Bits of Language: Architecture

Overview

- **Last two weeks.** Deep learning for images
  - Architecture. ConvNet
  - Training. Augmentation & Self-supervised training
  - Generation. VAE, GAN, Diffusion

- **This week.** Deep learning for text
  - Architecture. Preprocessing, RNN, Transformer
  - Generation. BERT & GPT

Overview

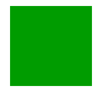
- **Last two weeks.** Deep learning for images
  - Architecture. ConvNet
  - Training. Augmentation & Self-supervised training
  - Generation. VAE, GAN, Diffusion

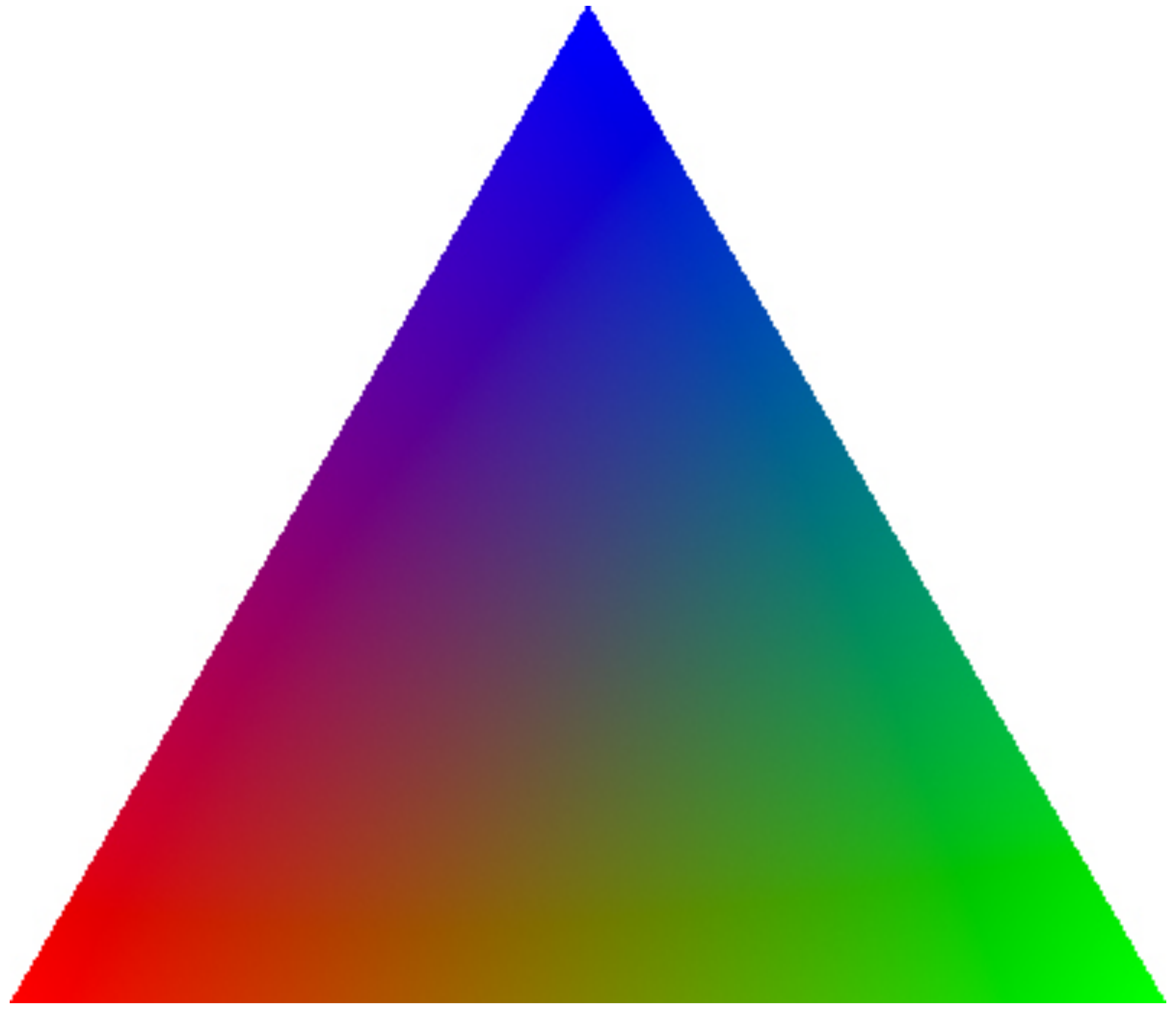
- **This week.** Deep learning for text
  - Architecture. Preprocessing, RNN, Transformer
  - Generation. BERT & GPT

Preview: Text vs. Image

- Text and image differs in many aspects

(1) Text is discrete

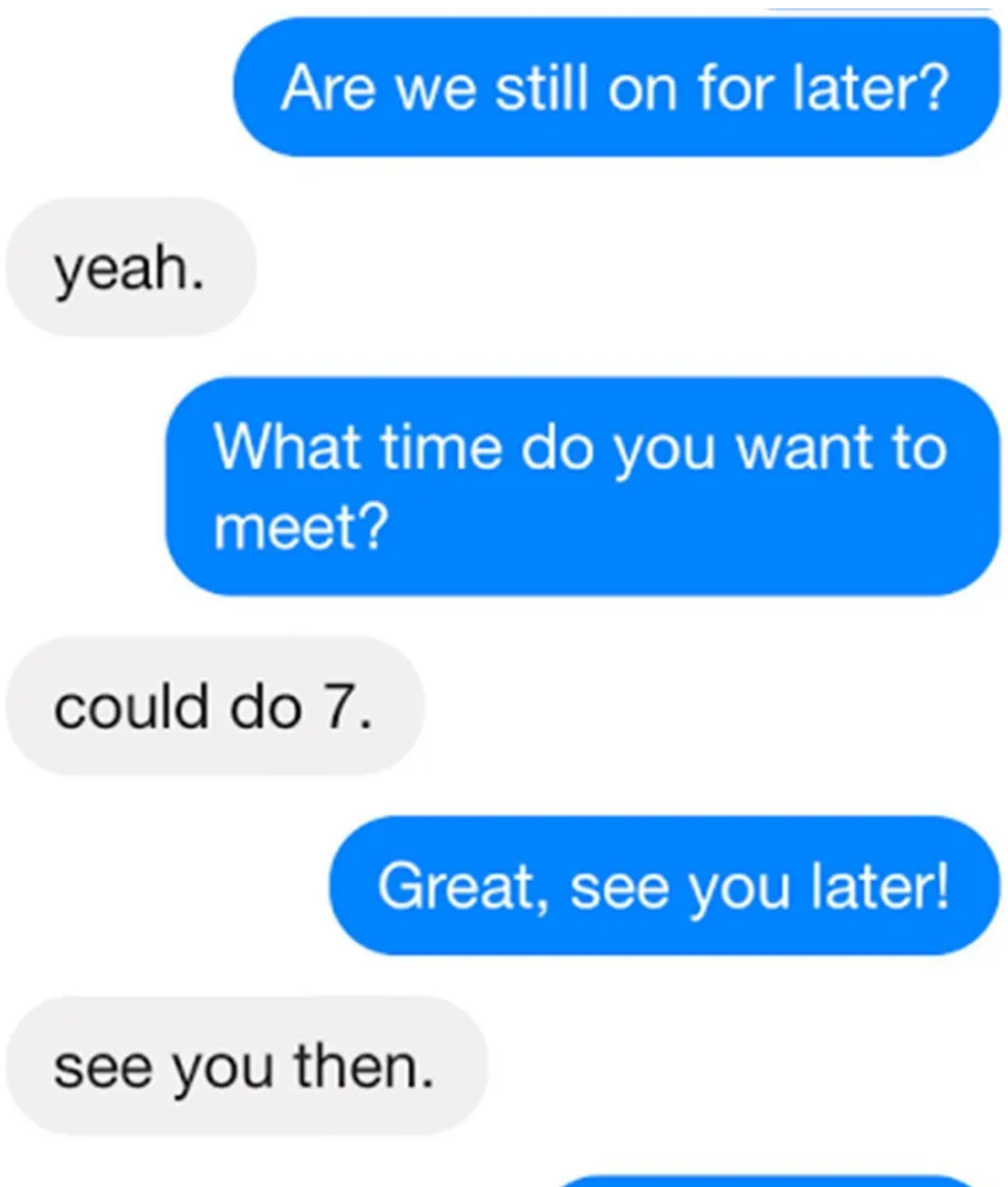
- Interpolating "" & "" vs. "A" & "C"
- <u>Required</u>. A nice text vectorization mechanism



Preview: Text vs. Image

(2) Text has variable length

- ⇔ image with fixed resolution — or can do downsampling
- Required. An architecture that can handle sequences effectively



Preview: Text vs. Image

(3) Text has weaker locality than images

- ⇔ image with high locality
- <u>Required</u>. Architecture that can cover far distances

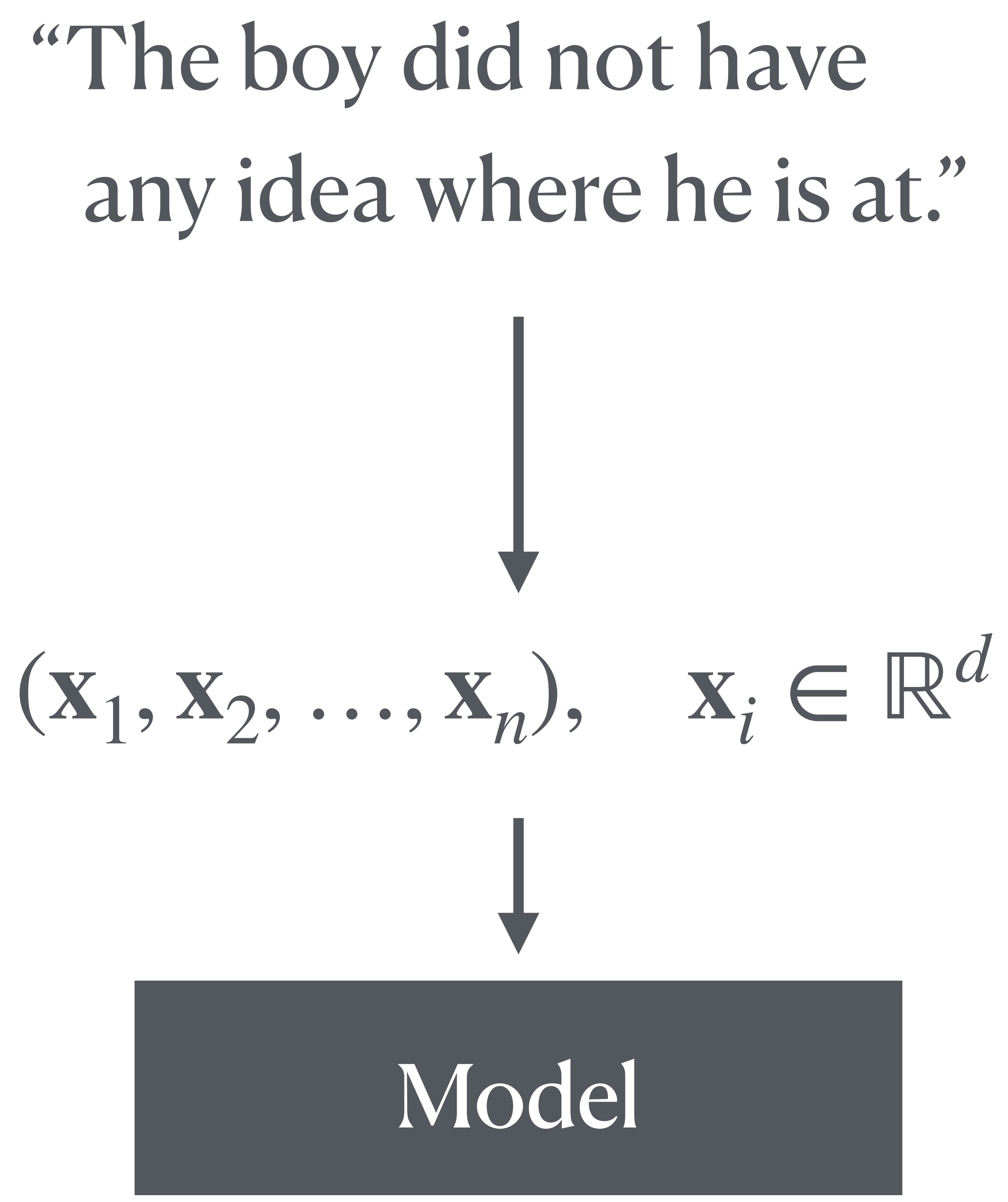
"The boy did not have any idea where he is at."

