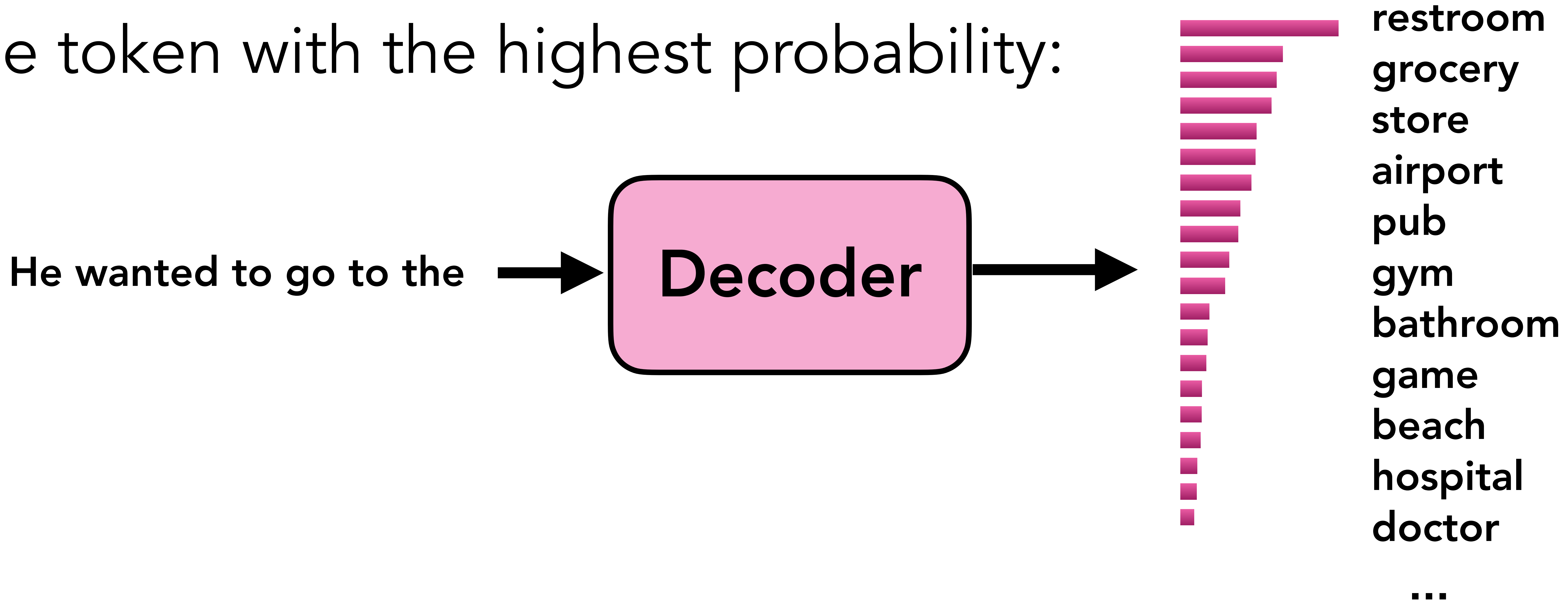
$$\hat{y}_t = g\big(P(y_t \mid X, \hat{y}_{<t})\big)$$



Greedy methods: Argmax Decoding

$$\hat{y}_t = \underset{w \in V}{\textbf{argmax}}\ P(y_t = w \mid \{y\}_{<t})$$

- g = select the token with the highest probability:



