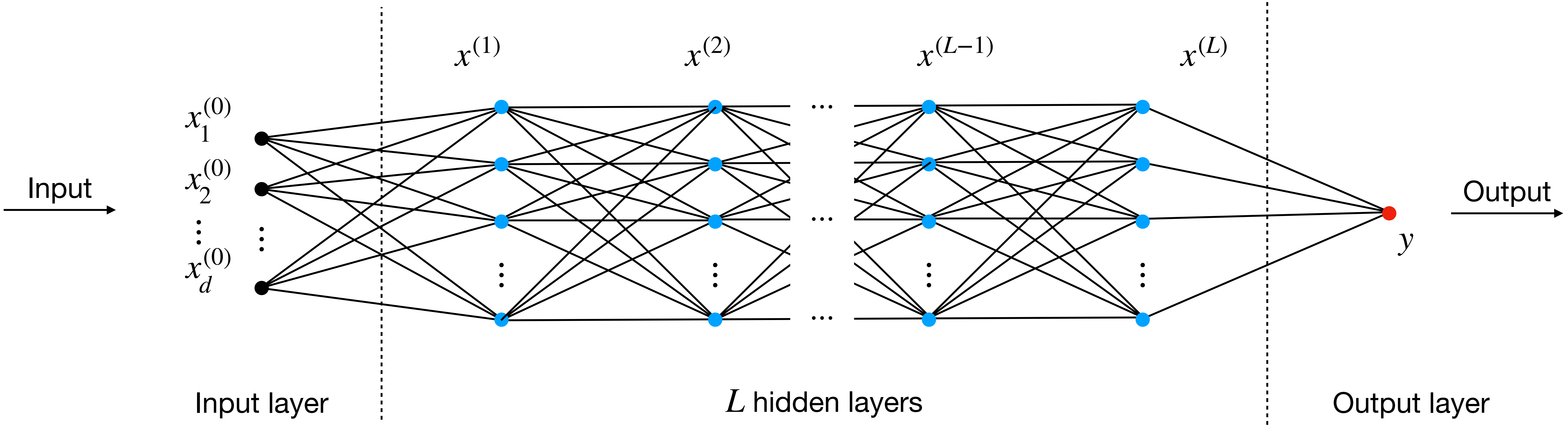
Transformers

Machine Learning Course - CS-433
11 Nov 2025
Robert West
(Slide credits: Nicolas Flammarion)

