Transformer

EECS 183/283a: Natural Language Processing

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i
- First, compute attention scores for each input hidden state using similarity between query (g_i) and keys ($\mathbf{h}$): $s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i
- First, compute attention scores for each input hidden state using similarity between query (g_i) and keys ($\mathbf{h}$): $s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$
- Then, take softmax of attention scores to get a distribution over keys:

$$\alpha_i = \text{softmax}(s_i) \in \Delta^{\mathbb{N}_{1:|\overline{x}|}}$$

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i
- First, compute attention scores for each input hidden state using similarity between query (g_i) and keys ($\mathbf{h}$): $s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$
- Then, take softmax of attention scores to get a distribution over keys:

$$\alpha_i = \text{softmax}(s_i) \in \Delta^{\mathbb{N}_{1:|\overline{x}|}}$$

- Finally, compute a weighted sum of values ($\mathbf{h}$) using this distribution

$$c_i = \sum_{j=1}^{|\overline{x}|} \alpha_{i,j} h_j \in \mathbb{R}^d$$

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i
- First, compute attention scores for each input hidden state using similarity between query (g_i) and keys ($\mathbf{h}$): $s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$
- Then, take softmax of attention scores to get a distribution over keys:

$$\alpha_i = \text{softmax}(s_i) \in \Delta^{\mathbb{N}_{1:|\overline{x}|}}$$

- Finally, compute a weighted sum of values ($\mathbf{h}$) using this distribution

$$c_i = \sum_{j=1}^{|\overline{x}|} \alpha_{i,j} h_j \in \mathbb{R}^d$$

- Use the weighted sum to predict the next word:

$$p(Y_{i+1} \mid \overline{x}, y_1, \ldots, y_i) = \text{softmax}(f(c_i, g_i))$$

Recap: Attention

- We encoded our input sequence into hidden states: $h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$
- Now we want to predict the next word y_{i+1}
- We have access to the previous decoder hidden state g_i
- First, compute attention scores for each input hidden state using similarity between query (g_i) and keys ($\mathbf{h}$): $s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$
- Then, take softmax of attention scores to get a distribution over keys:

$$\alpha_i = \text{softmax}(s_i) \in \Delta^{\mathbb{N}_{1:|\overline{x}|}}$$

- Finally, compute a weighted sum of values ($\mathbf{h}$) using this distribution

$$c_i = \sum_{j=1}^{|\overline{x}|} \alpha_{i,j} h_j \in \mathbb{R}^d$$

- Use the weighted sum to predict the next word:

$$p(Y_{i+1} \mid \overline{x}, y_1, \ldots, y_i) = \text{softmax}(f(c_i, g_i))$$

- Use the weighted sum to update the decoder hidden state: $g_i = g(g_{i-1}, c_{i-1}, y_i)$

Recap: Self-Attention

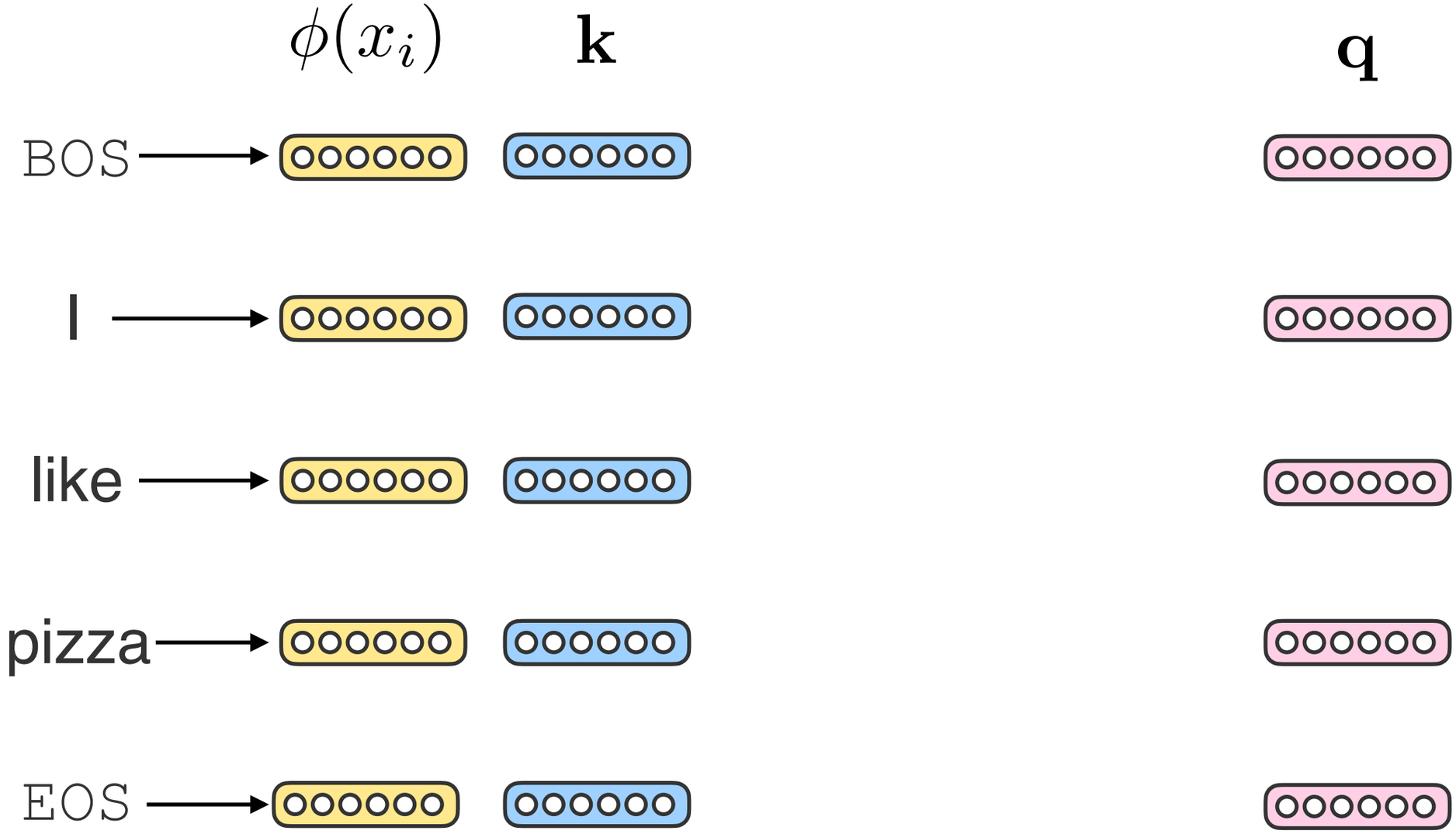
- For each token generated at index *i*, we have:
  - A key: $k_i = K\phi(x_i) \in \mathbb{R}^d \qquad K \in \mathbb{R}^{d \times d}$
  - A value: $v_i = V\phi(x_i) \in \mathbb{R}^d \quad V \in \mathbb{R}^{d \times d}$

- When trying to generate our next token at index *j*, we need a query: $q_j = Q\phi(x_j) \in \mathbb{R}^d \quad Q \in \mathbb{R}^{d \times d}$
  - Scores over keys: $s_{ji} = q_j^\intercal k_i$
  - Attention distribution over keys: $\alpha_{ji} = \frac{\exp(s_{ji})}{\sum_{i'=1}^{j} \exp(s_{ji'})}$
  - Weighted sum of values: $c_j = \sum_{i=1}^{j} \alpha_{ji} v_i$

$$x_{j+1} \sim p(X_{j+1} \mid x_1, \ldots, x_j) = \text{softmax}(f(c_j, \phi(x_j))$$

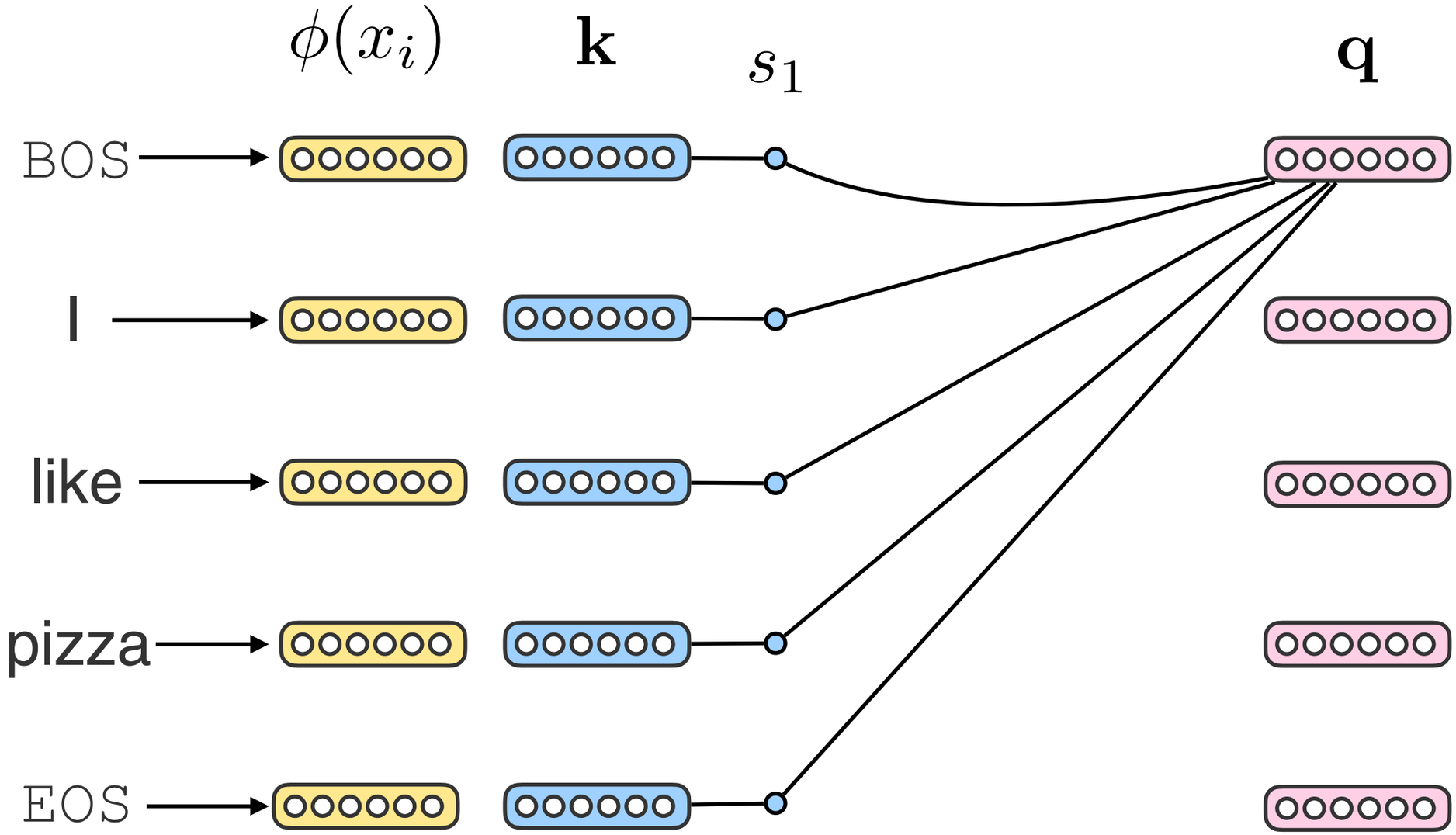
Self-Attention in Encoders

Encode



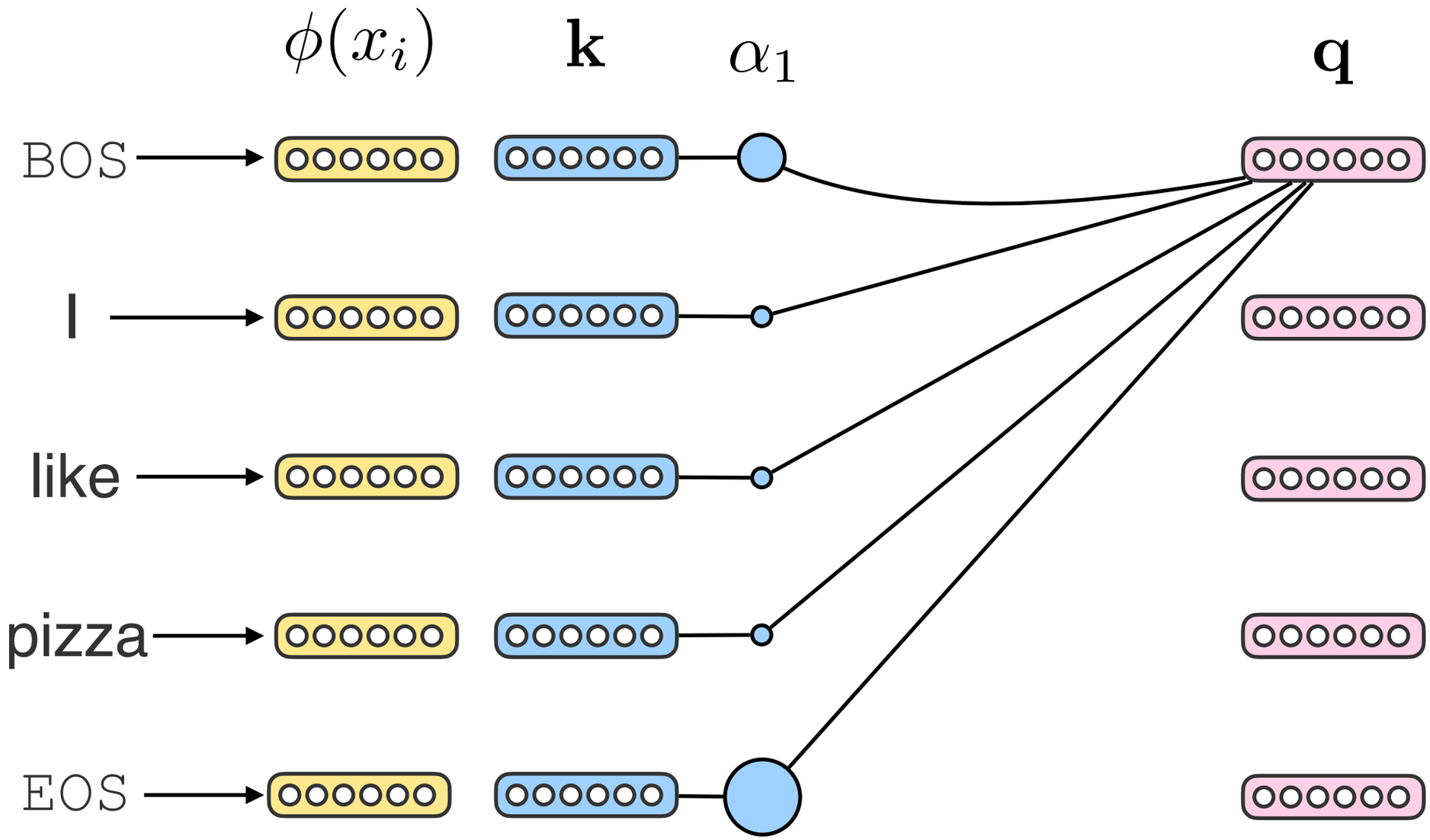
Self-Attention in Encoders

Encode



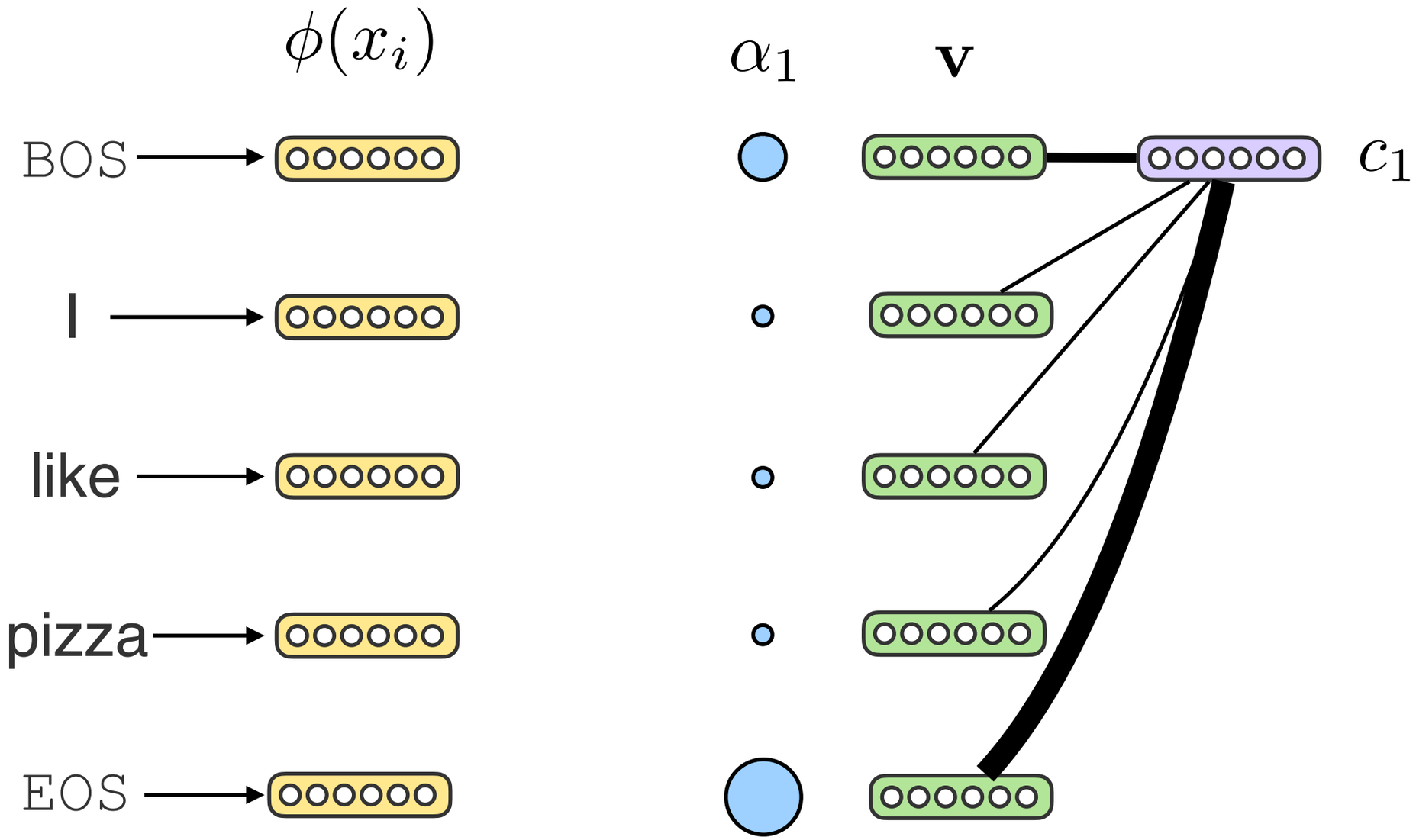
Self-Attention in Encoders

Encode



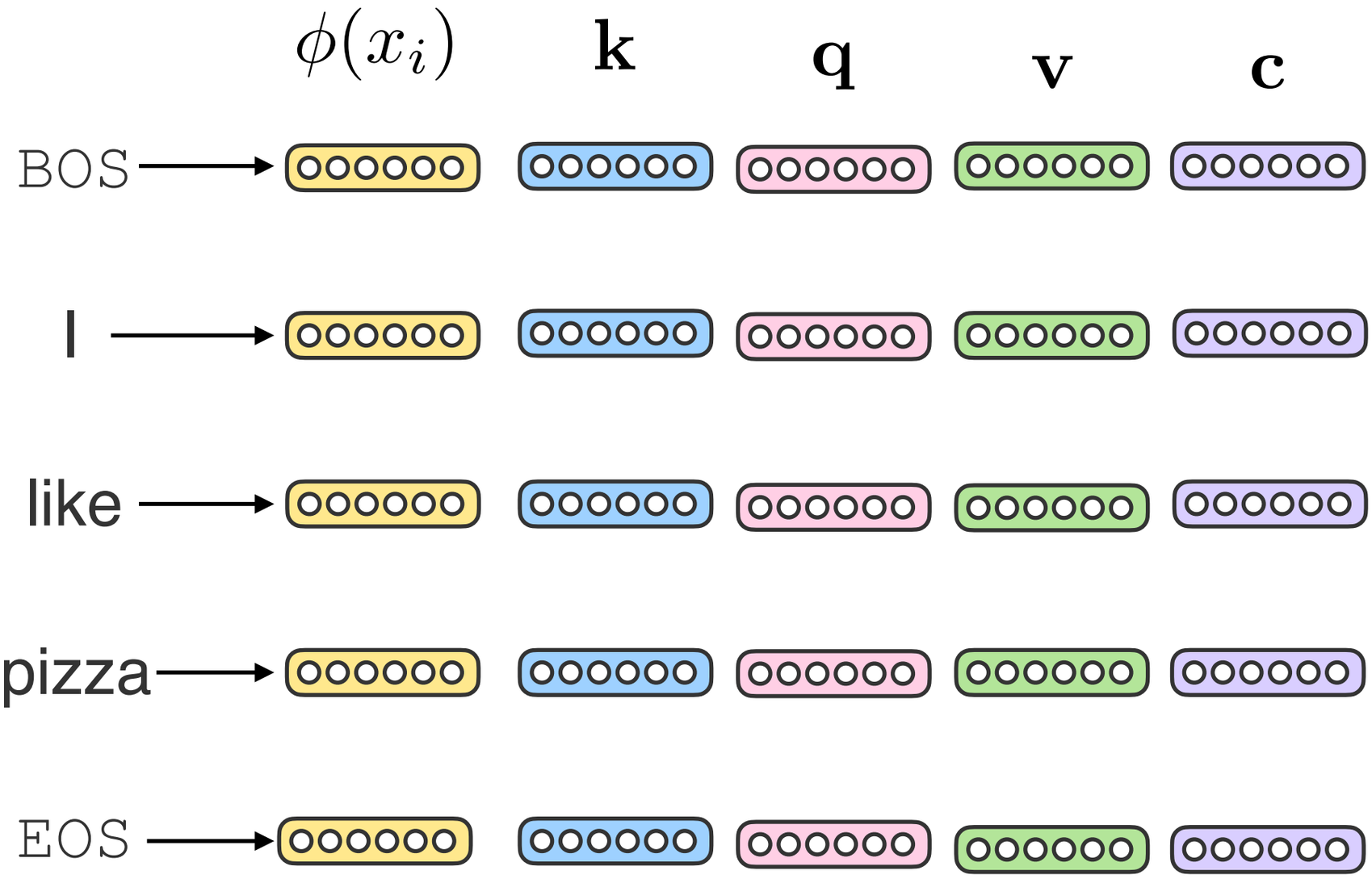
Self-Attention in Encoders

Encode



Self-Attention in Encoders

Encode



$$\mathbf{s} = \mathbf{q}^\intercal \mathbf{k} \in \mathbb{R}^{|\overline{x}| \times |\overline{x}|}$$

