Language: Architectures

EECE454 Intro. to Machine Learning Systems

Fall 2024

Overview

- **Last two weeks.** Deep learning for visual data (specifically, image)
    - Architectures
    - Scalable training
    - Generative model

Overview

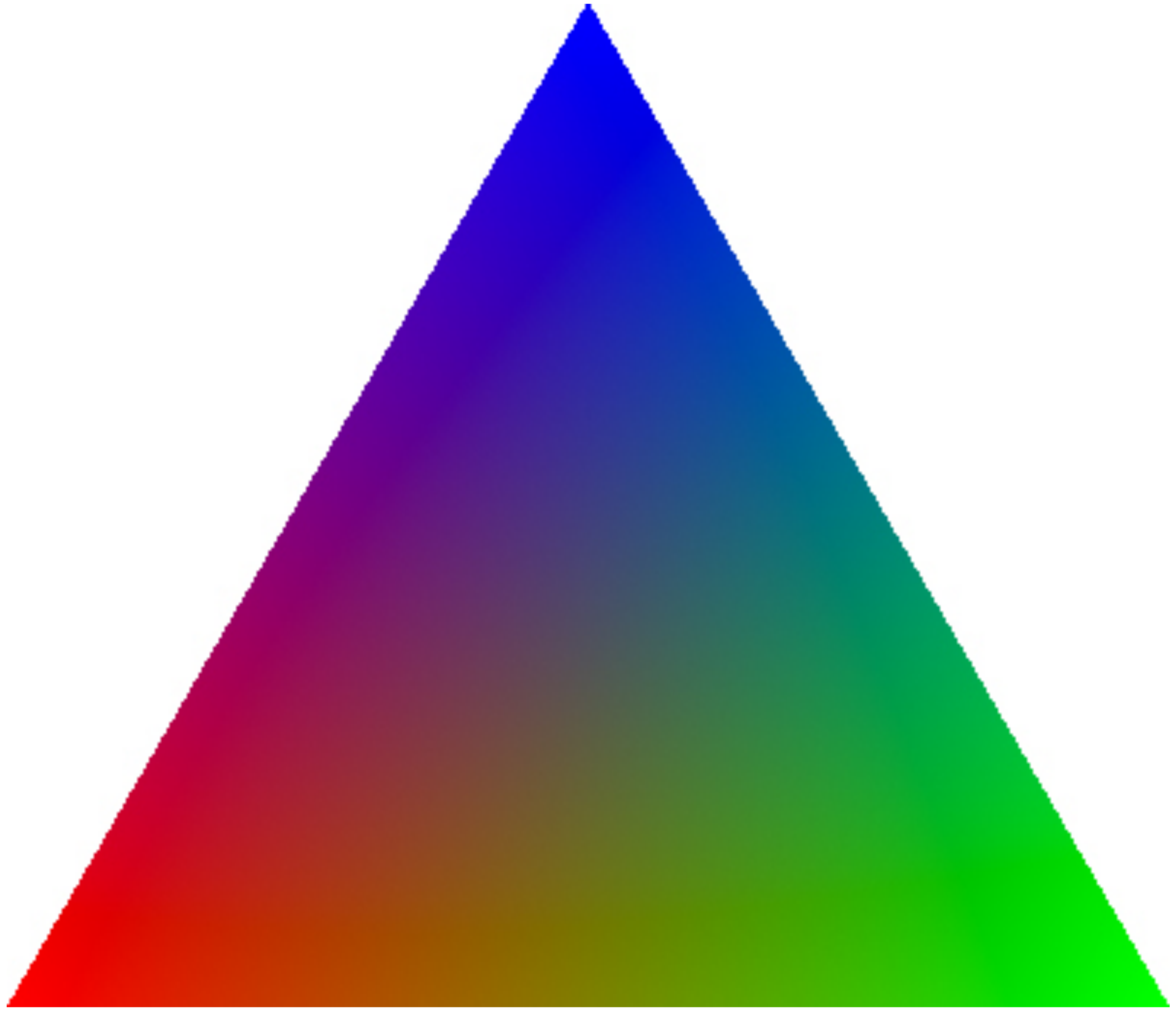
- **Last two weeks.** Deep learning for visual data (specifically, image)
  - Architectures
  - Scalable training
  - Generative model
- **This week.** Deep learning for language (specifically, text)
  - Architectures
    - Preprocessing
    - RNNs and Transformers
  - Language modeling

Preview: Text vs. Image

- **Question.** Why should language processing be different from image processing?

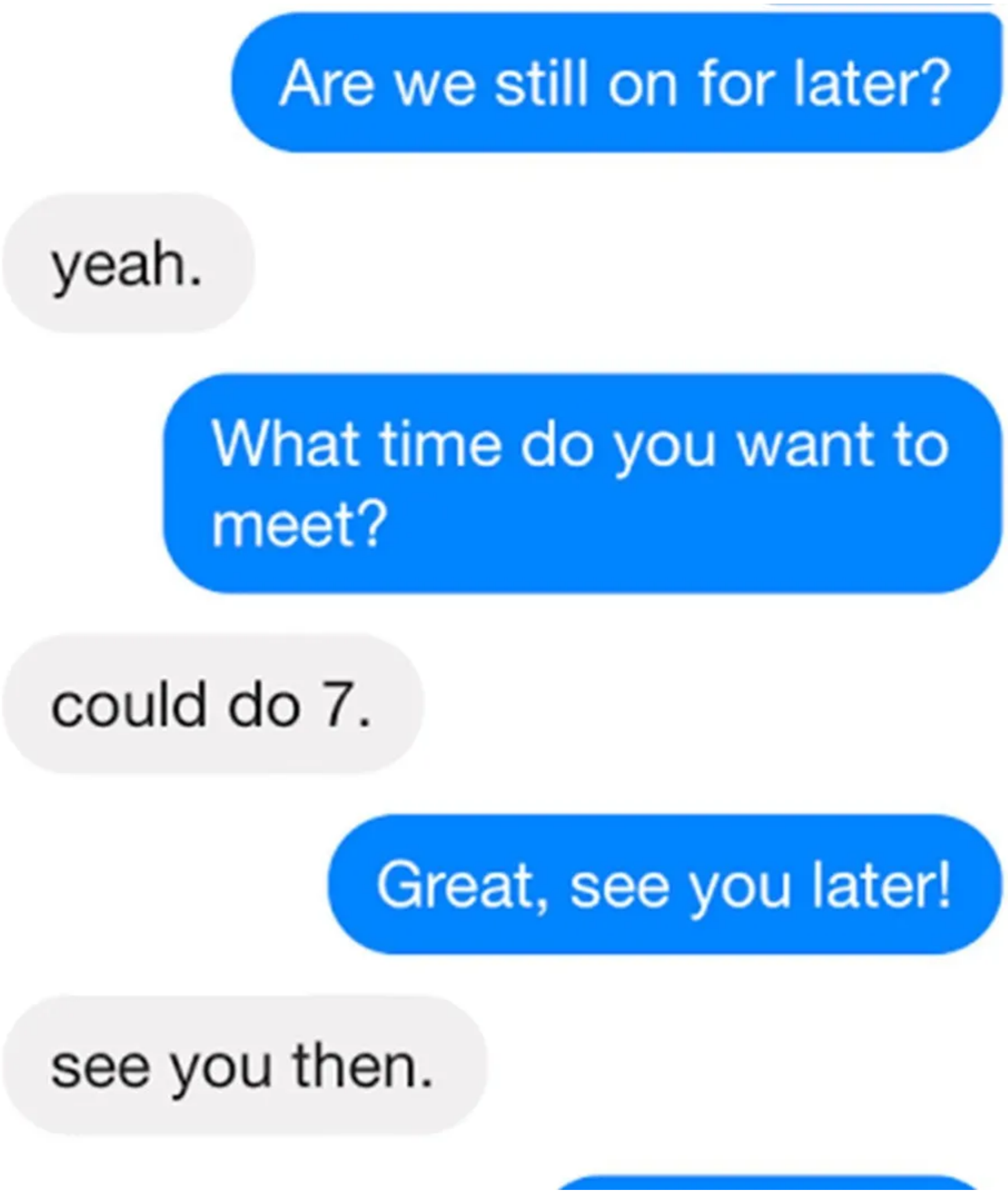
Preview: Text vs. Image

- **Question.** Why should language processing be different from image processing?
  - Language is discrete:
    - Interpolating "■" & "■" vs. "A" & "C"
    - To-do: Vectorization mechanism needed



Preview: Text vs. Image

- **Question.** Why should language processing be different from image processing?
  - Language is discrete:
    - Interpolating "■" & "■" vs. "A" & "C"
    - To-do: Vectorization mechanism needed
  - Language has variable length
    - To-do: Need a neural network architecture that can handle sequences effectively



Preview: Text vs. Image

- **Question.** Why should language processing be different from image processing?
  - Language is discrete:
    - Interpolating "■" & "■" vs. "A" & "C"
    - To-do: Vectorization mechanism needed
  - Language has variable length
    - To-do: Need a neural network architecture that can handle sequences effectively
  - Language has weaker locality than images
    - To-do: Architecture that can cover far distance

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

- Note. Later, we will see how image processing can be made similar to texts

Preprocessing

Pre-processing

- Translating text data into a sequence of vectors:
- Typically involves:
  - Normalization
  - Pre-tokenization
  - Tokenization
  - Embedding

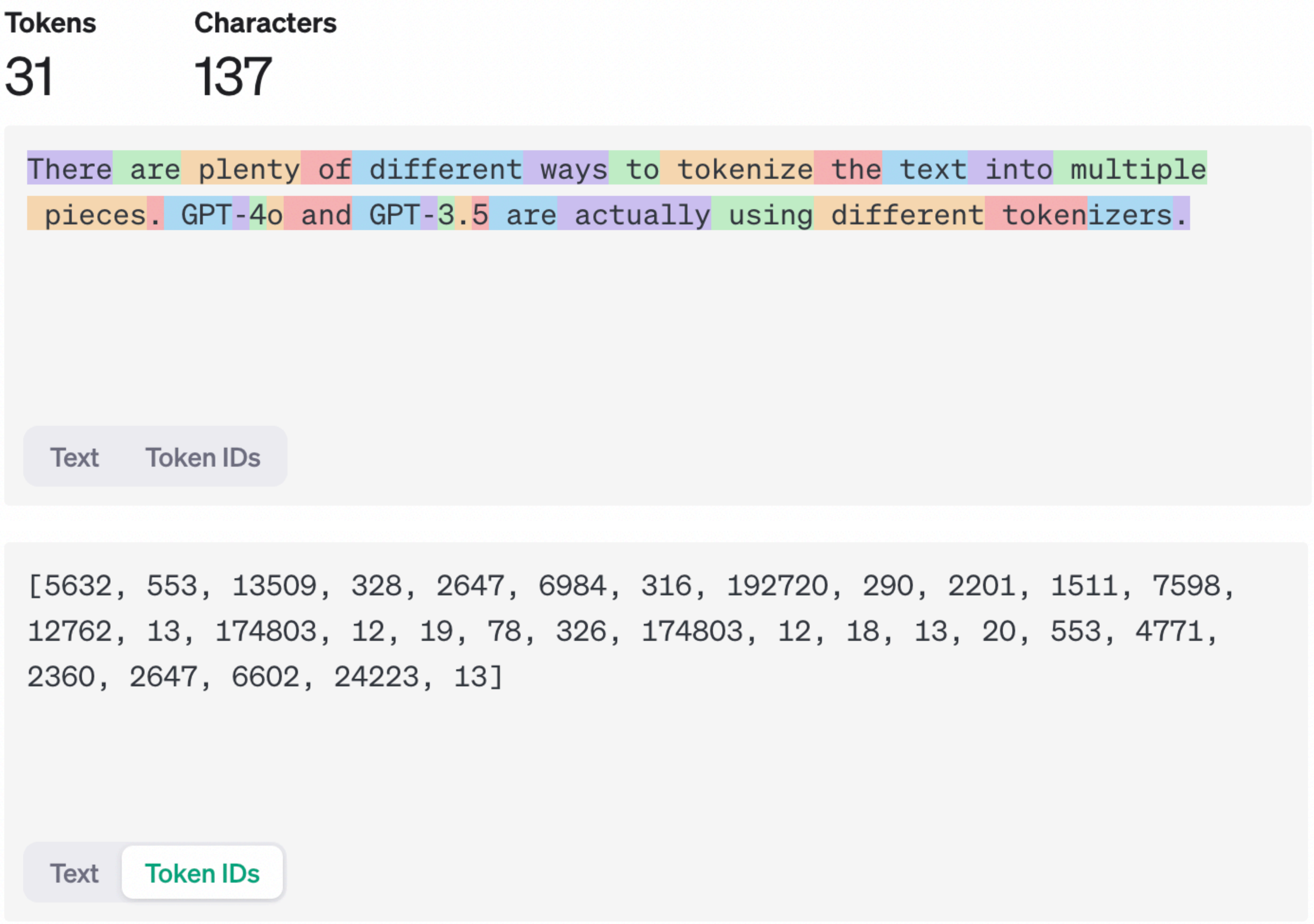
"The boy did not have any idea where he is at."

$$(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n), \quad \mathbf{x}_i \in \mathbb{R}^d$$

Model

Pre-processing

- Translating text data into a sequence
- Typically involves:
  - Normalization
  - Pre-tokenization
  - Tokenization
  - Embedding
- The first three are responsible for chunking the text and mapping them to codes.