EPFL

So far in this class…



Single datapoint x

Feature extractor:
maps input x
to high-dimensional
feature vector $f(x)$

Linear model:
maps feature vector $f(x)$
to one-dimensional
score $f(x)^\top w$

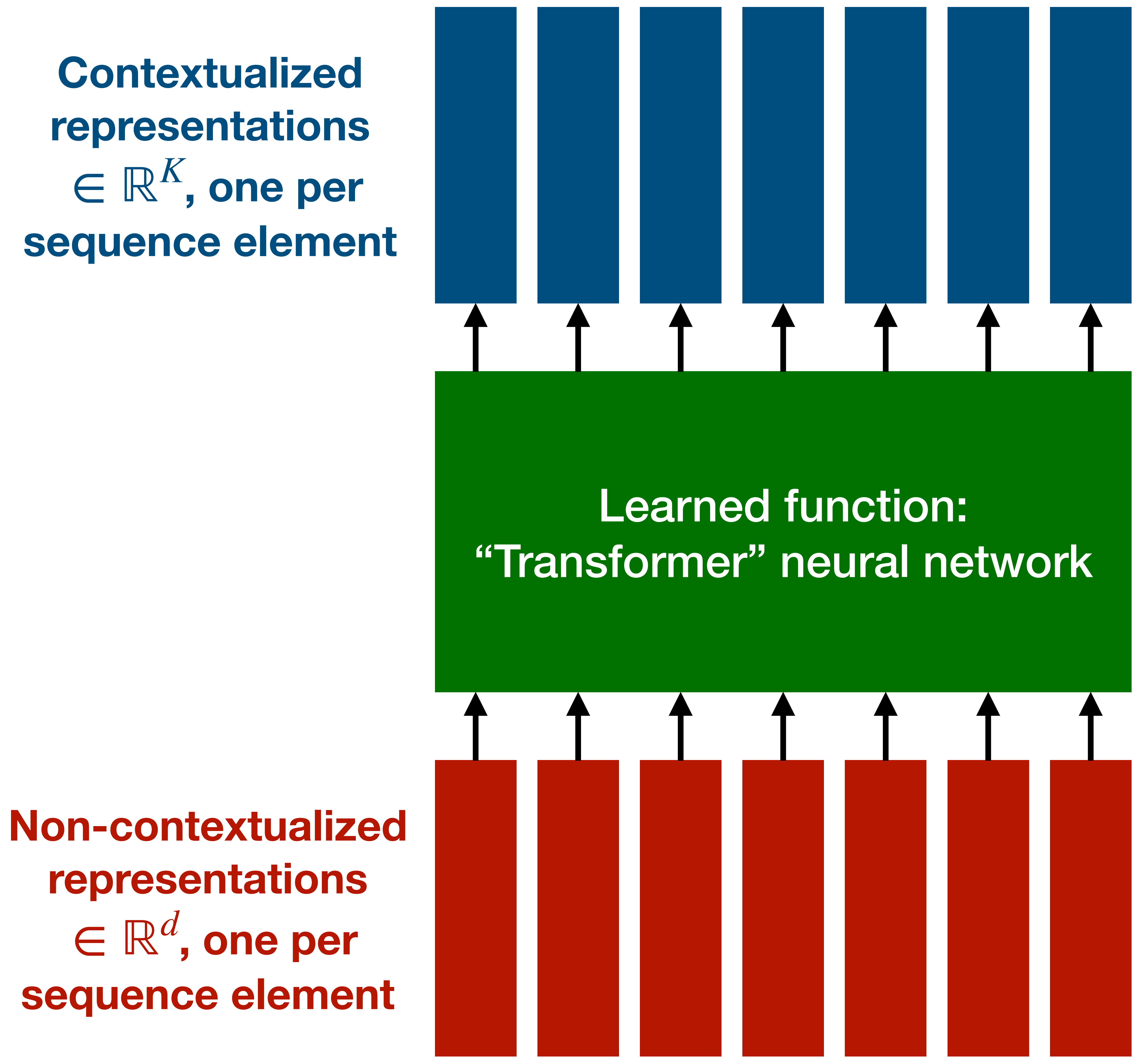
Towards richer inputs & outputs

Instead of single-datapoint inputs and 1-dimensional outputs, we may want to ingest and produce richer objects, e.g.,

- **Tagging:** for each element in an input *sequence* (or *set*), make one prediction (not independent of predictions for other elements); e.g., part-of-speech tagging, gene-structure annotation, …