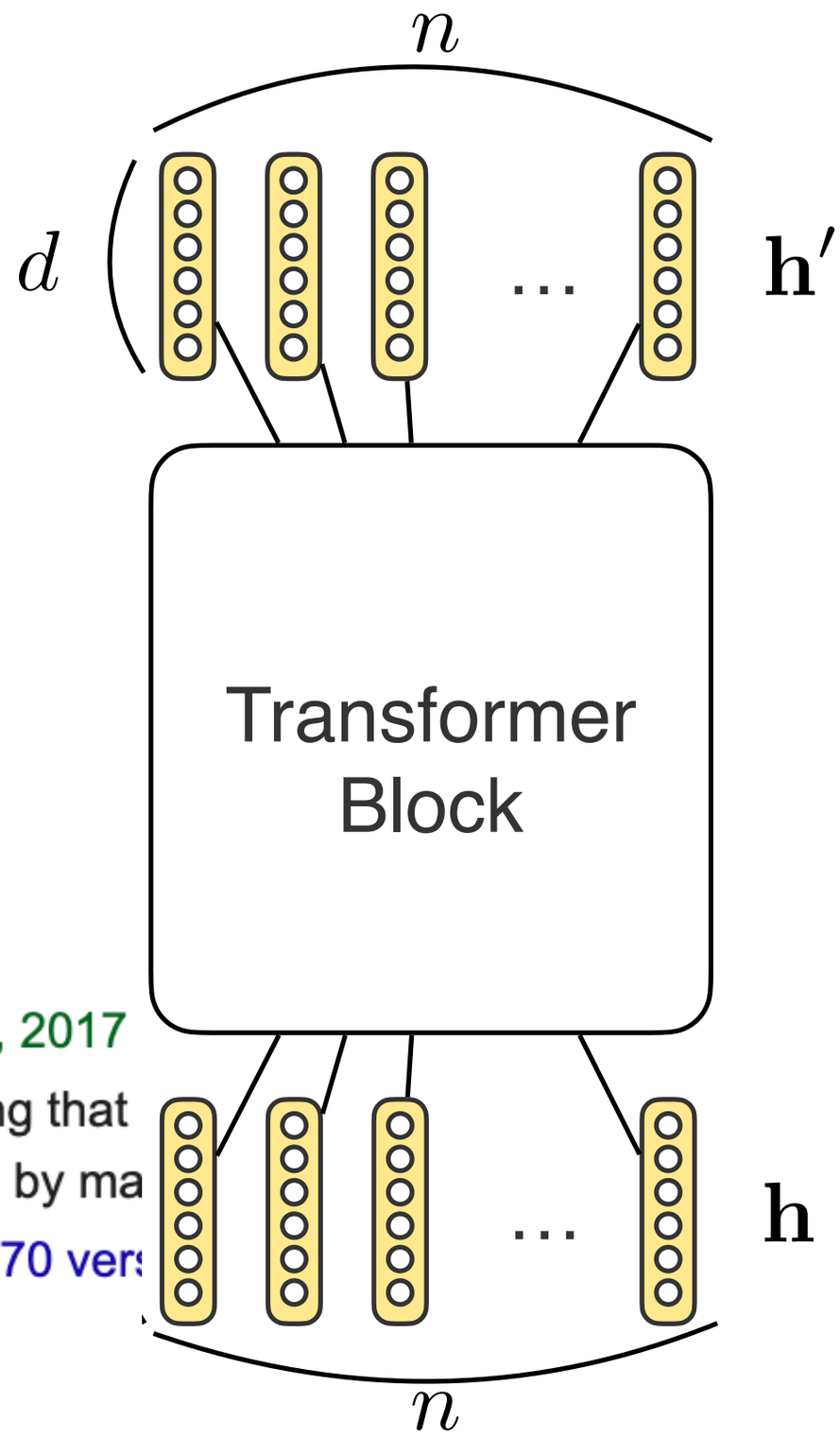
$$\alpha = \mathrm{softmax}_{\mathrm{dim}=1}(\mathbf{s}) \in \{\Delta^{\mathbb{N}_{1:|\overline{x}|}}\}^{|\overline{x}|}$$

$$\mathbf{c} = \alpha \mathbf{v} \in \mathbb{R}^{d \times |\overline{x}|}$$

We can get context-sensitive representations of each input token **in parallel**!

Building a Transformer Block

- A transformer block takes as input a sequence of input vectors $\mathbf{h} \in \mathbb{R}^{d \times n}$
- It generates a sequence of output vectors $\mathbf{h}' \in \mathbb{R}^{d \times n}$



Attention is **all you need**

A Vaswani, N Shazeer, N Parmar… - Advances in neural …, 2017

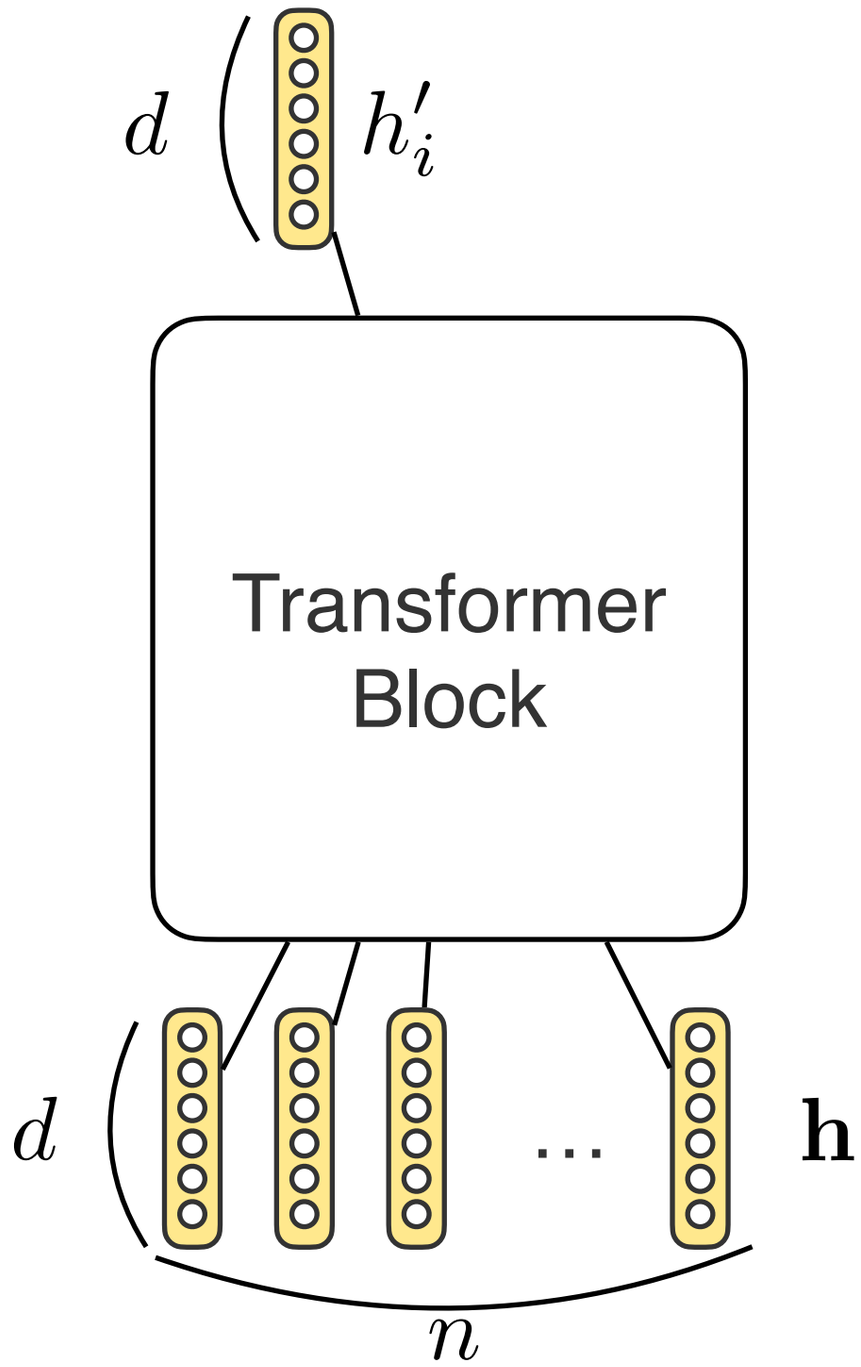
… to attend to **all** positions in the decoder up to and including that

… **We** implement this inside of scaled dot-product **attention** by ma

☆ Save 99 Cite Cited by 196986 Related articles All 70 vers

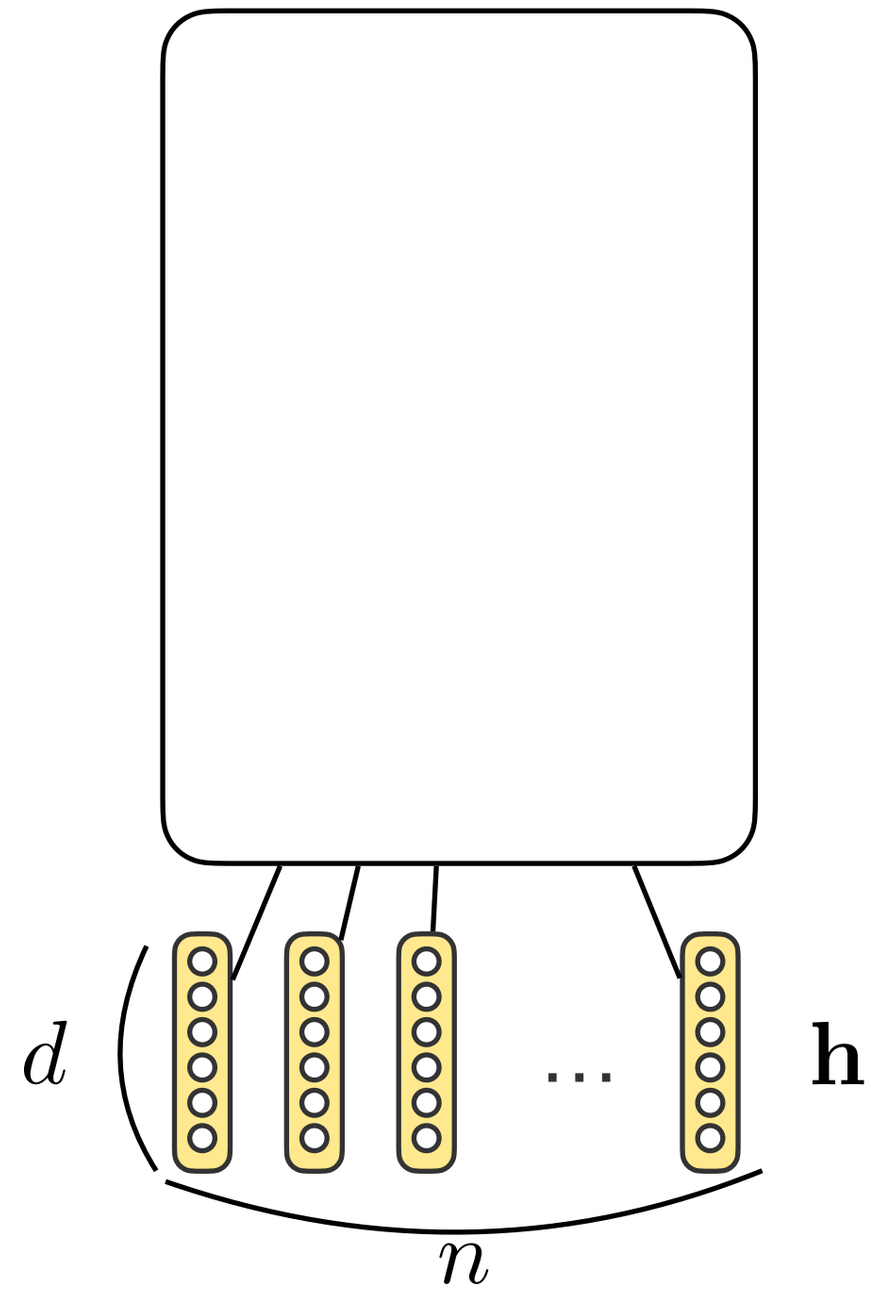
Building a Transformer Block

- A transformer block takes as input a sequence of input vectors $\mathbf{h} \in \mathbb{R}^{d \times n}$
- It generates a sequence of output vectors $\mathbf{h}' \in \mathbb{R}^{d \times n}$
- For now, let's focus on how it computes the output vector corresponding to the input at index *i*, $h'_i \in \mathbb{R}^d$



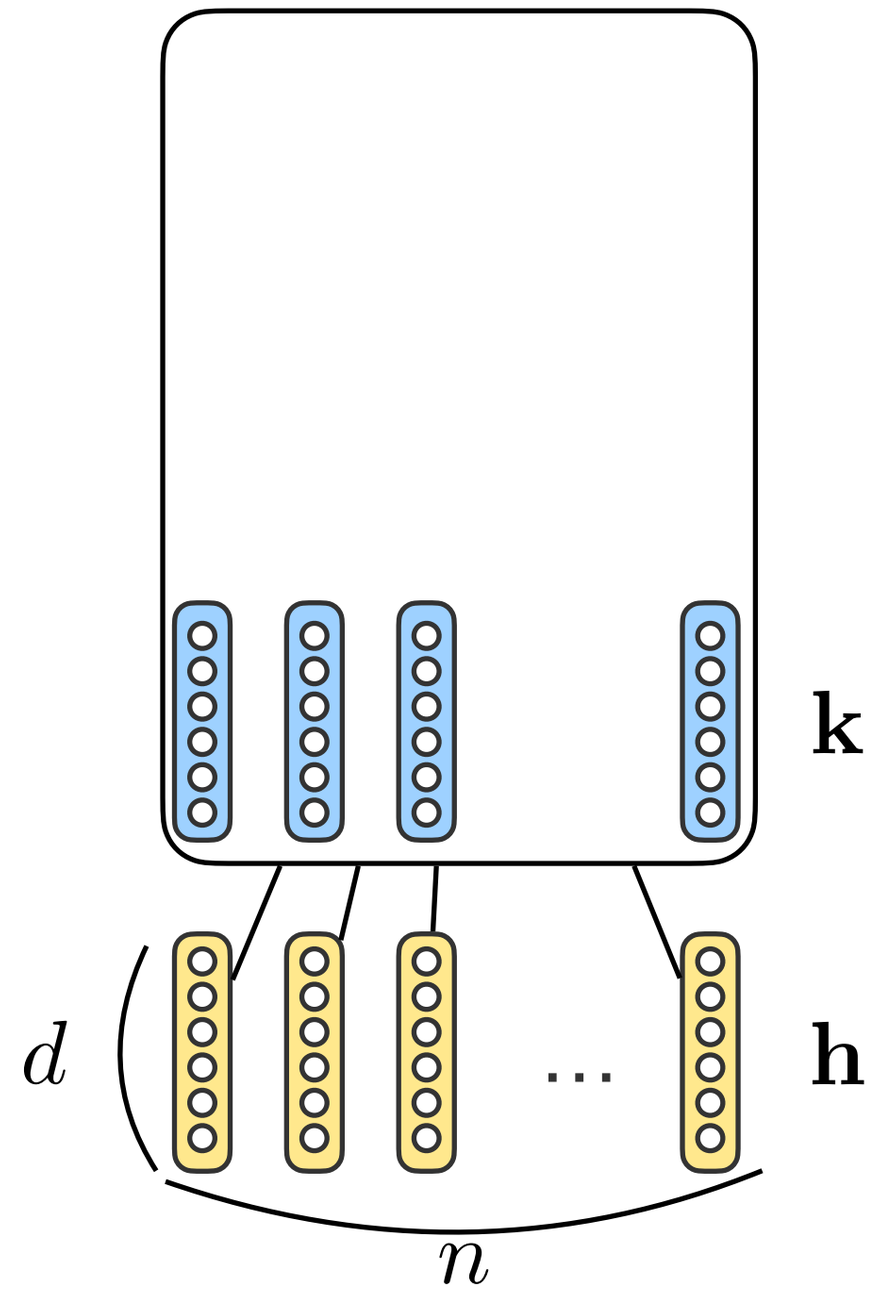
Self-Attention

1. Compute attention over input vectors



Self-Attention

1. Compute attention over input vectors
$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$

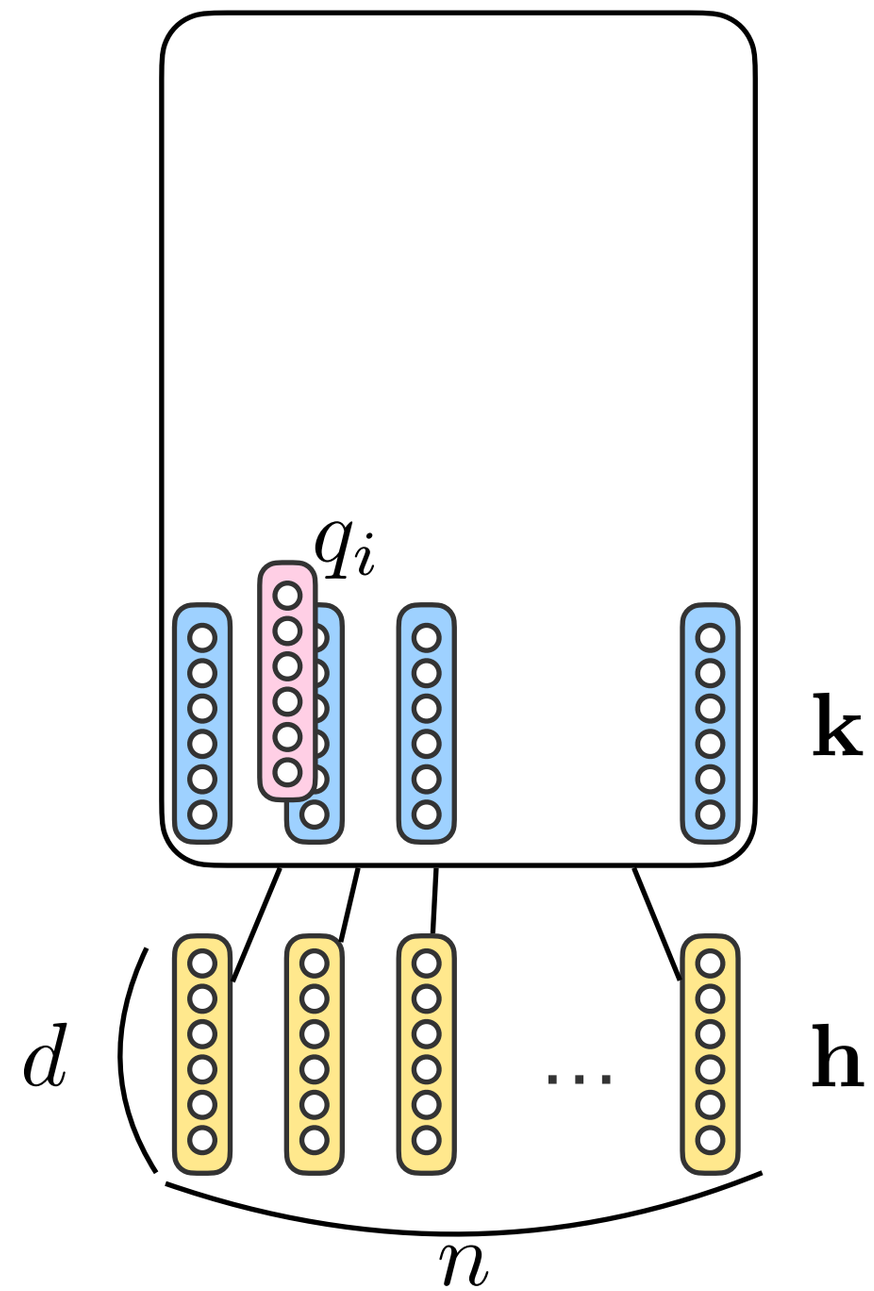


Self-Attention

1. Compute attention over input vectors

$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i = Qh_i \in \mathbb{R}^{d_k}$$



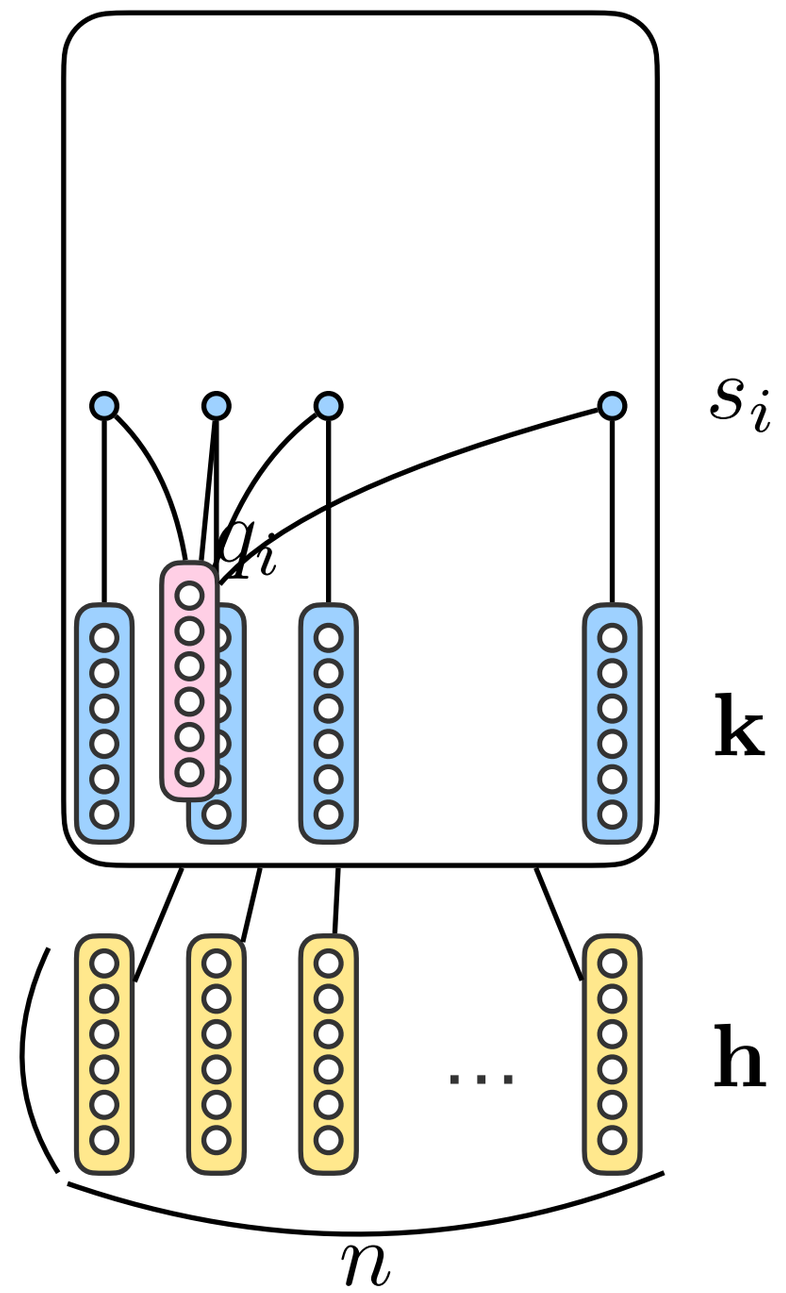
Self-Attention

1. Compute attention over input vectors

$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i = Qh_i \in \mathbb{R}^{d_k}$$

$$s_i = \frac{q_i^\intercal \mathbf{k}}{\sqrt{d_k}} \in \mathbb{R}^n$$



Scaling by vector dimension ensures the values don't become way too large.