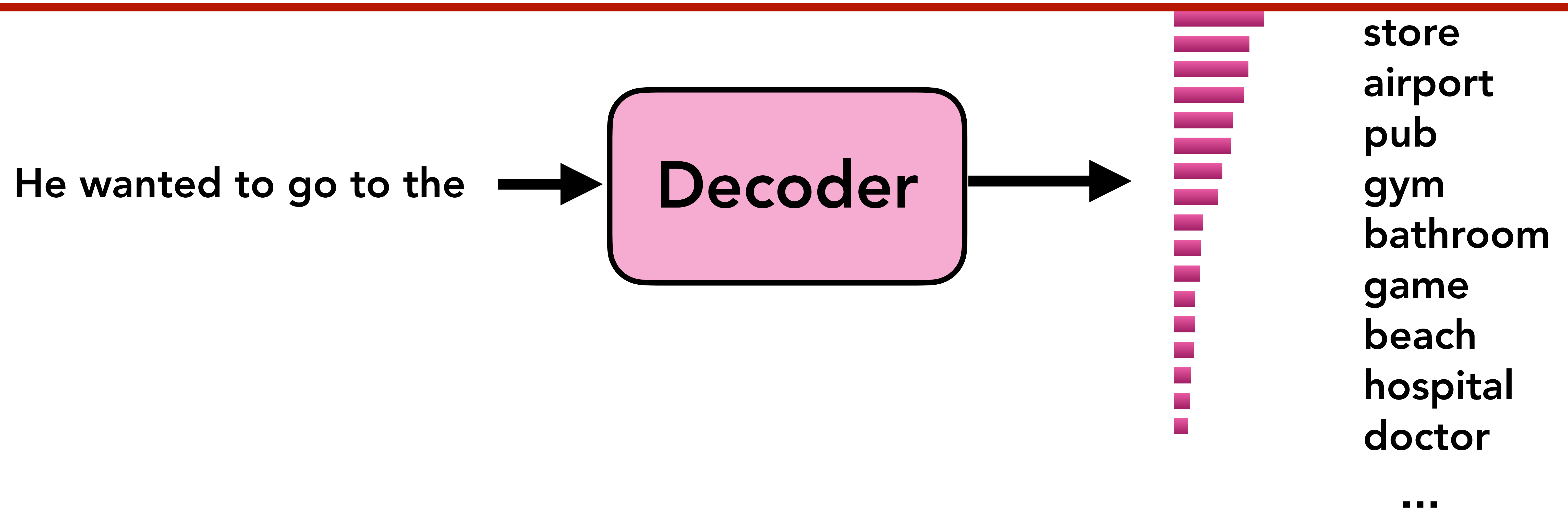
Greedy methods: Argmax Decoding

$$\hat{y}_t = \textbf{argmax}\, P(y_t = w \mid \{y\}_{<t})$$

Select **highest scoring** token

What's a potential problem with argmax decoding?

- g = selec



Issues with argmax decoding

- In argmax decoding, we cannot revise prior decisions
    - *les pauvres sont démunis (the poor don’t have any money)*
    - → *the* ____
    - → *the poor* ____
    - → *the poor are* ____

- Potential leads to sequences that are
    - **Ungrammatical**
    - **Unnatural**
    - **Nonsensical**
    - **Incorrect**