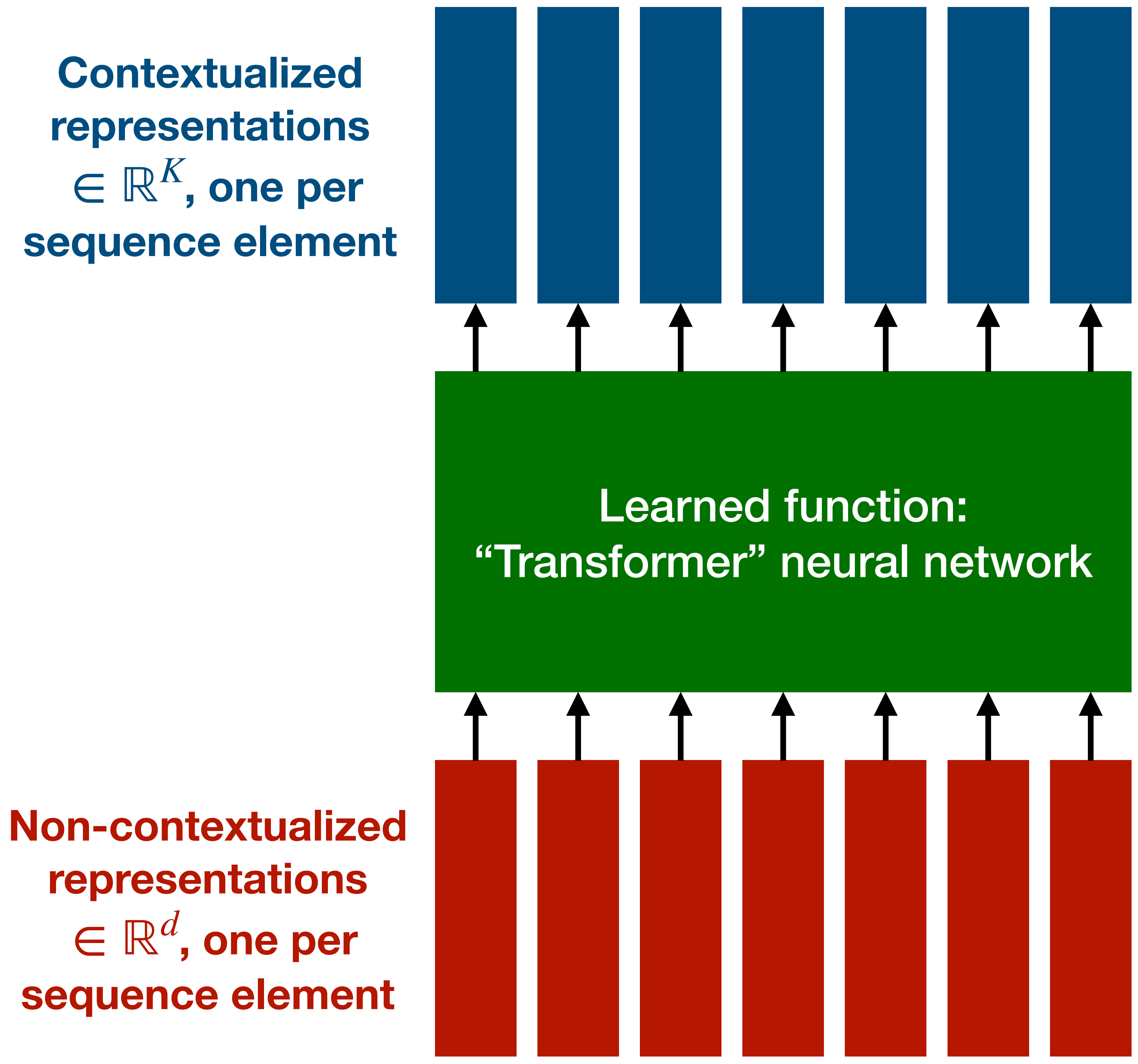
- **Sequence-to-sequence transduction:** machine translation, question answering, summarization, image-to-text, text-to-image, …