Self-Attention

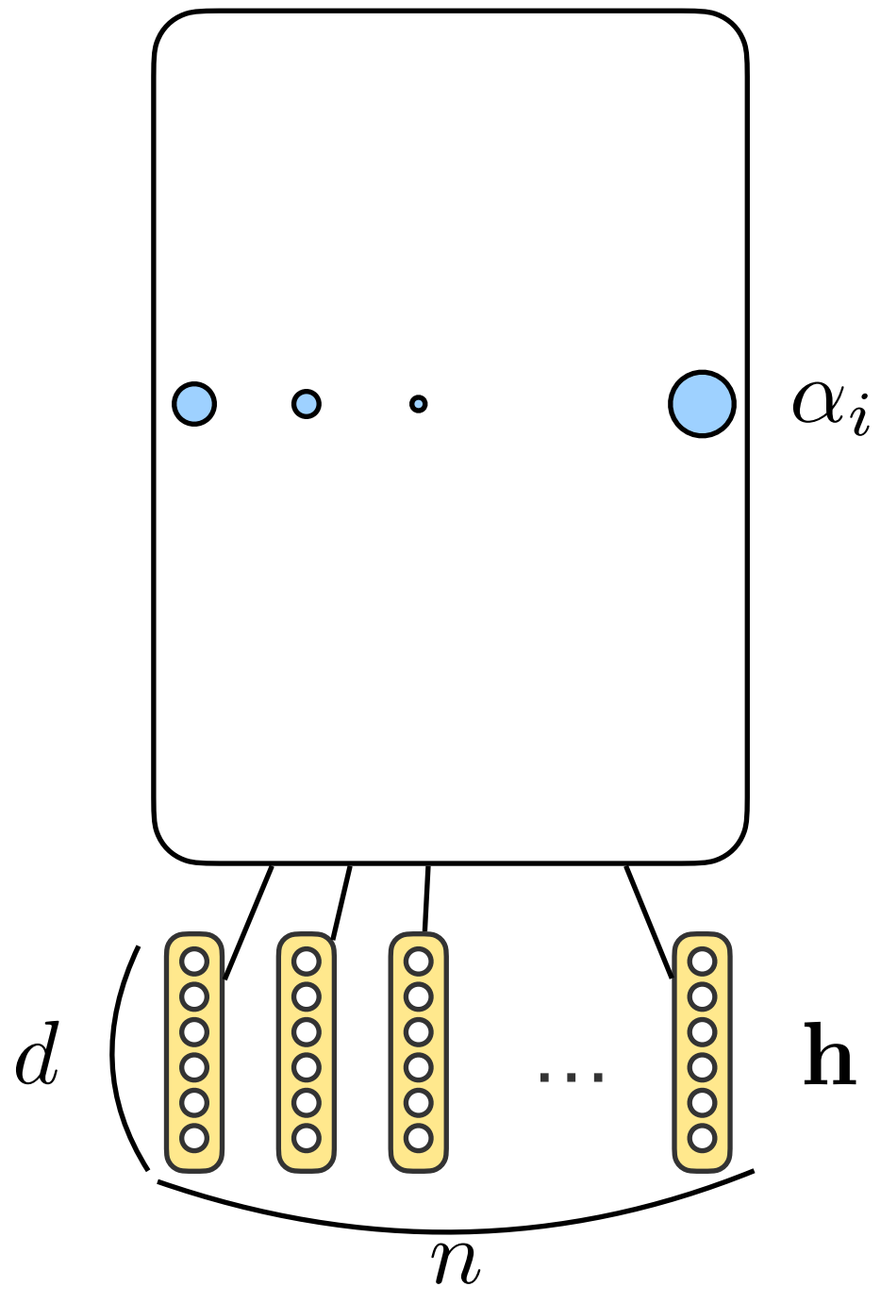
1. Compute attention over input vectors

$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i = Qh_i \in \mathbb{R}^{d_k}$$

$$s_i = \frac{q_i^\intercal \mathbf{k}}{\sqrt{d_k}} \in \mathbb{R}^n$$

$$\alpha_i = \text{softmax}\left(s_i\right) \in \Delta^{\mathbb{N}_{1:n}}$$



Self-Attention

1. Compute attention over input vectors

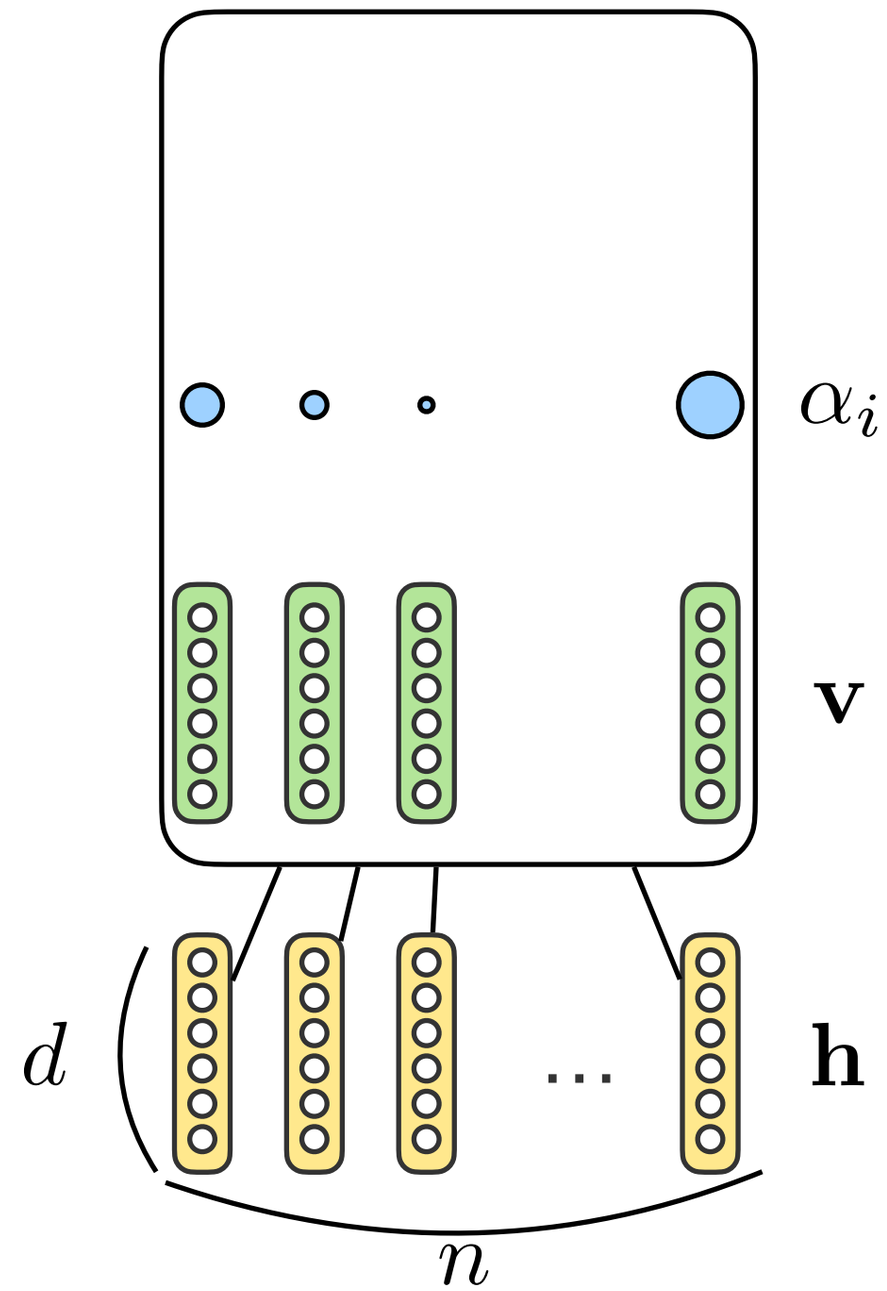
$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i = Qh_i \in \mathbb{R}^{d_k}$$

$$s_i = \frac{q_i^\intercal \mathbf{k}}{\sqrt{d_k}} \in \mathbb{R}^n$$

$$\alpha_i = \text{softmax}\left(s_i\right) \in \Delta^{\mathbb{N}_{1:n}}$$

$$\mathbf{v} = V\mathbf{h} \in \mathbb{R}^{d_k \times n}$$



Self-Attention

1. Compute attention over input vectors

$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

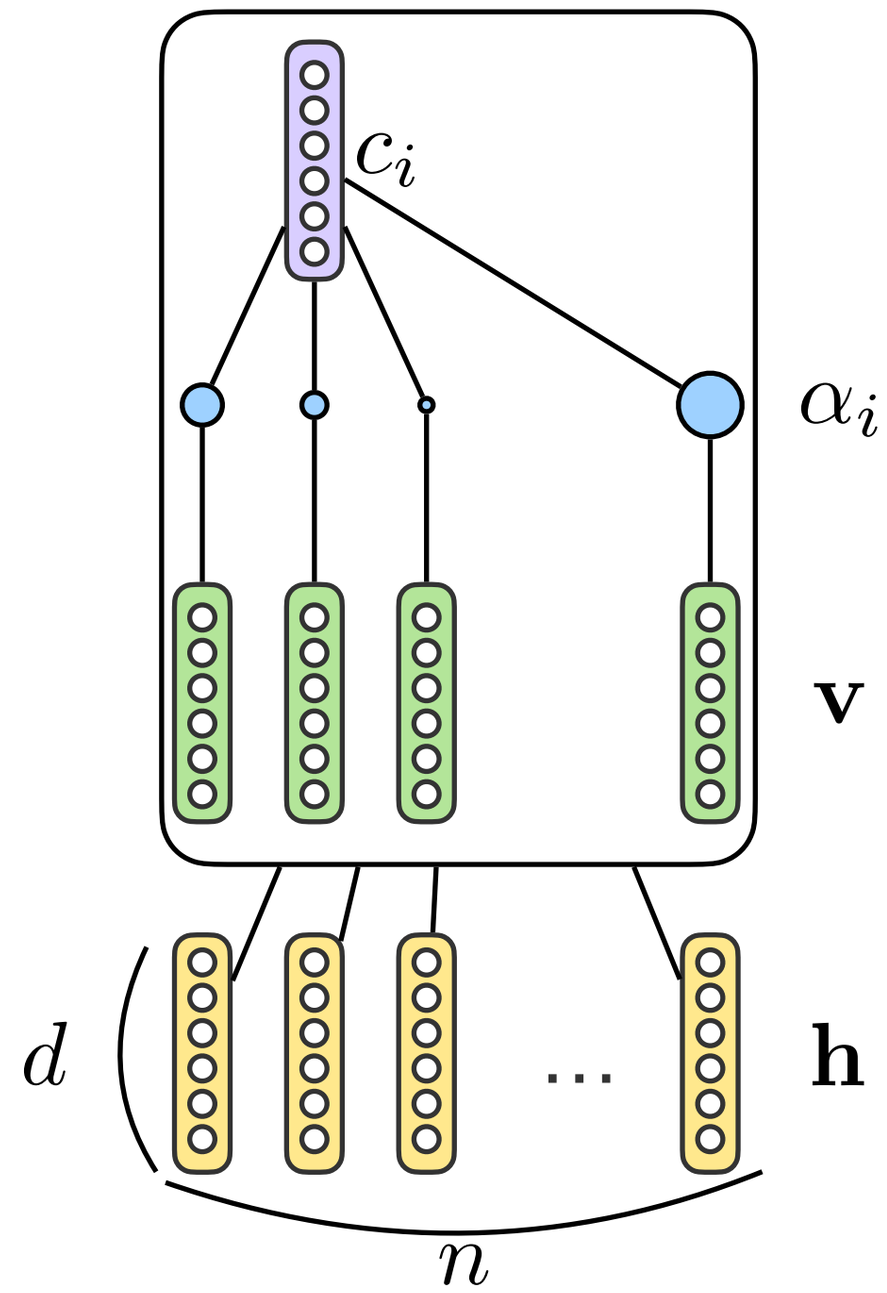
$$q_i = Qh_i \in \mathbb{R}^{d_k}$$

$$s_i = \frac{q_i^\intercal \mathbf{k}}{\sqrt{d_k}} \in \mathbb{R}^n$$

$$\alpha_i = \text{softmax}\left(s_i\right) \in \Delta^{\mathbb{N}_{1:n}}$$

$$\mathbf{v} = V\underset{n}{\mathbf{h}} \in \mathbb{R}^{d_k \times n}$$

$$c_i = \sum_{i'=1} \alpha_i v_{i'}$$



Self-Attention

1. Compute attention over input vectors

$$\mathbf{k} = K\mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i = Qh_i \in \mathbb{R}^{d_k}$$

$$s_i = \frac{q_i^\intercal \mathbf{k}}{\sqrt{d_k}} \in \mathbb{R}^n$$

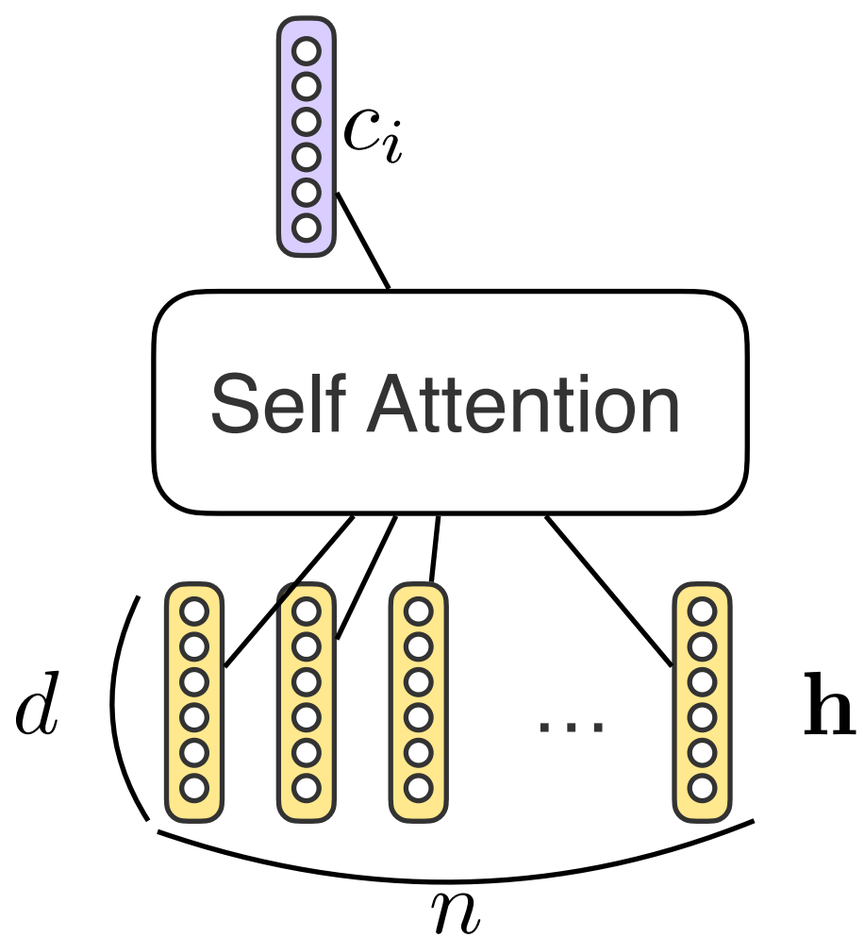
$$\alpha_i = \text{softmax}\left(s_i\right) \in \Delta^{\mathbb{N}_{1:n}}$$

$$\mathbf{v} = V\underset{n}{\mathbf{h}} \in \mathbb{R}^{d_k \times n}$$

$$c_i = \sum_{i'=1} \alpha_i v_{i'}$$

$$c_i = \text{SelfAttention}(\mathbf{h})_i$$

$$\theta_{\text{SelfAttention}} = \{K, Q, V\}$$



Multi-Head Attention

1. Compute attention over input vectors

$$\mathbf{k}^{(j)} = K_j \mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$q_i^{(j)} = Q_j h_i \in \mathbb{R}^{d_k}$$

$$s_i^{(j)} = \frac{q_i^{(j)\intercal} \mathbf{k}^{(j)}}{\sqrt{d_k}} \in \mathbb{R}^n$$

$$\alpha_i^{(j)} = \text{softmax}\left(s_i^{(j)}\right) \in \Delta^{\mathbb{N}_{1:n}}$$

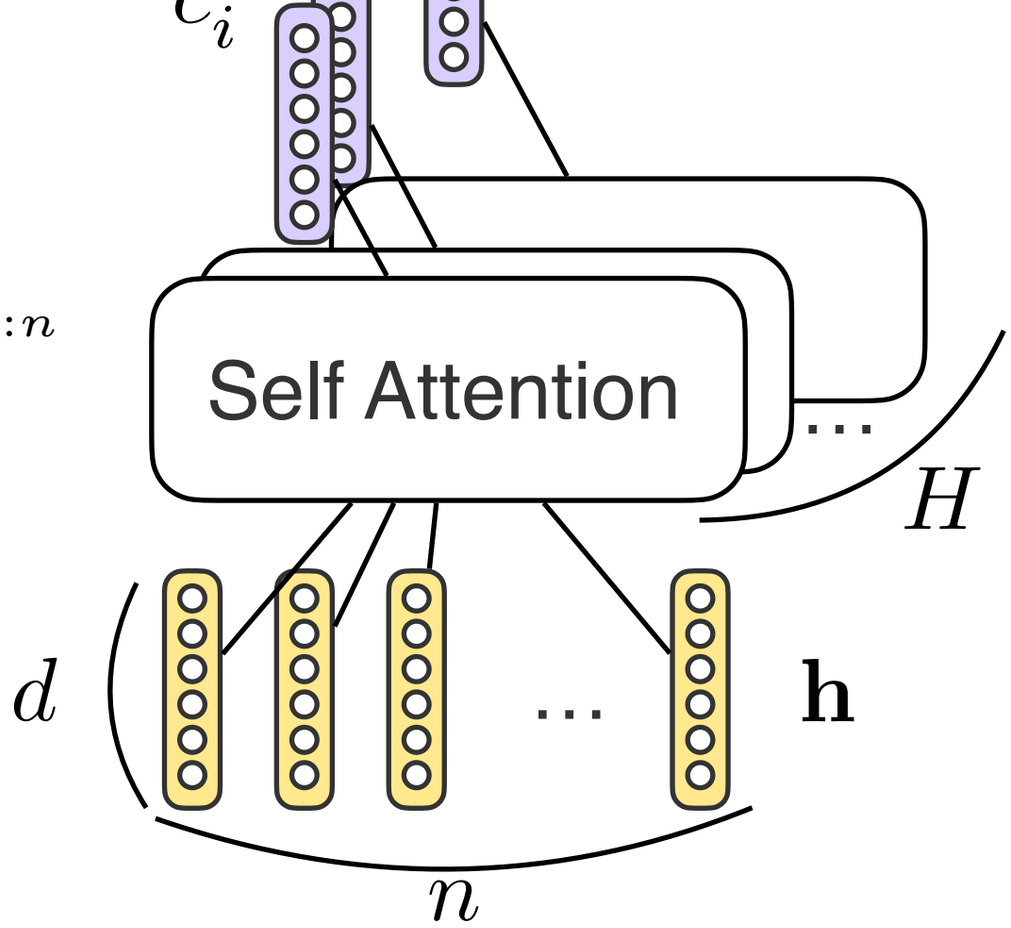
$$\mathbf{v}^{(j)} = V_j \mathbf{h} \in \mathbb{R}^{d_k \times n}$$