Text preprocessing

Preprocessing

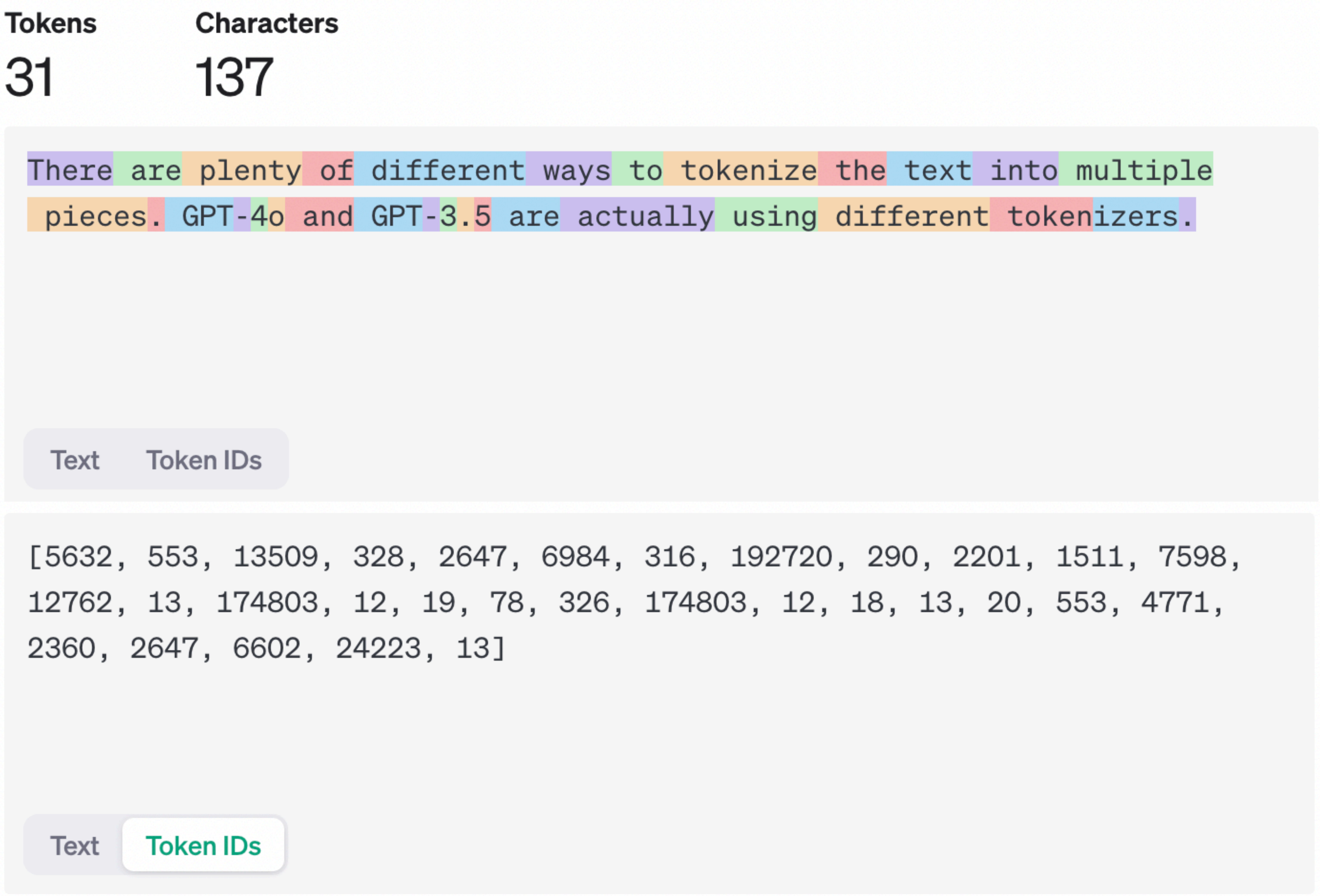
- Unlike images, translating text into vectors is not straightforward
  - Unicode? ASCII? A vs. B vs. C

- Typically goes through:
  - Normalization
  - Pre-tokenization
  - Tokenization
  - Embedding



Preprocessing

- The first three steps (normalization ~ tokenization) are responsible for chunking the text and mapping them into codes.



Preprocessing

- The last step (embedding) maps each chunk to a vector
  - Want to keep our dictionary small enough for handling

```
[5632, 553, 13509, 328, 2647, 6984, 316, 192720, 290, 2201, 1511, 7598,
12762, 13, 174803, 12, 19, 78, 326, 174803, 12, 18, 13, 20, 553, 4771,
2360, 2647, 6602, 24223, 13]
```

Text | Token IDs

$$[\text{token } 1] \longrightarrow \mathbf{x}_1 \in \mathbb{R}^d$$

$$[\text{token } 2] \longrightarrow \mathbf{x}_2 \in \mathbb{R}^d$$

$$\cdots$$

$$[\text{token } 30522] \longrightarrow \mathbf{x}_{30522} \in \mathbb{R}^d$$

Step 1. Normalization

- Various cleanups on the given text to reduce the data complexity
  - Remove unnecessary variations
    - Hello → hello # uppercase
    - I ate it all → I ate it all # whitespace
    - café → cafe # accent
    - e-mail → email # punctuation
  - Unify date & numeric formats
    - 01/31/2024 → 2024-01-31
    - 31st Jan. 2024 → 2024-01-31

Step 1. Normalization

- Often, this is done at a unicode level
  - There are many equivalences…
  - https://www.unicode.org/reports/tr15/

- **Note.** Some LLMs are known to use a specific unicode for “ “
  - Easy to filter out LLM-generated data from their own training data
  - Copyright uses
  - Catching cheating ;)

Subtype	Examples		
Font variants	ℌ	→	H
	ℍ	→	H
Linebreaking differences	[NBSP]	→	[SPACE]
Positional variant forms	ﻉ	→	ع
	ﻊ	→	ع
	ﻋ	→	ع
	ﻌ	→	ع
Circled variants	①	→	1
Width variants	ｶ	→	カ
Rotated variants	︷	→	{
	︸	→	}
Superscripts/subscripts	i^9	→	i9
	i_9	→	i9
Squared characters	㌀	→	アパート
Fractions	¼	→	1/4
Other	ǆ	→	dž

Step 2. Pre-tokenization

- Break down text into manageable units
  - Facilitate more accurate tokenization — i.e., chunking
  - Sometimes, prevent breaking down

- can’t $\rightarrow$ can + ’t # contraction
- some sentence. $\rightarrow$ some sentence + . # punctuation
- DMZ $\not\rightarrow$ D + MZ # abbreviation & acronym

Step 3. Tokenization

- Break down each sentence into tokens
- Many variants…

(1) Word-based tokenization

- Good semantics
- Too many vocabularies

Split on spaces

Let’s	do	tokenization!

Split on punctuation

Let	’s	do	tokenization	!

Step 3. Tokenization

(2) Character-based tokenization

- Small vocabulary size
- Bad semantics

Step 3. Tokenization

(3) Subword tokenization

- Frequent words are kept as a single token
- Rare words are subdivided
    - Reduces the expected sequence length
- How to take whitespaces into account differs from a tokenizer to another

Let’s </w>	do</w>	token	ization</w>	!</w>

Step 3. Tokenization

- As an example, we'll take a look at Byte-Pair Encoding (BPE)
  - A data-driven method to generate subword tokenization policy
  - Similar: WordPiece

- **Idea.** Merge frequent character combinations into tokens
  - Begin from the character-level tokens
  - If certain token appears together frequently, merge them
  - Repeat

Tokenization: Byte-Pair Encoding

```
"hug", "pug", "pun", "bun", "hugs"
```

- **Example.** Suppose that our text corpus consists of five words
  - Initial vocabulary: [b, g, h, n, p, s, u]
  - Count the word frequencies: `("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)`
  - Use this to count subword frequencies

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

Tokenization: Byte-Pair Encoding

```
"hug", "pug", "pun", "bun", "hugs"
```

- **Example.** Suppose that our text corpus consists of five words
  - Initial vocabulary: [b, g, h, n, p, s, u]
  - Count the word frequencies: `("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)`
  - Use this to count subword frequencies