Pre-processing

- Translating text data into a sequenc
- Typically involves:
  - Normalization
  - Pre-tokenization
  - Tokenization
  - Embedding
- The first three are responsible for chunking the text and mapping them to codes.
- 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

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

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

$$\ldots$$

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

Normalization

- Various cleanups on the given text to reduce data complexity
  - Lowercasing
    - e.g., “hello” and “Hello” has the same meaning
  - Removing unnecessary whitespaces, accents, punctuations
    - e.g., “I ate it all” —> “I ate it all”
      “café” —> “cafe”   “e-mail” —> “email”

Normalization

- Various cleanups on the given text to reduce data complexity
  - Lowercasing
    - e.g., “hello” and “Hello” has the same meaning
  - Removing unnecessary whitespaces, accents, punctuations
    - e.g., “I ate it all” —> “I ate it all”
      “café” —> “cafe”   “e-mail” —> “email”
  - Date & Numerics
    - “01/31/2024,” “31st Jan. 2024” —> “2024-01-31”
  - Unicode normalization
    - handling many equivalences
    - **https://www.unicode.org/reports/tr15/**

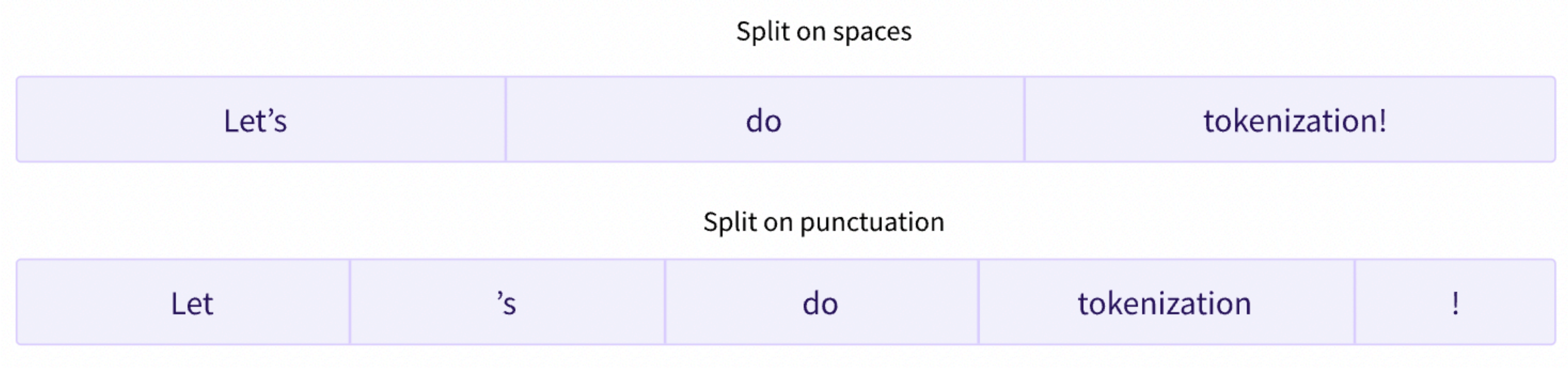
Subtype	Examples		
Font variants	ℌ	→	H
	ℍ	→	H
Linebreaking differences	[NBSP]	→	[SPACE]
Positional variant forms	ﻉ	→	ع
	ﻊ	→	ع
	ﻋ	→	ع
	ﻌ	→	ع
Circled variants	①	→	1
Width variants	ｶ	→	カ
Rotated variants	︷	→	{
	︸	→	}
Superscripts/subscripts	i^9	→	i9
	i_9	→	i9
Squared characters	㌀	→	アパート
Fractions	¼	→	1/4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

Pre-tokenization

- Facilitate more accurate tokenization (chunking) by breaking down text into manageable units.
  - Handling contractions
    - “can’t” —> “can” + “’t"
  - Dealing with punctuations
    - “(some sentence).” —> “(some sentence)” + “.”
  - Abbreviations and acronyms
    - “DMZ” should not be “D” + “MZ”

Tokenization

- Breaking the sentence down into tokens
  - Word-based tokenization
    - Good semantics
    - Too many vocabularies...



Tokenization

- Breaking the sentence down into tokens
  - Word-based tokenization
  - Character-based tokenization
    - Smaller vocabulary size
    - Bad semantics

Tokenization

- Breaking the sentence down into tokens
  - Word-based tokenization
  - Character-based tokenization
  - Subword tokenization
    - Frequent words are kept as a single token
    - Rare words are subdivided
      - Reduces expected sequence length
    - How to take “spaces” into account differs from tokenizer to tokenizer

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

Byte-Pair Encoding

- Data-driven generation of tokenization policy
  - Start from the character-level tokens
  - Generate combined codes for the frequent tokens

https://huggingface.co/learn/nlp-course/en/chapter6/5?fw=pt

Byte-Pair Encoding

- Data-driven generation of tokenization policy
  - Start from the character-level tokens
  - Generate combined codes for the frequent tokens
- **Example.**
- Suppose that our text corpus consists of five words:

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

  - Then our initial vocabulary will be: ["b", "g", "h", "n", "p", "s", “u”]

https://huggingface.co/learn/nlp-course/en/chapter6/5?fw=pt

Byte-Pair Encoding

- Data-driven generation of tokenization policy
  - Start from the character-level tokens
  - Generate combined codes for the frequent tokens
- **Example.**
- Suppose that our text corpus consists of five words.
  - Then our initial vocabulary will be: ["b", "g", "h", "n", "p", "s", "u"]
- Count the word frequencies.

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

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

Byte-Pair Encoding

- Data-driven generation of tokenization policy
  - Start from the character-level tokens
  - Generate combined codes for the frequent tokens
- **Example.**
- Suppose that our text corpus consists of five words.
  - Then our initial vocabulary will be: ["b", "g", "h", "n", "p", "s", “u”]
- Count the word frequencies.
- Use this to count subword frequencies, and expand the vocabulary

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

```
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)
```