$$c_i^{(j)} = \sum_{i'=1} \alpha_i^{(j)} v_{i'}^{(j)}$$

$$c_i^{(j)} = \text{SelfAttention}_j(\mathbf{h})_i$$

Multi-Head Attention

$$\theta_{\text{SelfAttention}_j} = \{K_j, Q_j, V_j\}$$



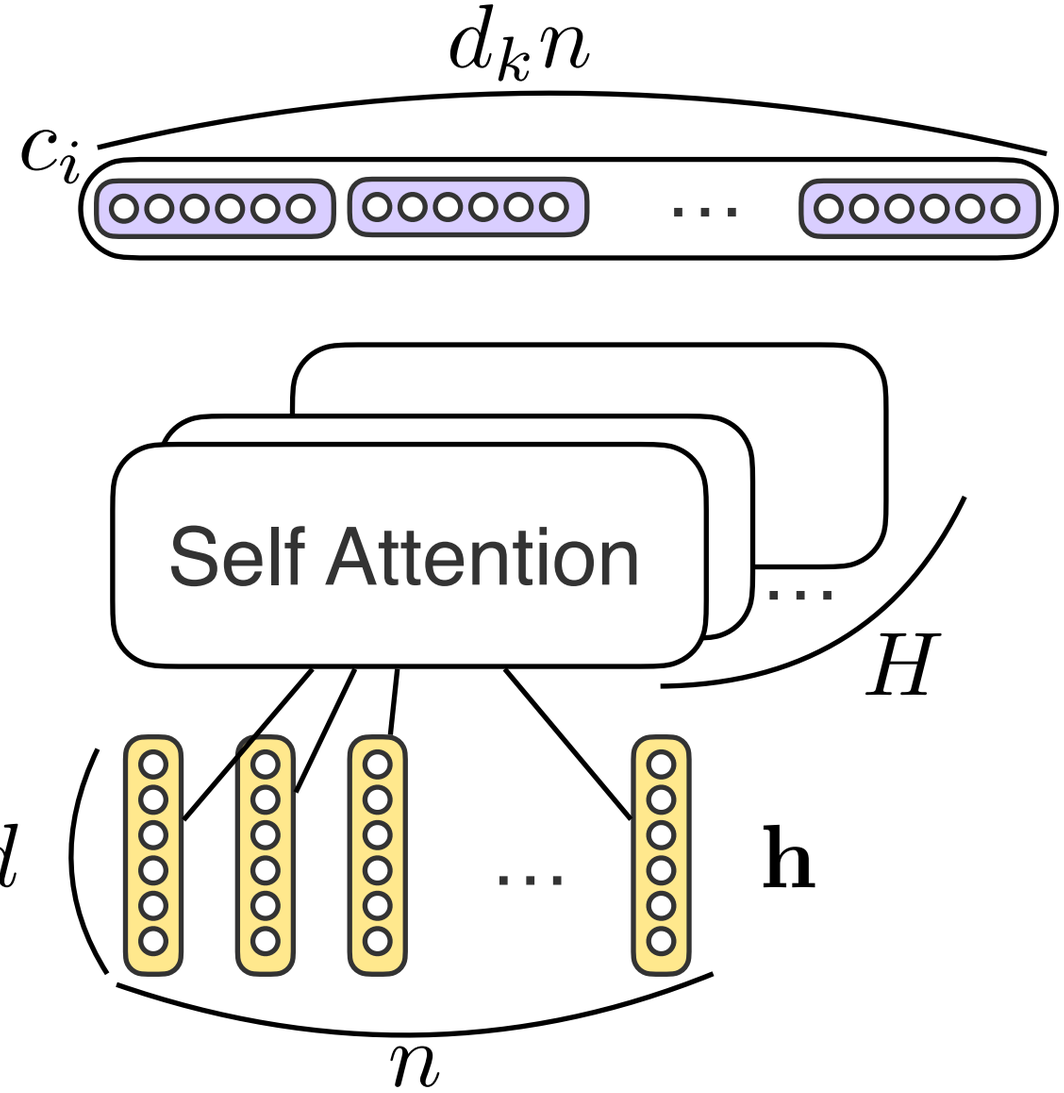
Multi-Head Attention

Multi-Head Attention

1. Compute attention over input vectors

$$c_i^{(j)} = \text{SelfAttention}_j(\mathbf{h})_i$$

$$c_i = \left[c_i^{(1)}; \ldots; c_i^{(H)}\right]$$



Multi-Head Attention

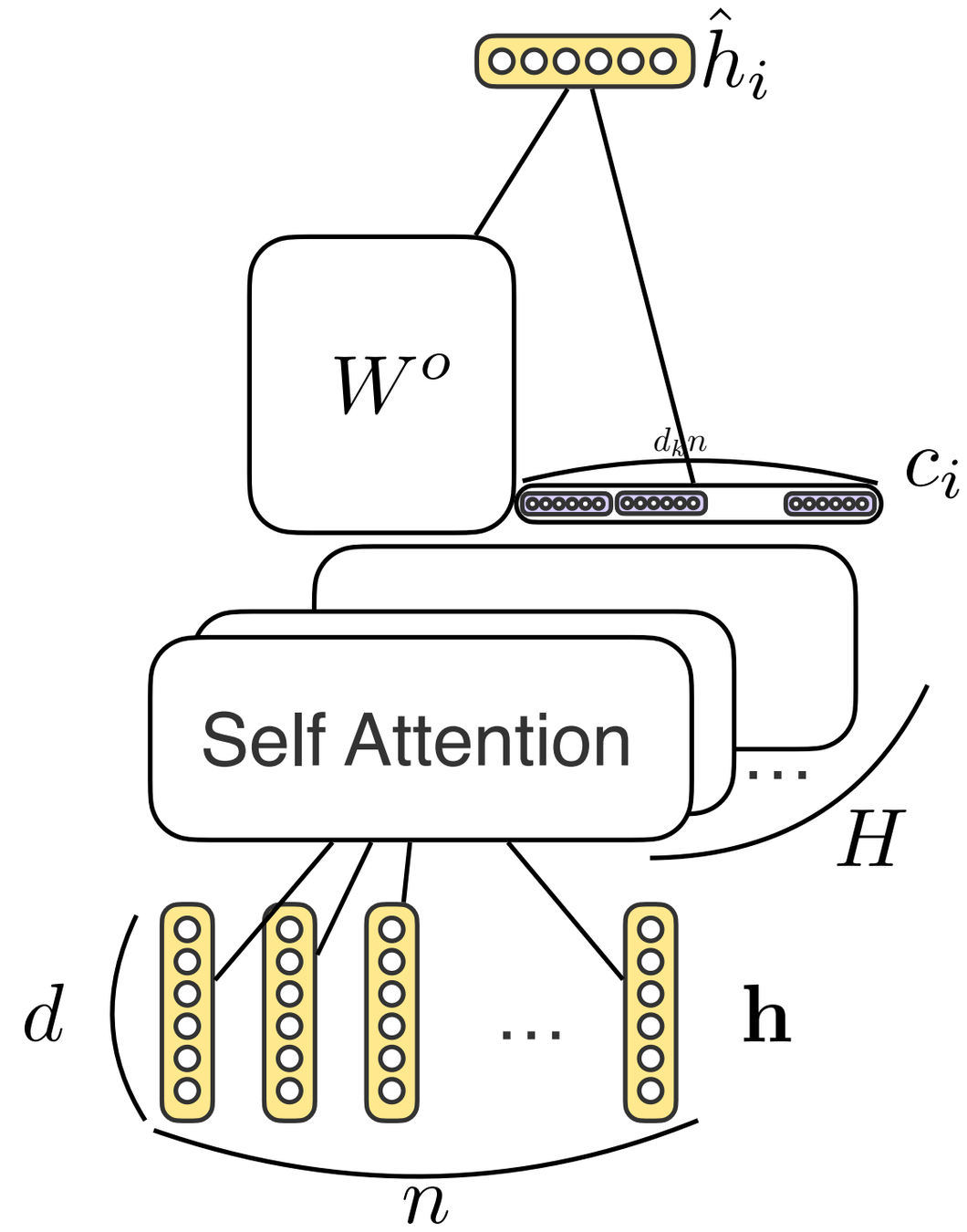
1. Compute attention over input vectors

$$c_i^{(j)} = \text{SelfAttention}_j(\mathbf{h})_i$$

$$c_i = \left[c_i^{(1)}; \ldots; c_i^{(H)}\right]$$

$$\hat{h}_i = W^o c_i$$

Multi-Head Attention



Multi-Head Attention

Multi-Head Attention

1. Compute attention over input vectors

$$c_i^{(j)} = \text{SelfAttention}_j(\mathbf{h})_i$$

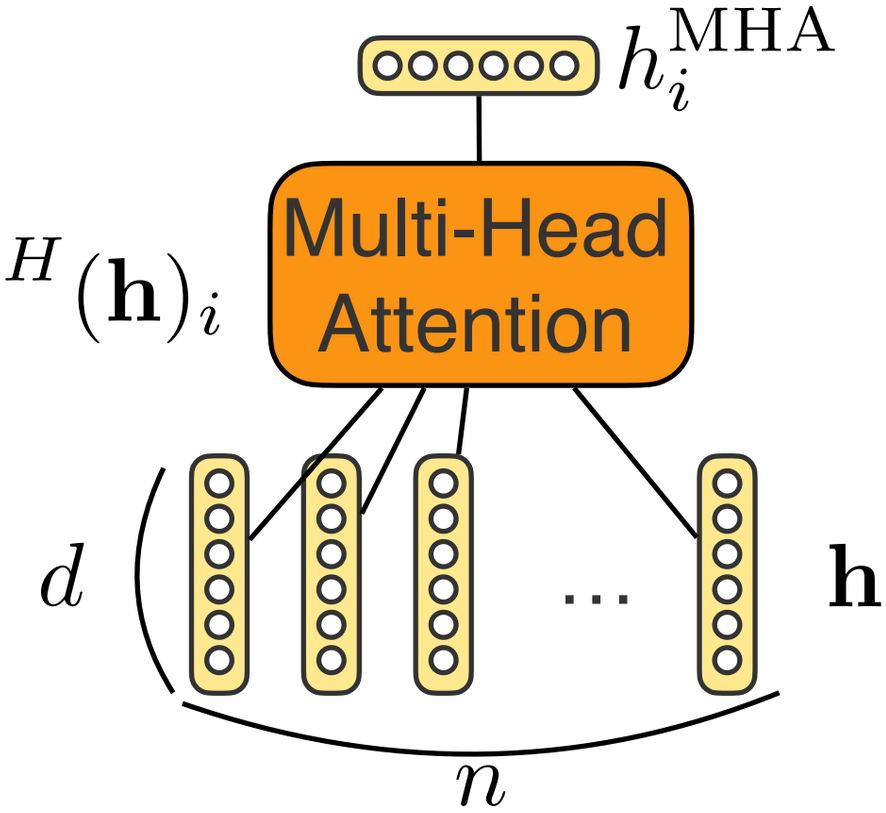
$$c_i = \left[c_i^{(1)}; \ldots; c_i^{(H)}\right]$$

$$\hat{h}_i = W^o c_i$$

$$h_i^{\text{MHA}} = \text{MultiHeadAttention}^H(\mathbf{h})_i$$

$$\theta_{\text{MultiHeadAttention}^H} = \{W^o\} \cup \{K_j, Q_j, V_j\}_{j=1}^H$$

Multi-headed attention allows us to look at multiple places in the input sequence at once.



Layer Normalization

Residual connection allows for smoother gradients.

Layer normalization cuts down on uninformative variation in activations, speeding up training.

1. Compute attention over input vectors

$$h_i^{\text{MHA}} = \text{MultiHeadAttention}^H(\mathbf{h})_i$$

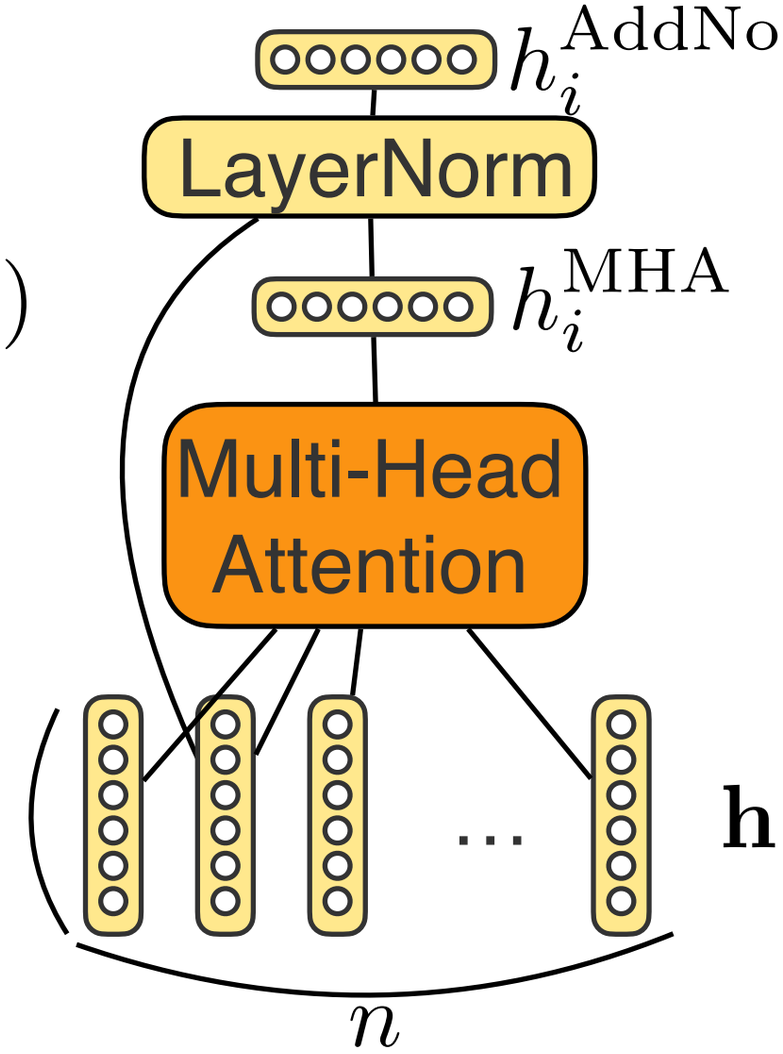
2. Add and norm

$$h_i^{\text{AddNorm}_1} = \text{LayerNorm}(h_i + h_i^{\text{MHA}})$$

$$\text{LayerNorm} : \mathbb{R}^d \to \mathbb{R}^d$$

$$\text{LayerNorm}(x) = \frac{g}{\sigma(x)}(x - \mu(x))$$

$$\mu(x) = \frac{1}{d}\sum_{i=1}^{d} x_i \quad \sigma(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$



Feedforward Network

1. Compute attention over input vectors

$$h_i^{\text{MHA}} = \text{MultiHeadAttention}^H(\mathbf{h})_i$$

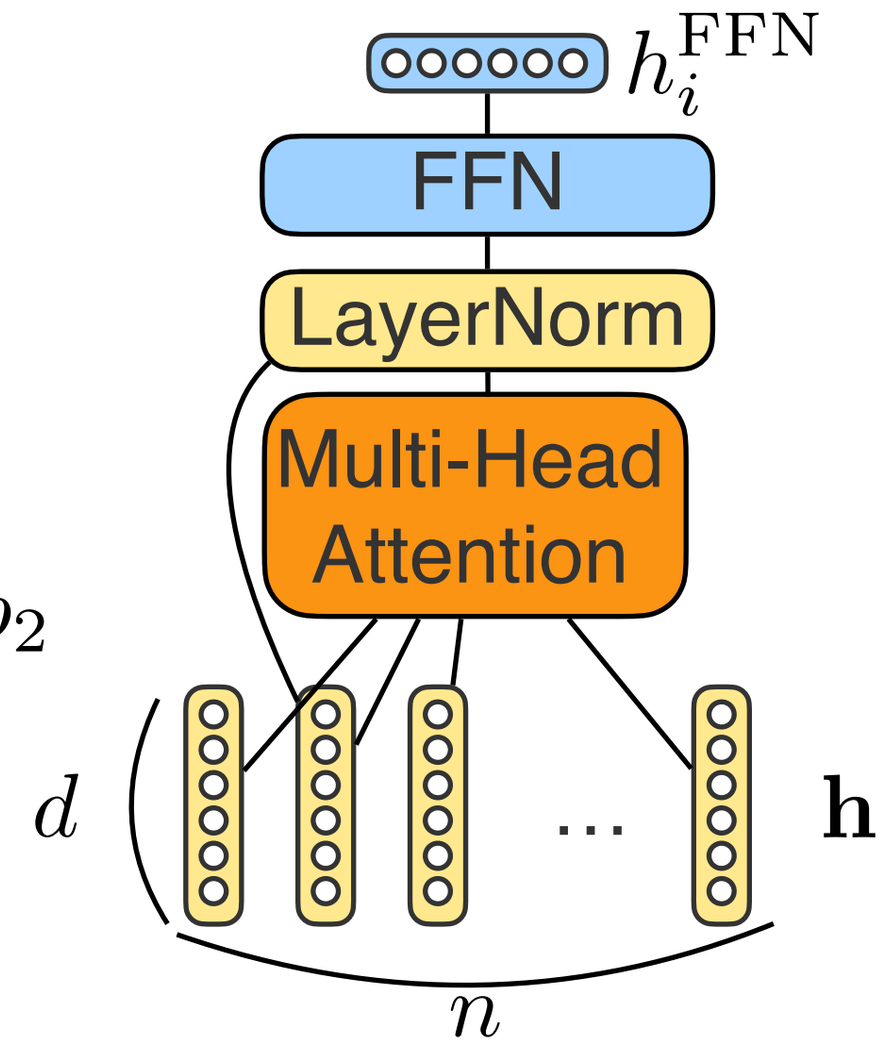
2. Add and norm

$$h_i^{\text{AddNorm}_1} = \text{LayerNorm}(h_i + h_i^{\text{MHA}})$$

3. Feedforward layer

$$h_i^{\text{FFN}} = \text{ReLU}(h_i^{\text{AddNorm}_1} W_1 + b_1)W_2 + b_2$$

Here is a nonlinearity that makes learning easier!



More Layer Normalization

1. Compute attention over input vectors

$$h_i^{\text{MHA}} = \text{MultiHeadAttention}^H(\mathbf{h})_i$$

2. Add and norm

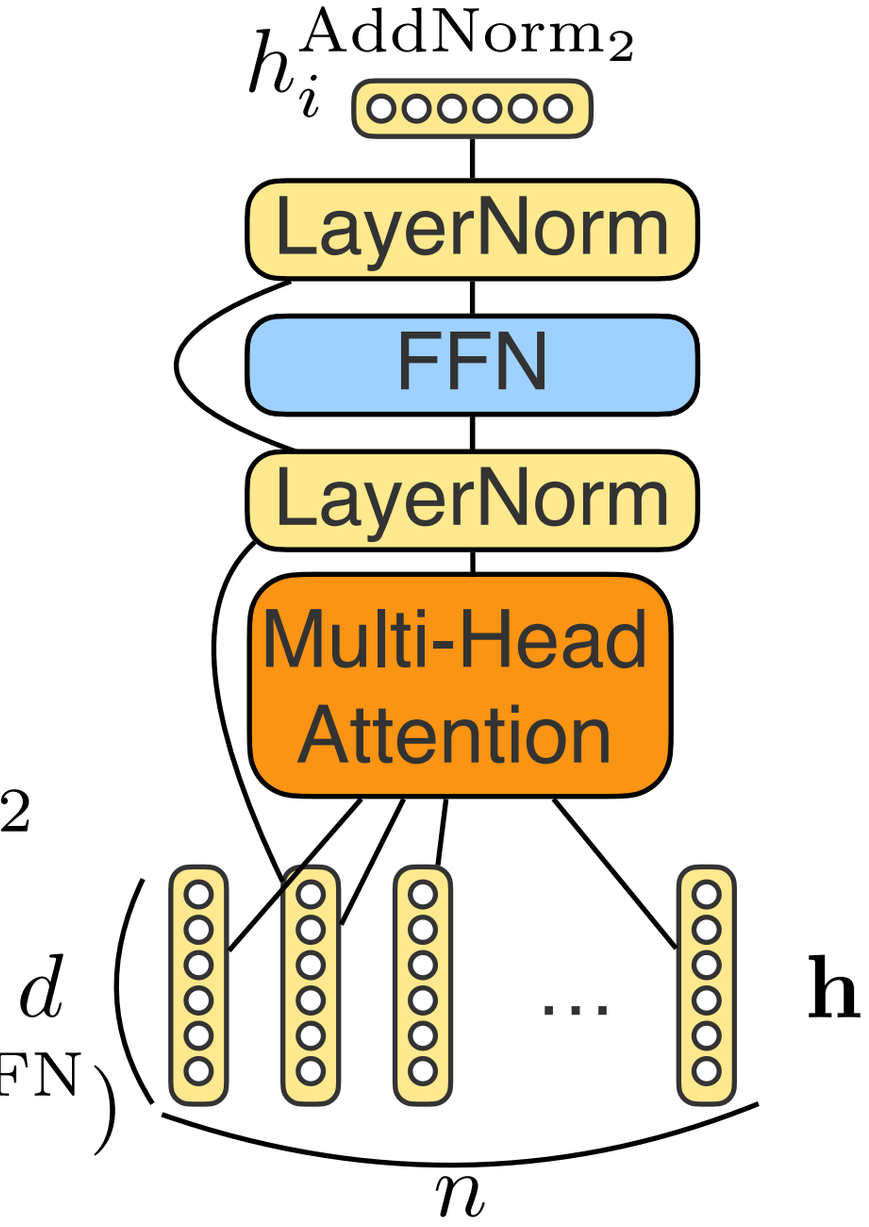
$$h_i^{\text{AddNorm}_1} = \text{LayerNorm}(h_i + h_i^{\text{MHA}})$$

3. Feedforward layer

$$h_i^{\text{FFN}} = \text{ReLU}(h_i^{\text{AddNorm}_1} W_1 + b_1)W_2 + b_2$$

4. Another add and norm

$$h_i^{\text{AddNorm}_2} = \text{LayerNorm}(h_i^{\text{AddNorm}_1} + h_i^{\text{FFN}})$$