Beam Search

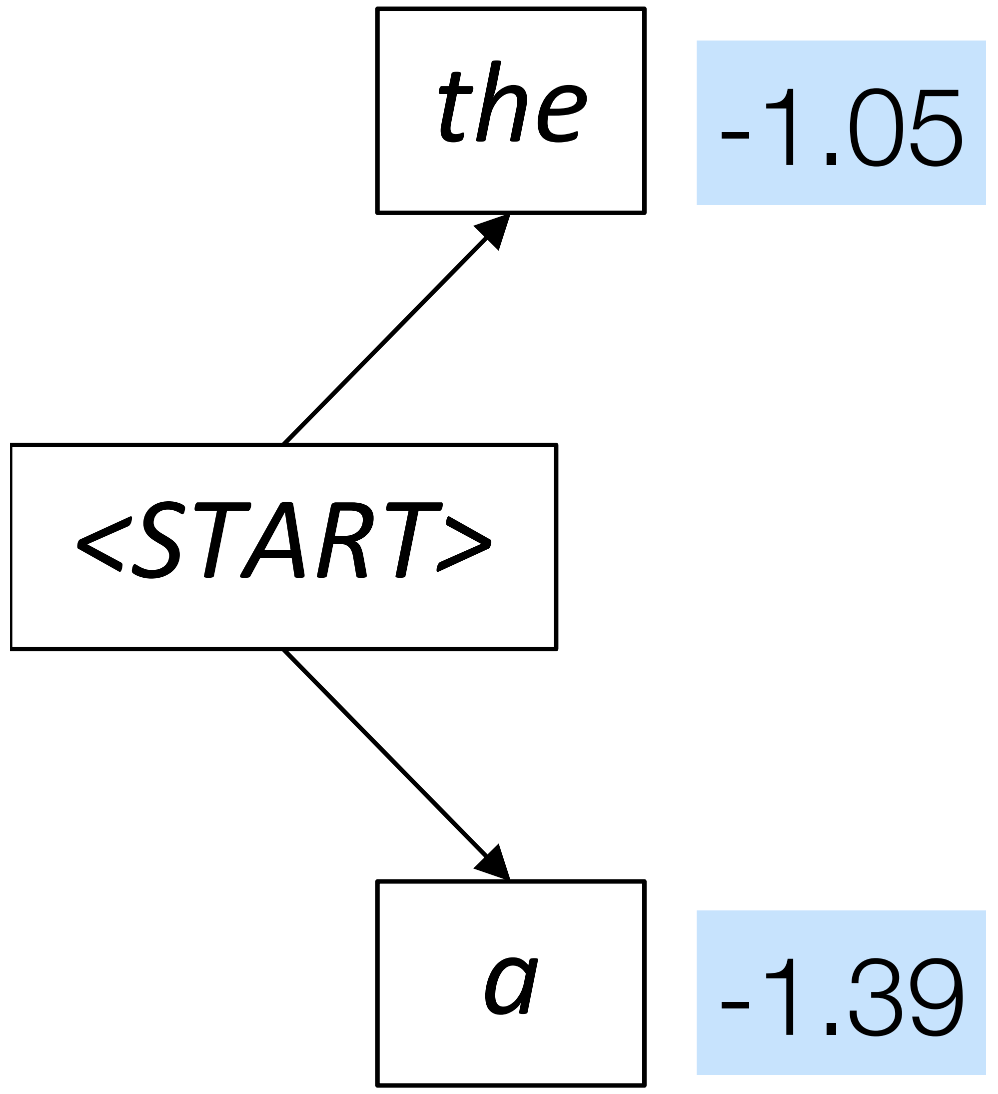
- *les pauvres sont démunis (the poor don't have any money)*
- → *the* ____
- → *the poor* ____
- → *the poor are* ____

- **Beam Search**: Explore several different hypotheses instead of just one
  - Track of the *b* highest scoring sequences at each decoder step instead of just one
  - Score at each step: $\sum_{t=1}^{j} \log P(\hat{y}_t \mid \hat{y}_1, \ldots, \hat{y}_{t-1}, X)$
  - *b* is called the **beam size**