https://huggingface.co/learn/nlp-course/en/chapter6/5?fw=pt

Byte-Pair Encoding

- Data-driven generation of tokenization policy
  - Start from the character-level tokens
  - Generate combined codes for the frequent tokens
- **Example.**
- Suppose that our text corpus consists of five words.
  - Then our initial vocabulary will be: ["b", "g", "h", "n", "p", "s", "u"]
- Count the word frequencies.
- Use this to count subword frequencies, and expand the vocabulary
- Repeat until the desired vocab. 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)
```

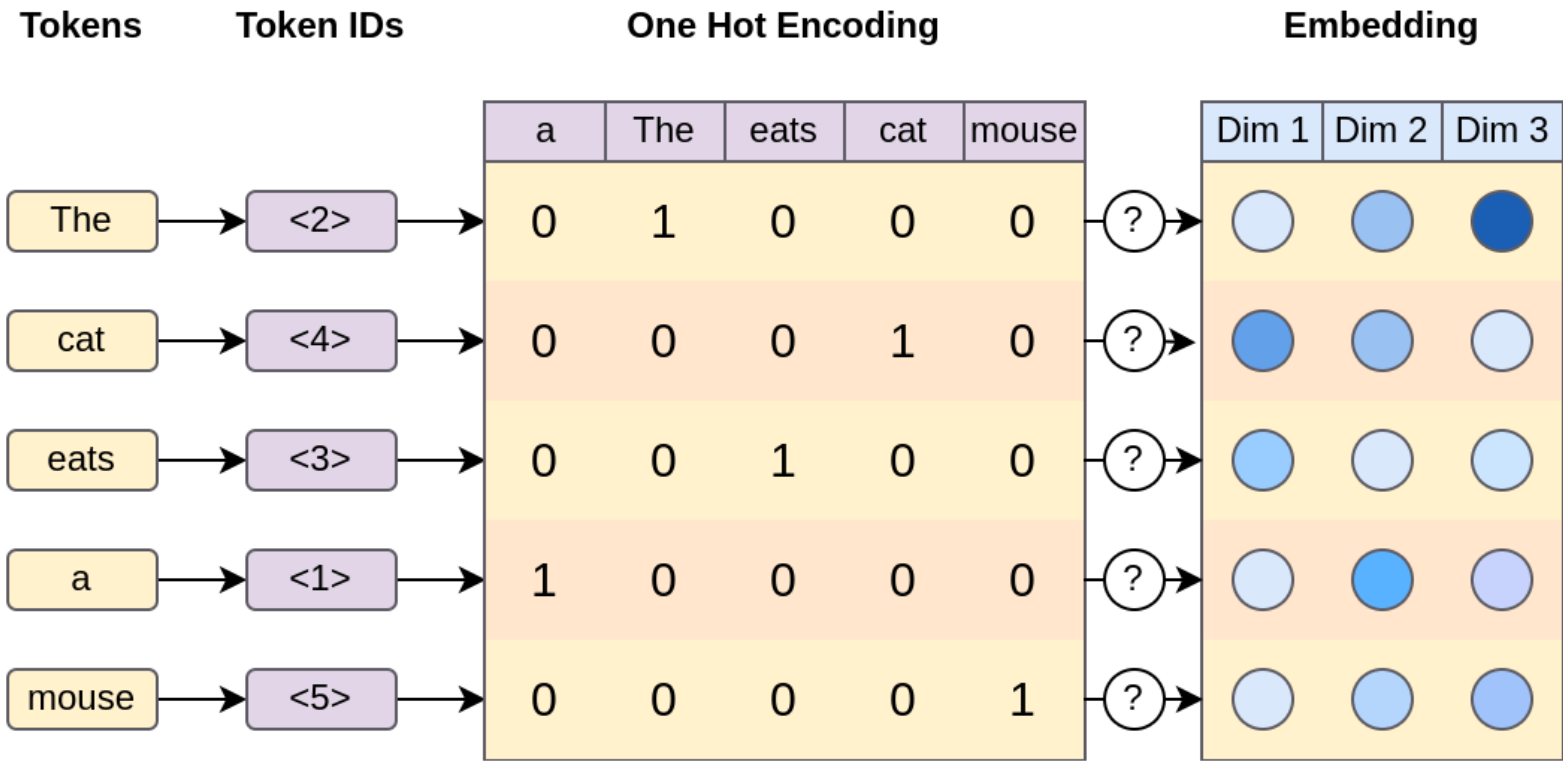
Byte-Pair Encoding

- Data-driven generation of tokenization policy
    - Start from the character-level tokens
    - Generate combined codes for the frequent tokens
- **Example.**
- Suppose that our text corpus consists of five words.
    - Then our initial vocabulary will be: ["b", "g", "h", "n", "p", "s", "u"]
- Count the word frequencies.
- Use this to count subword frequencies, and expand the vocabulary
- Repeat until the desired vocab. size is met.

- **Note.** Many other ways to do it, e.g., WordPiece.

Embedding

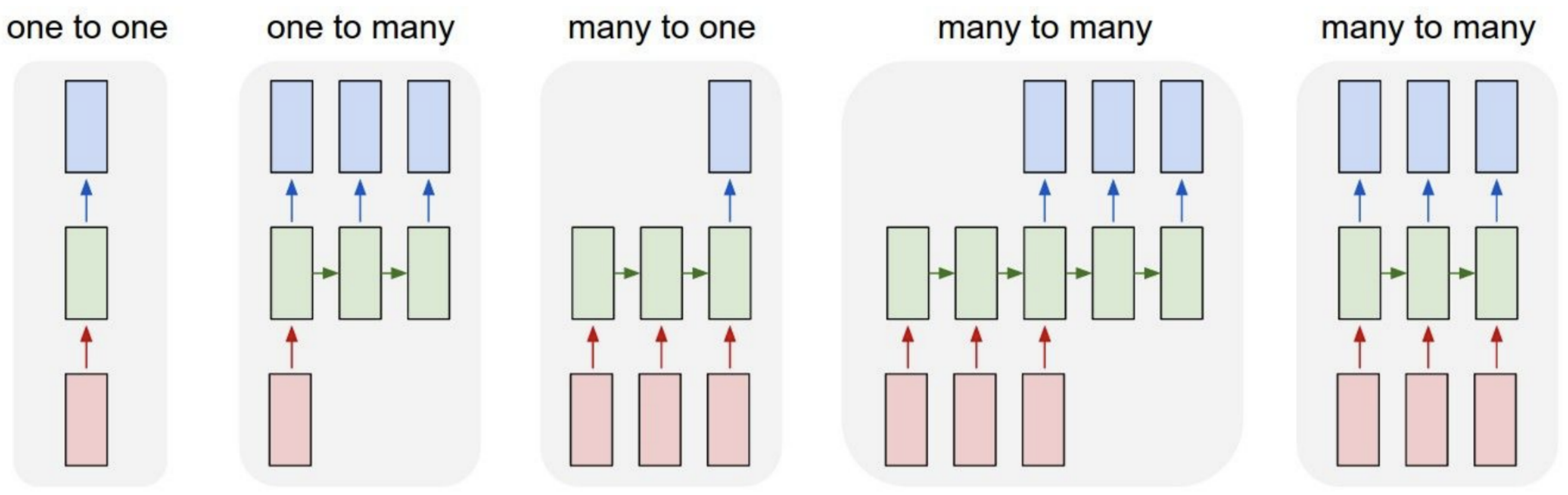
- Each token IDs is translated into one-hot encodings, and then to embeddings
  - Implementable with lookup tables
  - Embedding is trainable as well — more details on this later



Architectures

Architectures