One Transformer Block

1. Compute attention over input vectors

$$h_i^{\text{MHA}} = \text{MultiHeadAttention}^H(\mathbf{h})_i$$

2. Add and norm

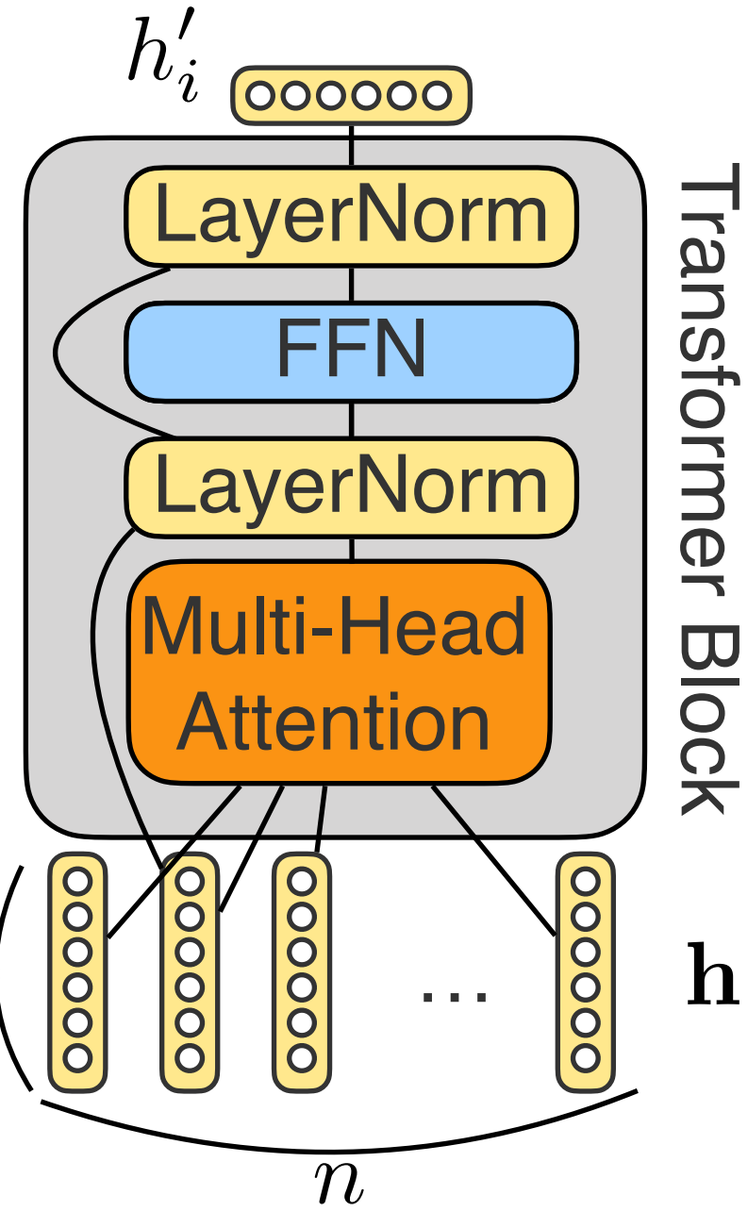
$$h_i^{\text{AddNorm}_1} = \text{LayerNorm}(h_i + h_i^{\text{MHA}})$$

3. Feedforward layer

$$h_i^{\text{FFN}} = \text{ReLU}(h_i^{\text{AddNorm}_1} W_1 + b_1) W_2 + b_2$$

4. Another add and norm

$$h_i' = \text{LayerNorm}(h_i^{\text{AddNorm}_1} + h_i^{\text{FFN}})$$



One Transformer Block

$$\mathbf{h}^{\text{MHA}} = \text{MultiHeadAttention}^{H}(\mathbf{h})$$

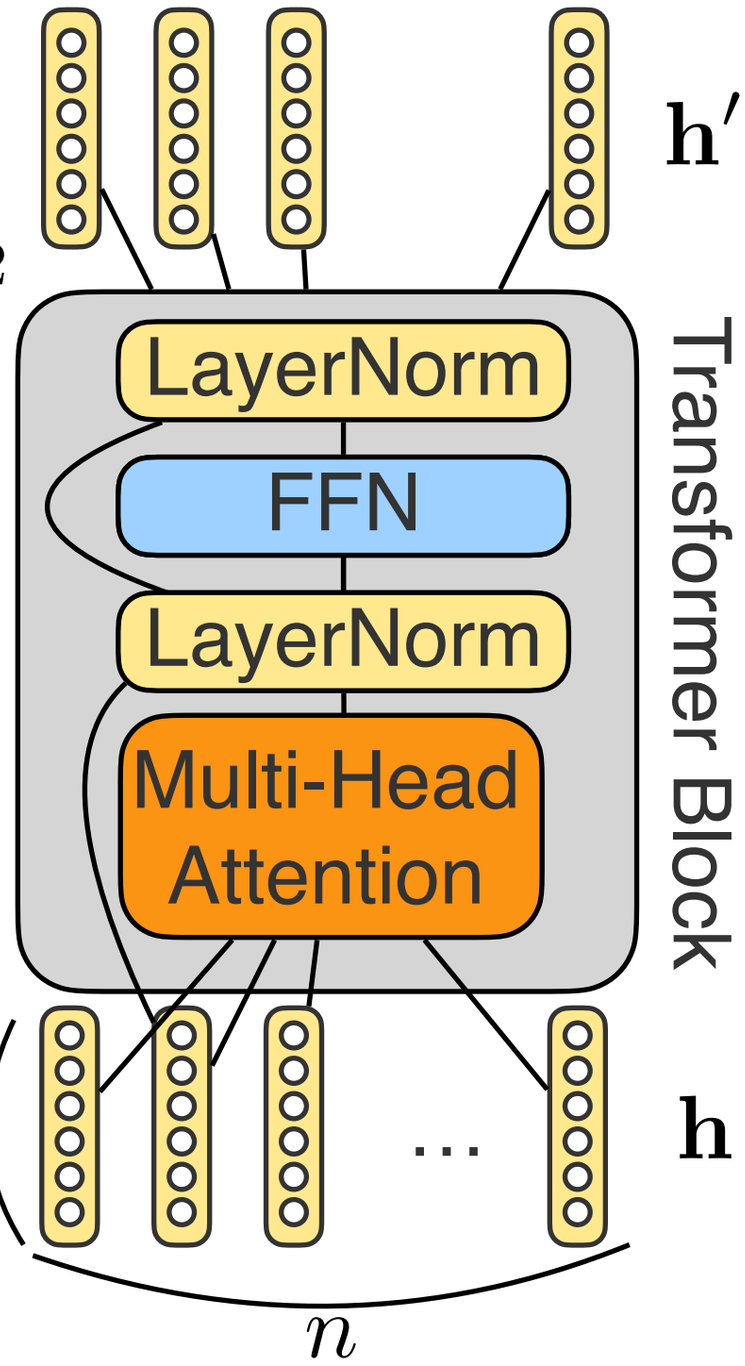
$$\mathbf{h}^{\text{AddNorm}} = \text{LayerNorm}(\mathbf{h} + \mathbf{h}^{\text{MHA}})$$

$$\mathbf{h}_i^{\text{FFN}} = \text{ReLU}(\mathbf{h}^{\text{AddNorm}} W_1 + b_1) W_2 + b_2$$

$$\mathbf{h}' = \text{LayerNorm}(\mathbf{h}^{\text{AddNorm}} + \mathbf{h}^{\text{FFN}})$$

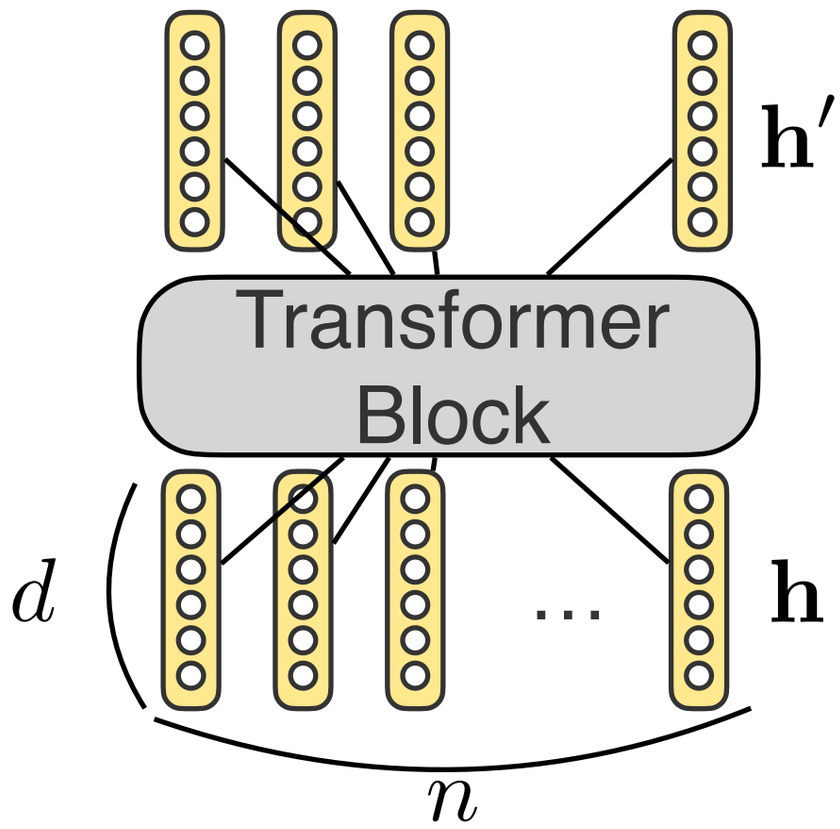
$$\mathbf{h}' = \text{TransformerBlock}(\mathbf{h})$$

$$\theta_{\text{TransformerBlock}} = \theta_{\text{MultiHeadAttention}^{H}} \cup \{W_1, W_2, b_1, b_2\}$$



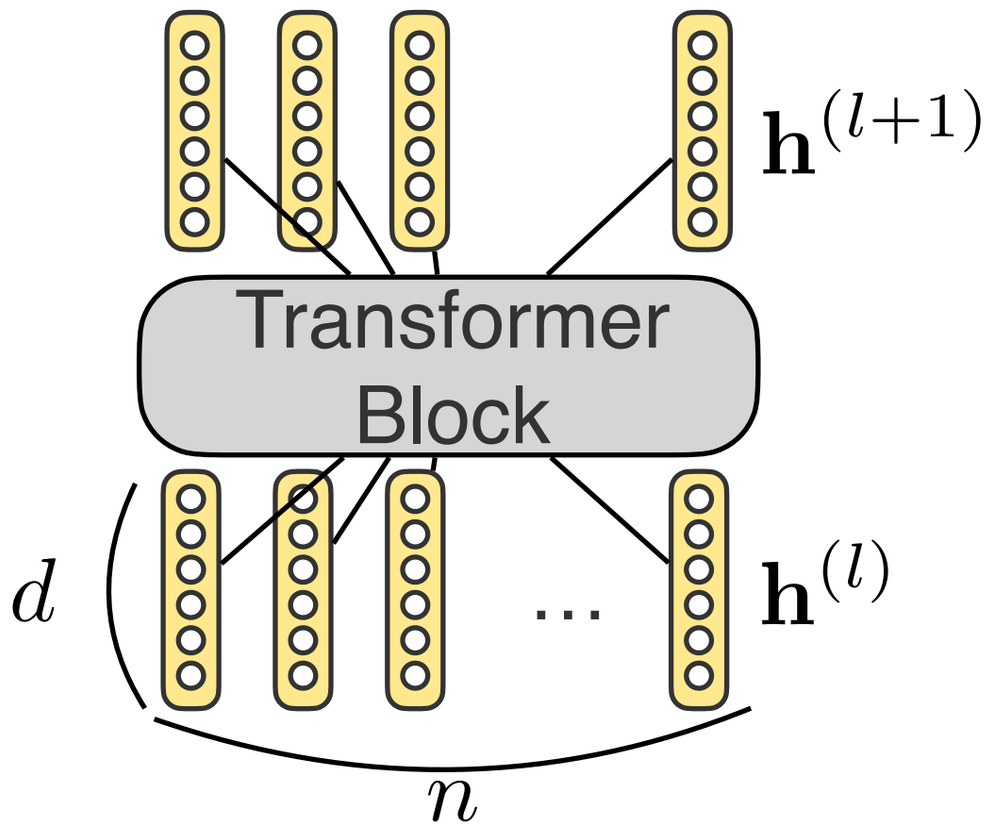
Multi-Layer Transformer

$$\mathbf{h}' = \text{TransformerBlock}(\mathbf{h})$$



Multi-Layer Transformer

$$\mathbf{h}^{(\mathrm{l}+1)} = \mathrm{TransformerBlock}_{l+1}(\mathbf{h}^{(l)})$$



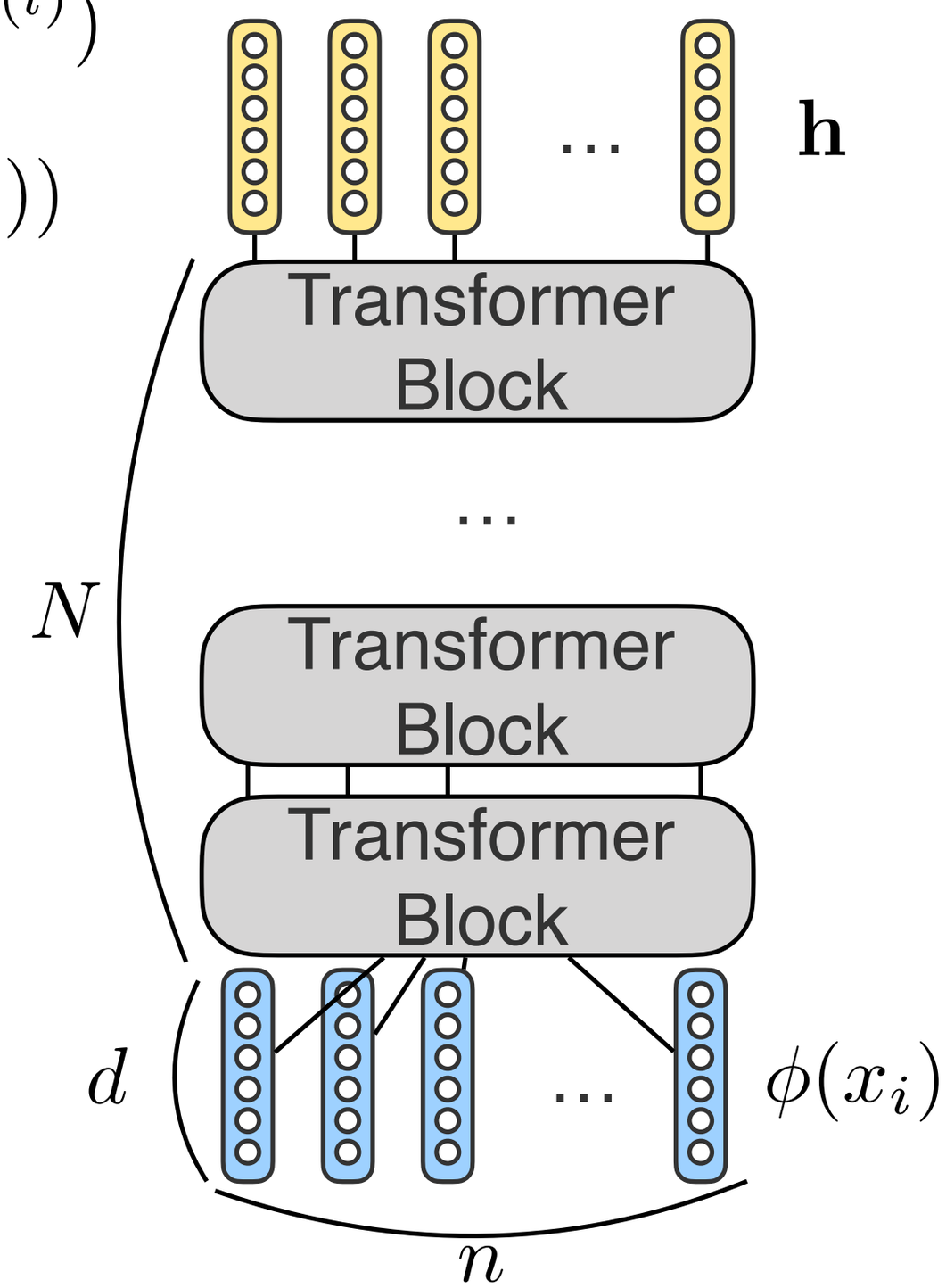
Transformer Encoder

$$\mathbf{h}^{(l+1)} = \text{TransformerBlock}_{l+1}(\mathbf{h}^{(l)})$$

$$\mathbf{h}^{N} = \underset{\text{Encoder}}{\text{Transformer}}(\phi(x_1), \ldots, \phi(x_n))$$

$$\mathbf{h}^{N} \in \mathbb{R}^{d \times n}$$

Input: $\phi(x_i) = \phi(x) + \phi(i)$
word + position embeddings
$\overline{x} = \langle x_1, \ldots, x_n \rangle$



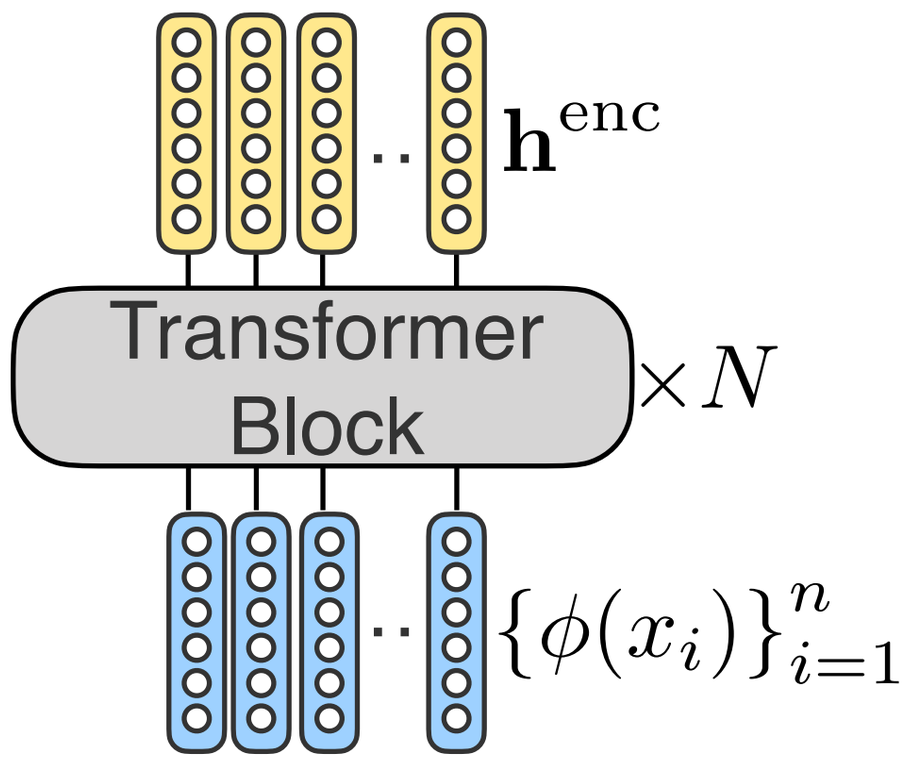
Transformer Encoder

$$\mathbf{h}^N = \underset{\text{Encoder}}{\text{Transformer}}(\phi(x_1), \ldots, \phi(x_n))$$

$$\underset{\text{Encoder}}{\text{Transformer}} : \mathcal{V}^+ \rightarrow \mathbb{R}^{d \times n}$$

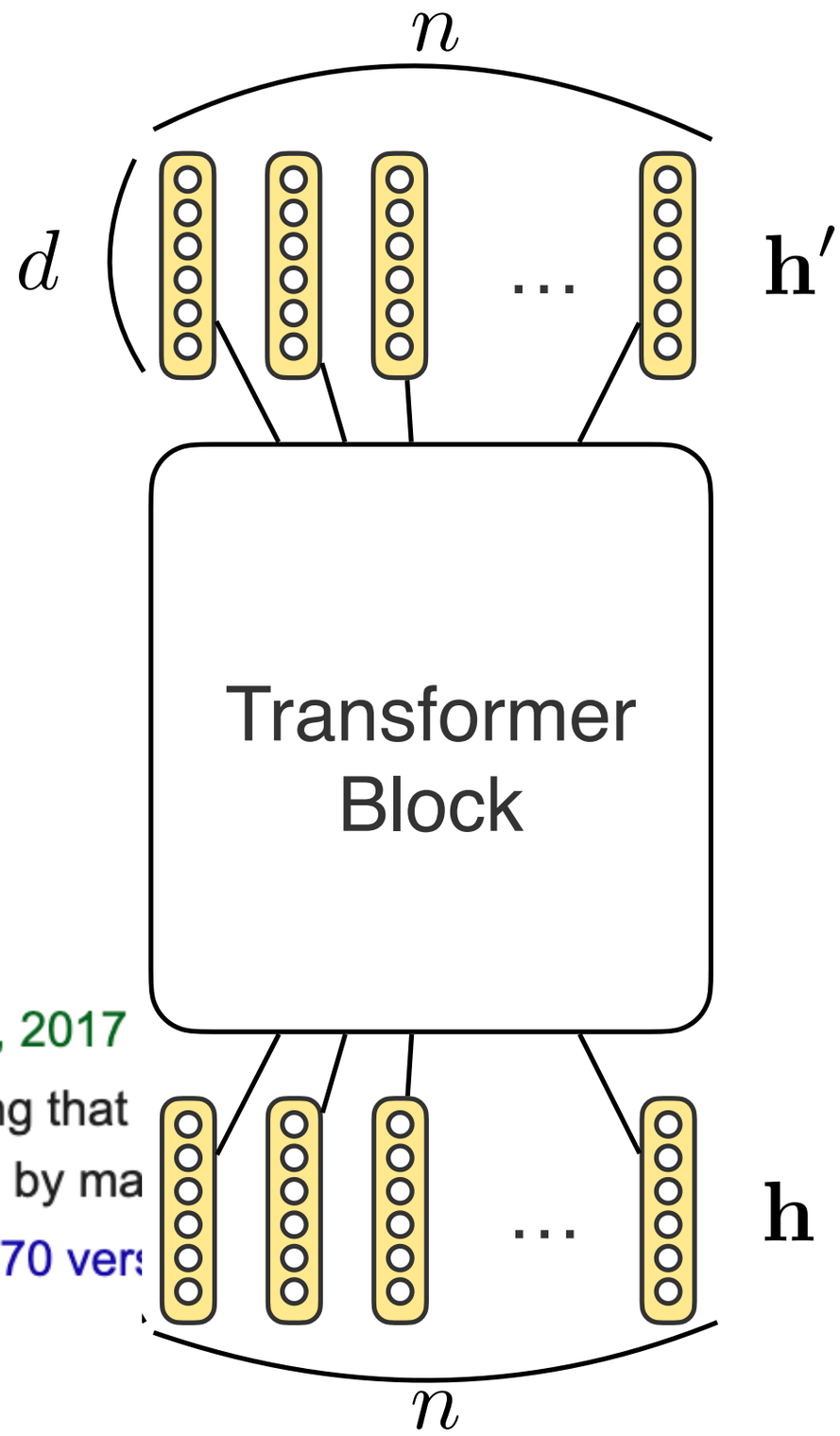
The transformer encoder maps from a sequence of tokens to a set of vectors, each corresponding to a token in the input sequence

This is exactly what we got from an RNN encoder and used for conditional generation in a decoder!