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

  - Expand the vocabulary

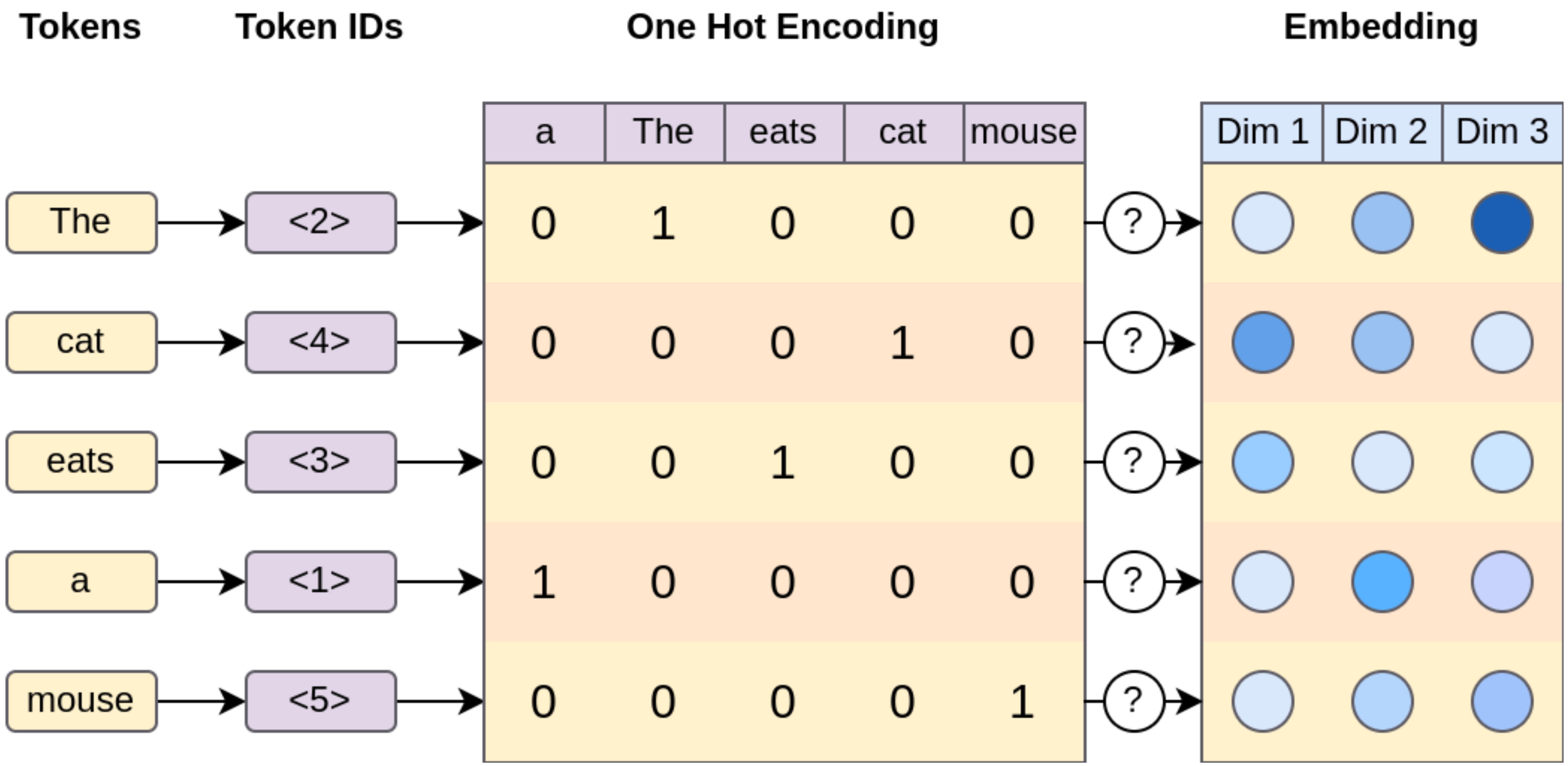
```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

  - Repeat, until the desired vocabulary size is met

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

Step 4. Embedding

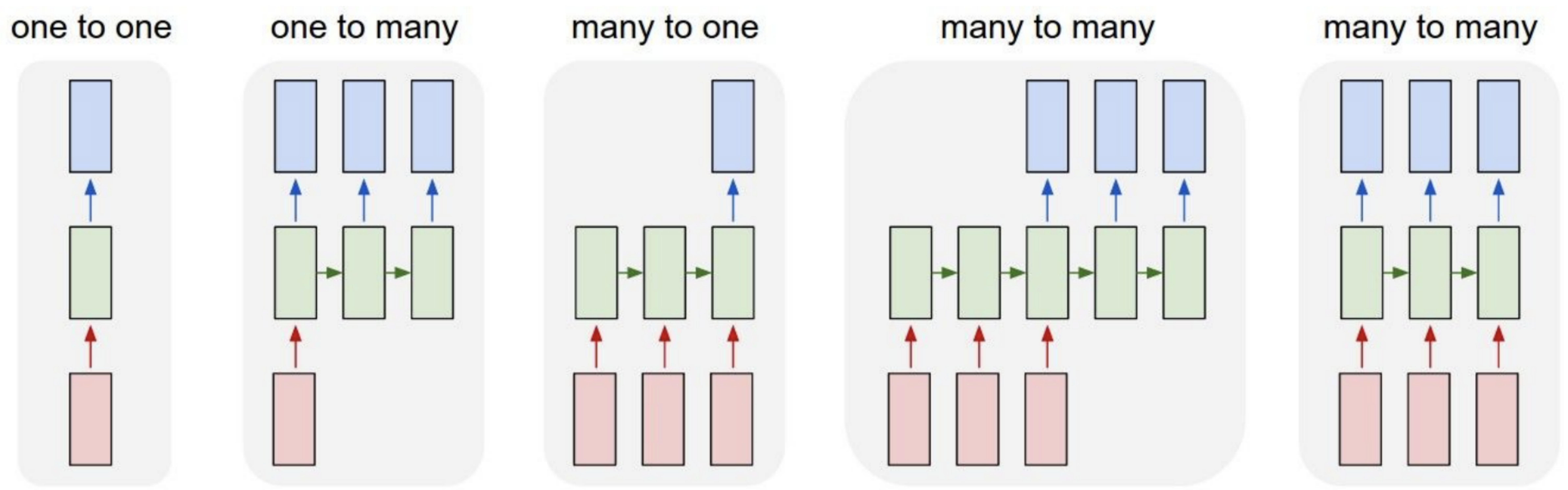
- Token IDs are translated into one-hot encodings, and then to embeddings
  - Implementable with lookup tables
  - Embedding is trainable as well — more on this later



Architectures

Architectures

- We will cover two architectures that are designed for sequences
  - Recurrent neural nets (RNNs)
  - Transformers

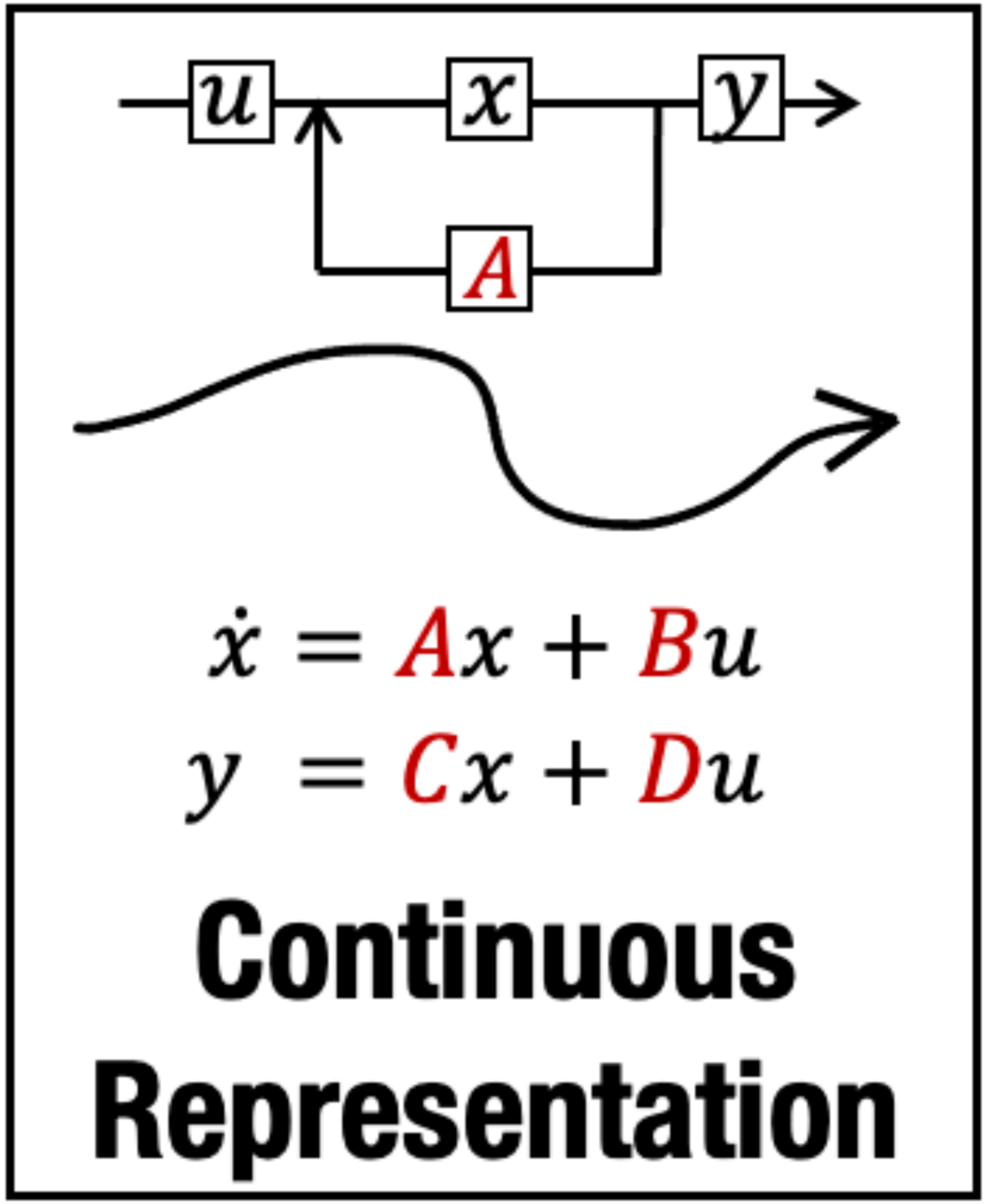


RNNs

State-space models

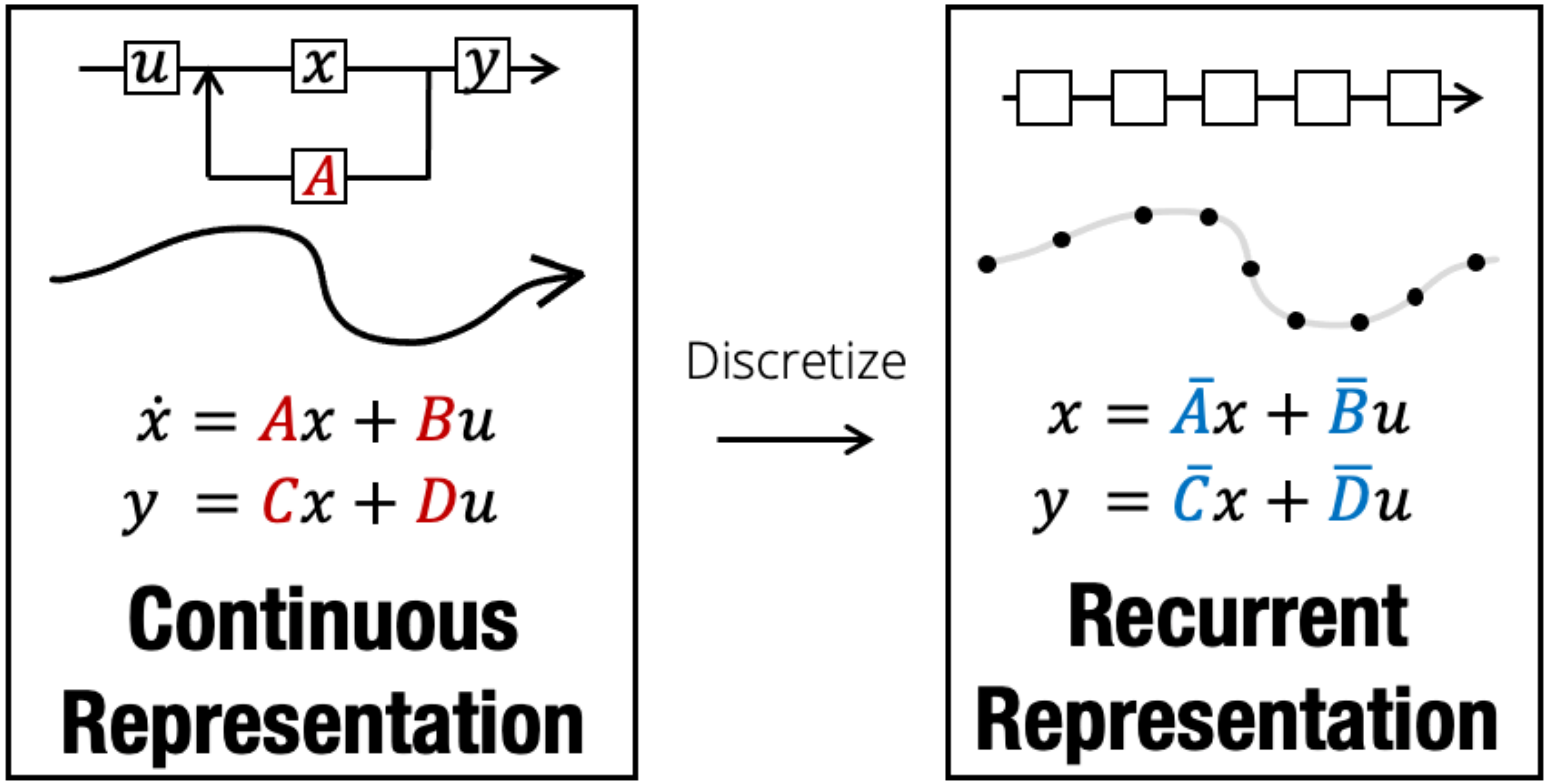
- **Idea.** Handle sequential input using a state-space model (SSM)

- **Recall: Continuous SSMs**
  - Handles a sequential input $u(t)$ by accumulating the internal state $x(t)$
  - Sequential output $y(t)$ is determined by both $u(t), x(t)$
  - Parameterized by transition matrices A, B, C, D



State-space models

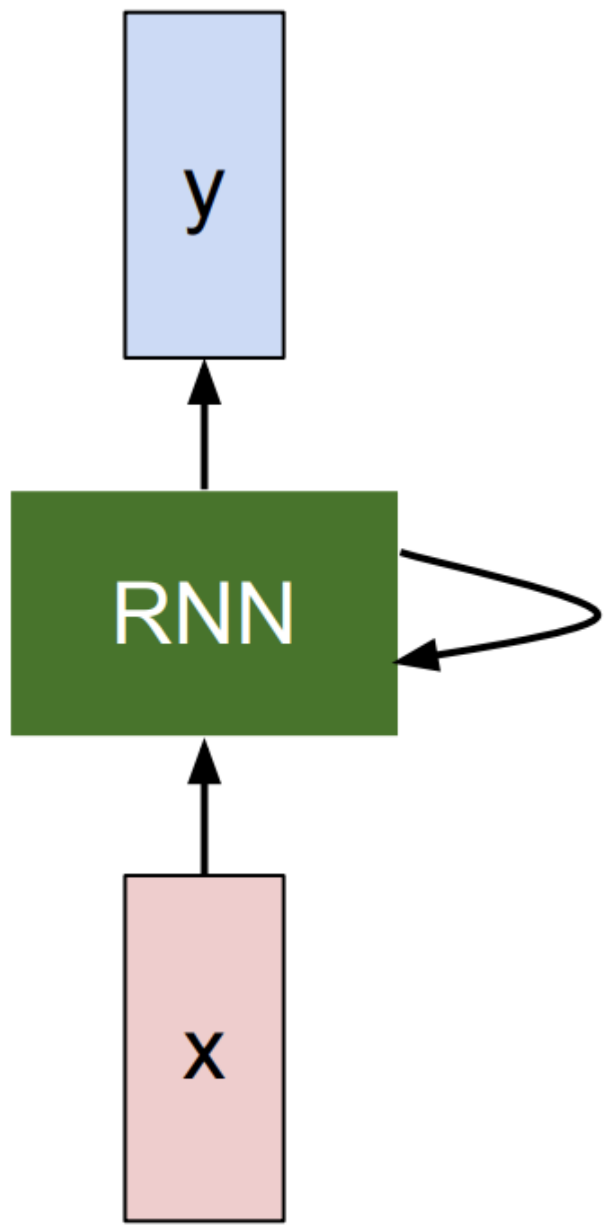
- Discrete inputs can be handled similarly, with discrete SSMs
  - One can use $$x(t) = \bar{A}x(t-1) + \bar{B}u(t)$$ $$y(t) = \bar{C}x(t) + \bar{D}u(t)$$