- We will cover two architectures that are designed for sequence-like inputs / outputs
  - RNNs
  - Transformers
- Should be able to handle all following cases…

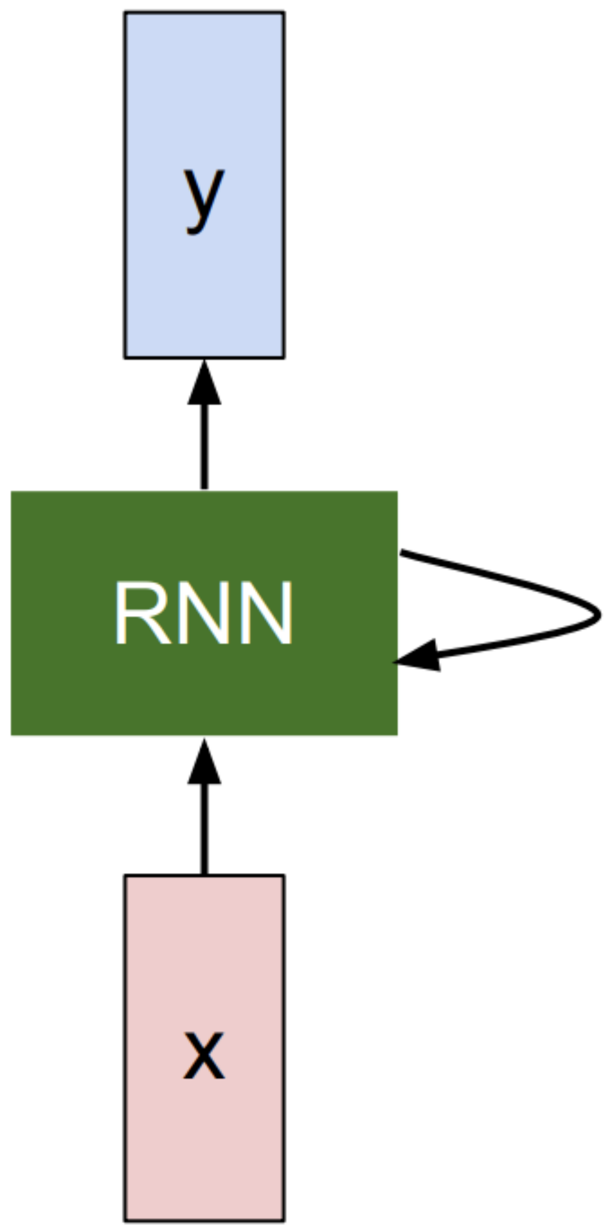


RNNs

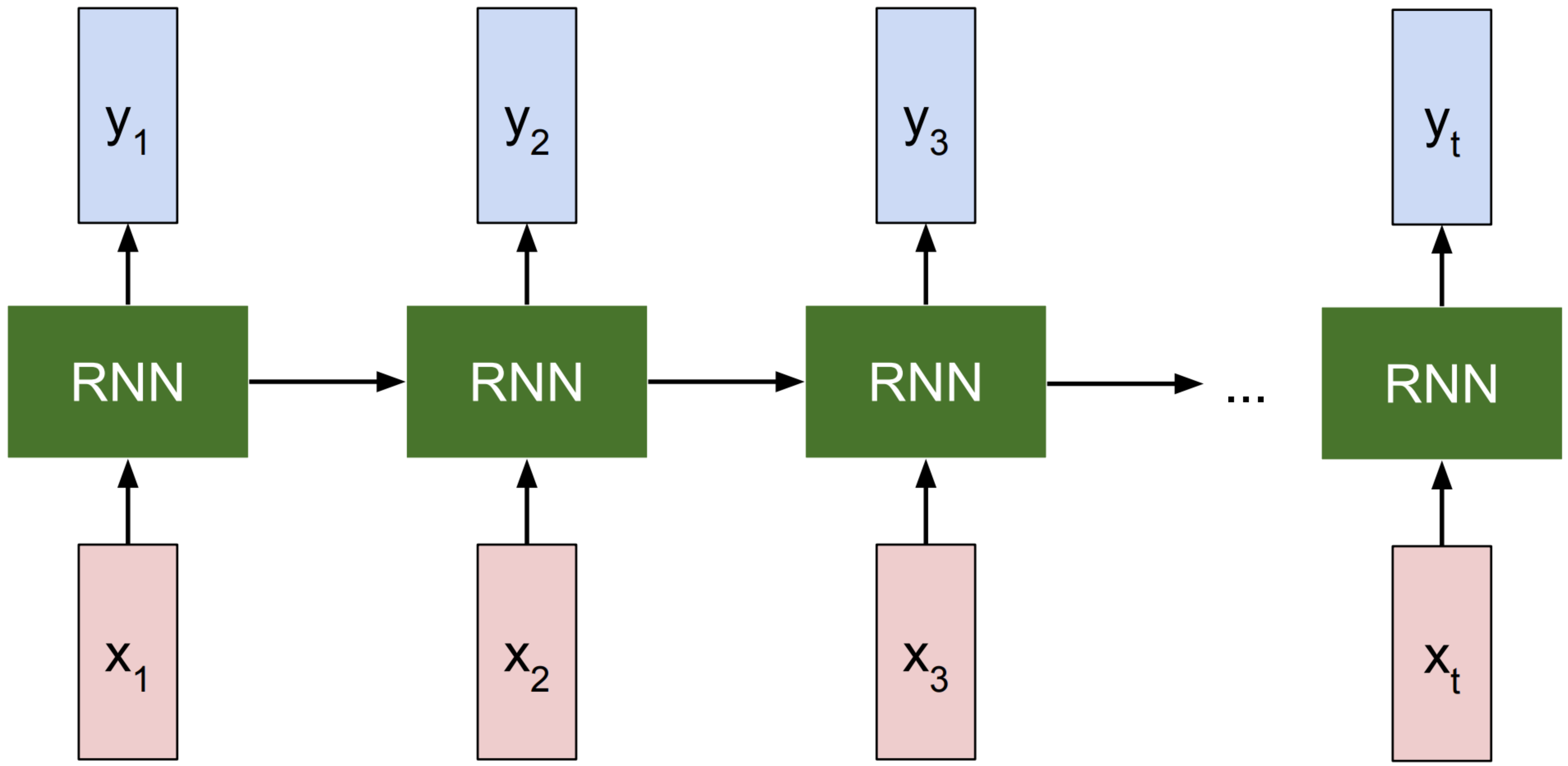
(follows exposition of https://cs231n.github.io/rnn/)

Recurrent Neural Networks

- **Idea.** Handle sequential input using a state-space model $\hat{\mathbf{y}}_t = f_\theta(\mathbf{x}_t; \mathbf{h}_{t-1})$
    - The internal state $\mathbf{h}_{t-1} = g_\theta(\mathbf{x}_{t-1}; \mathbf{h}_{t-2})$ contains the (compressed) information from the past history of inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{t-1}$.



RNN



RNN (unrolled)

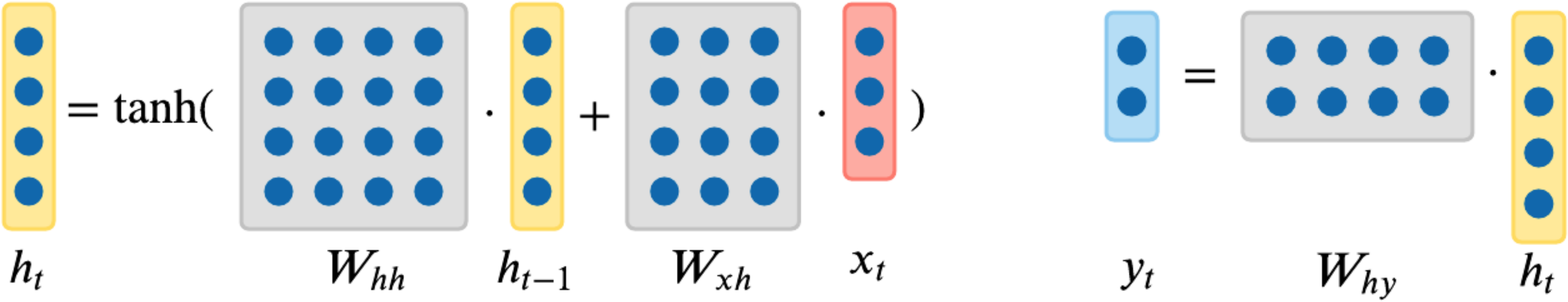
Recurrent Neural Networks

- **Parameterization.**
  In the simplest form (Rumelhart, 1986), the recurrence can be formalized as:

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

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

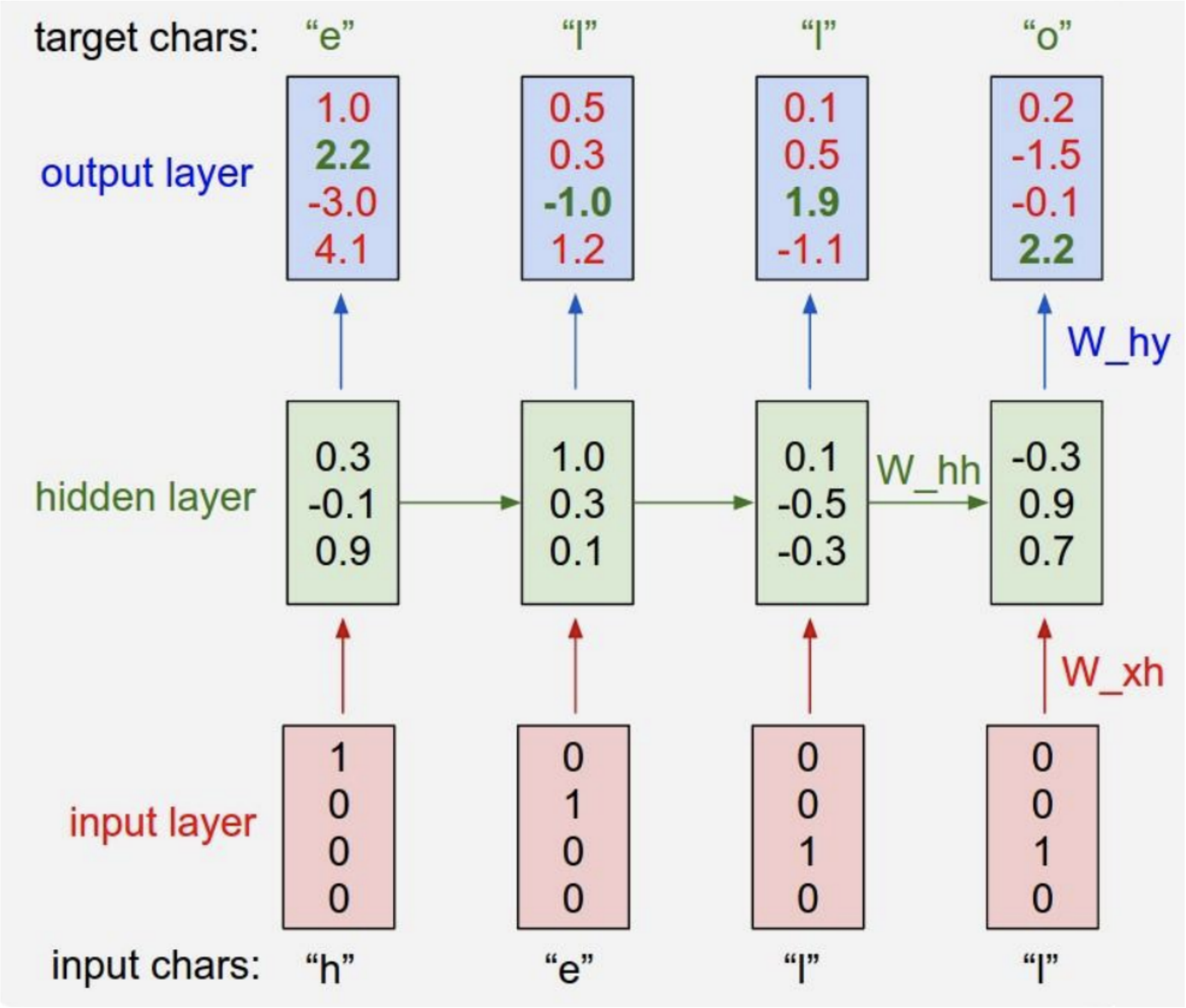
  (recall: hidden Markov models)



RNN for language modeling

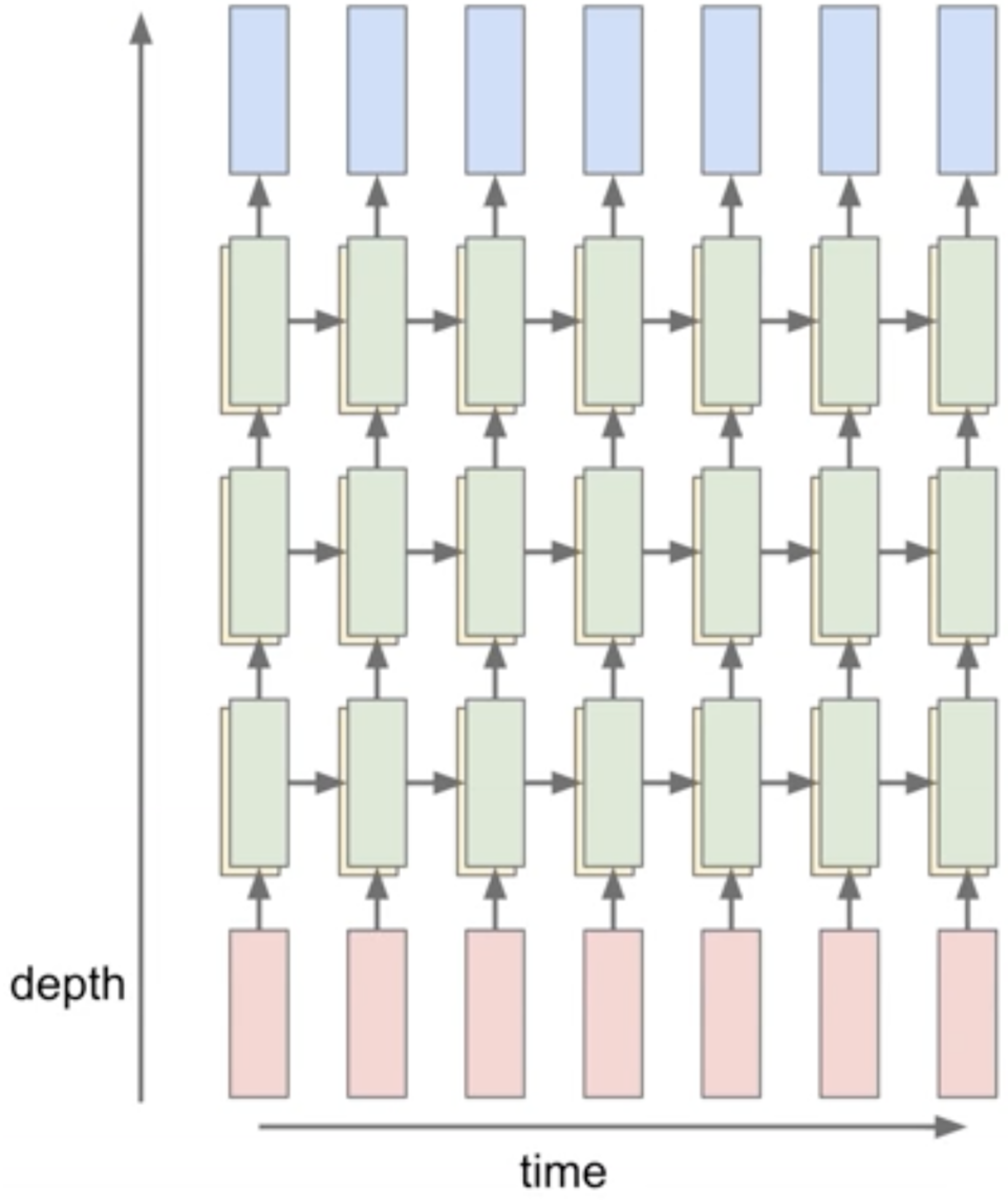
- **Example (Language Model).**
  Suppose that we want to generate new sentences with:
  - Character-level tokens
  - Single-layer RNN
  - No embedding layer

- Then, we can feed the generated character as an RNN input to keep on generating new characters.