Building a Transformer Block

- A transformer block takes as input a sequence of input vectors $\mathbf{h} \in \mathbb{R}^{d \times n}$
- It generates a sequence of output vectors $\mathbf{h}' \in \mathbb{R}^{d \times n}$



Attention is all you need

A Vaswani, N Shazeer, N Parmar… - Advances in neural …, 2017

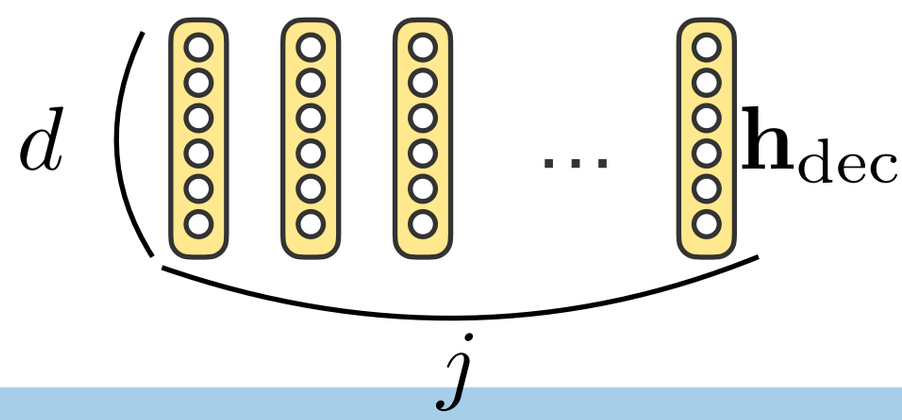
… to attend to **all** positions in the decoder up to and including that

… **We** implement this inside of scaled dot-product **attention** by ma

☆ Save 99 Cite Cited by 196986 Related articles All 70 vers

Decoder Block

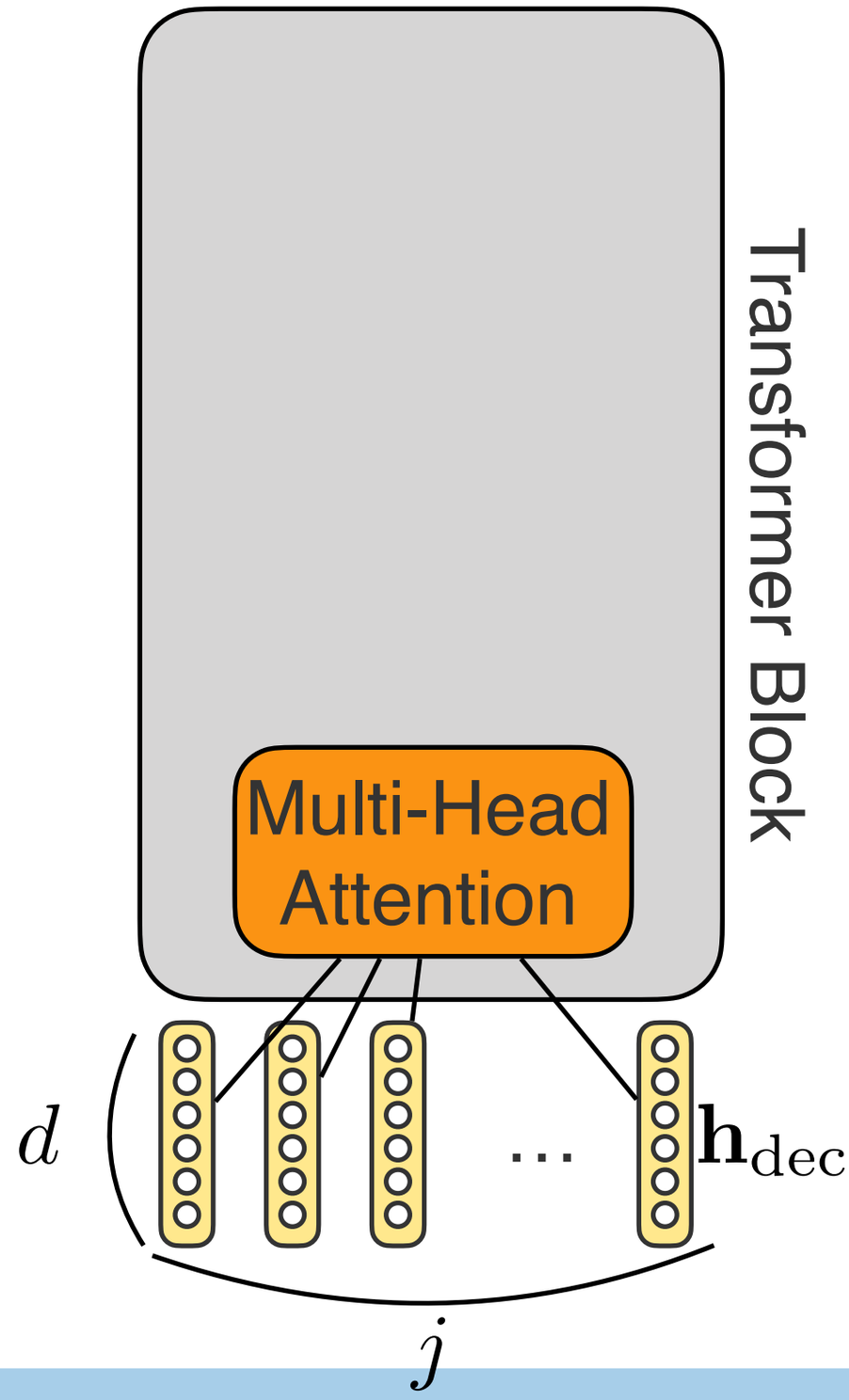
We’re at generation step *j*, and have our previous decoder steps available $\mathbf{h}_{\text{dec}}$



Decoder Block

$$\mathbf{h}^{\mathrm{MHA}_1} = \mathrm{MultiHeadAttention}^{H}(\mathbf{h}_{\mathrm{dec}})$$

First, we do self-attention over the tokens we've already generated

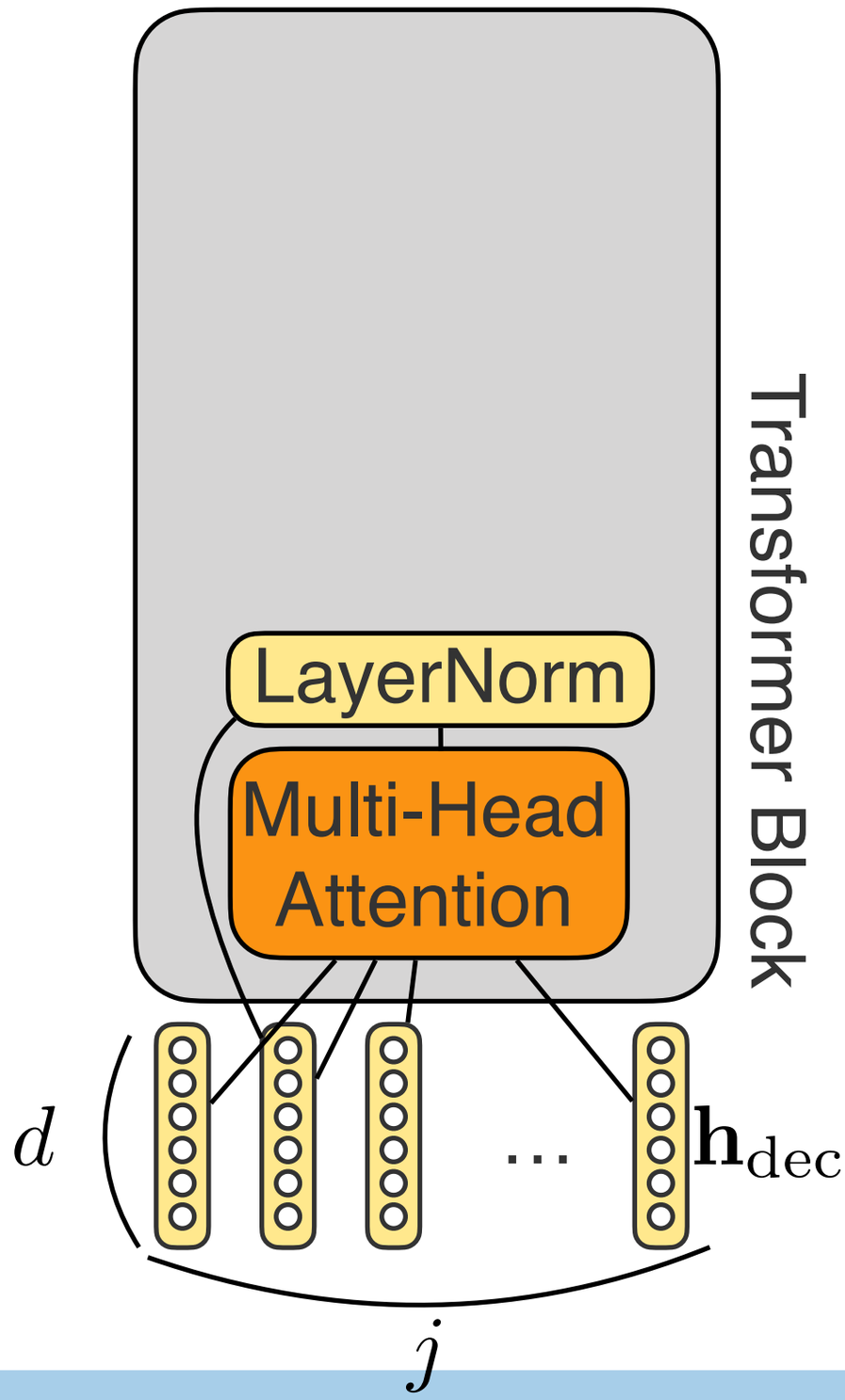


Decoder Block

$$\mathbf{h}^{\mathrm{MHA}_1} = \mathrm{MultiHeadAttention}^{H}(\mathbf{h}_{\mathrm{dec}})$$

$$\mathbf{h}^{\mathrm{AddNorm}_1} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{dec}} + \mathbf{h}^{\mathrm{MHA}_1})$$

Layer norm on the output of the self-attention



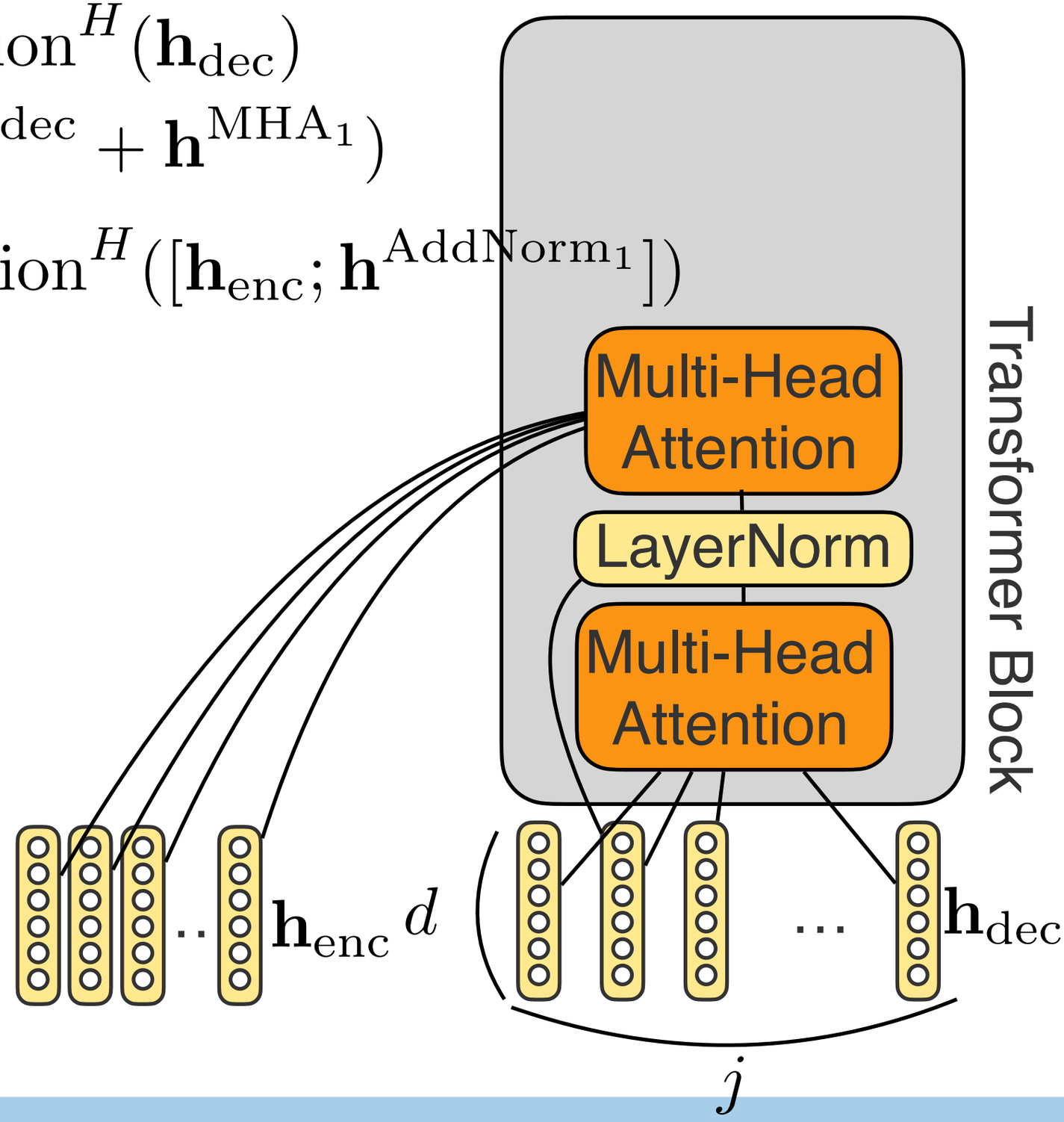
Decoder Block

$$\mathbf{h}^{\mathrm{MHA}_1} = \mathrm{MultiHeadAttention}^H(\mathbf{h}_{\mathrm{dec}})$$

$$\mathbf{h}^{\mathrm{AddNorm}_1} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{dec}} + \mathbf{h}^{\mathrm{MHA}_1})$$

$$\mathbf{h}^{\mathrm{MHA}_2} = \mathrm{MultiHeadAttention}^H([\mathbf{h}_{\mathrm{enc}}; \mathbf{h}^{\mathrm{AddNorm}_1}])$$

Then we attend to the encoder hidden states using the output of the previous layer normalization



Decoder Block

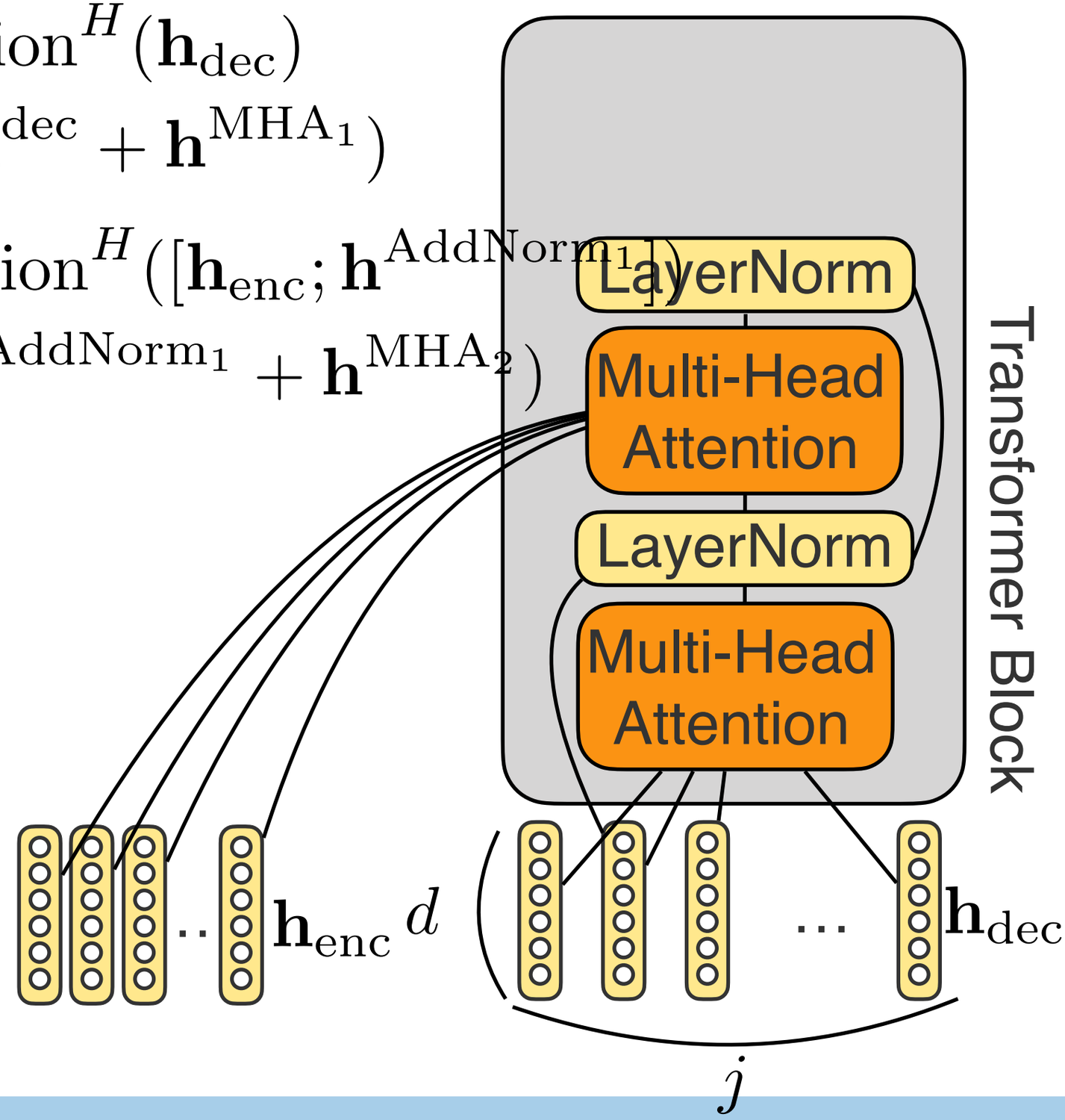
$$\mathbf{h}^{\mathrm{MHA}_1} = \mathrm{MultiHeadAttention}^H(\mathbf{h}_{\mathrm{dec}})$$

$$\mathbf{h}^{\mathrm{AddNorm}_1} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{dec}} + \mathbf{h}^{\mathrm{MHA}_1})$$

$$\mathbf{h}^{\mathrm{MHA}_2} = \mathrm{MultiHeadAttention}^H([\mathbf{h}_{\mathrm{enc}}; \mathbf{h}^{\mathrm{AddNorm}_1}])$$

$$\mathbf{h}^{\mathrm{AddNorm}_2} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{AddNorm}_1} + \mathbf{h}^{\mathrm{MHA}_2})$$

Yet another layer norm



Decoder Block

$$\mathbf{h}^{\mathrm{MHA}_1} = \mathrm{MultiHeadAttention}^H(\mathbf{h}_{\mathrm{dec}})$$

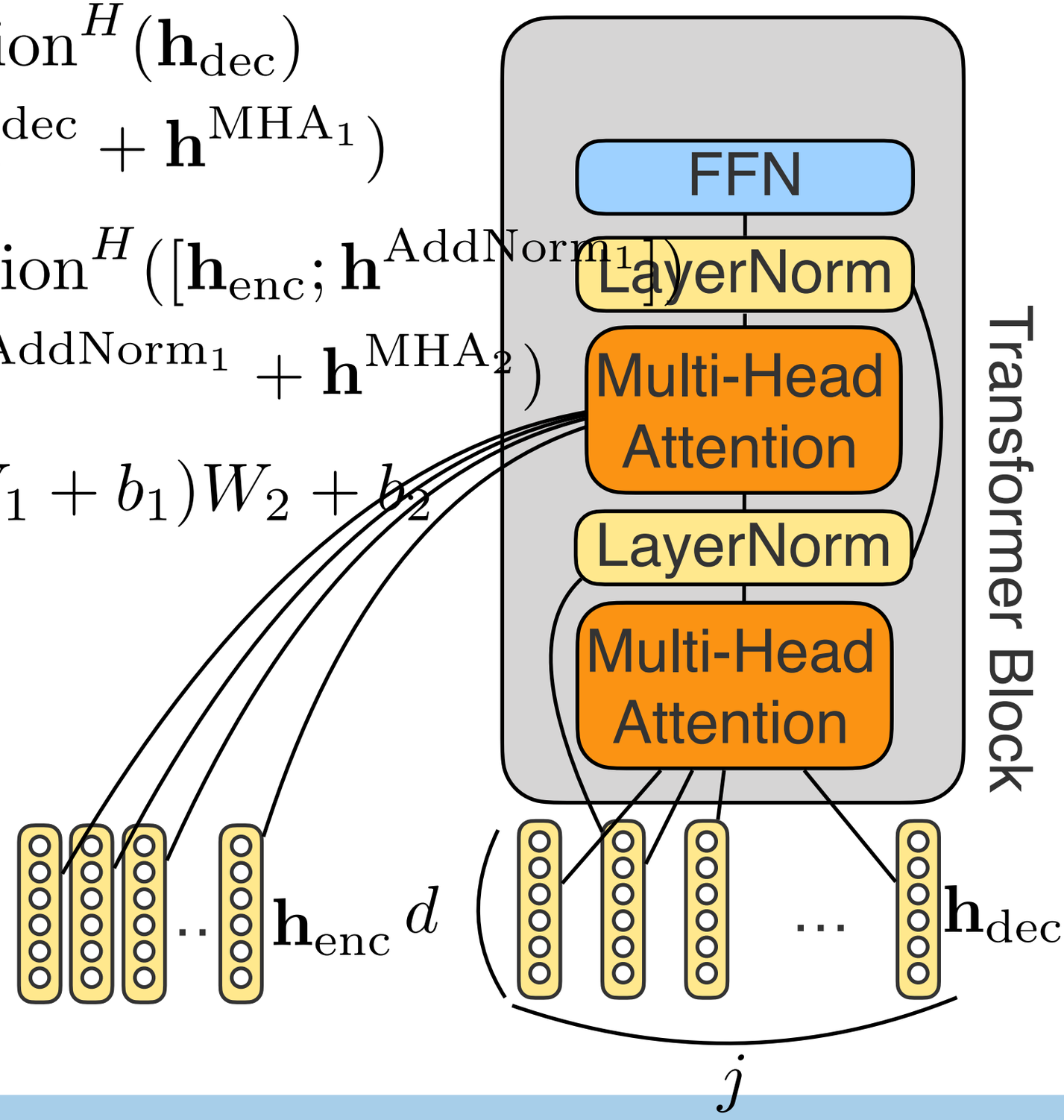
$$\mathbf{h}^{\mathrm{AddNorm}_1} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{dec}} + \mathbf{h}^{\mathrm{MHA}_1})$$

$$\mathbf{h}^{\mathrm{MHA}_2} = \mathrm{MultiHeadAttention}^H([\mathbf{h}_{\mathrm{enc}}; \mathbf{h}^{\mathrm{AddNorm}_1}])$$

$$\mathbf{h}^{\mathrm{AddNorm}_2} = \mathrm{LayerNorm}(\mathbf{h}^{\mathrm{AddNorm}_1} + \mathbf{h}^{\mathrm{MHA}_2})$$

$$\mathbf{h}^{\mathrm{FFN}} = \mathrm{ReLU}(\mathbf{h}^{\mathrm{AddNorm}_2} W_1 + b_1) W_2 + b_2$$