Basis: contextualized representations



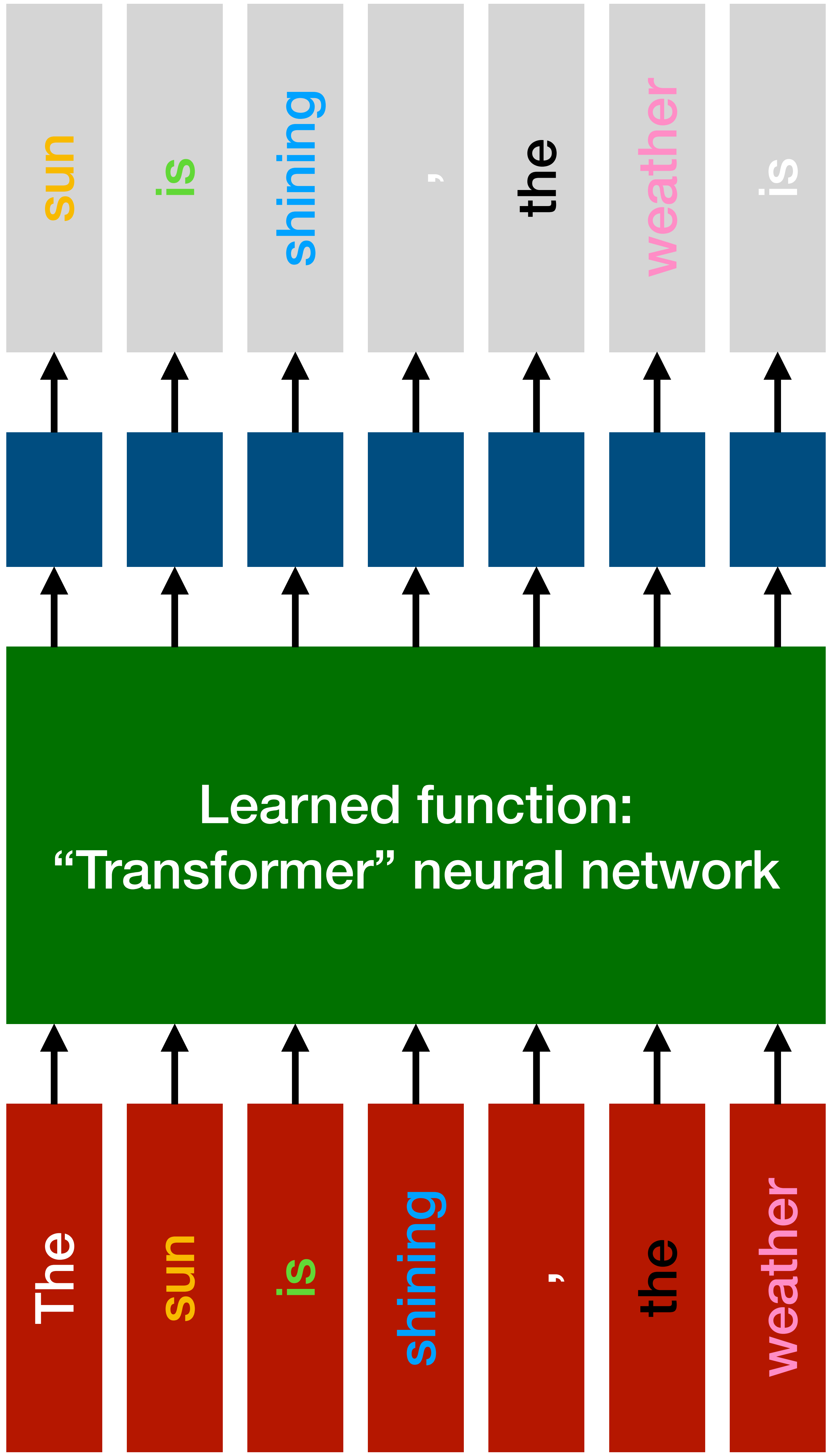
- **Tagging:** linear classifier on top of each contextualized representation, learned together with other params
- **Sequence-to-sequence transduction:** learn function for mapping seq of contextualized representations to seq of outputs (e.g., via next-token prediction, see next slides)
- **Classification** also straightforward: use representation of final seq element as feature vector for entire seq; feed to logistic regression layer (trained together with other params)

Loss



- Depending on your task, feed contextualized representations into an appropriate loss function; minimize via backprop
- Tagging: e.g., one logistic loss per element
- Classification: e.g., logistic loss on final sequence element