  - Similar in transformers (much compute!)



Deep RNNs

- Stack multiple RNN blocks to build a deep RNN
    - Strengthens the “memory” of RNNs
    - Can capture longer-term relationships, theoretically
        - but this is actually quite difficult!



Limitations

- **Hard to capture long-term dependencies.** Due to vanishing/exploding gradients from $\tanh(\cdot)$
  - Suppose that we want to use the loss at time t (i.e., L_t),
    to update the information that we should have kept at time 1 (i.e., $\mathbf{h}_1$).
    - The partial derivative of current state w.r.t. past state is:

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

Limitations

- **Hard to capture long-term dependencies.** Due to vanishing/exploding gradients from $\mathbf{tanh}(\cdot)$
  - Suppose that we want to use the loss at time t (i.e., L_t),
    to update the information that we should have kept at time 1 (i.e., $\mathbf{h}_1$).
    - The partial derivative of current state w.r.t. past state is:

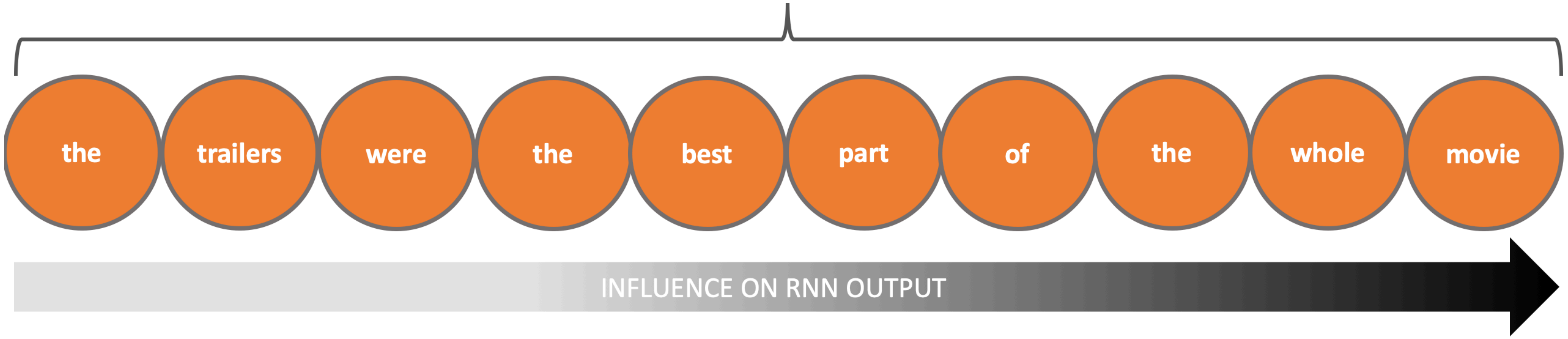
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \tanh'(\mathbf{W}_{\mathrm{hh}}\mathbf{h}_{t-1} + \mathbf{W}_{\mathrm{xh}}\mathbf{x}_t)\mathbf{W}_{\mathrm{hh}}$$

    - The gradient with respect to the loss at time t (L_t) can be written as:

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

Limitations