Feedforward network



Decoder Block

$$\mathbf{h}^{\text{MHA}_1} = \text{MultiHeadAttention}^H(\mathbf{h}_{\text{dec}})$$

$$\mathbf{h}^{\text{AddNorm}_1} = \text{LayerNorm}(\mathbf{h}^{\text{dec}} + \mathbf{h}^{\text{MHA}_1})$$

$$\mathbf{h}^{\text{MHA}_2} = \text{MultiHeadAttention}^H([\mathbf{h}_{\text{enc}}; \mathbf{h}^{\text{AddNorm}_1}])$$

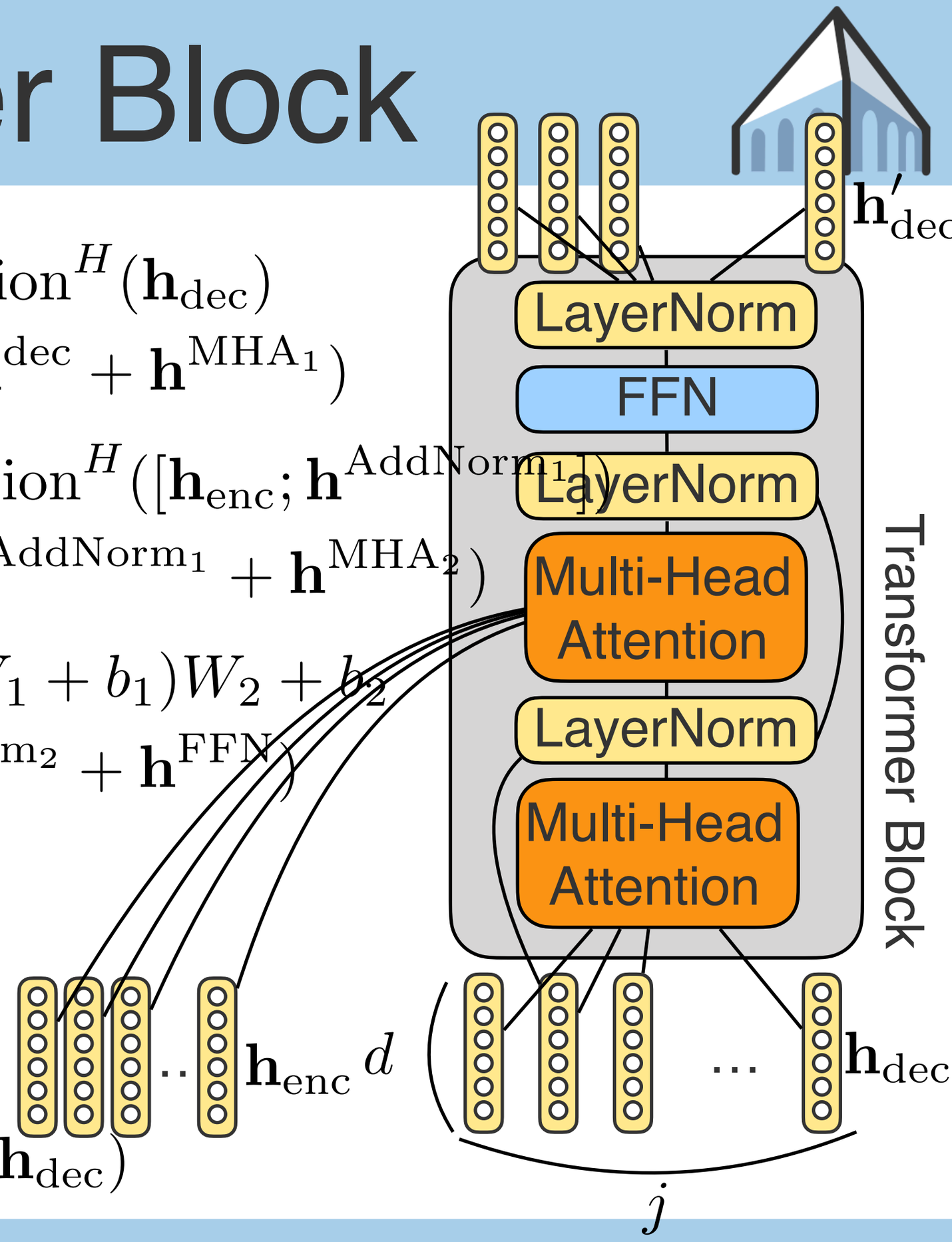
$$\mathbf{h}^{\text{AddNorm}_2} = \text{LayerNorm}(\mathbf{h}^{\text{AddNorm}_1} + \mathbf{h}^{\text{MHA}_2})$$

$$\mathbf{h}^{\text{FFN}} = \text{ReLU}(\mathbf{h}^{\text{AddNorm}_2} W_1 + b_1) W_2 + b_2$$

$$\mathbf{h}'_{\text{dec}} = \text{LayerNorm}(\mathbf{h}^{\text{AddNorm}_2} + \mathbf{h}^{\text{FFN}})$$

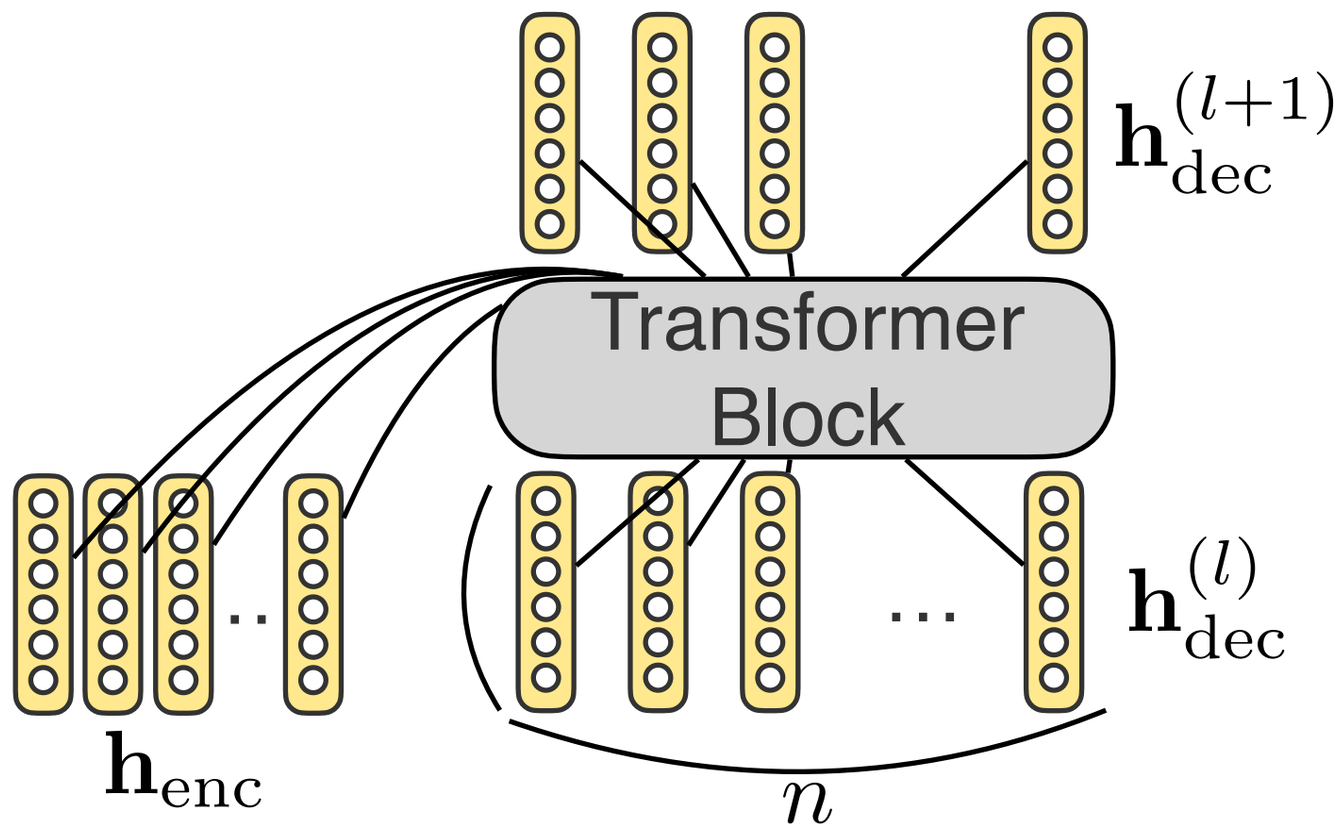
One last layer norm to get our final output vectors for each token generated so far!

$$\mathbf{h}'_{\text{dec}} = \text{DecoderBlock}(\mathbf{h}_{\text{enc}}, \mathbf{h}_{\text{dec}})$$



Multi-Layer Decoder

$$\mathbf{h}_{\text{dec}}^{(l+1)} = \text{DecoderBlock}_{l+1}(\mathbf{h}_{\text{enc}}, \mathbf{h}_{\text{dec}}^{(l+1)})$$



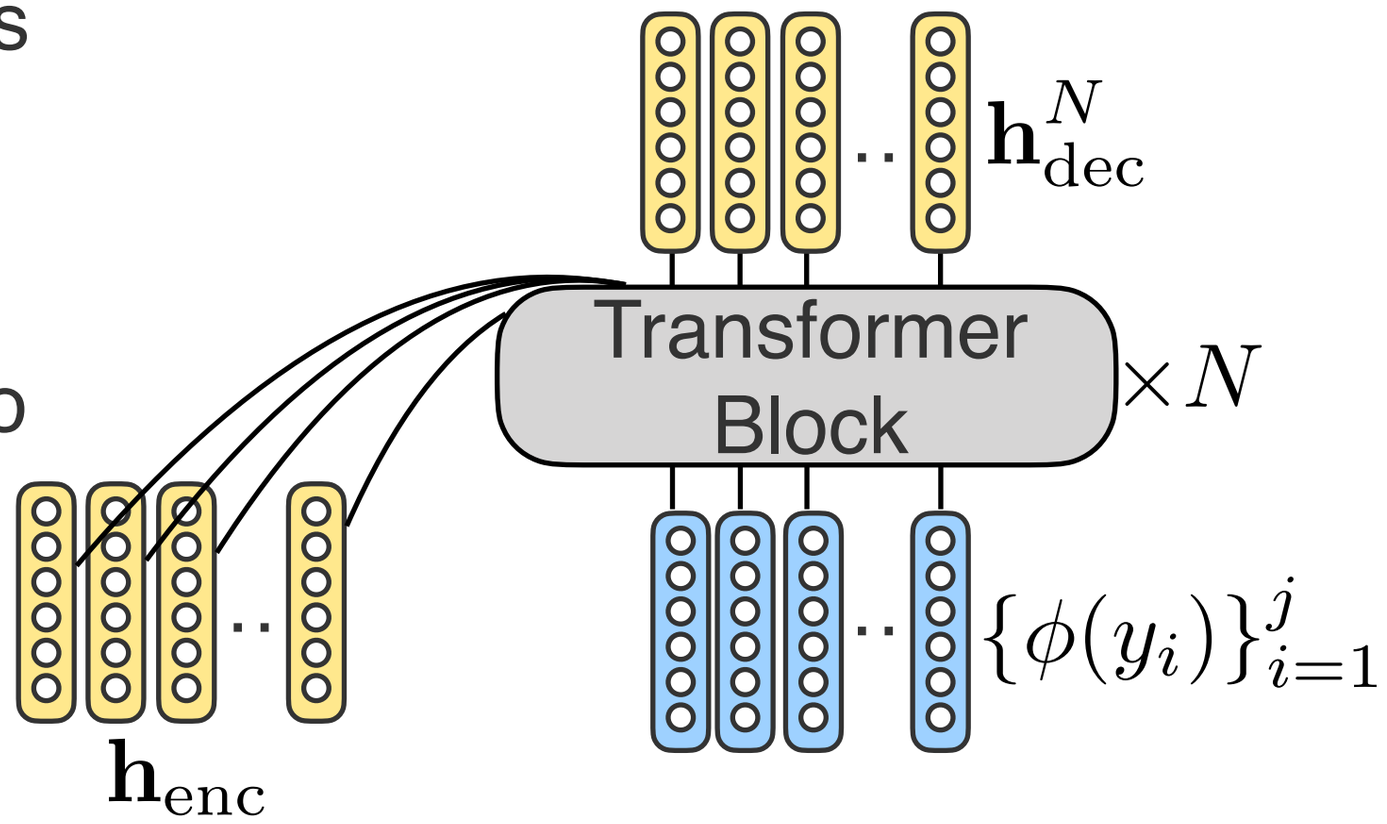
Transformer Decoder

$$\mathbf{h}_{\text{dec}}^{(l+1)} = \text{DecoderBlock}_{l+1}(\mathbf{h}_{\text{enc}}, \mathbf{h}_{\text{dec}}^{(l+1)})$$

$$\mathbf{h}_{\text{dec}}^{N} = \underset{\text{Decoder}}{\text{Transformer}}(\mathbf{h}_{\text{enc}}, \phi(y_1), \ldots, \phi(y_j))$$

$$\underset{\text{Decoder}}{\text{Transformer}} : \mathbb{R}^{d \times n} \times \mathcal{V}^j \rightarrow \mathbb{R}^{d \times j}$$

The transformer decoder maps from a set of input encodings and a sequence of tokens generated so far to a set of vectors, each corresponding to a token in the currently-generated sequence



Decoding

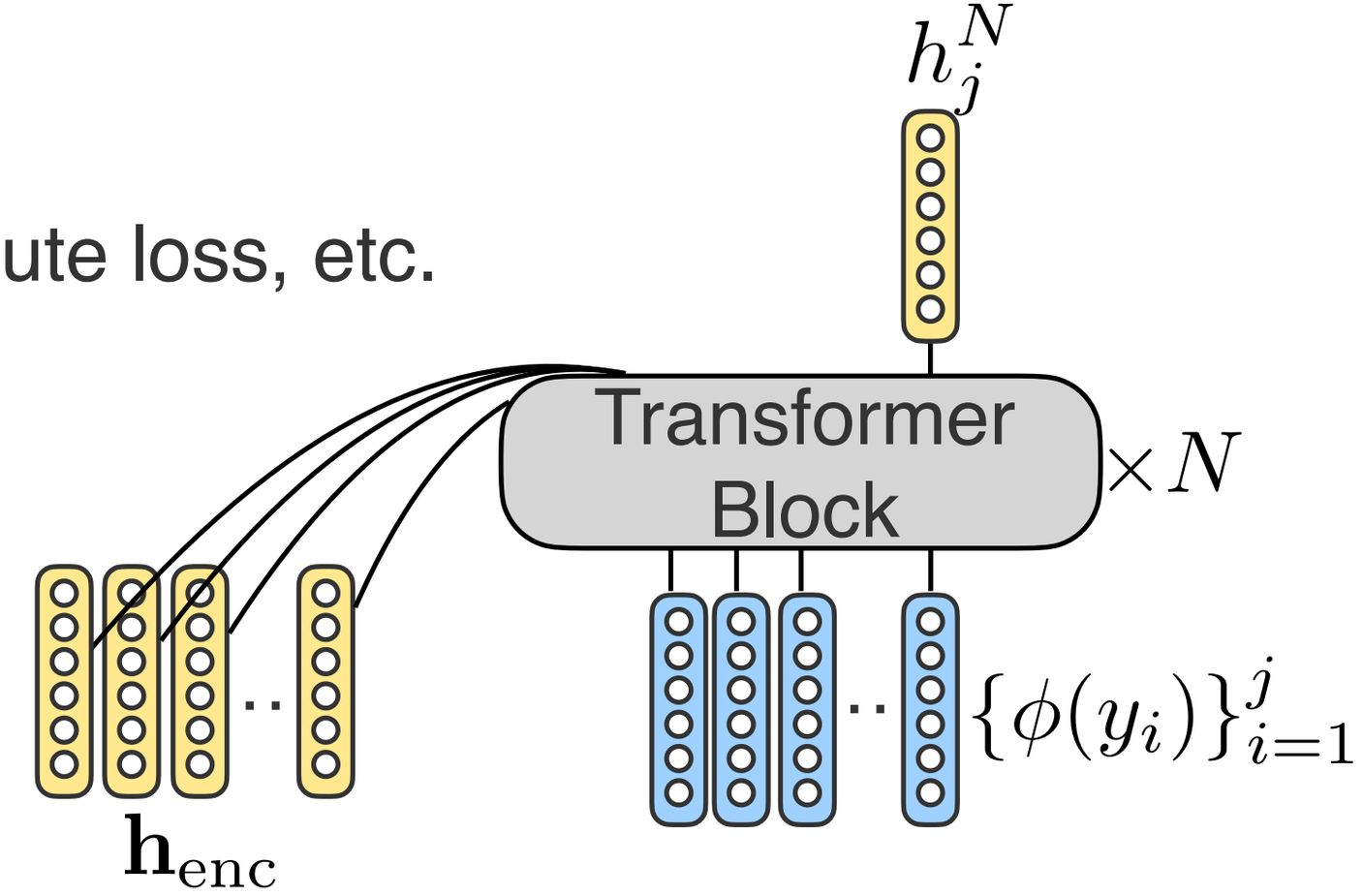
- Let's say we want to predict the word at index *j* + 1

- All we need to do is transform this into a distribution over the vocabulary

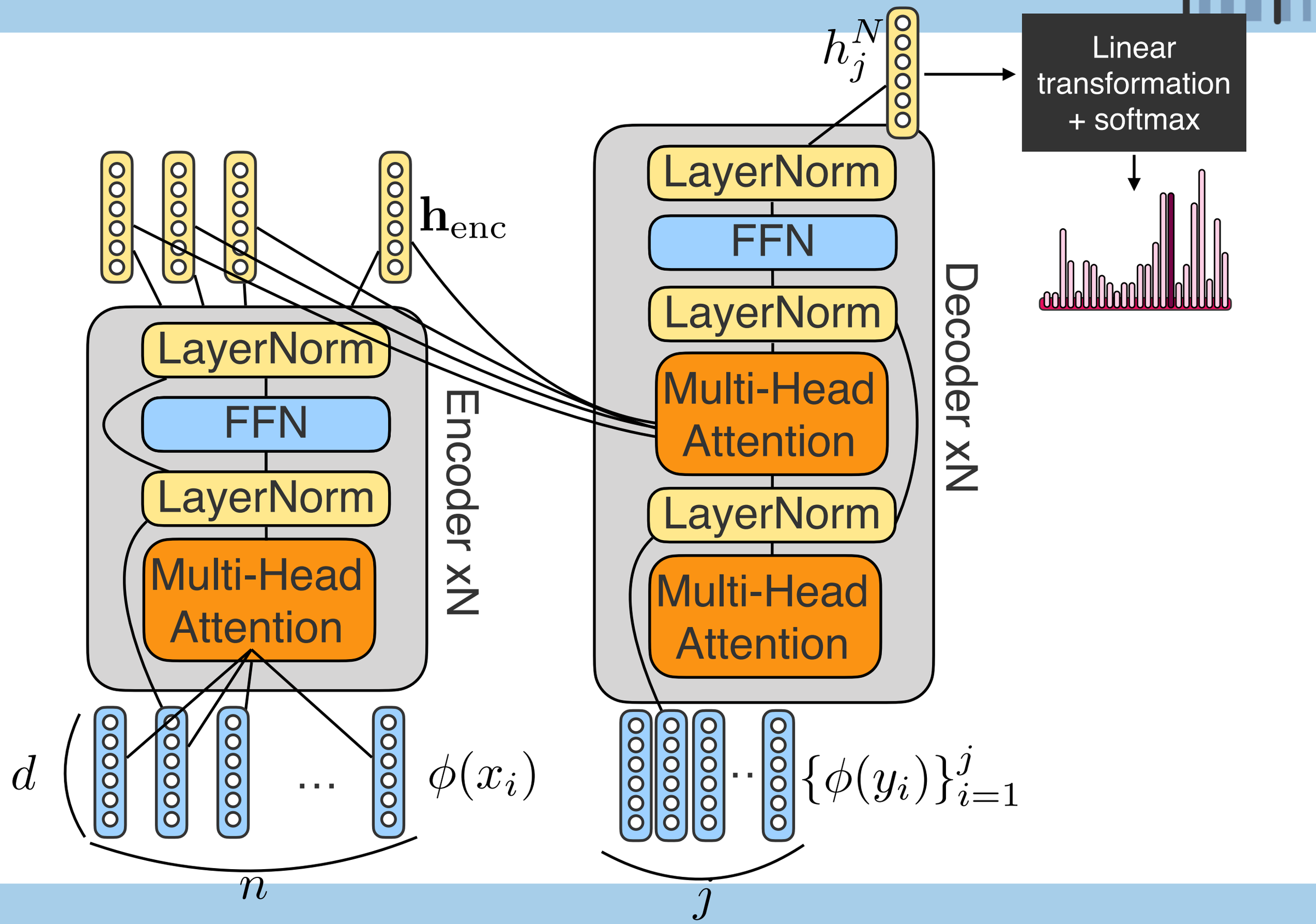
$$p(Y_{j+1} \mid x_1, \ldots, x_n, y_1, \ldots, y_j) = \text{softmax}(W h_j^N)$$

$$W \in \mathbb{R}^{|\mathcal{V}| \times d}$$

- Then we can sample, compute loss, etc.