Large language models: training

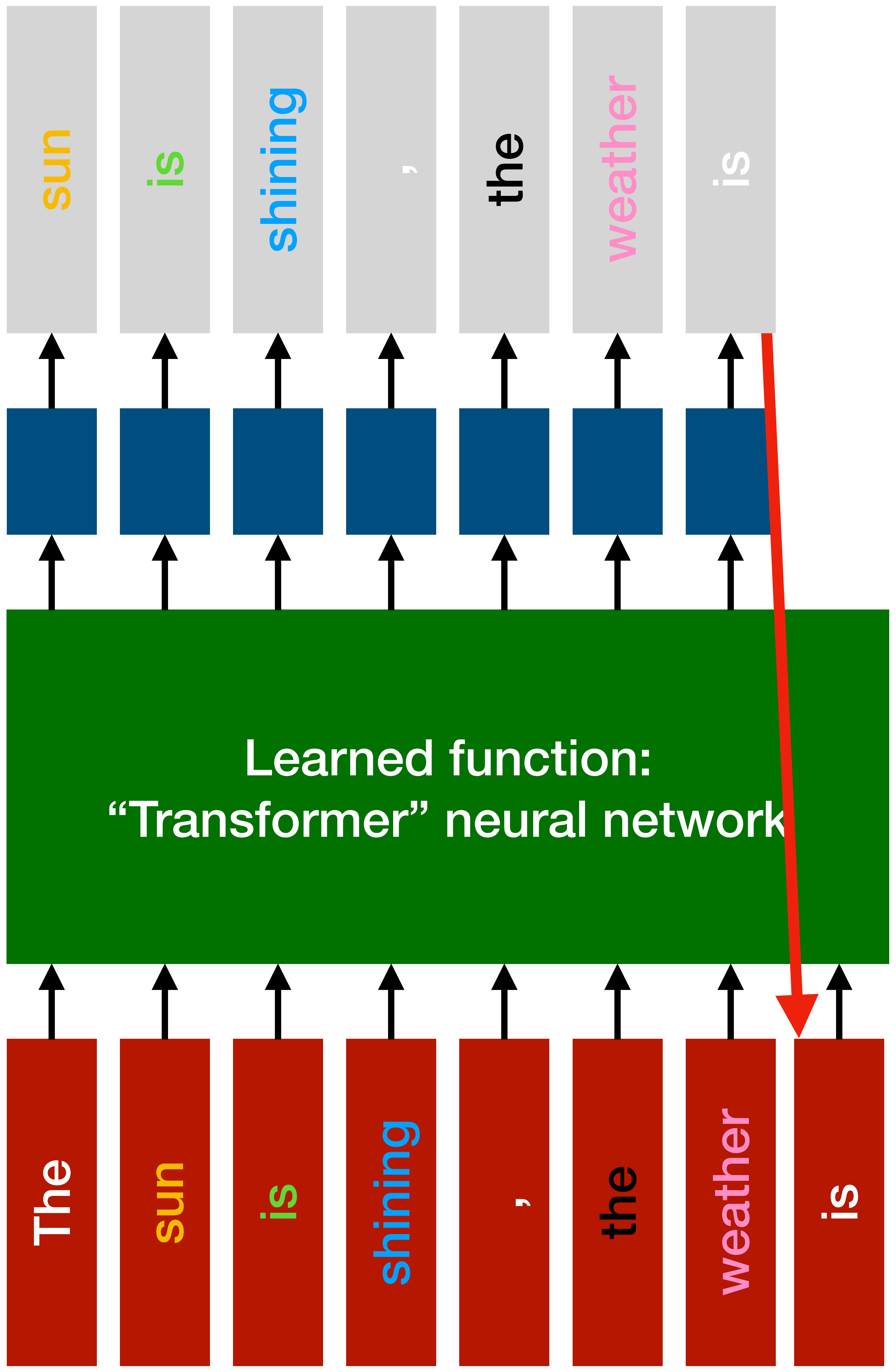


- **Next-word prediction:** train the network to tag every element with the word that comes next

- **Via logistic regression:** contextualized representation of word i is used to predict word $i + 1$ (#classes = #words in vocabulary)