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

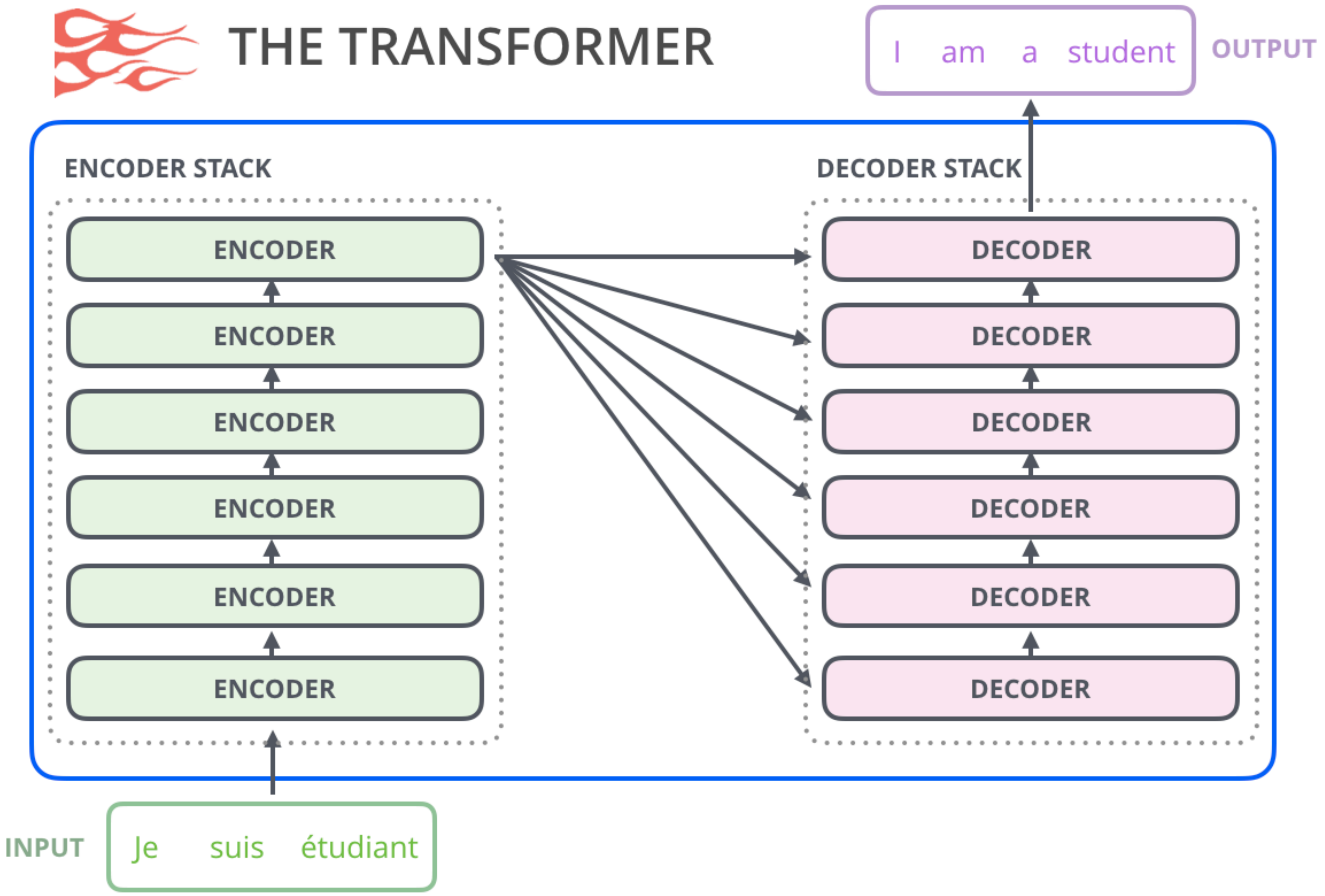


- **Solution.**
  - Adopt extra modules that is designed for long-term dependencies
    - called LSTM (not covered in this course)
  - Let the very old input directly affect the new output
    - called Transformers

Transformers

Transformers

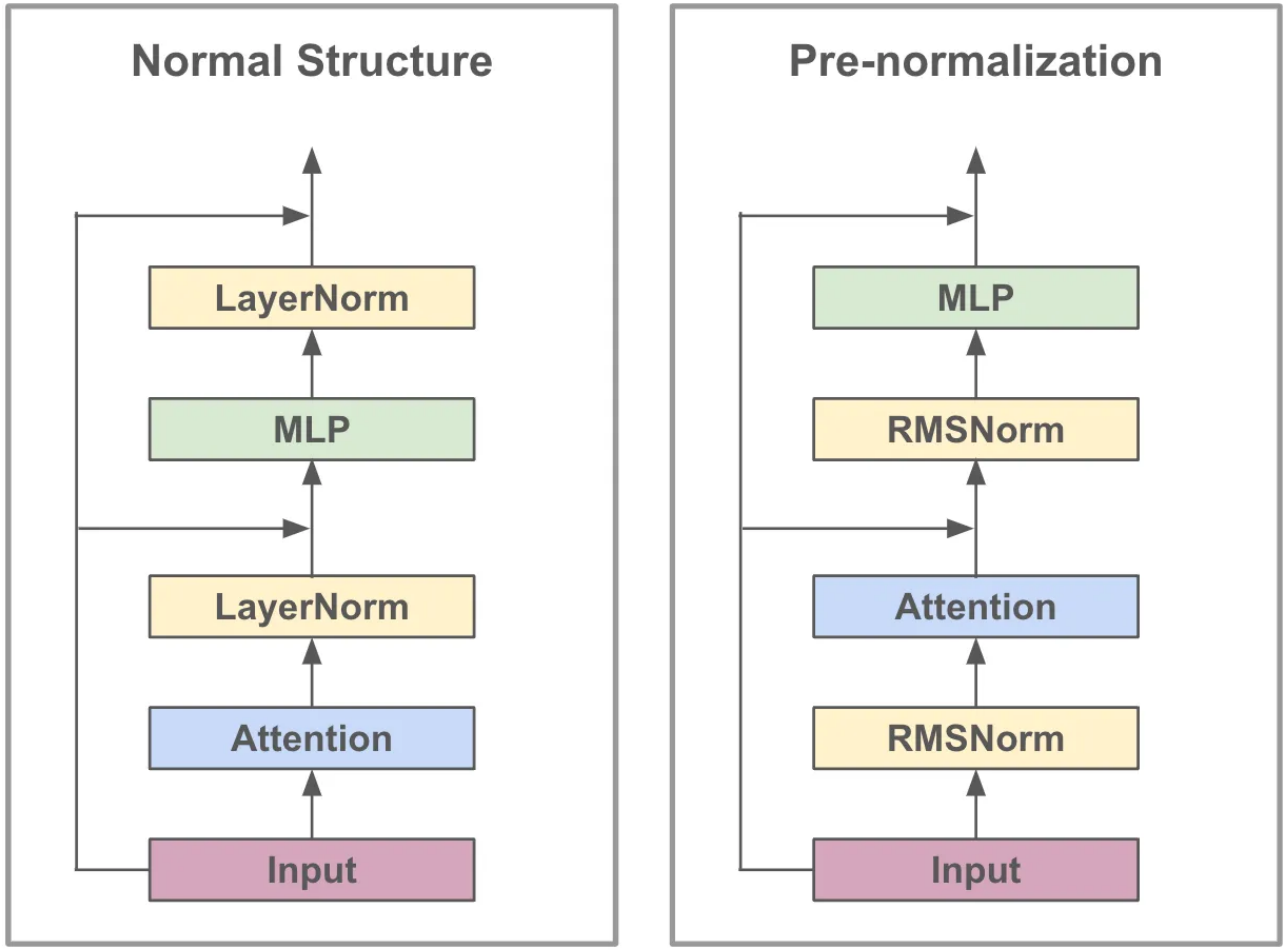
- Consists of a stack of encoders blocks, and a stack of decoder blocks
  - **Encoder-only.** BERT
  - **Decoder-only.** GPT (our focus)



Transformers

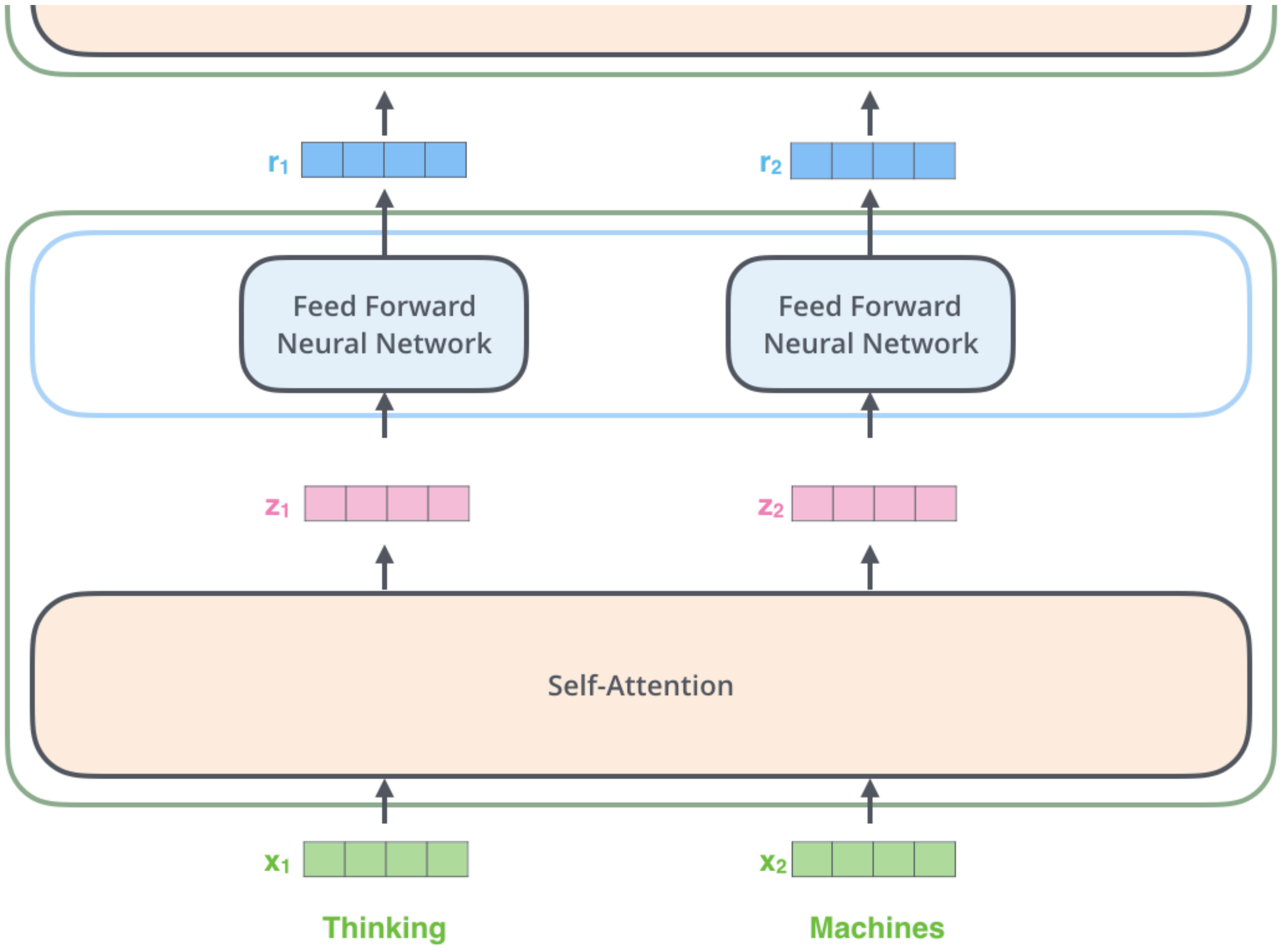
- Consists of a stack of encoders blocks, and a stack of decoder blocks
  - **Encoder-only.** BERT
  - **Decoder-only.** GPT (our focus)

- Each block consists of four elements:
  - Multi-head self-attention (MHA)
  - Feed-forward network (FFN)
  - LayerNorm / RMSNorm
  - Residual connections



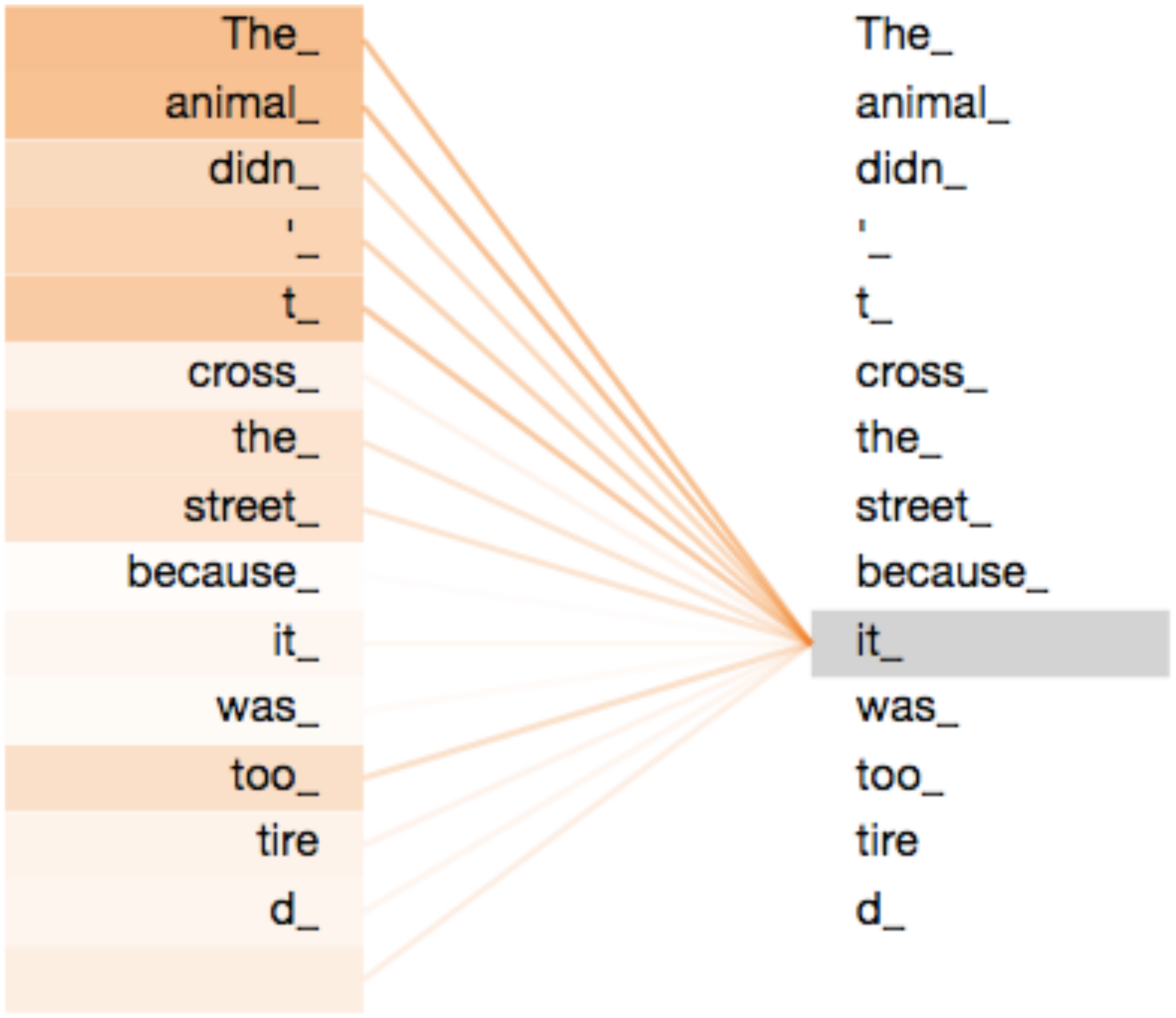
MHA and FFN

- MHA and FFN plays a complementary role
  - **MHA.** Captures inter-token dependency
  - **FFN.** Applies intra-token operations
    - Same operation for all tokens



Self-Attention

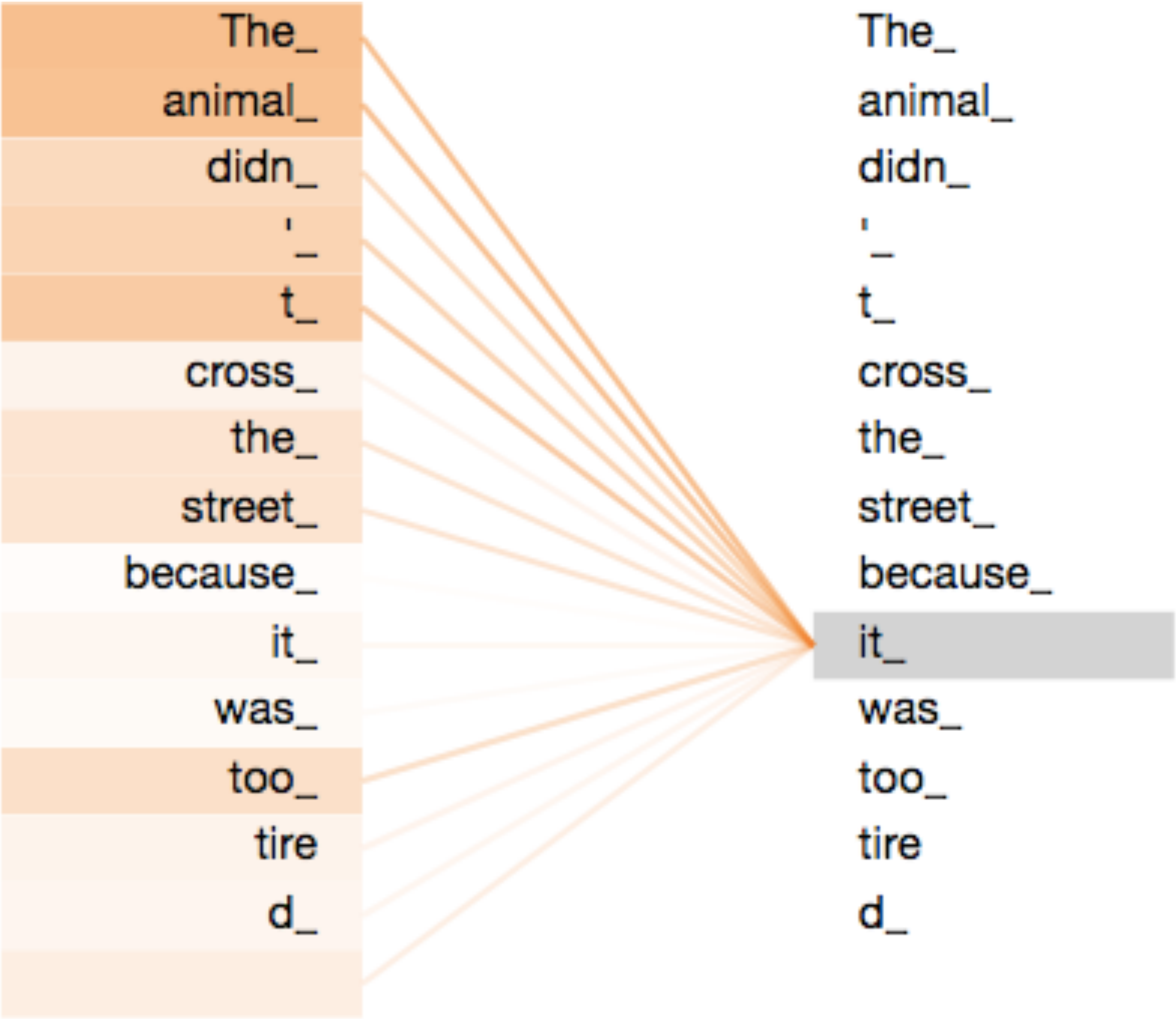
- **Idea.** Measures the relevance of other tokens for processing the target token
  - The token output will be a weighted sum of “values” from other tokens



Self-Attention

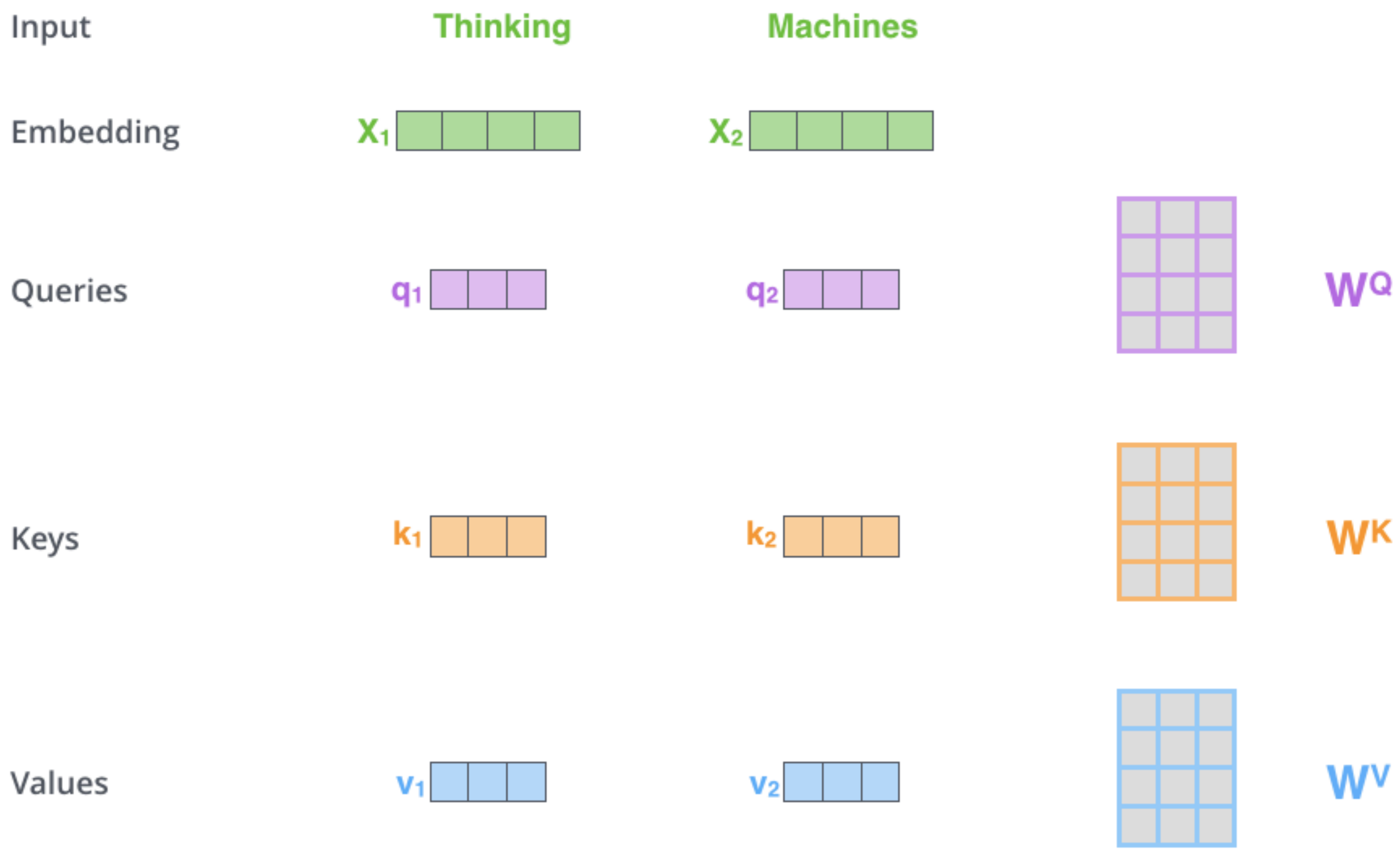