Beam Search

Beam size = 2

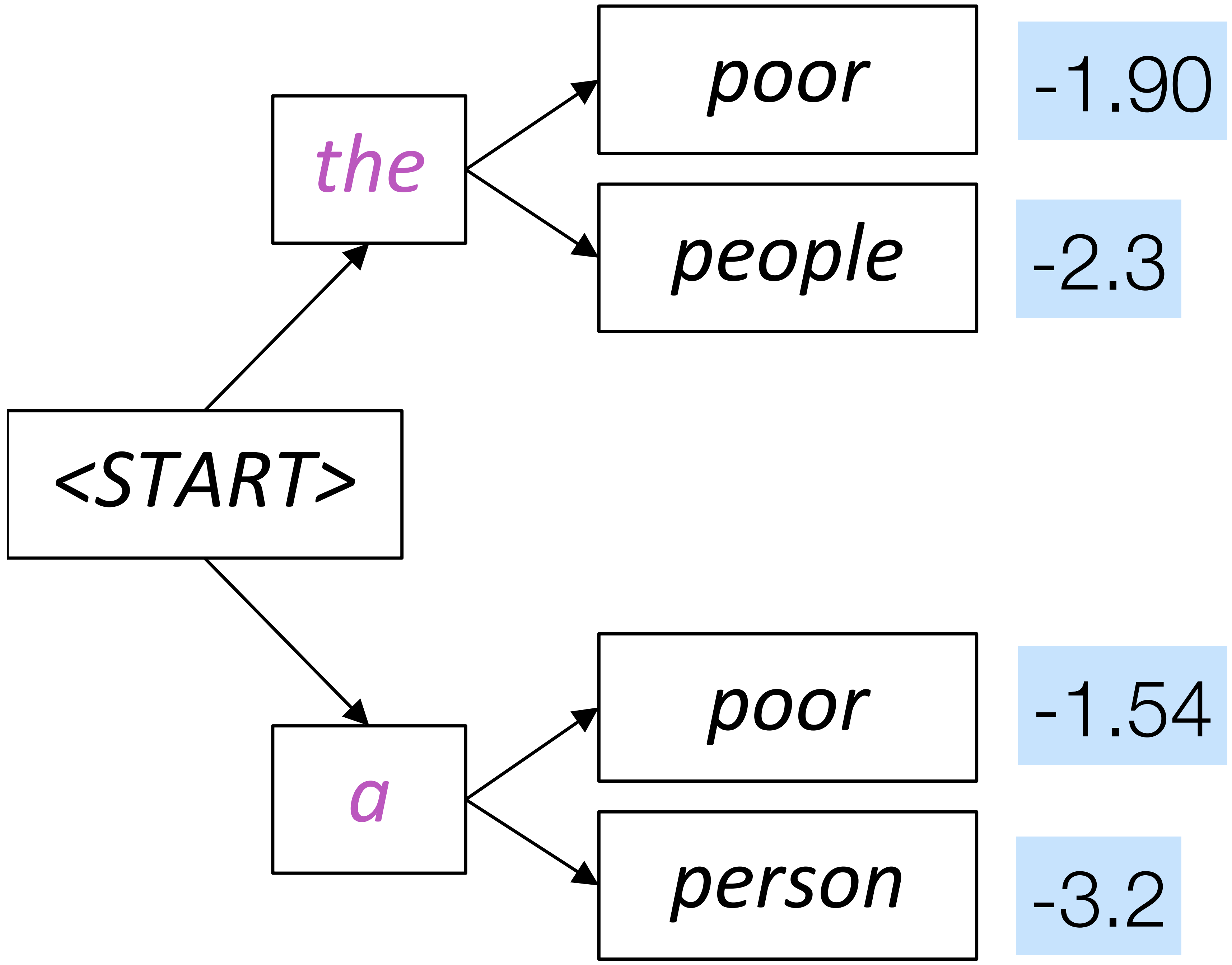
$\log P(\hat{y}_1 \mid y_0)$



Beam Search

Beam size = 2