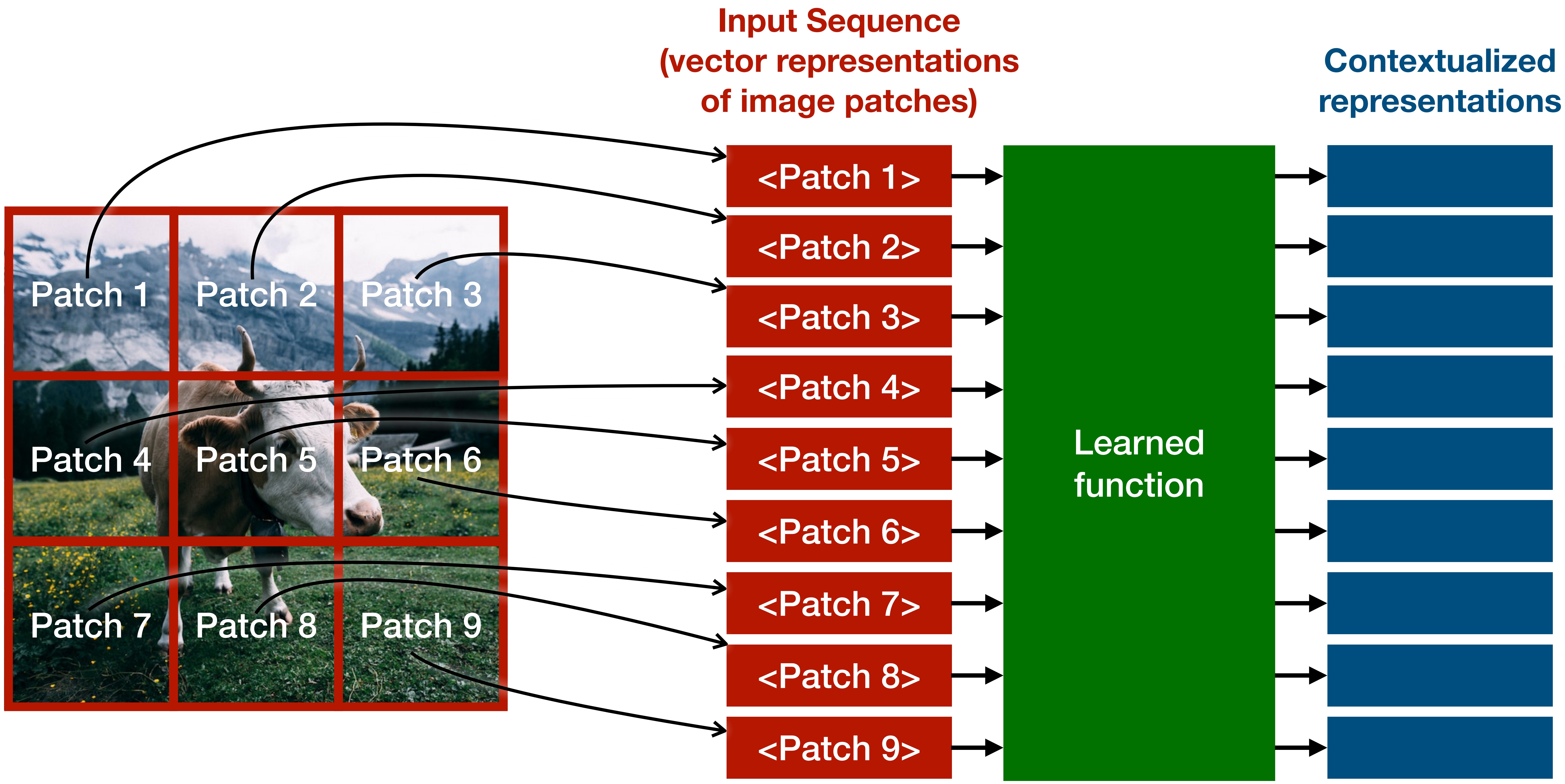
- **Masking:** To avoid "cheating", next-word prediction for word i can only see words $1, \ldots, i$