- **Idea.** Measures the relevance of other tokens for processing the target token
  - The token output will be a weighted sum of "values" from other tokens

- To measure the relevance, we use the so-called attention score
  - Expressed as a softmax of the dot products of query (self) and key (other tokens)
  - Also pays attention to the self
    - thus called self-attention



Self-Attention

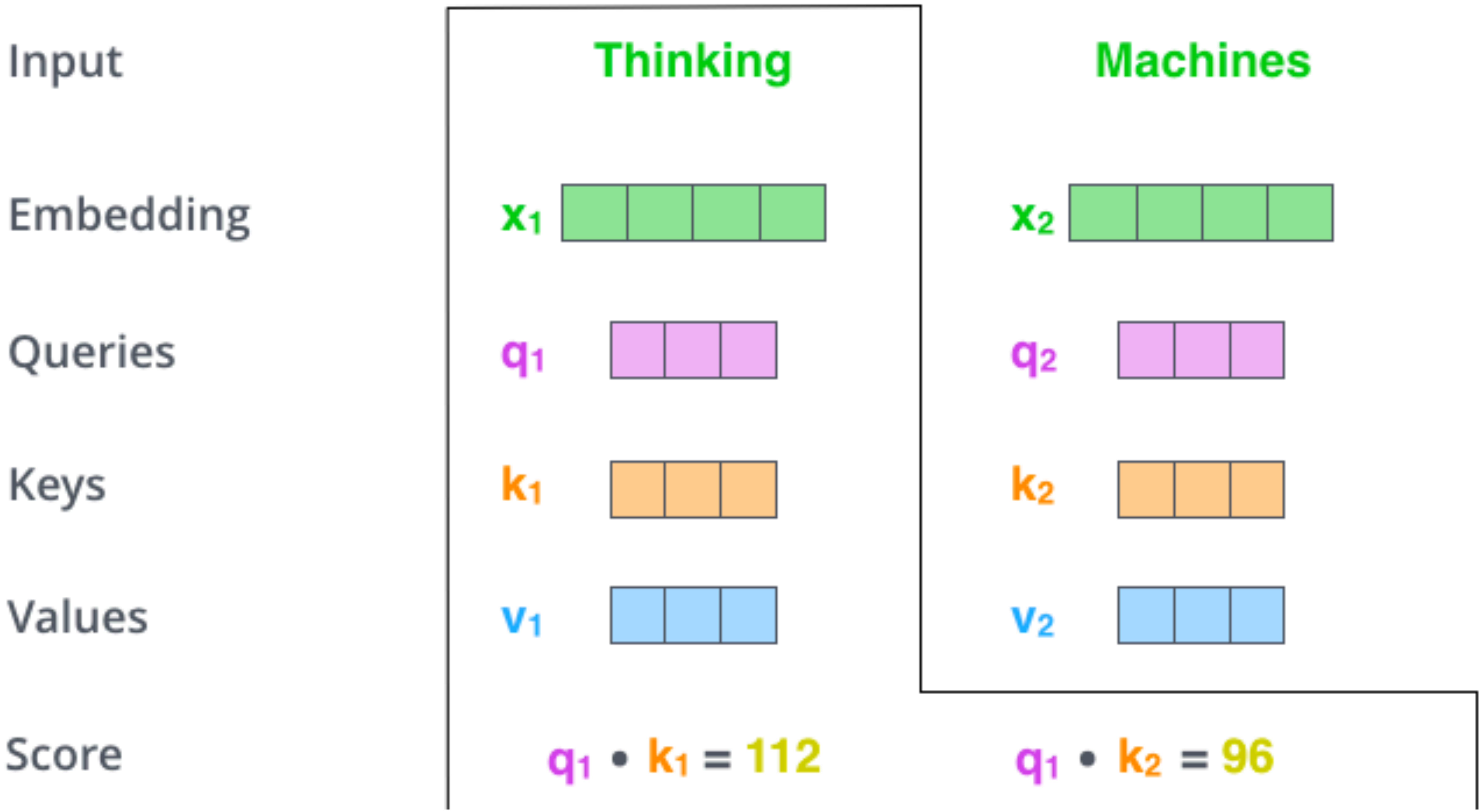
- **Step 1.** For each **token**, we compute **query**, **key**, and **value**.
  - Weight matrices are shared over the tokens



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

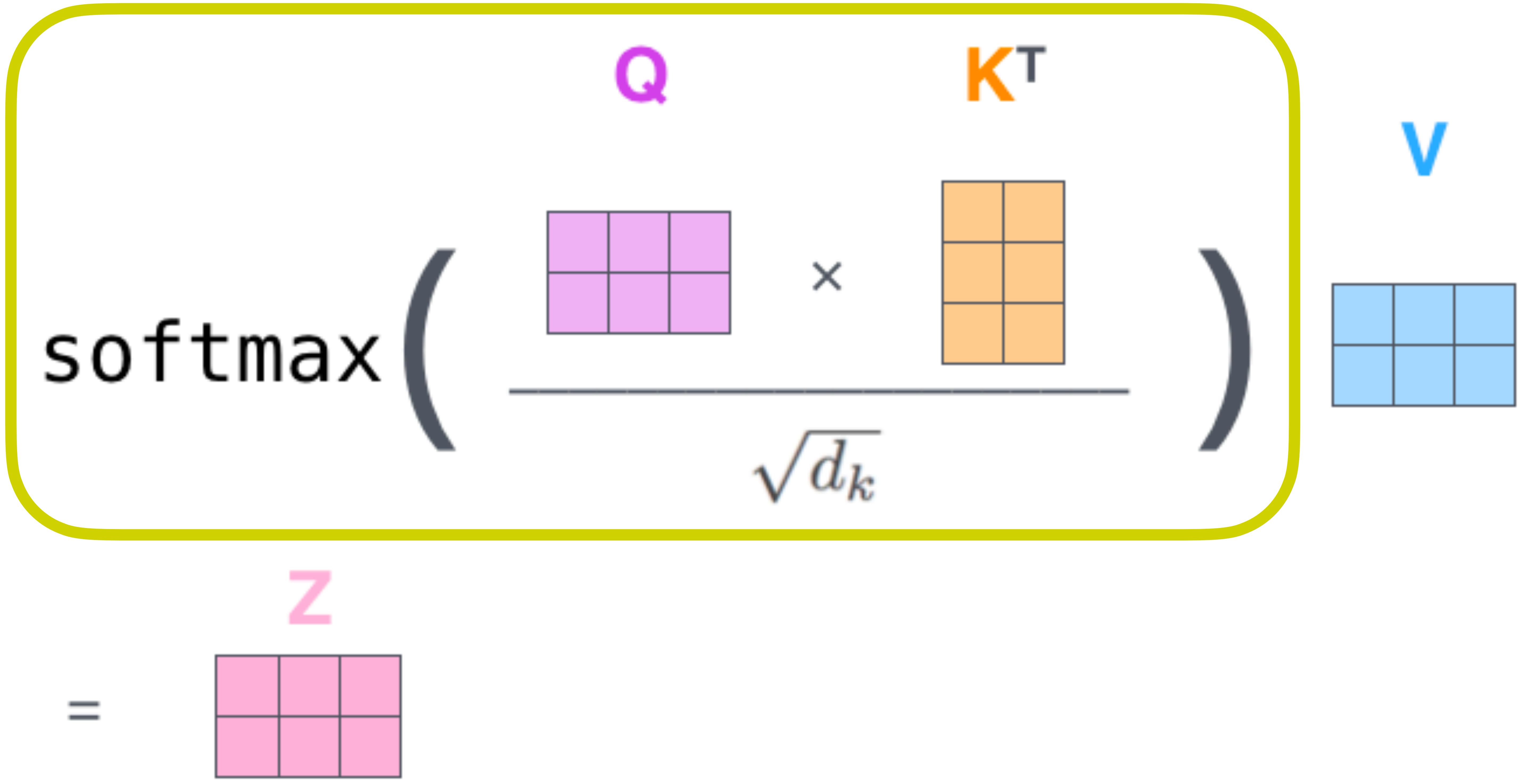
Self-Attention

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



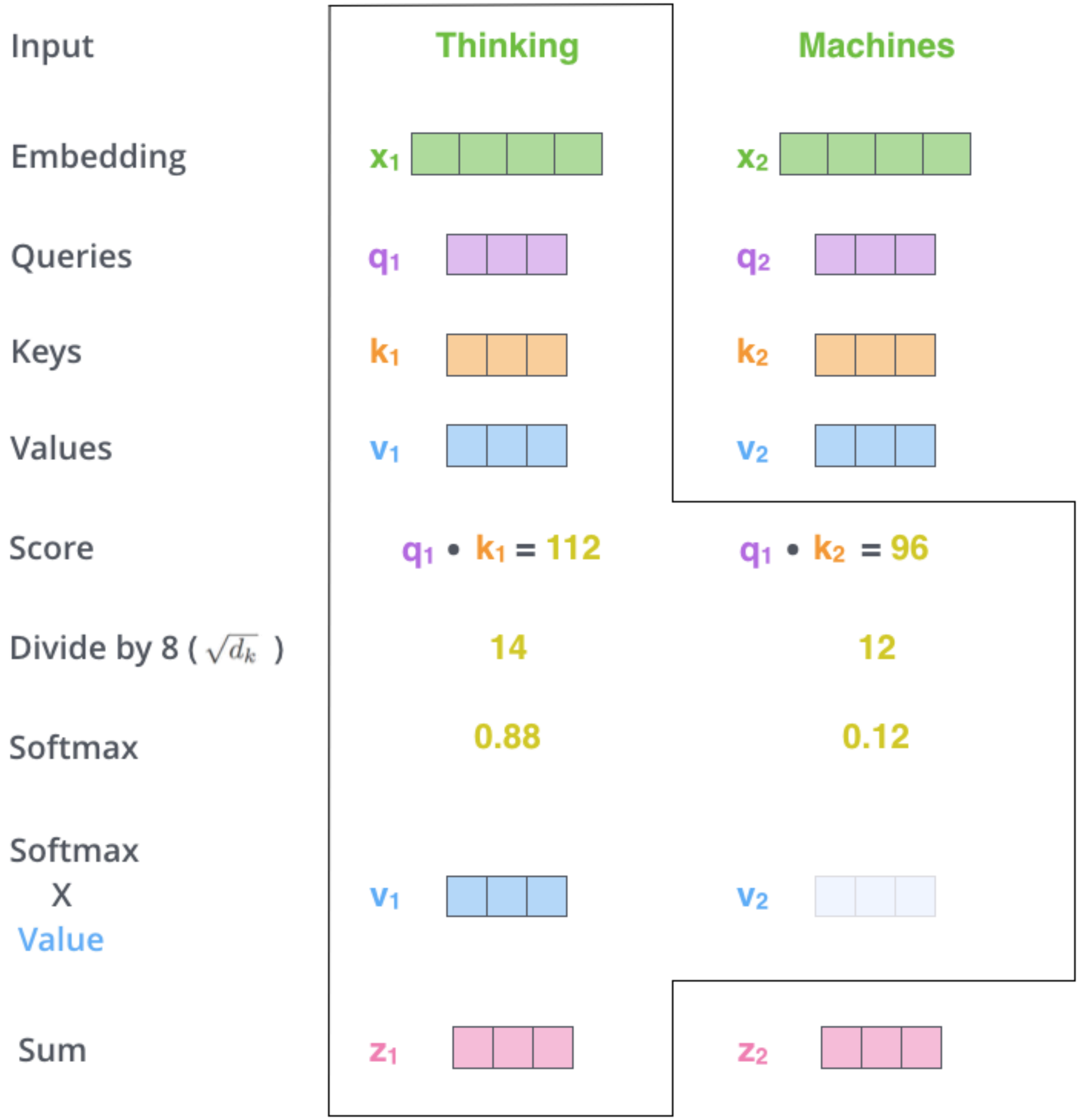
Self-Attention

- **Step 3.** Compute output as a weighted sum of values, weighted by the softmax of dot products.
    - Normalized by the dimensions



Self-Attention

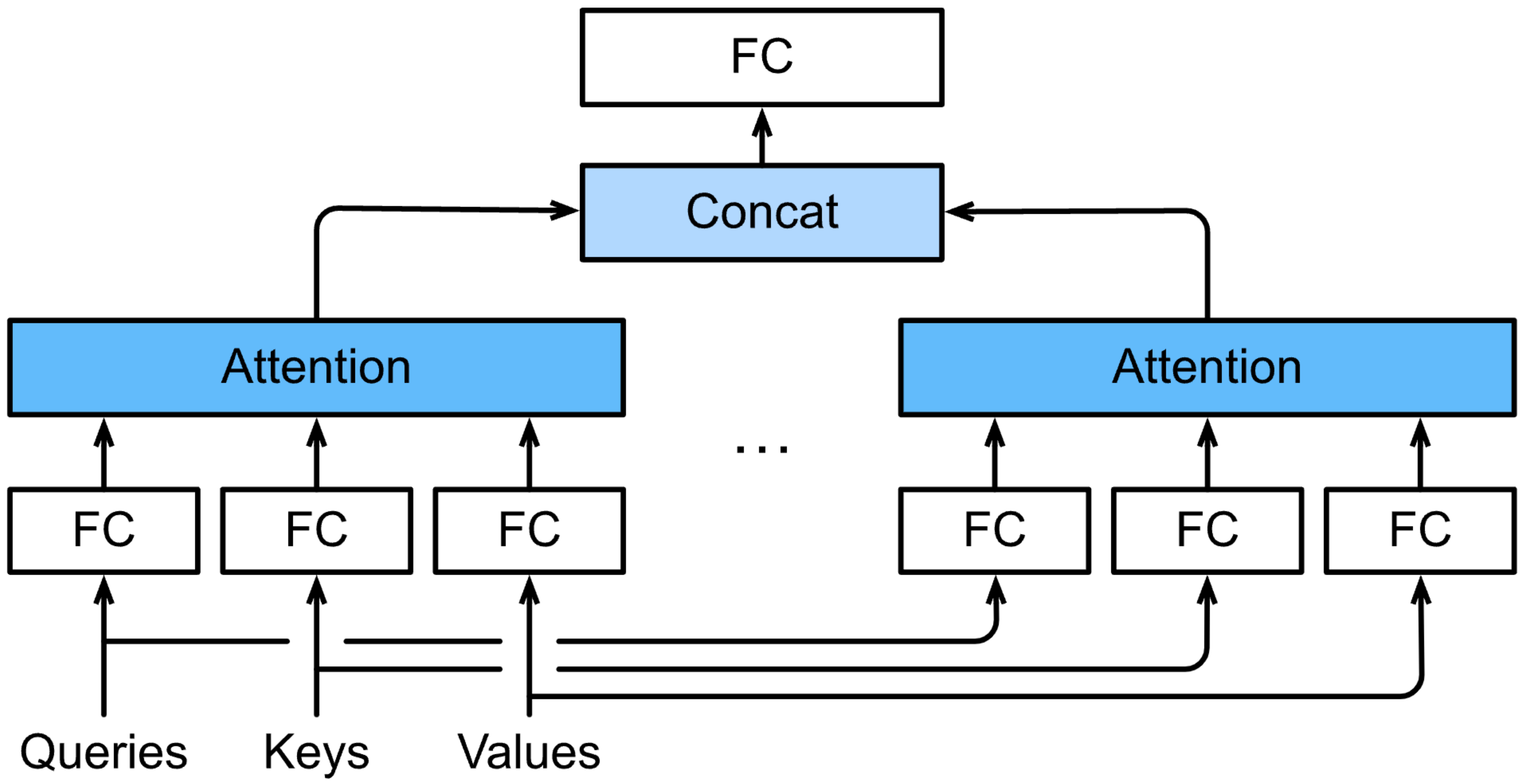
- **Computation & Memory.**
  Suppose that we have n tokens.
  - Q/K/V computation.
    - $O(n)$
  - Attention for each Q-K pairs.
    - $O(n^2)$
  - Weighted sum.
    - $O(n^2)$
- Unlike RNN, requires quadratic operation with respect to the sequence length!



https://d2l.ai/chapter_attention-mechanisms-and-transformers/multihead-attention.html

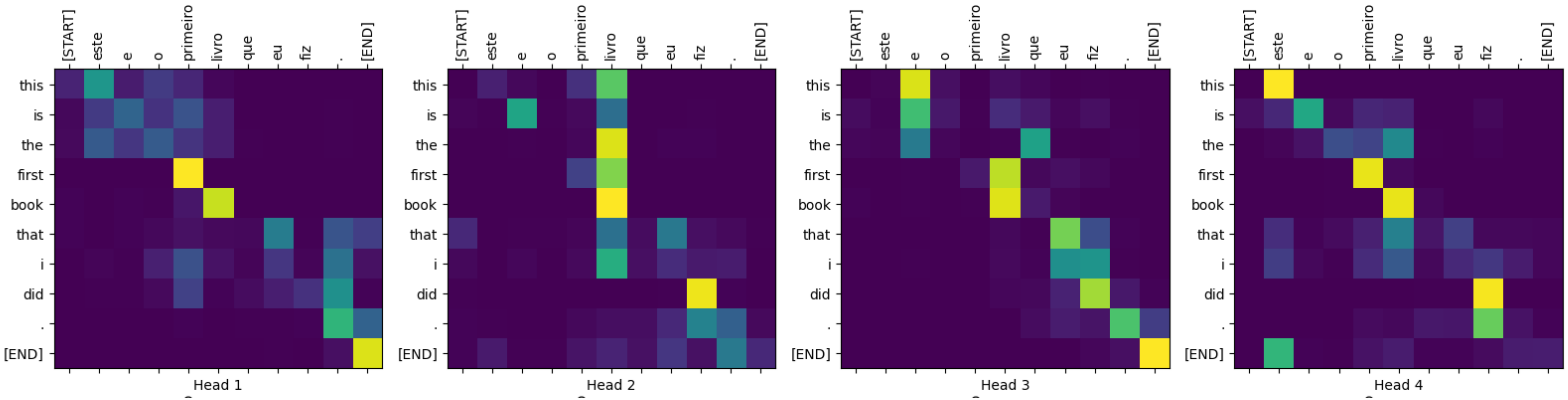