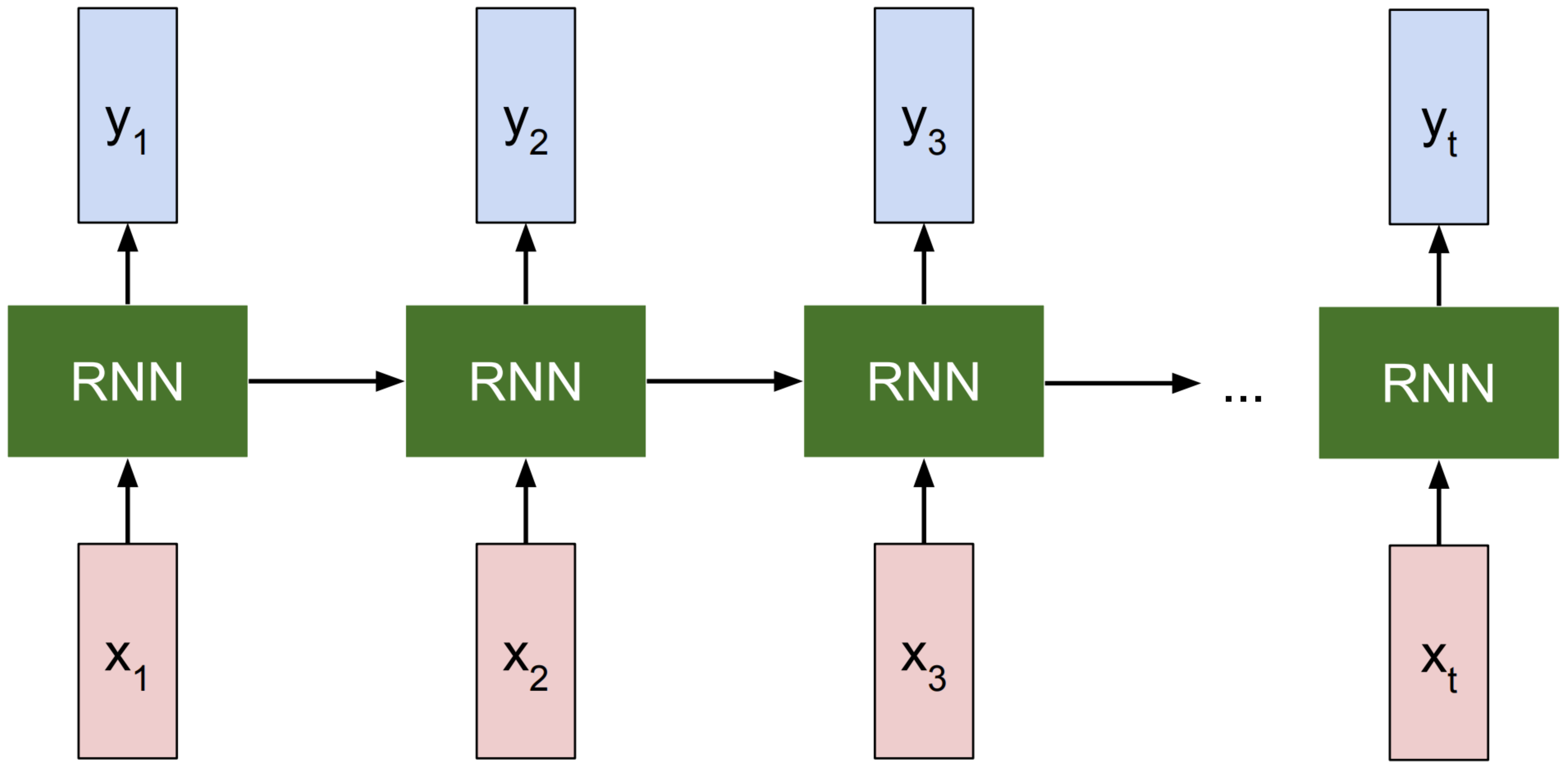
Recurrent layer

- More generally, we can use nonlinear modules:
  - $\mathbf{h}_t = g_\theta(\mathbf{x}_t; \mathbf{h}_{t-1})$
  - $\hat{\mathbf{y}}_t = f_\theta(\mathbf{x}_t; \mathbf{h}_{t-1})$

note: change of notations



RNN



RNN (unrolled)

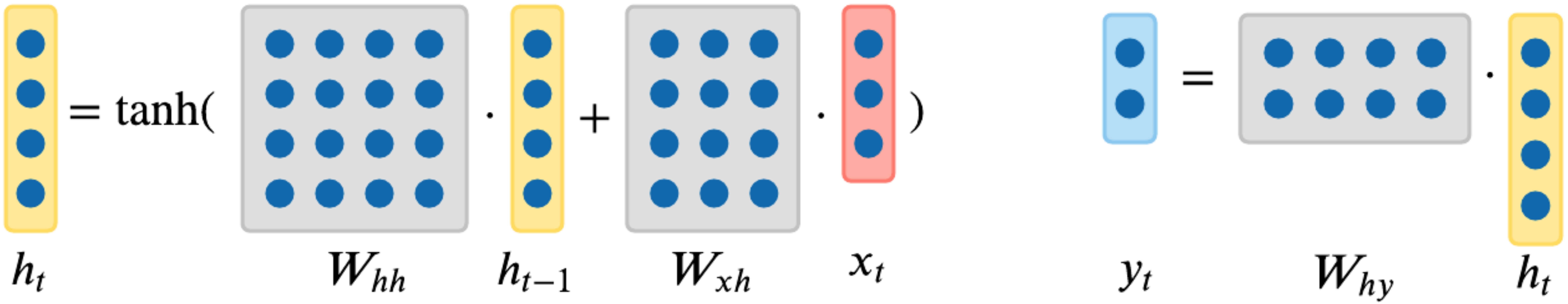
Recurrent layer

- In the simplest parameterization, the recurrence is formalized as:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{\mathrm{hh}}\mathbf{h}_{t-1} + \mathbf{W}_{\mathrm{xh}}\mathbf{x}_t)$$

$$\mathbf{y}_t = \mathbf{W}_{\mathrm{hy}}\mathbf{h}_t$$

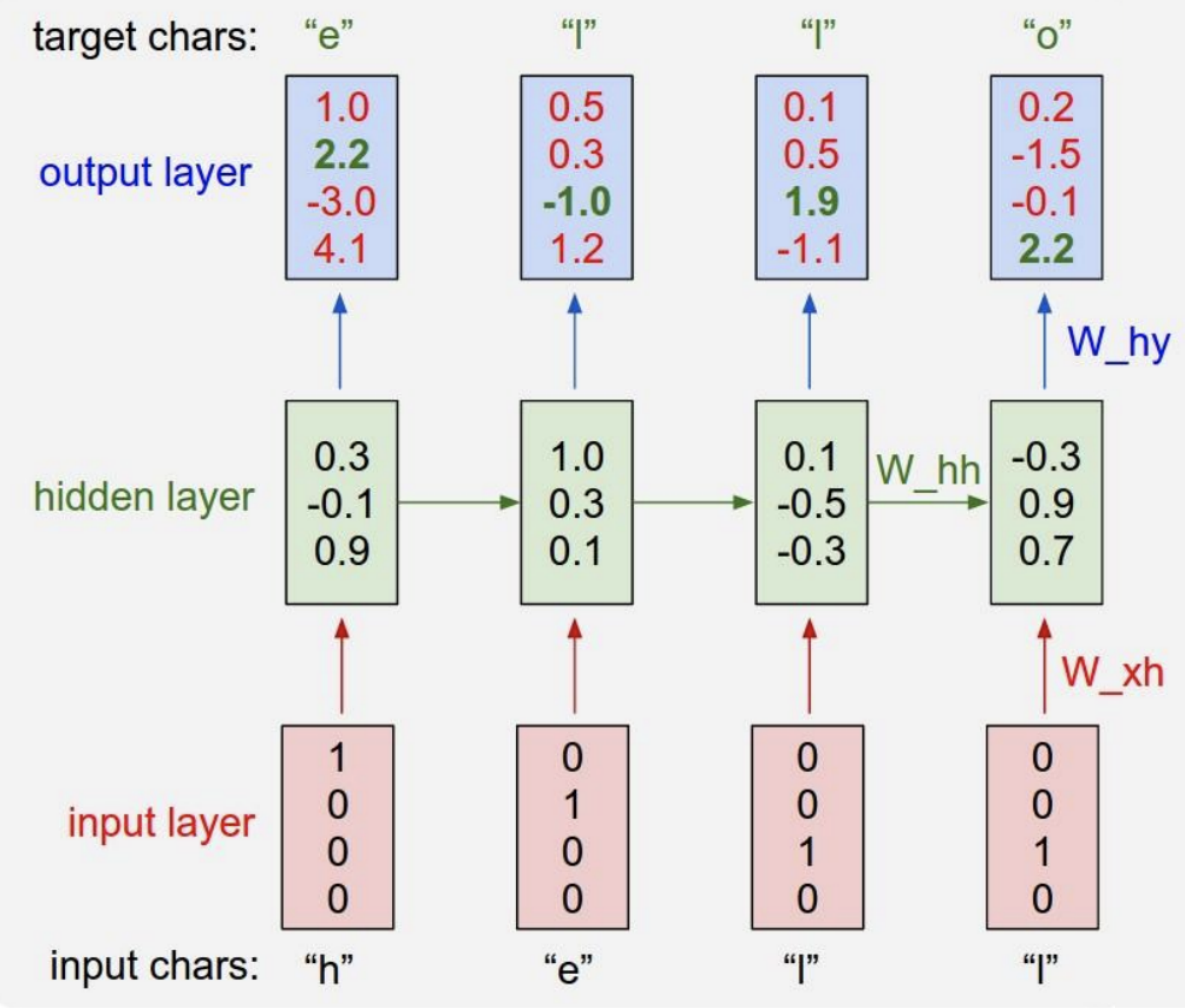
Rumelhart (1986)



Example: RNN for sentence generation

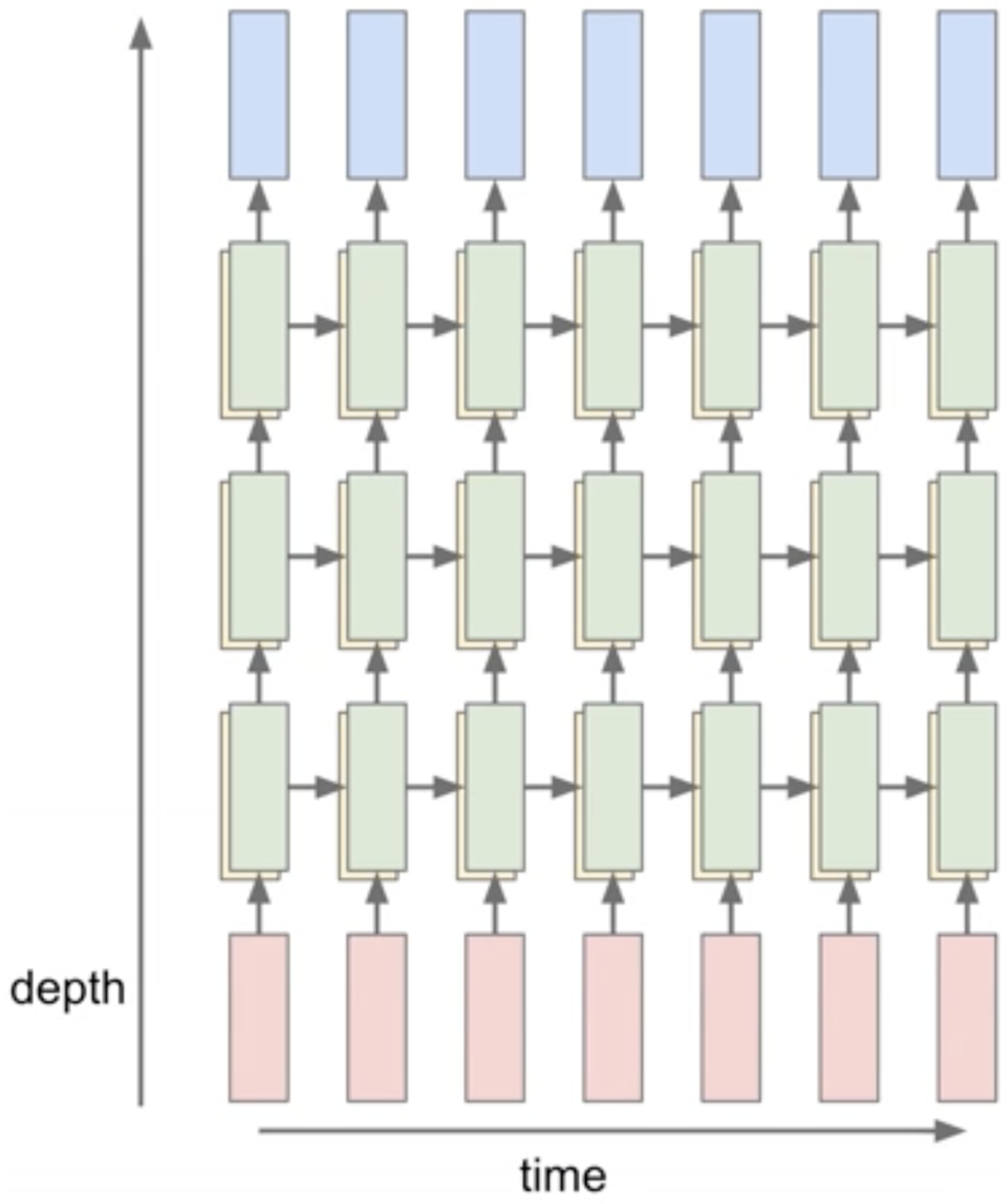
- Consider the case with:
  - Character-level tokens
  - Single-layer RNN
  - Identity embedding

- Sentence generation can be done by feeding the generated character as a new input
  - Similar in GPT



Deep recurrent neural network

- Recurrent neural networks can be stacked to build a deep network
    - Strengthens the “memory”
    - Quite difficult to train
        - Vanishing/exploding gradient



Gradients of RNNs

- Suppose that we have computed the loss at time t # i.e., L_t
  - We want to use this to update the hidden state at time 1 # i.e., $\mathbf{h}_1$