Encoder-Decoder



Training

- Given some paired data $(\overline{x}, \overline{y})$:

$$\mathbf{h}_{\text{enc}} = \underset{\text{Encoder}}{\text{Transformer}}(\overline{x})$$

We can compute these in parallel on the GPU

$$p(y_{i+1}) = \underset{\text{Decoder}}{\text{Transformer}}(\mathbf{h}_{\text{enc}}, y_1, \ldots, y_i)$$

Probability of a token should only depend on the ones that come before it! But we still want to take advantage of parallelism…

Training

- Given some paired data $(\overline{x}, \overline{y})$:

$$\mathbf{h}_{\text{enc}} = \underset{\text{Encoder}}{\text{Transformer}}(\overline{x})$$

We can compute these in parallel on the GPU

$$p(y_{i+1}) = \underset{\text{Decoder}}{\text{Transformer}}(\mathbf{h}_{\text{enc}}, y_1, \ldots, y_i)$$

Probability of a token should only depend on the ones that come before it! But we still want to take advantage of parallelism…

- **Masking:**
  - When doing the forward pass on the decoder, self-attention can be parallelized
  - But we don't want the model to learn to rely on "future" words in the sequence!
  - Solution: during training, set $s_{ij} = -\infty$ if *i* (query index) < *j* (key/value index)

Unconditional Sequence Generation

- Recall the autoregressive approximation of sequence probability:

$$p(\overline{x}) = \prod_{i=1}^{|\overline{x}|} p(x_i \mid x_1, \ldots, x_{i-1})$$

- Can we do this with a transformer model?

Unconditional Sequence Generation

- Recall the autoregressive approximation of sequence probability:

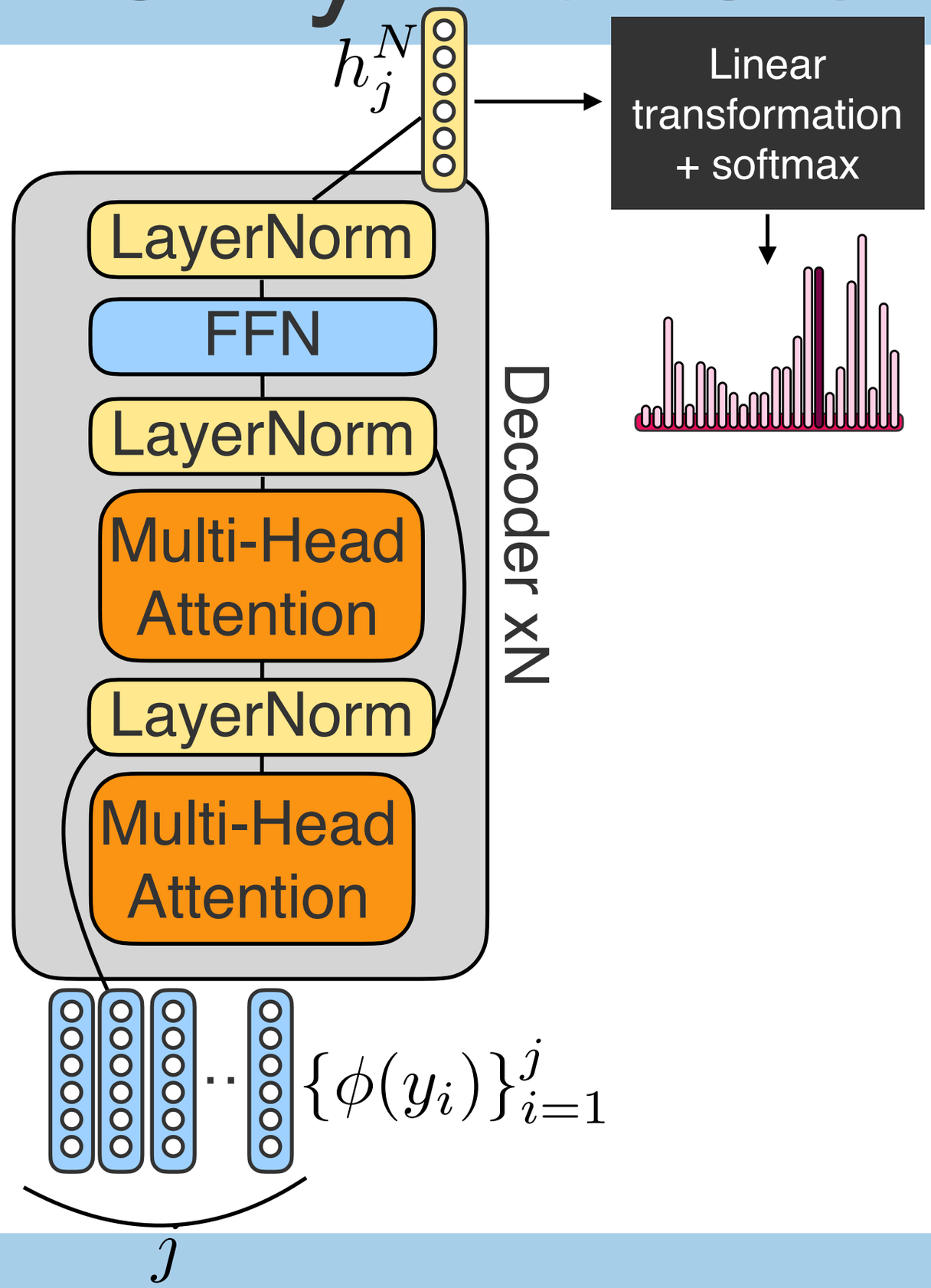
$$p(\overline{x}) = \prod_{i=1}^{|\overline{x}|} p(x_i \mid x_1, \ldots, x_{i-1})$$

- Can we do this with a transformer model?

- **Yes!** We just don't have an encoder (i.e., decoder-only LLM).

- We still have to be careful about masking while training.

Decoder-Only Transformer



Sequence Encoding

- What if we just want some representation of a sentence, and we don't really care about being able to generate sentences? $\mathrm{Enc}(\overline{x})$
- We get this with the transformer encoder:

$$\mathbf{h}_{\mathrm{enc}} = \underset{\mathrm{Encoder}}{\mathrm{Transformer}}(\overline{x})$$

- But how might we train it?

Masked Language Modeling

Masked language modeling:

- The probability of a sequence is a product of local token probabilities
- The probability of a token depends on the ones that came before *and after* it

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$

The name *anteater* refers to the species' [redacted], which consists mainly of ants and termites.

Masked Language Modeling

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$

- From our transformer encoder, we get

$$h_i = \underset{\text{Encoder}}{\text{Transformer}}(\overline{x})_i \in \mathbb{R}^d$$

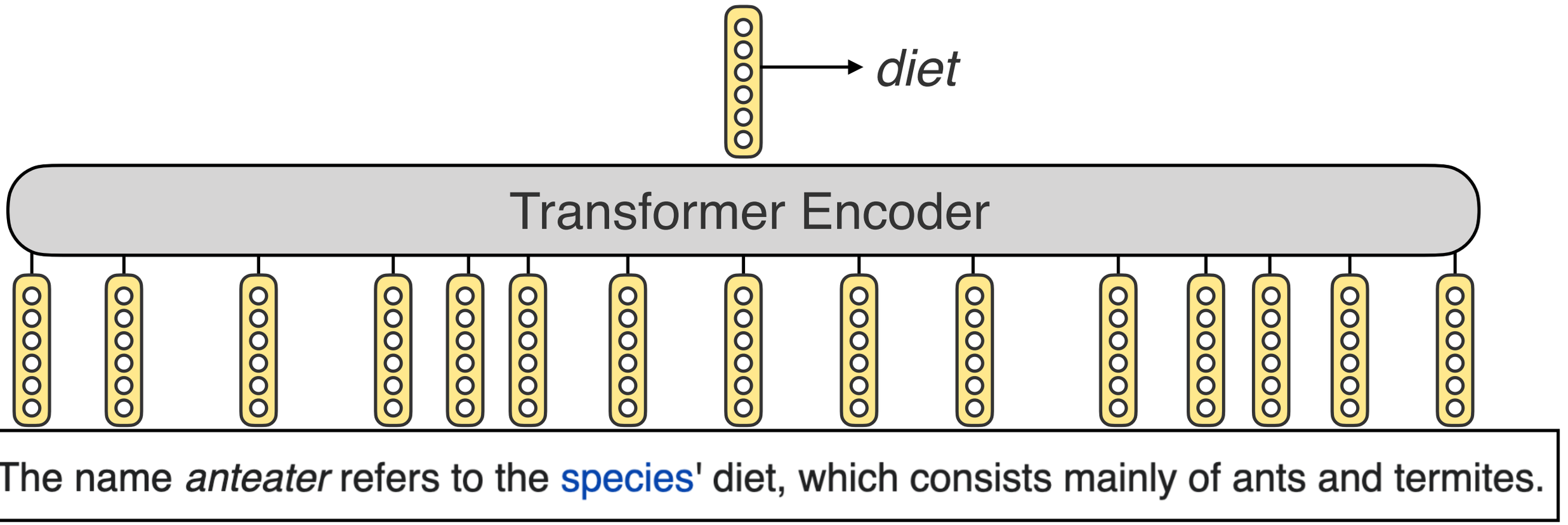
- We could just transform this to the vocabulary space, and compute the loss from there:

$$p(X_i) = \text{softmax}(W h_i)$$

- Do you anticipate any problems?

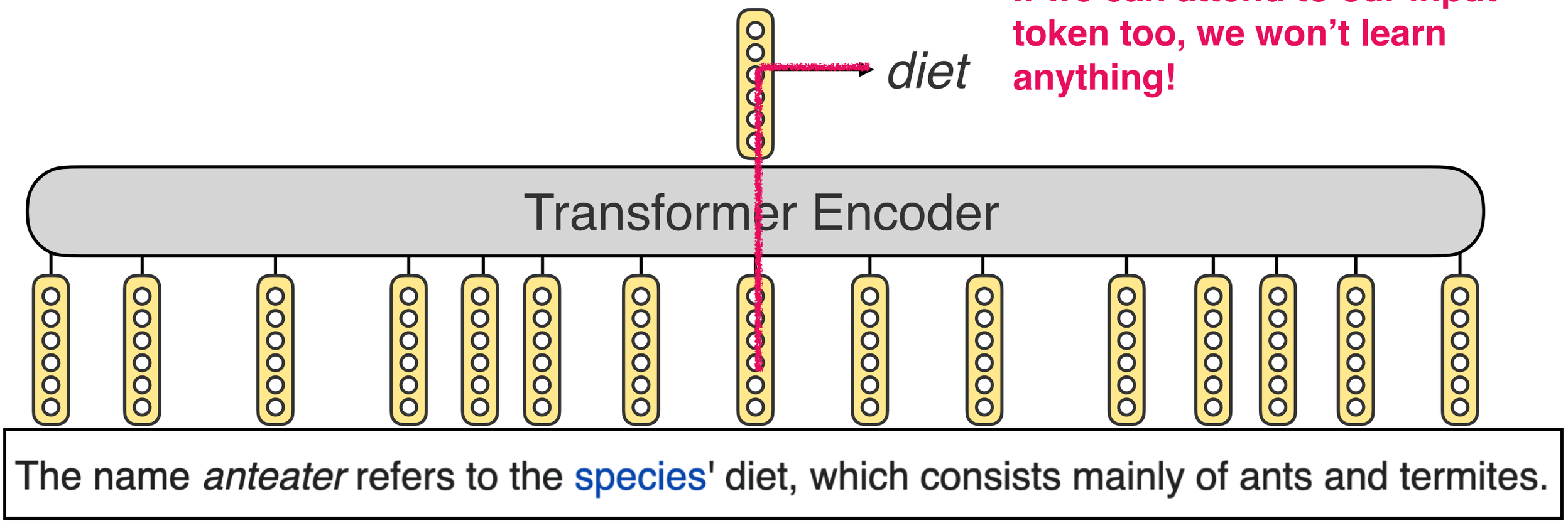
Masked Language Modeling

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$



Masked Language Modeling

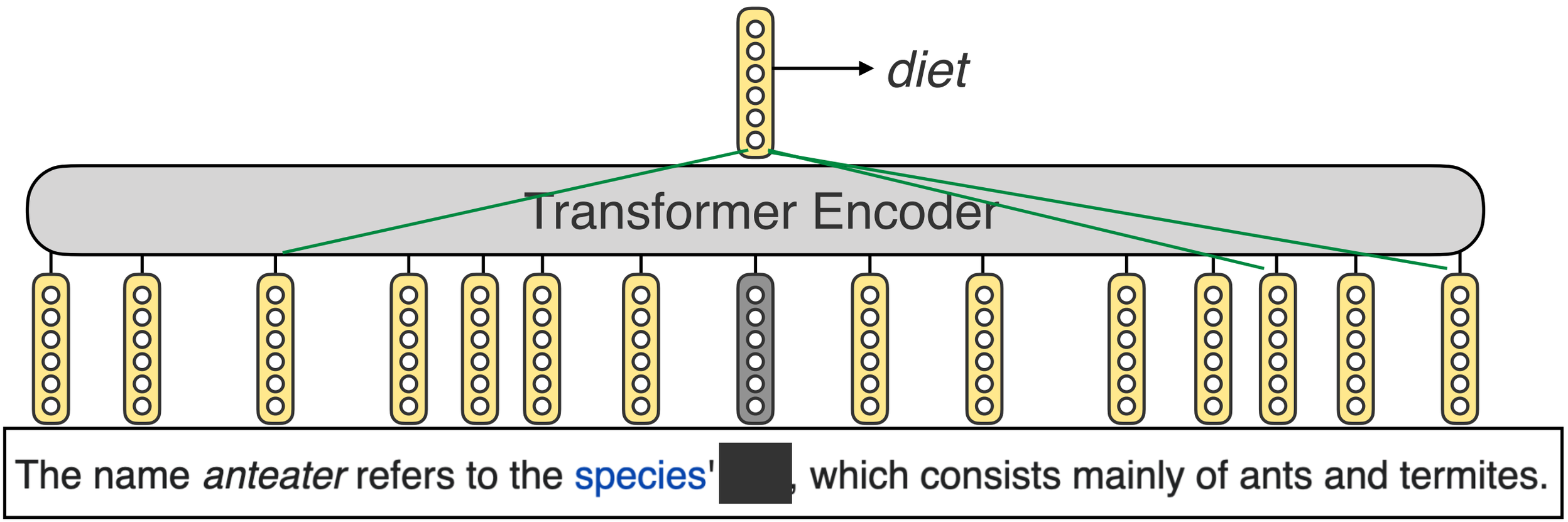
$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$



If we can attend to our input token too, we won't learn anything!

Masked Language Modeling

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$



Instead: randomly sample some tokens to replace with a MASK placeholder token in the input, then train the model to predict those words

Masked Language Modeling

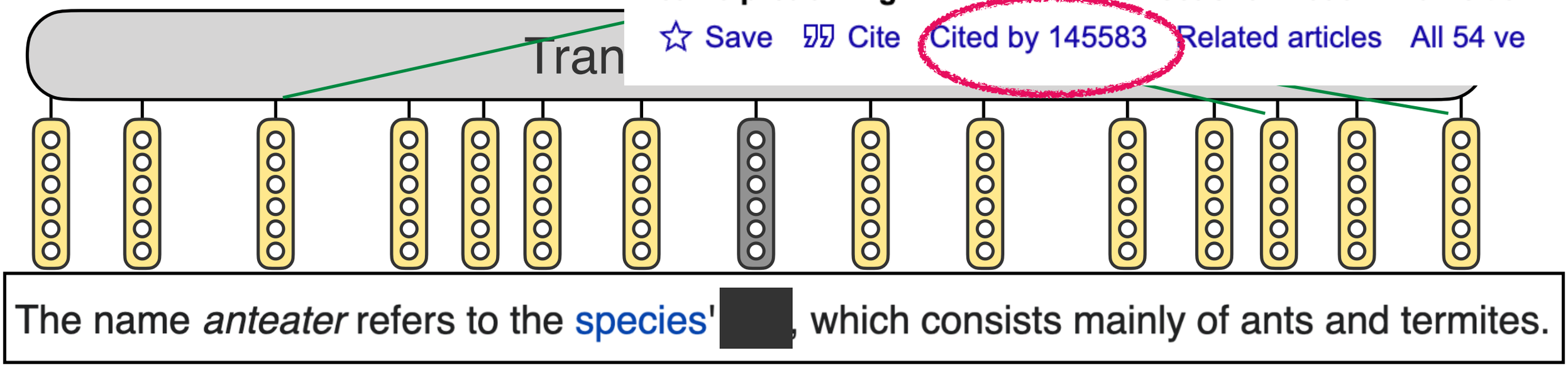
$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$

Bert: Pre-training of deep bidirectional transfor understanding

J Devlin, MW Chang, K Lee… - … : human **language** …, 2019 - a

… **deep** bidirectionality of **BERT** by evaluating two **pretraining** ob same **pretraining** … No NSP: A **bidirectional** model which is trai

☆ Save 99 Cite Cited by 145583 Related articles All 54 ve



Instead: randomly sample some tokens to replace with a MASK placeholder token in the input, then train the model to predict those words

Overview: Sequence Modeling

- A sequence model imposes a probability distribution over $\mathcal{V}^+$

$$p(\overline{X}) \in \Delta^{\mathcal{V}^+} \qquad \overline{x} \in \mathcal{V}^+ \qquad p(\overline{x})$$

- One common approximation for computing this is autoregressive decomposition:

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

- (Or sometimes, masked language modeling:)

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, x_{i+1} \ldots x_n)$$

Overview: Sequence Generation

- From an autoregressive language model, we can generate sequences using a variety of decoding methods that make local decisions about what words to add to a sequence

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

- Ancestral sampling: sample directly from $p(X_i \mid x_1, \ldots, x_{i-1})$
- Adjusting the temperature of a distribution

$$p(X_i = w \mid x_1, \ldots, x_{i-1}) = \frac{\exp(s(w)/\tau)}{\sum_{w' \in \mathcal{V}} \exp(s(w')/\tau)}$$

Overview: Sequence Generation

- From an autoregressive language model, we can generate sequences using a variety of decoding methods that make local decisions about what words to add to a sequence

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

- Estimating the argmax via greedy sampling

$$x_i \leftarrow \arg\max_{x \in \mathcal{V}} p(X_i \mid x_1, \ldots, x_{i-1})$$

- Or, estimate this using beam search

Overview: Sequence Generation

- From an autoregressive language model, we can generate sequences using a variety of decoding methods that make local decisions about what words to add to a sequence

$$p(\overline{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

- To better trade off between diversity and coherence, we can manipulate the local probability distributions by masking out wordtypes with lower probabilities:
  - Epsilon sampling
  - Top-k sampling
  - Top-p / nucleus sampling

Overview: Language Models

- How do we actually model local probabilities in the autoregressive approximation?

$$p(X_i = x) = p(x \mid x_1, \ldots, x_{i-1})$$

- N-gram approximation: the probability of a word at index *i* only depends on the *n* - 1 words that came before it

$$\approx p(x \mid x_{i-n+1}, \ldots, x_{i-1})$$

- We can compute this using counts from a corpus

$$\approx \frac{C(x_{i-n+1}, \ldots, x_{i-1}, x)}{C(x_{i-n+1}, \ldots, x_{i-1})}$$

Overview: Language Models

- How do we actually model local probabilities in the autoregressive approximation?

$$p(X_i = x) = p(x \mid x_1, \ldots, x_{i-1})$$

- N-gram approximation: the probability of a word at index *i* only depends on the *n* - 1 words that came before it

$$\approx p(x \mid x_{i-n+1}, \ldots, x_{i-1})$$

- Or using a neural network and word embeddings

$$\approx p(x_i \mid x_{i-n+1}, \ldots, x_i; \theta)$$

- But this approximation misses long-distance dependencies in sequences

Overview: RNNs

- Instead, let's have a model that operates on and updates a hidden state as it processes inputs and generates outputs

$$h_i = \sigma(W_h h_{i-1} + W_e \phi(x_i) + b_1)$$

- Some variations, such as LSTM, solve issues with learning, e.g. vanishing gradients

- Now we can also get context-dependent token-specific embeddings (i.e. hidden states produced for each token)

Overview: Sequence Embedding and Seq2Seq

- We can use hidden states to do word-level classification

- We can also use them to get a single representation of an entire sequence, given some pooling method:

$$\text{Enc}(\overline{x}) = \text{Pool}\left(h_1, \ldots, h_{|\overline{x}|}\right) \in \mathbb{R}^d$$

- Given a single representation of a sentence, we can do conditional text generation, e.g. for machine translation:

$$p(\overline{y}) = \prod_{i=1}^{m} p(y_i \mid \text{Enc}(\overline{x}), y_1, \ldots, y_{i-1})$$

Overview: Attention

- But using a single, fixed-size hidden state acts as a bottleneck that limits the expressivity of the model

- Instead, we can weight the contribution of each input state according to some current state during *decoding*:

$$h_1, \ldots, h_{|\overline{x}|} = \mathbf{h} \in \mathbb{R}^{d \times |\overline{x}|}$$

$$s_i = a(g_i, \mathbf{h}) \in \mathbb{R}^{|\overline{x}|}$$

$$\alpha_i = \text{softmax}(s_i) \in \Delta^{\mathbb{N}_{1:|\overline{x}|}}$$

$$c_i = \sum_{j=1}^{|\overline{x}|} \alpha_{i,j} h_j \in \mathbb{R}^d$$

$$p(Y_{i+1} \mid \overline{x}, y_1, \ldots, y_i) = \text{softmax}(f(c_i, g_i))$$

$$g_i = g(g_{i-1}, c_{i-1}, y_i)$$

Overview: Self-Attention

- We can even remove the recurrence relation in the encoder and use attention exclusively!

$$k_i = K\phi(x_i) \in \mathbb{R}^d$$
$$v_i = V\phi(x_i) \in \mathbb{R}^d$$
$$q_j = Q\phi(x_j) \in \mathbb{R}^d$$

$$s_{ji} = q_j^\intercal k_i$$

$$\alpha_{ji} = \frac{\exp(s_{ji})}{\sum_{i'=1}^{j} \exp(s_{ji'})}$$

$$c_j = \sum_{i=1}^{j} \alpha_{ji} v_i$$

$$x_{j+1} \sim p(X_{j+1} \mid x_1, \ldots, x_j) = \text{softmax}(f(c_j, \phi(x_j))$$

Overview:
Encoder-Decoder Transformer