Multi-head SA

- Typically, we use multiple parallel self-attention layers in a transformer block
  - The outputs of the SA blocks are concatenated, and linearly projected.



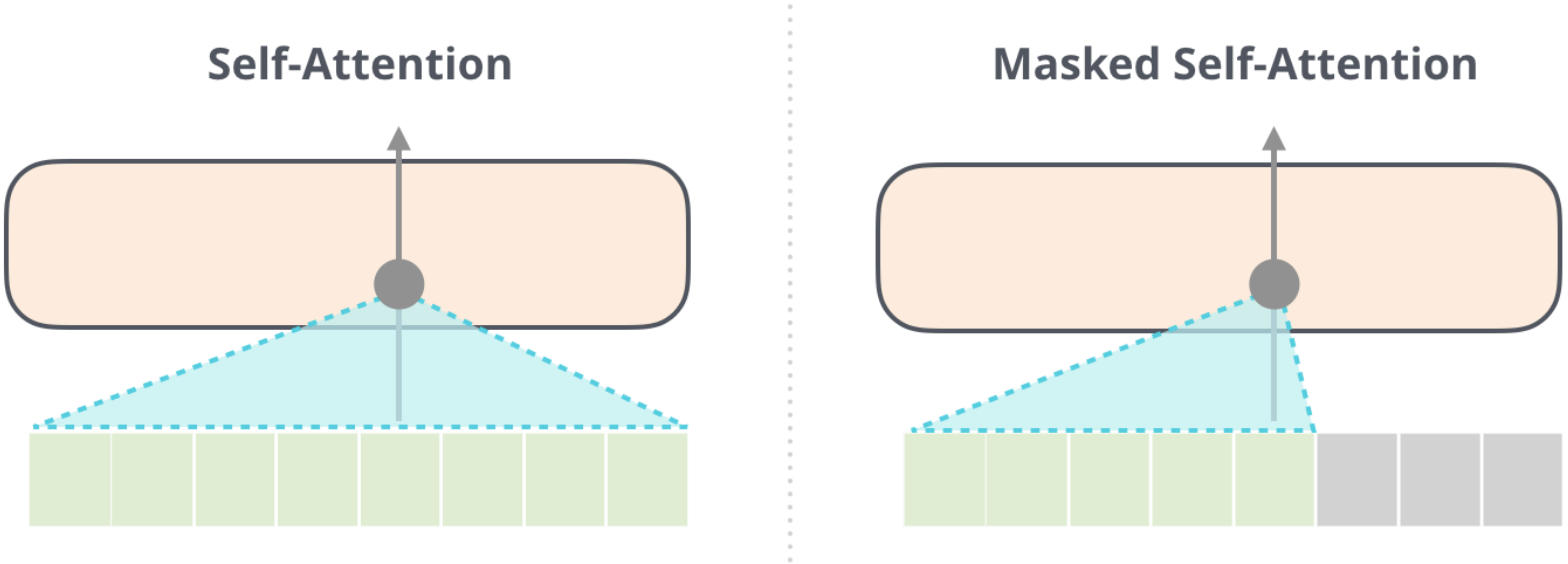
Multi-head SA

- Typically, we use multiple parallel self-attention layers in a transformer block
    - The outputs of the SA blocks are concatenated, and linearly projected
    - The heads indeed tend to capture diverse attention patterns



Causal masking for attention

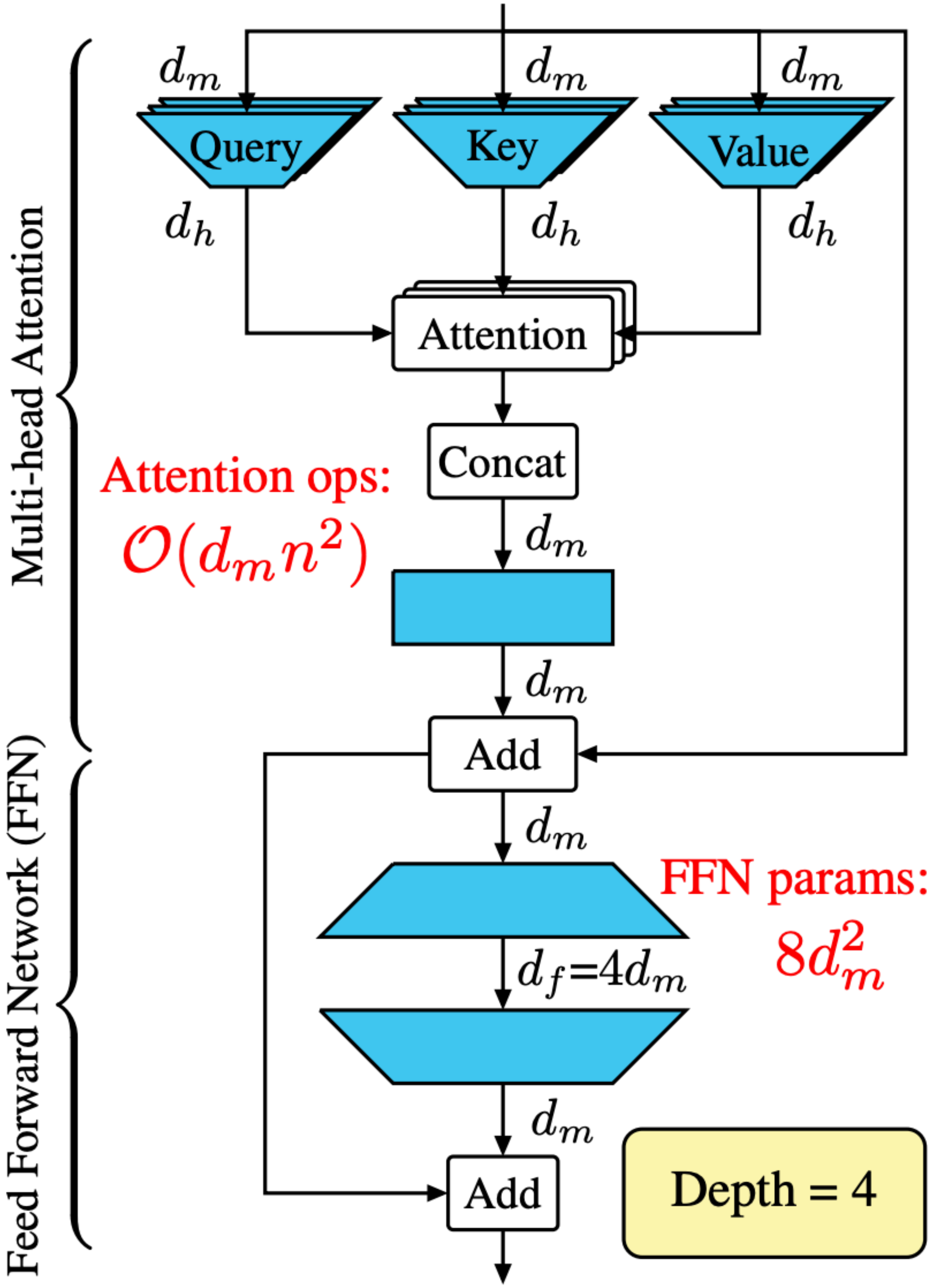
- In decoder-only transformers (like GPT), the self-attention layers are masked
  - For generating tth token, one can only see $\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}$



Feed-forward network

- Fully-connected layers that follow the MHA
  - If very basic, simply use two-layer nets
    - Takes the 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%

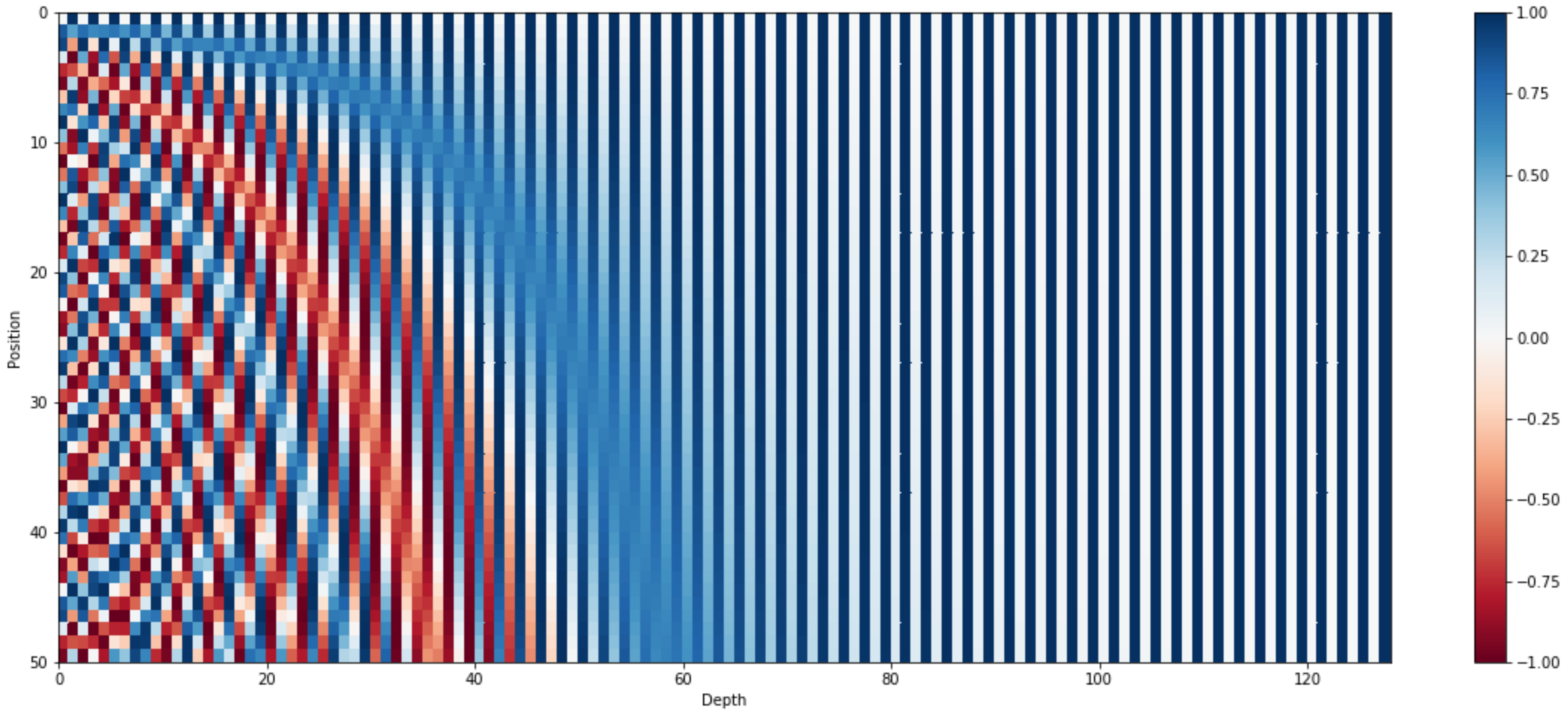


https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Positional encoding

- **Observation.** Self-attention mechanism is neat, but it disregards positional information!
  - Solution. To resolve this, it is common to add position-specific information to the data (positional encoding; added to initial embeddings)

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



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

Cheers