- The partial derivative of the current state w.r.t. past state is:

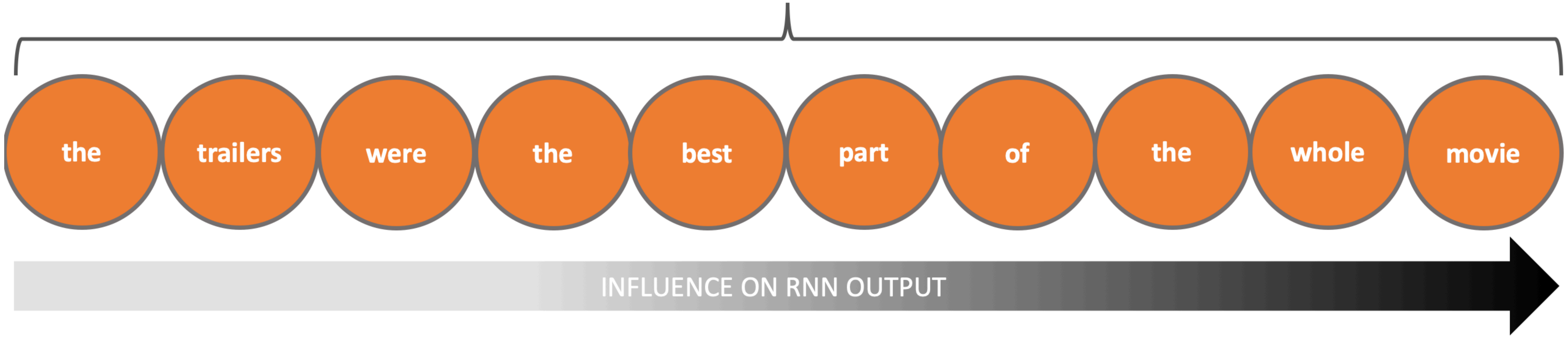
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \tanh'(\mathbf{W}_{\mathrm{hh}}\mathbf{h}_{t-1} + \mathbf{W}_{\mathrm{xh}}\mathbf{x}_t)\mathbf{W}_{\mathrm{hh}}$$

- Then, we have:

$$\frac{\partial L_t}{\partial \mathbf{h}_1} = \frac{\partial L_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \cdot \cdots \cdot \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} = \frac{\partial L_t}{\partial \mathbf{h}_t} \cdot \left(\prod_{i=2}^{t} \tanh'(\mathbf{W}_{\mathrm{hh}}\mathbf{h}_{i-1} + \mathbf{W}_{\mathrm{xh}}\mathbf{x}_i) \right) \mathbf{W}_{\mathrm{hh}}^{t-1}$$

Gradients of RNNs

"the trailers were the best part of the whole movie."



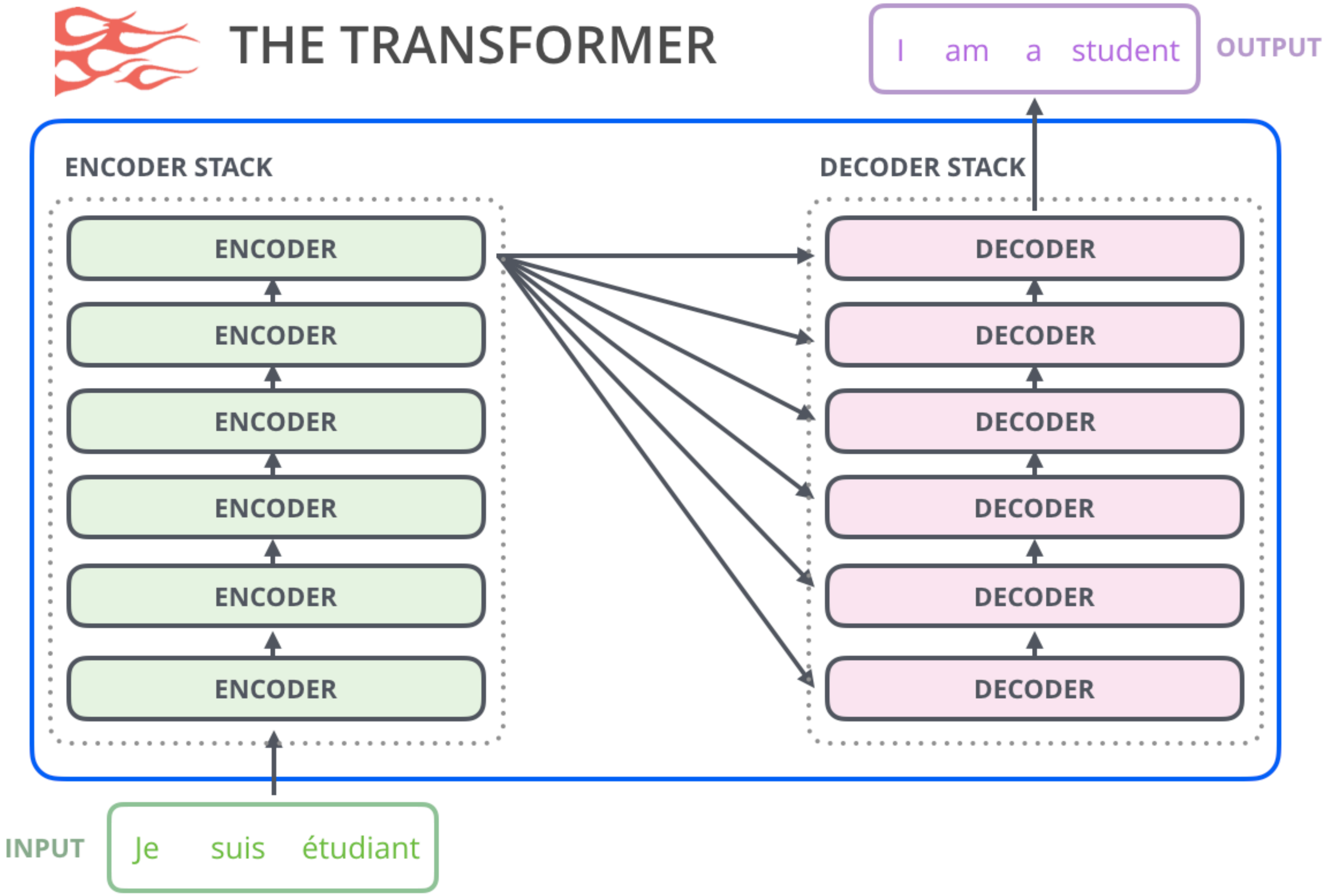
- **Solution.**
  - Adopt extra modules that are designed for long-term dependencies
    - e.g., LSTM
  - Let the very old input directly affect the new output
    - e.g., Transformers

Transformers

Transformers

- Consists of:
  - a stack of encoder blocks
  - a stack of decoder blocks

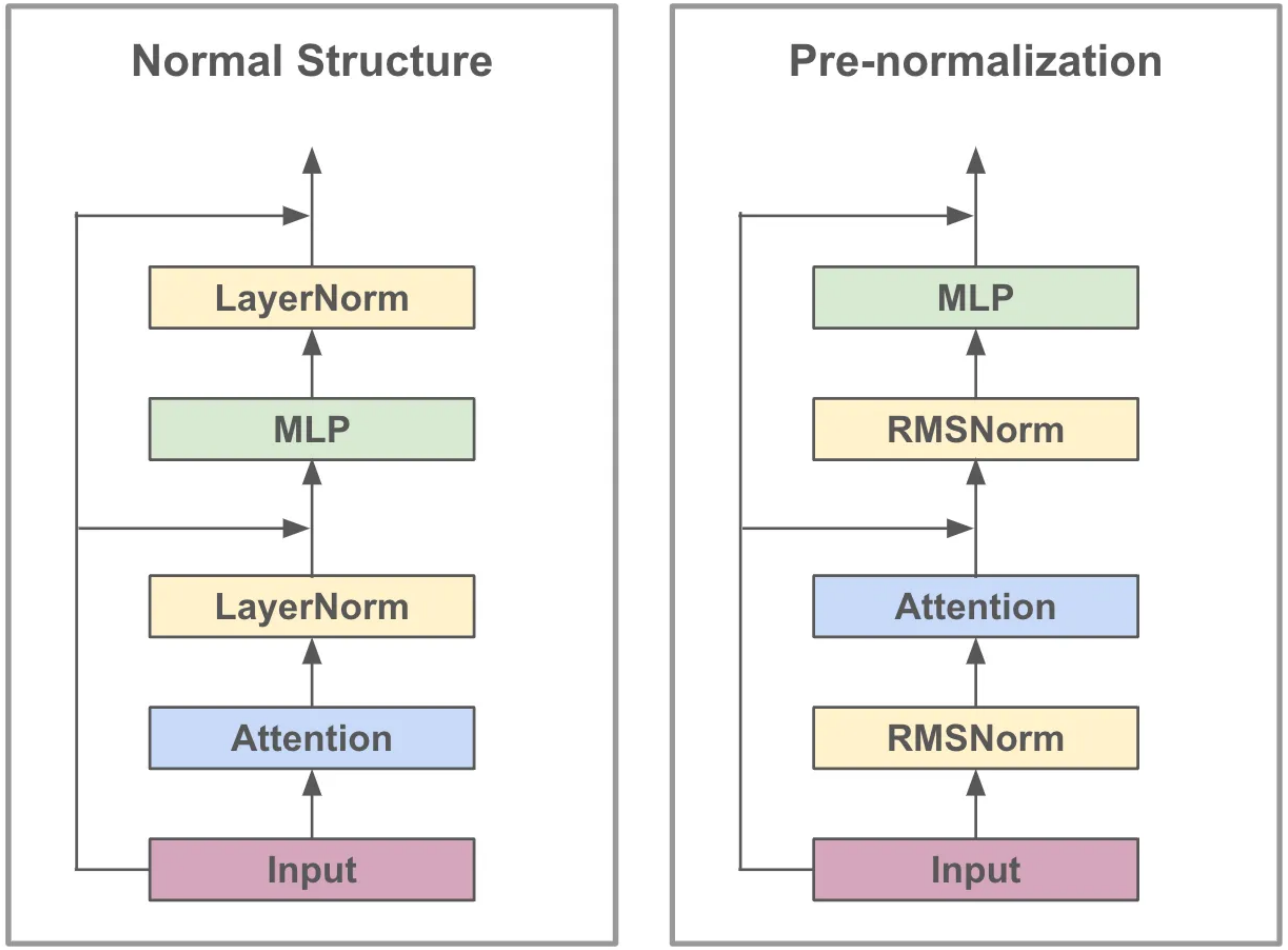
- **Encoder-only.** BERT
- **Decoder-only.** GPT (our focus)



Transformers

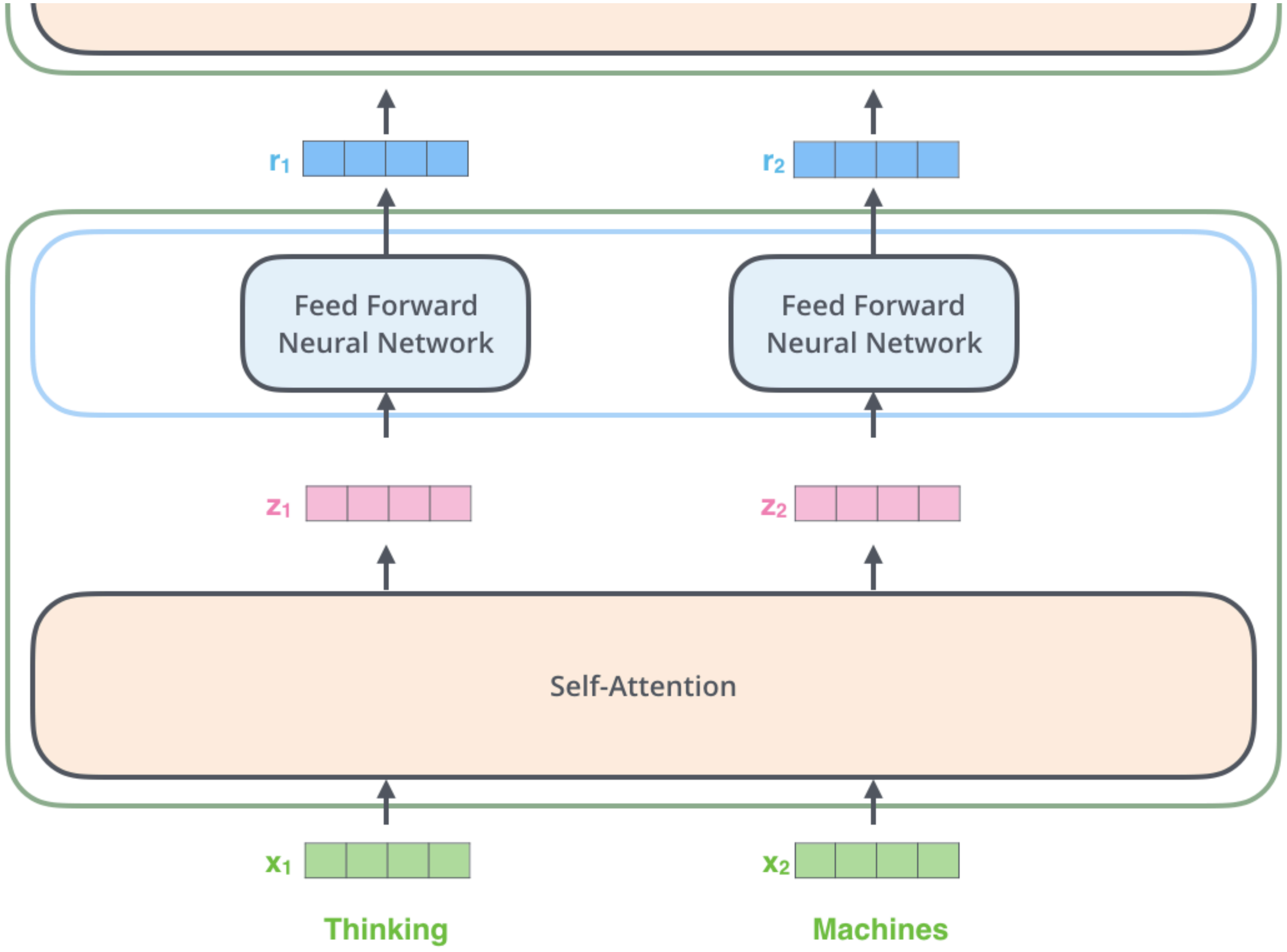
- Each block consists of four elements