Large language models: inference



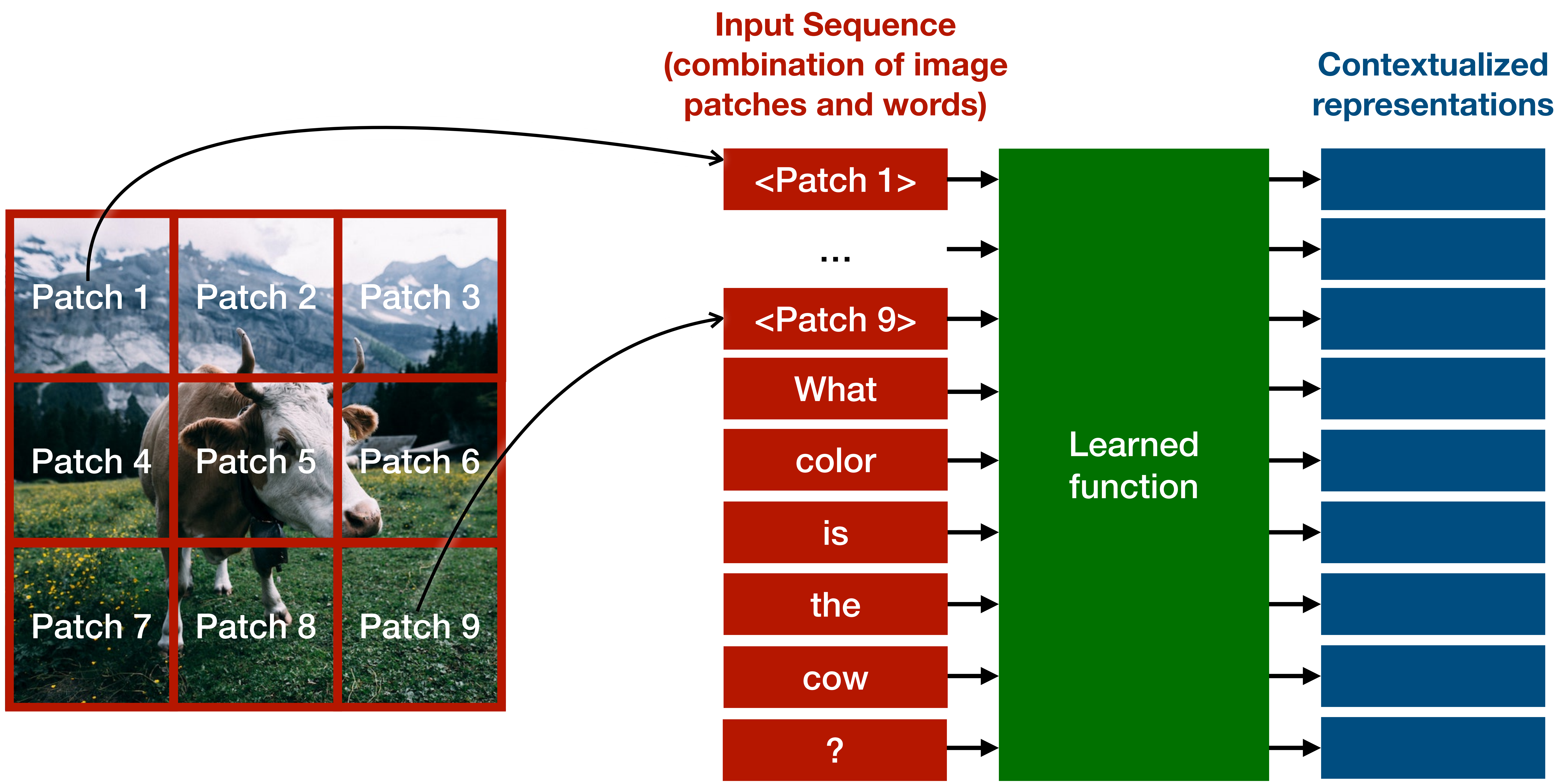
- Obtain contextualized representation of last input element
- Make prediction for next word
- Add it to input
- Repeat