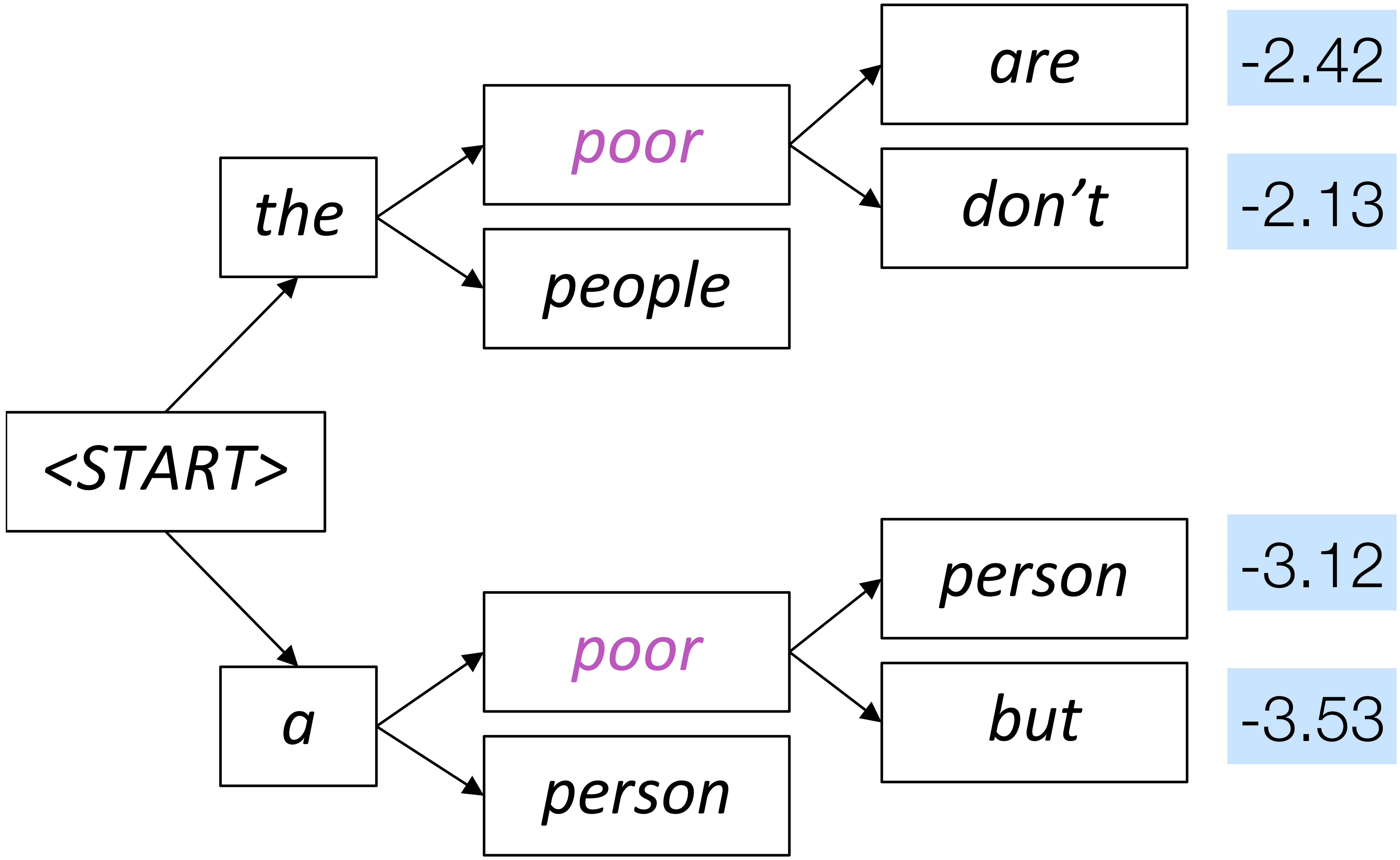
$$\sum_{t=1}^{2} \log P(\hat{y}_t \mid \hat{y}_0, \ldots, \hat{y}_{t-1})$$