  - Multi-head Self Attention (MHA)
  - Feedforward network (FFN)
  - LayerNorm / RMSNorm
  - Residual connection

- Only new component is the MHA



MHA and FFN

- MHA and FFN plays a complementary role
  - **MHA.** Models inter-token interaction
  - **FFN.** Applies intra-token operation # i.e., same op. for all tokens



Self-Attention

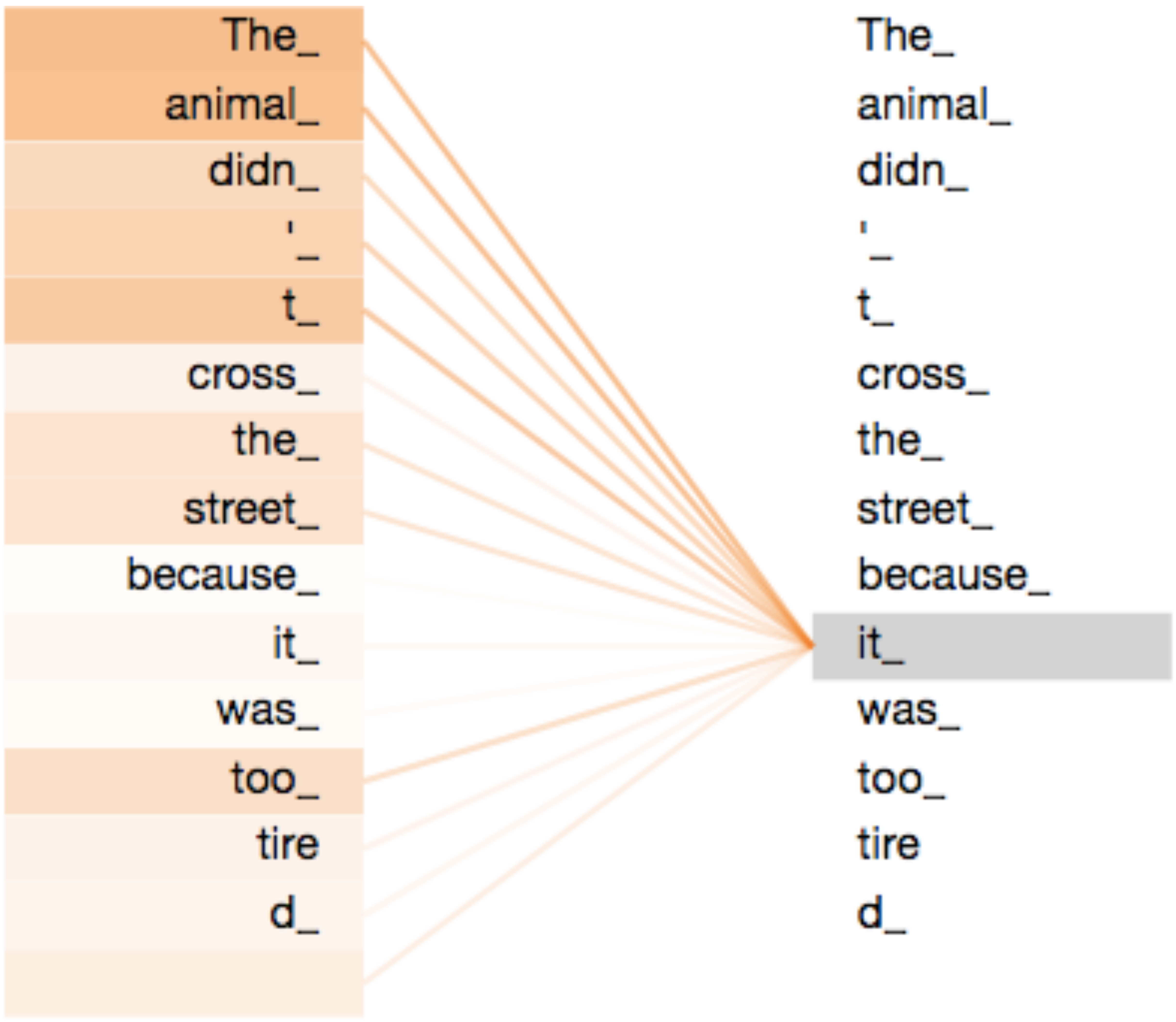
- Roughly, the self-attention layer does two things

(1) For i-th token, we measure the relevance of {1,…,N}th tokens

- called “attention score”

(2) Take a relevance-weighted sum of other token’s “values” to compute the i-th token’s output

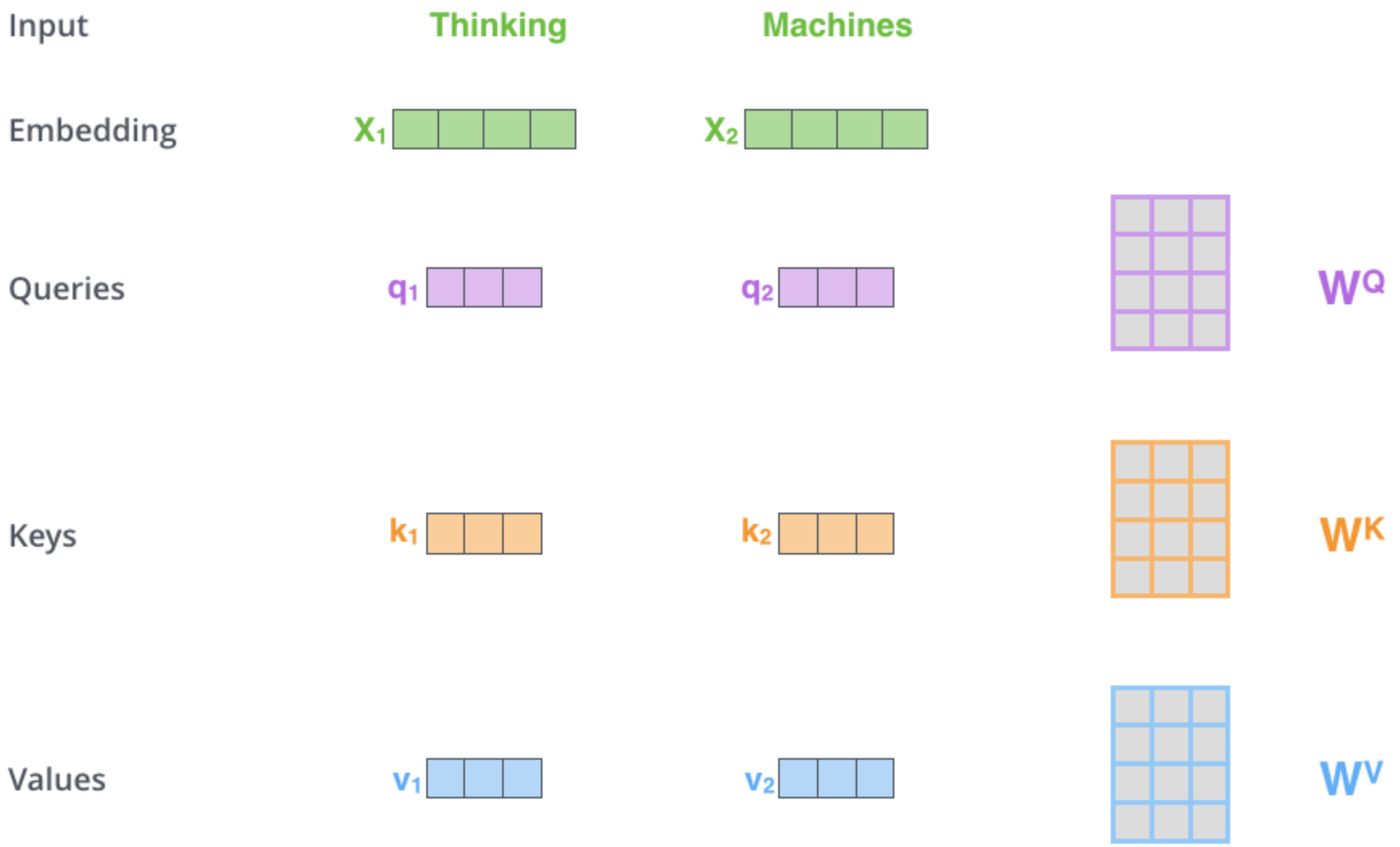
$$o_i = \sum_j \text{attention}(i,j) \cdot \text{val}(j)$$



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

Self-Attention

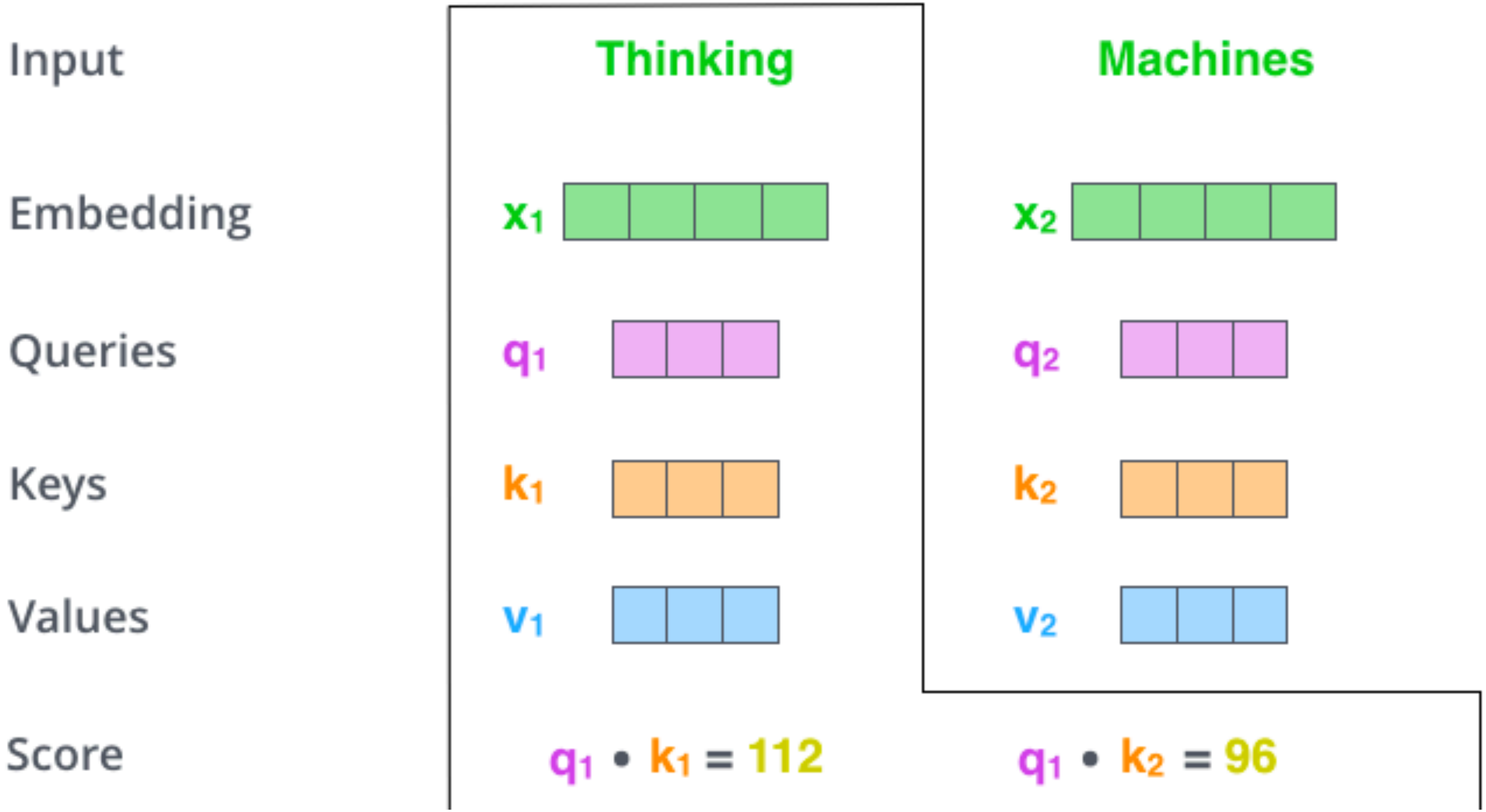
- **Step 1.** For each token, we compute query, key, and value
  - Weight matrices are shared over all tokens



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

Self-Attention

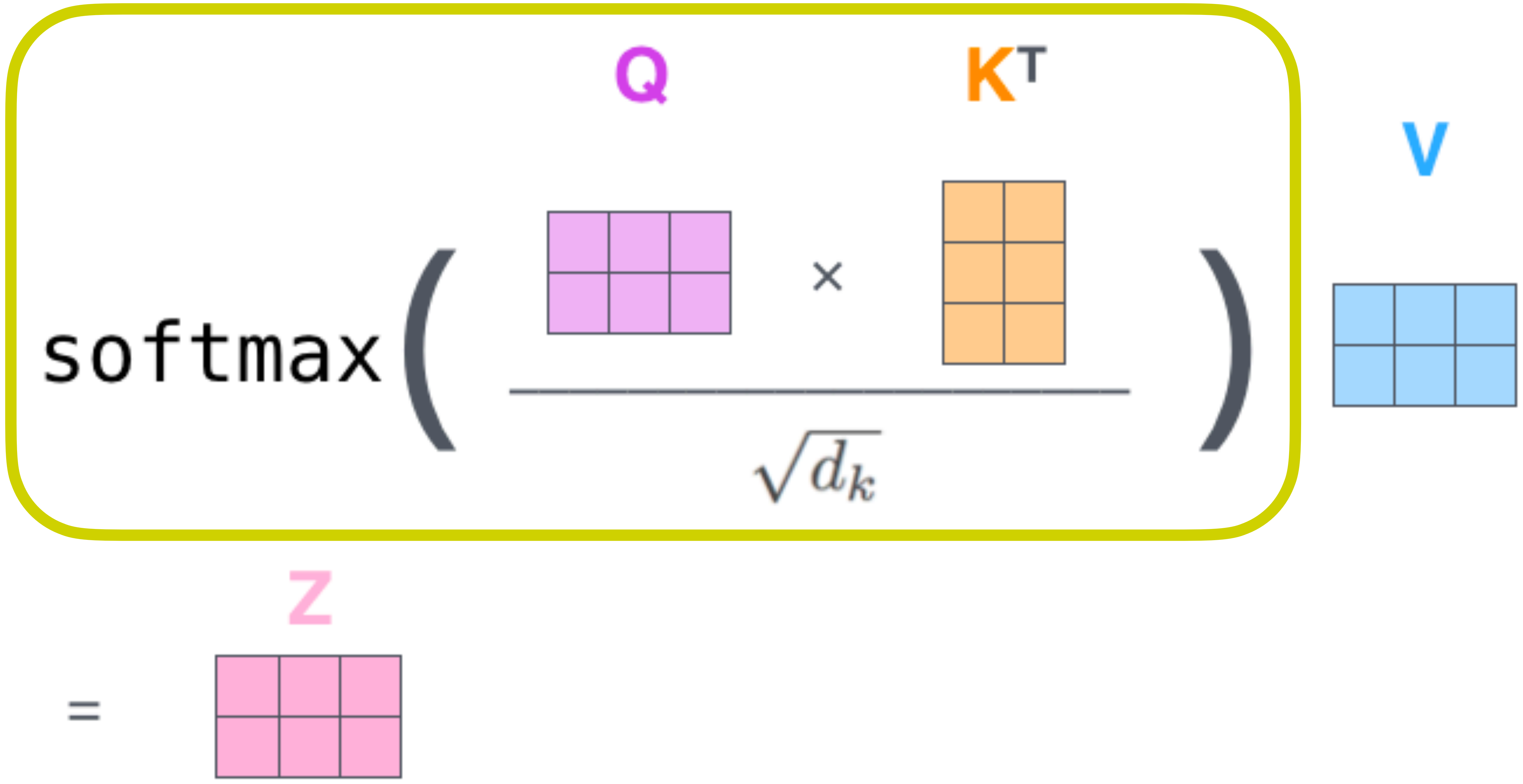
- **Step 2.** Compute dot products of the query (self) and key (self, others)



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

Self-Attention

- **Step 3.** Compute output as a weighted sum of values
  - weighted by the softmax of dot products (attention score)
  - normalized by the dimensions

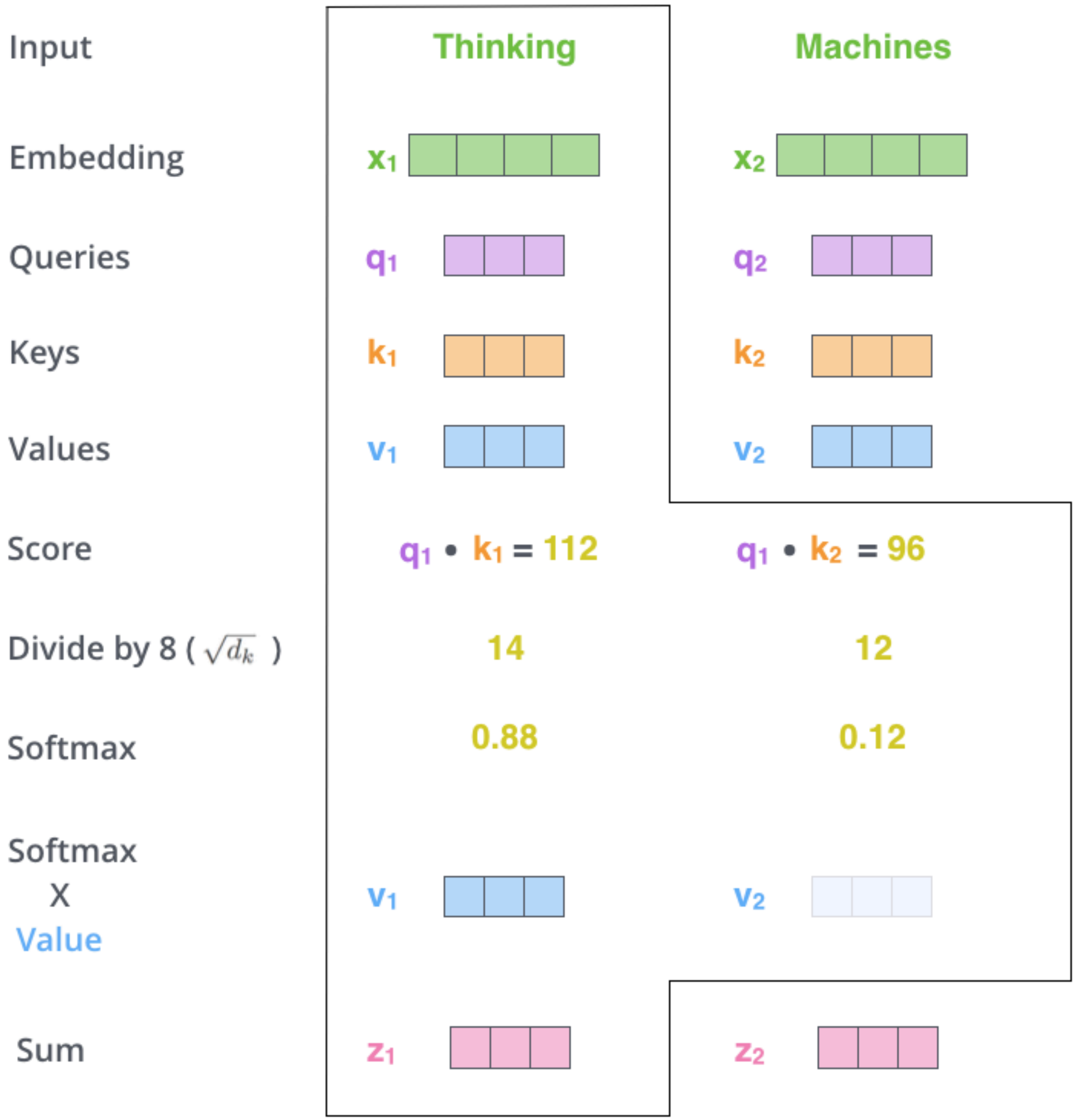


https://jalammar.github.io/illustrated-transformer/

Computation & Memory

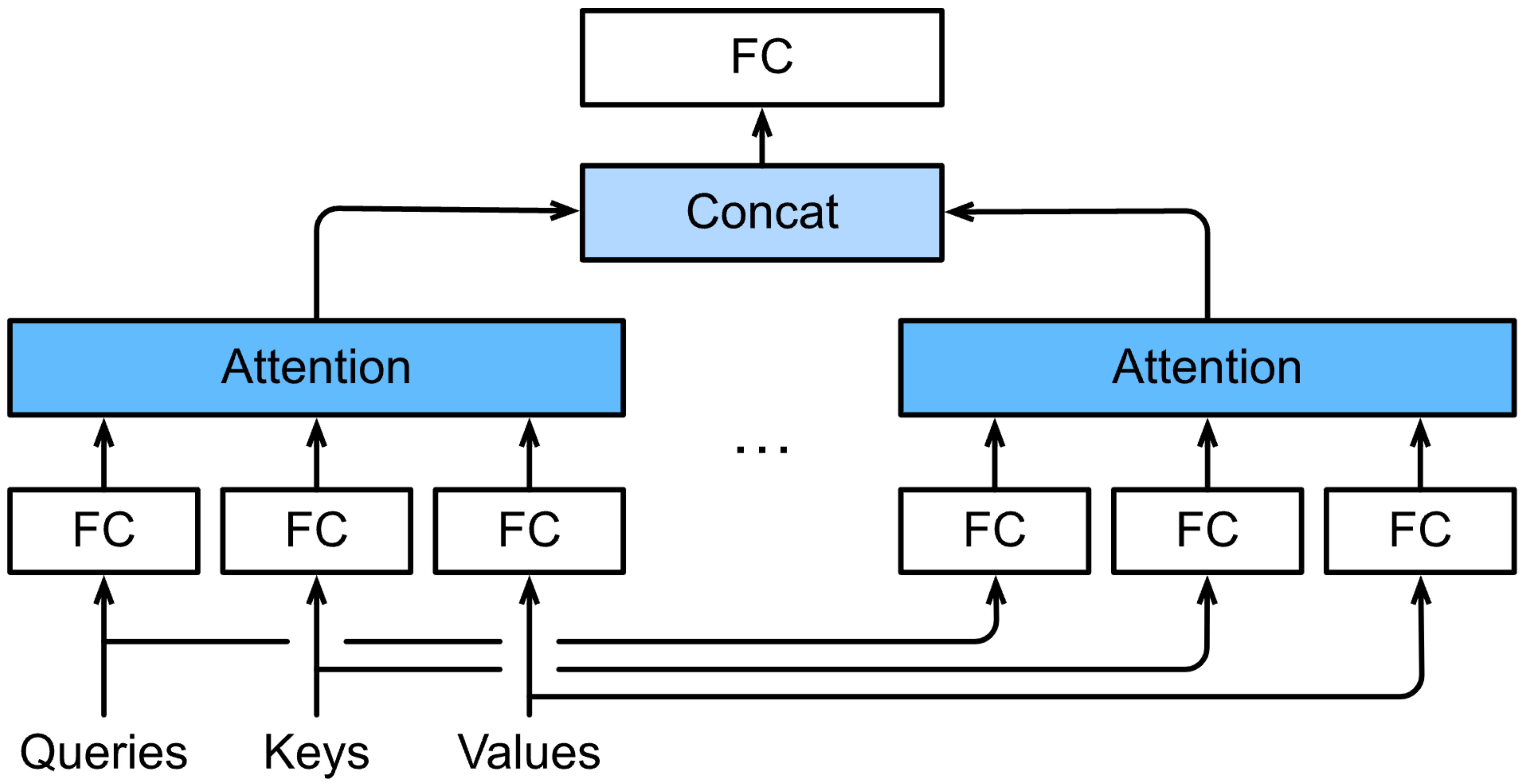
- Suppose that we have n tokens
  - Q/K/V computation: $O(n)$
  - Attention: $O(n^2)$
  - Weighted sum: $O(n^2)$

- Unlike RNNs, requires quadratic operations with respect to the sequence length!
  - Why we need many GPUs



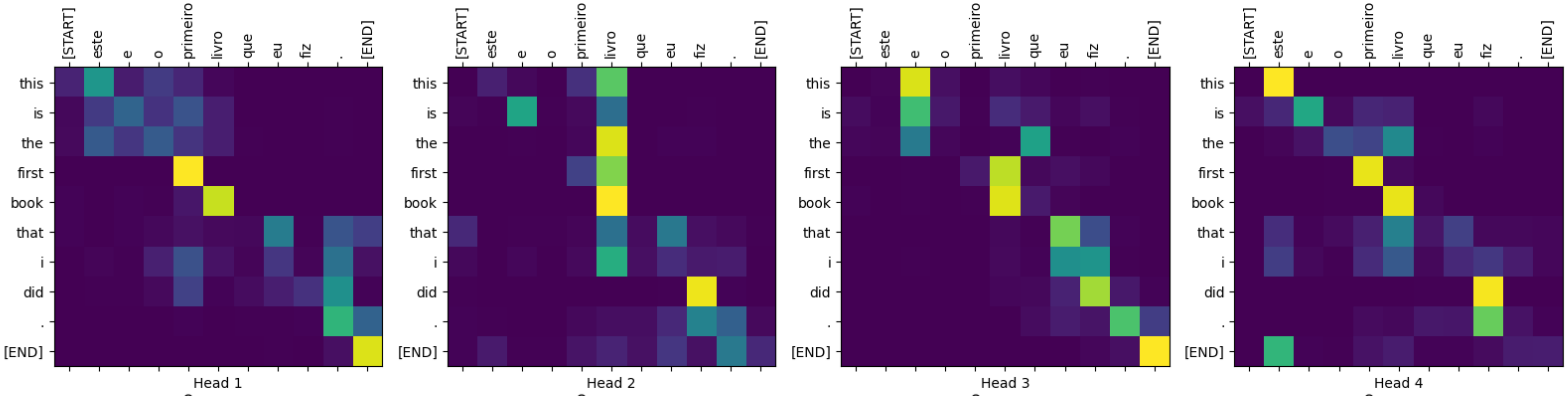
Multi-head Self-Attention

- Typically, we use multiple self-attention layers in a transformer block
  - Computed in parallel
  - Outputs are concatenated and linearly projected



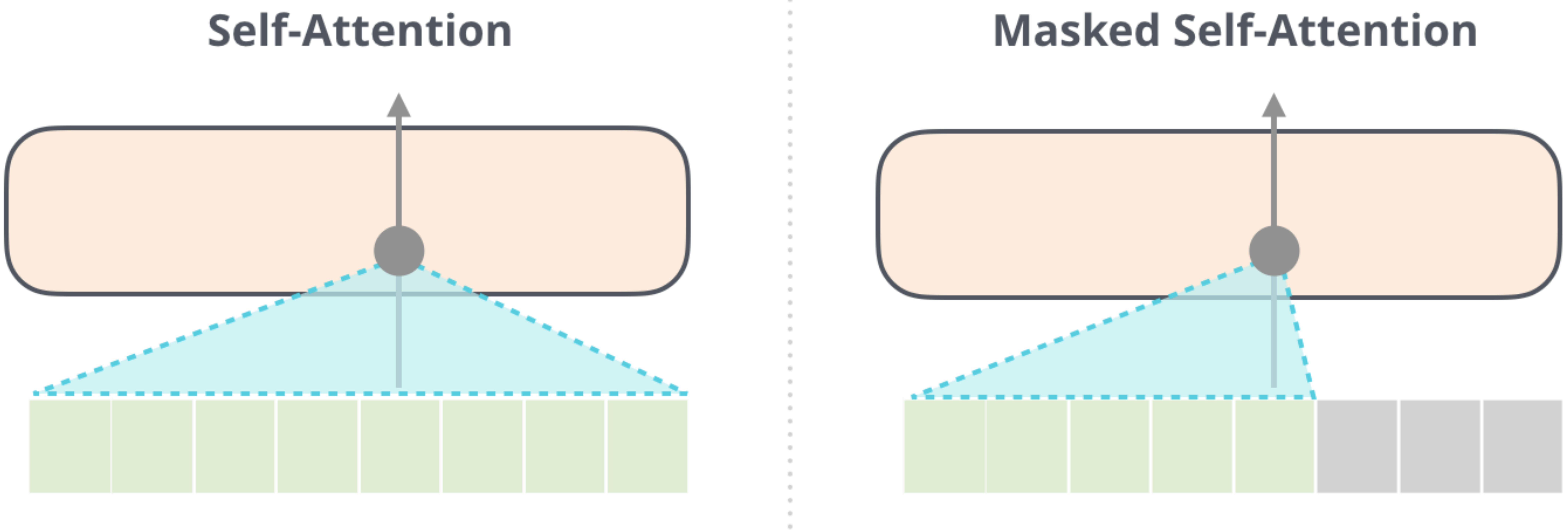
Multi-head Self-Attention

- Each attention head tend to capture diverse attention patterns
    - Similar to multiple convolution filters in a layer



Causal masking for attention

- In decoder-only models (like GPT), the self-attention layers are masked
  - For generating t-th token, the model can only utilize $\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}$

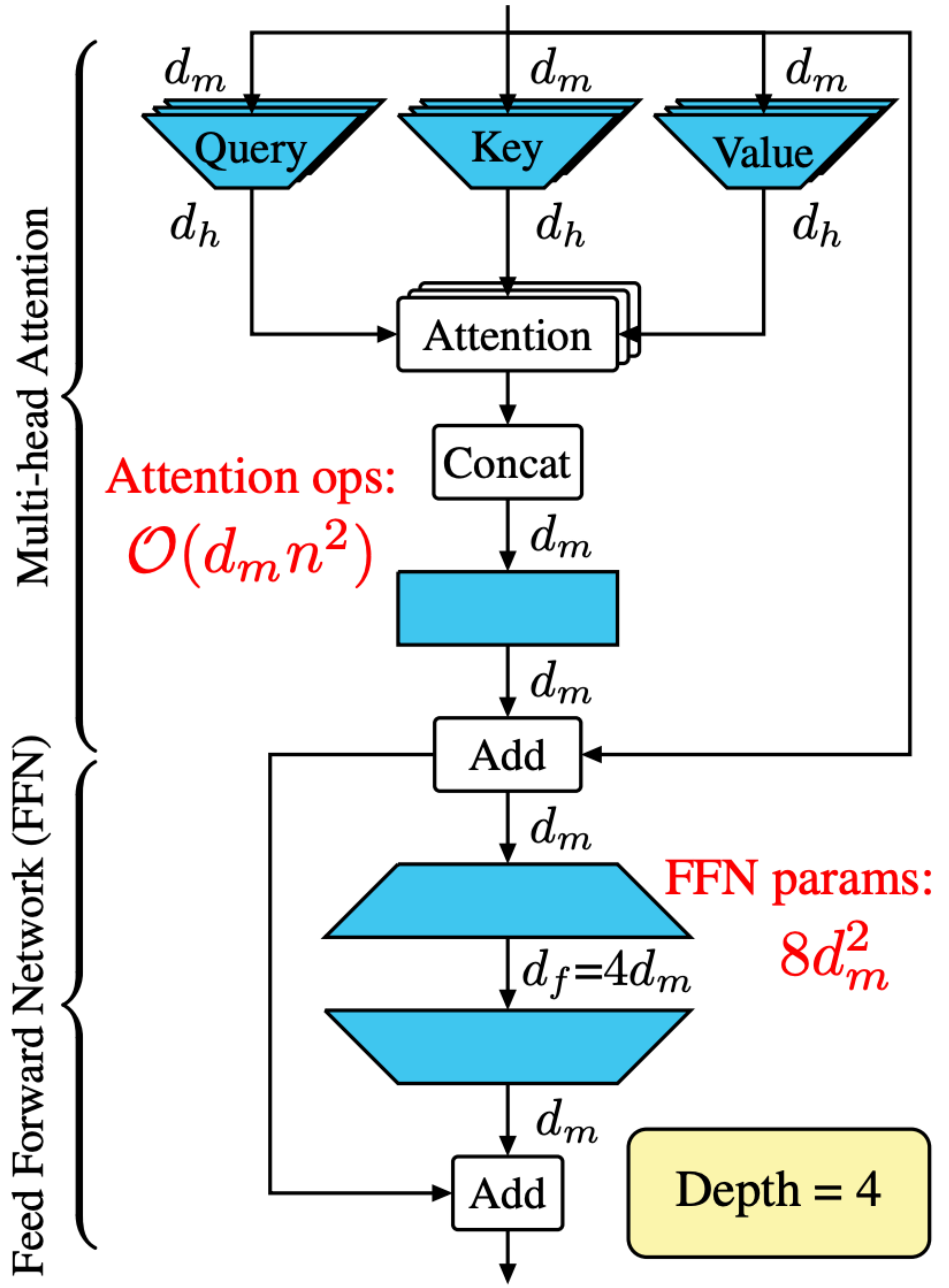


Mehta et al., "DeLight: Deep and lightweight transformer," ICLR 2021

Feedforward network

- Fully-connected layers that follow the MHA
  - **Basic.** Use two-layer MLP