Beam Search

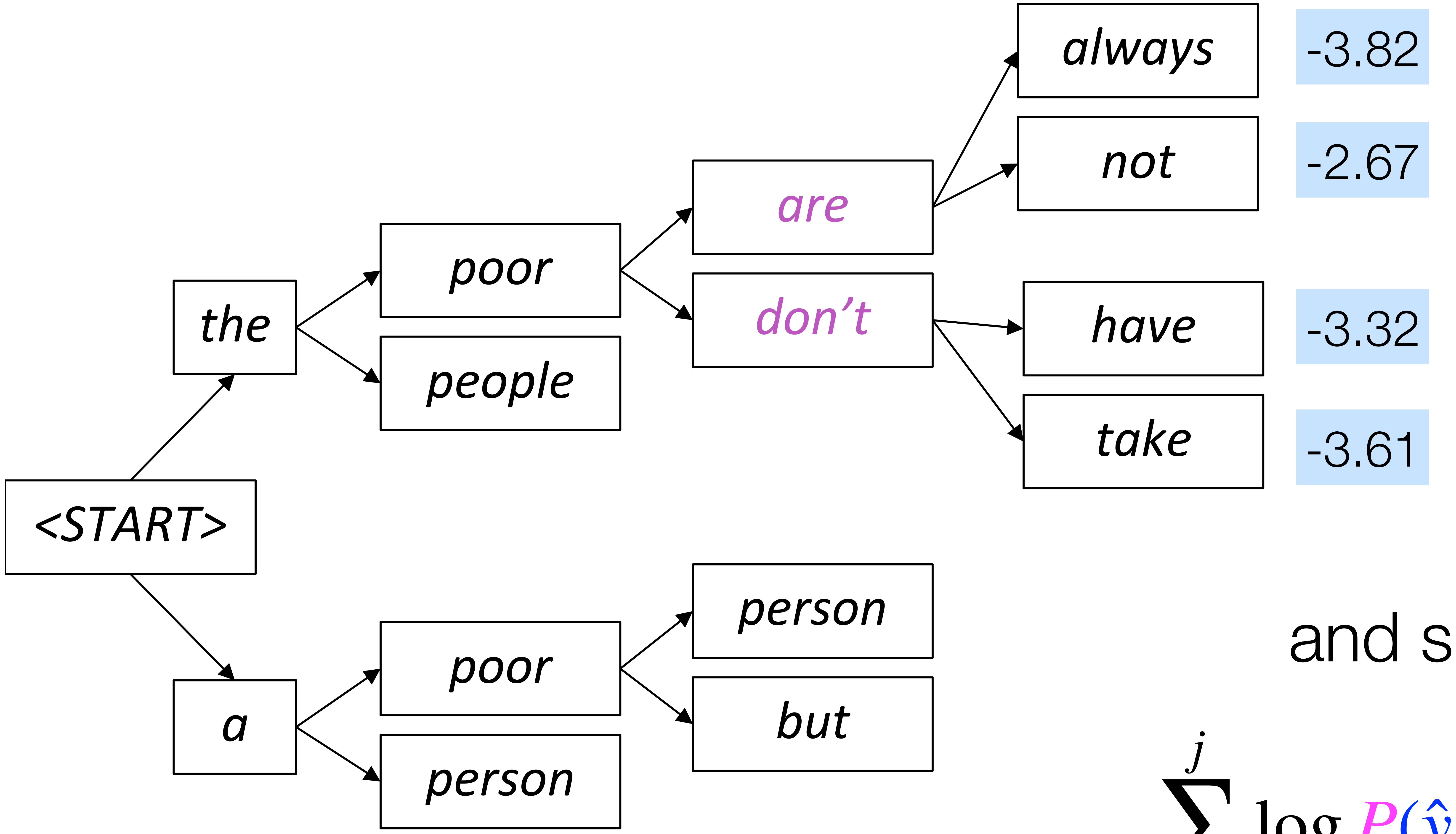
Beam size = 2

$$\sum_{t=1}^{3} \log P(\hat{y}_t \mid y_0, \hat{y}_1, \ldots, \hat{y}_{t-1})$$



Beam Search

Beam size = 2

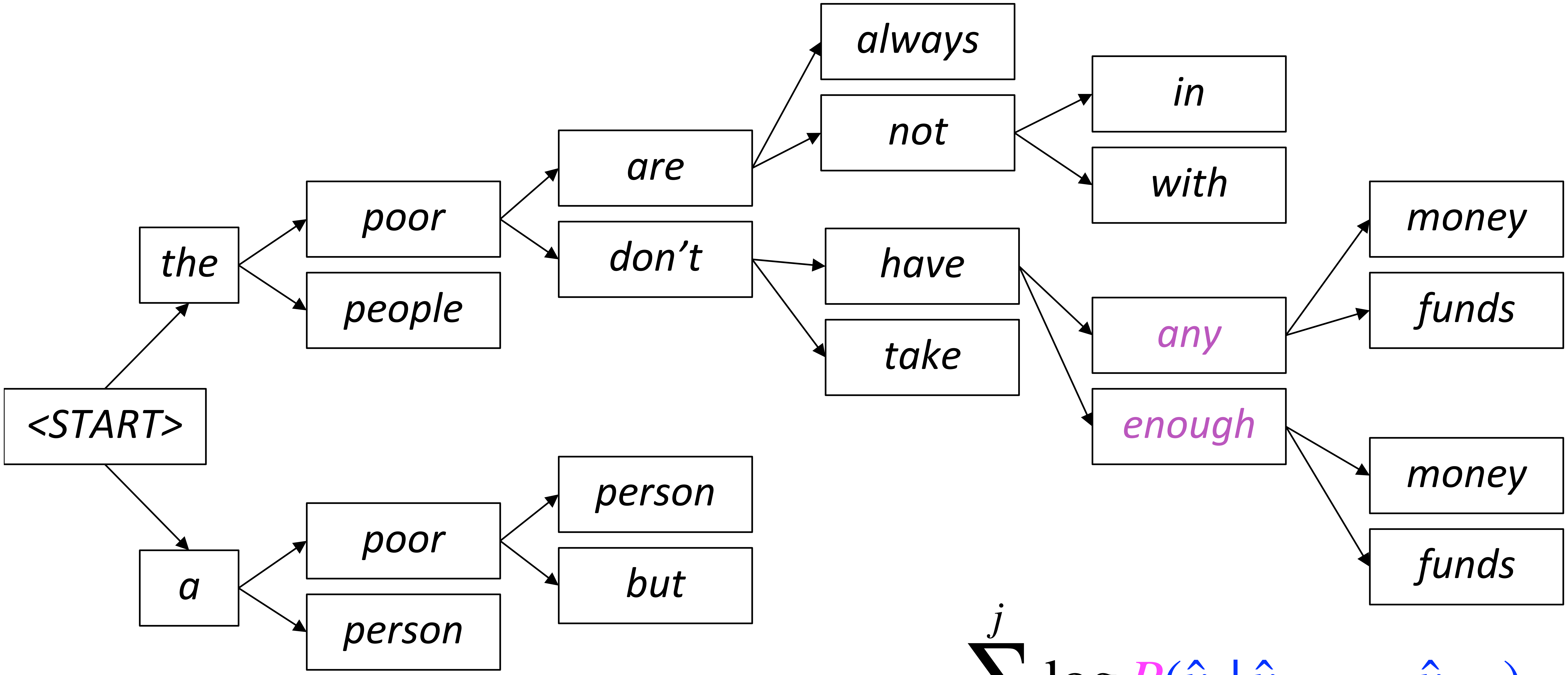


and so on…

$$\sum_{t=1}^{j} \log P(\hat{y}_t \mid \hat{y}_1, \ldots, \hat{y}_{t-1})$$

Beam Search

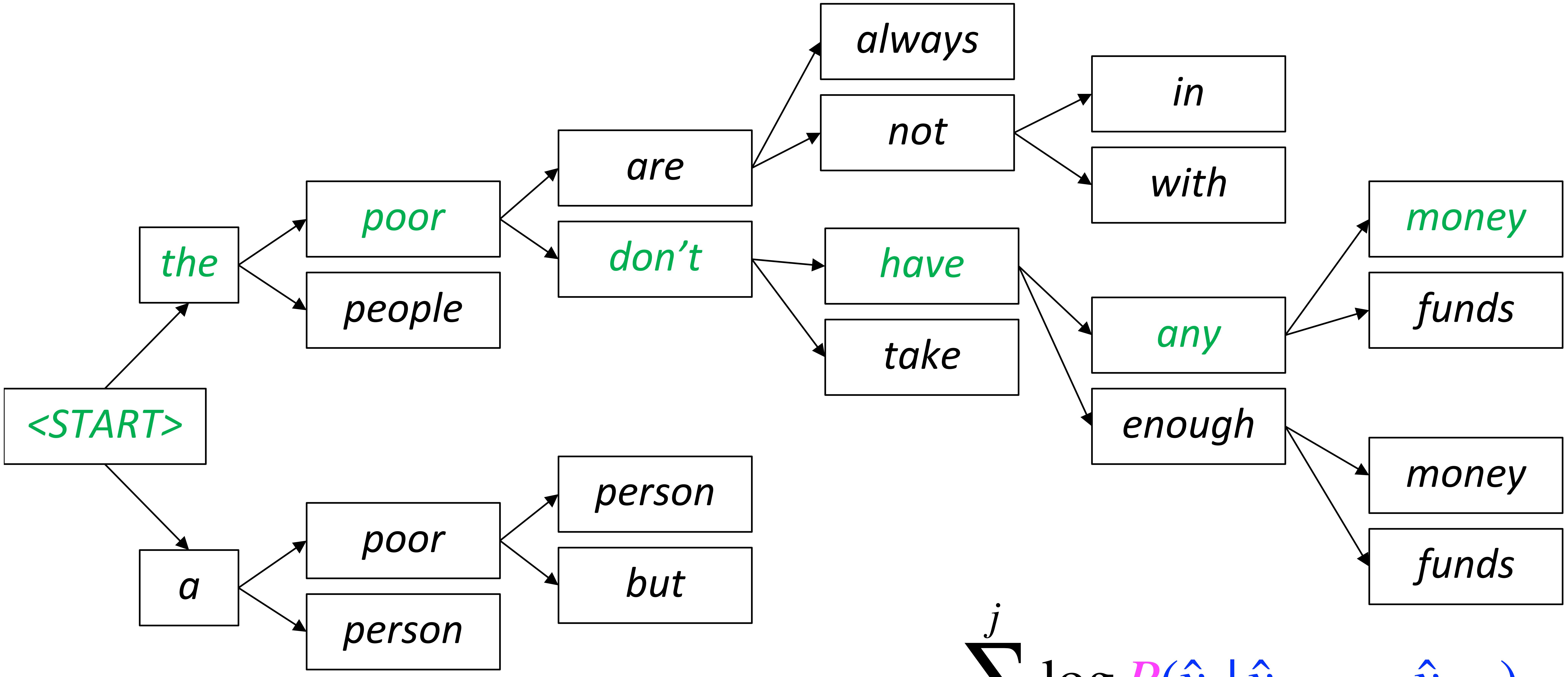
Beam size = 2



$$\sum_{t=1}^{j} \log P(\hat{y}_t \mid \hat{y}_1, \ldots, \hat{y}_{t-1})$$

Beam Search

Beam size = 2



$$\sum_{t=1}^{j} \log P(\hat{y}_t | \hat{y}_1, \ldots, \hat{y}_{t-1})$$

Beam Search

- Different hypotheses may produce <END> token at different time steps
  - When a hypothesis produces <END>, stop expanding it and place it aside
- Continue beam search until:
  - All b beams (hypotheses) produce <END> OR
  - Hit max decoding limit T
- Select top hypotheses using the *normalized* likelihood score

$$\frac{1}{T}\sum_{t=1}^{T}\log P(\hat{y}_t \mid \hat{y}_1, \ldots, \hat{y}_{t-1}, X)$$

  - Otherwise shorter hypotheses have higher scores

What do you think might happen if we increase the beam size?

Effect of beam size

- Small beam size *b* has similar issues as argmax decoding
  - **Outputs that are ungrammatical, unnatural, nonsensical, incorrect**
  - b=1 is the same as argmax decoding
- Larger beam size *b* reduces some of these problems
  - Potentially much more computationally expensive
  - Outputs tend to get shorter and more generic

Looking Forward

- **Tomorrow**: Pretraining Transformers - GPT
- **Next week:** Pretraining masked language models (BERT), Transfer Learning
- **Exercise Session:** Transformers, Decoding

References

- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR, abs/1409.0473*.
- Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention is All you Need. *ArXiv, abs/1706.03762*.
- Wu et al., Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arxiv 2016