    - Inverted bottleneck structure
  - Tend to be very compute-heavy
    - Especially so for larger models

1	description	FLOPs / update	% FLOPS MHA	% FLOPS FFN	% FLOPS attn	% FLOPS logit
8	OPT setups					
9	760M	4.3E+15	35%	44%	14.8%	5.8%
10	1.3B	1.3E+16	32%	51%	12.7%	5.0%
11	2.7B	2.5E+16	29%	56%	11.2%	3.3%
12	6.7B	1.1E+17	24%	65%	8.1%	2.4%
13	13B	4.1E+17	22%	69%	6.9%	1.6%
14	30B	9.0E+17	20%	74%	5.3%	1.0%
15	66B	9.5E+17	18%	77%	4.3%	0.6%
16	175B	2.4E+18	17%	80%	3.3%	0.3%

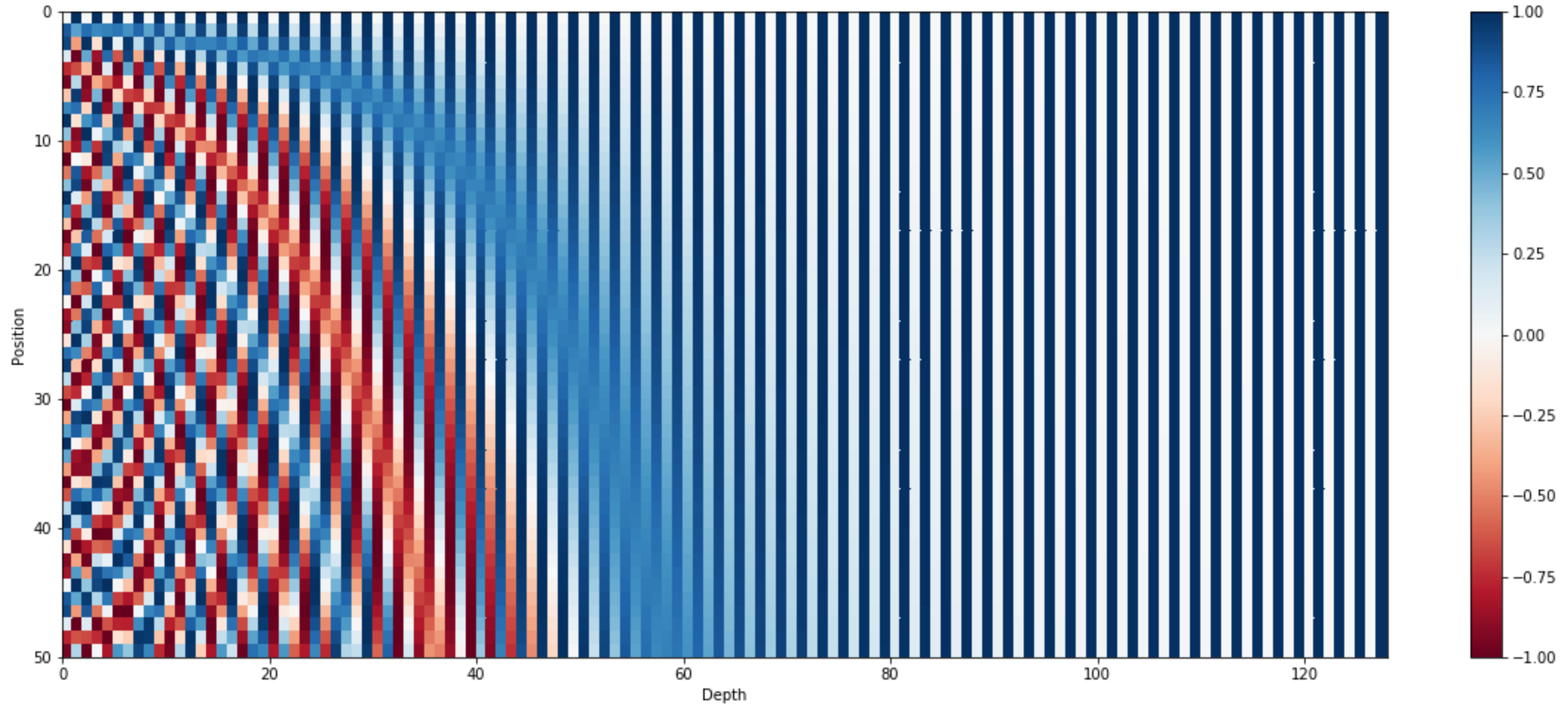


https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Positional encoding

- **Problem.** Self-attention ignores the positional information of each token
- Solution. Add position-specific vector to the token embedding
  - called positional encoding
  - added at initial embedding or in each block

$$\vec{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases} \qquad \omega_k = \frac{1}{10000^{2k/d}}$$



More references

- **Beginner.** Jay Alammar’s blog posts
  - https://jalammar.github.io/illustrated-transformer/

- **Advanced.**
  - Phuong and Hutter, “Formal algorithms for Transformers,” 2022
    - https://arxiv.org/abs/2207.09238
  - He and Hoffman, “Simplifying Transformer Blocks,” 2023
    - https://arxiv.org/abs/2311.01906

</lecture 19>