Images can also be represented as sequences

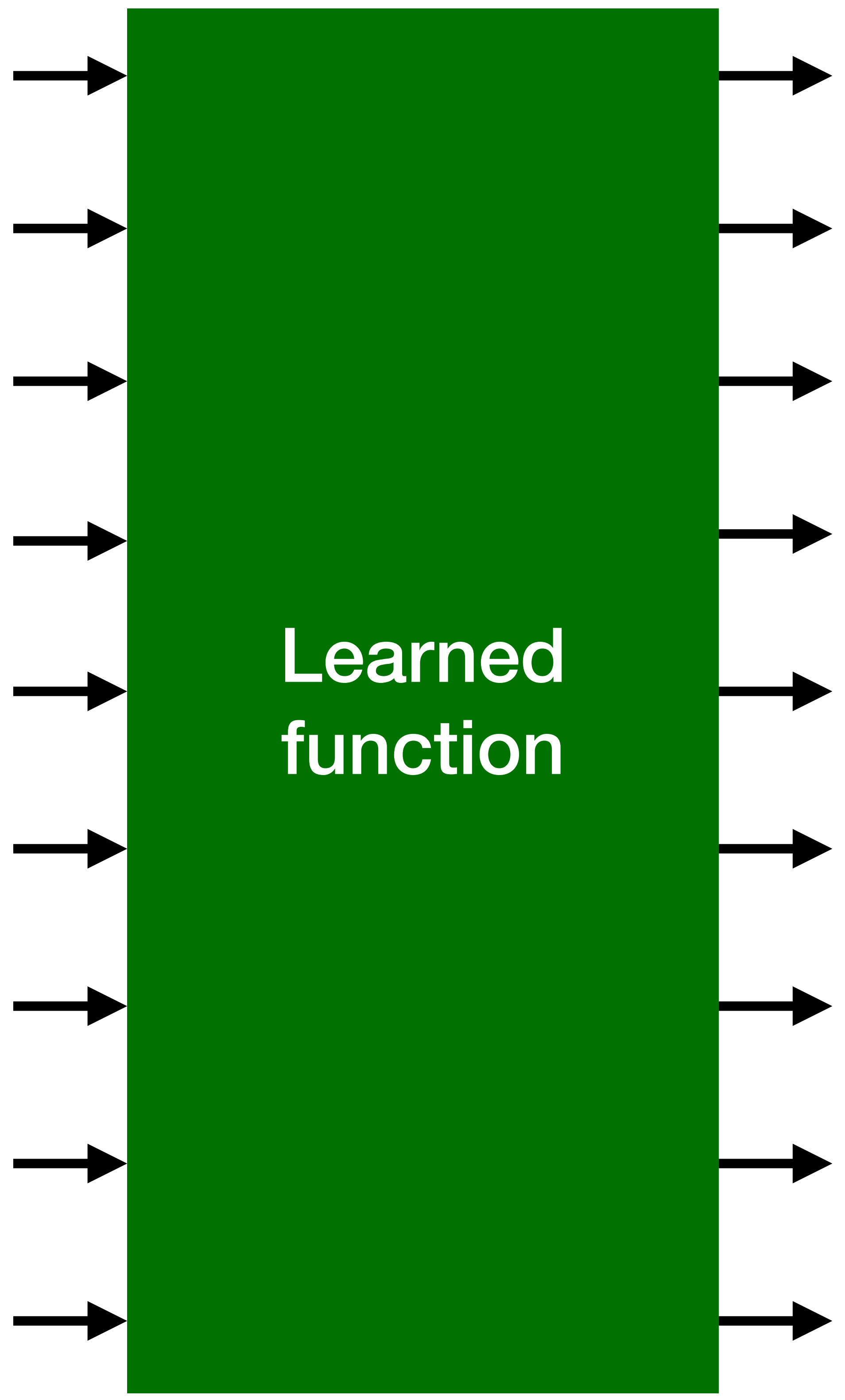


Note: In practice the patches are typically much smaller

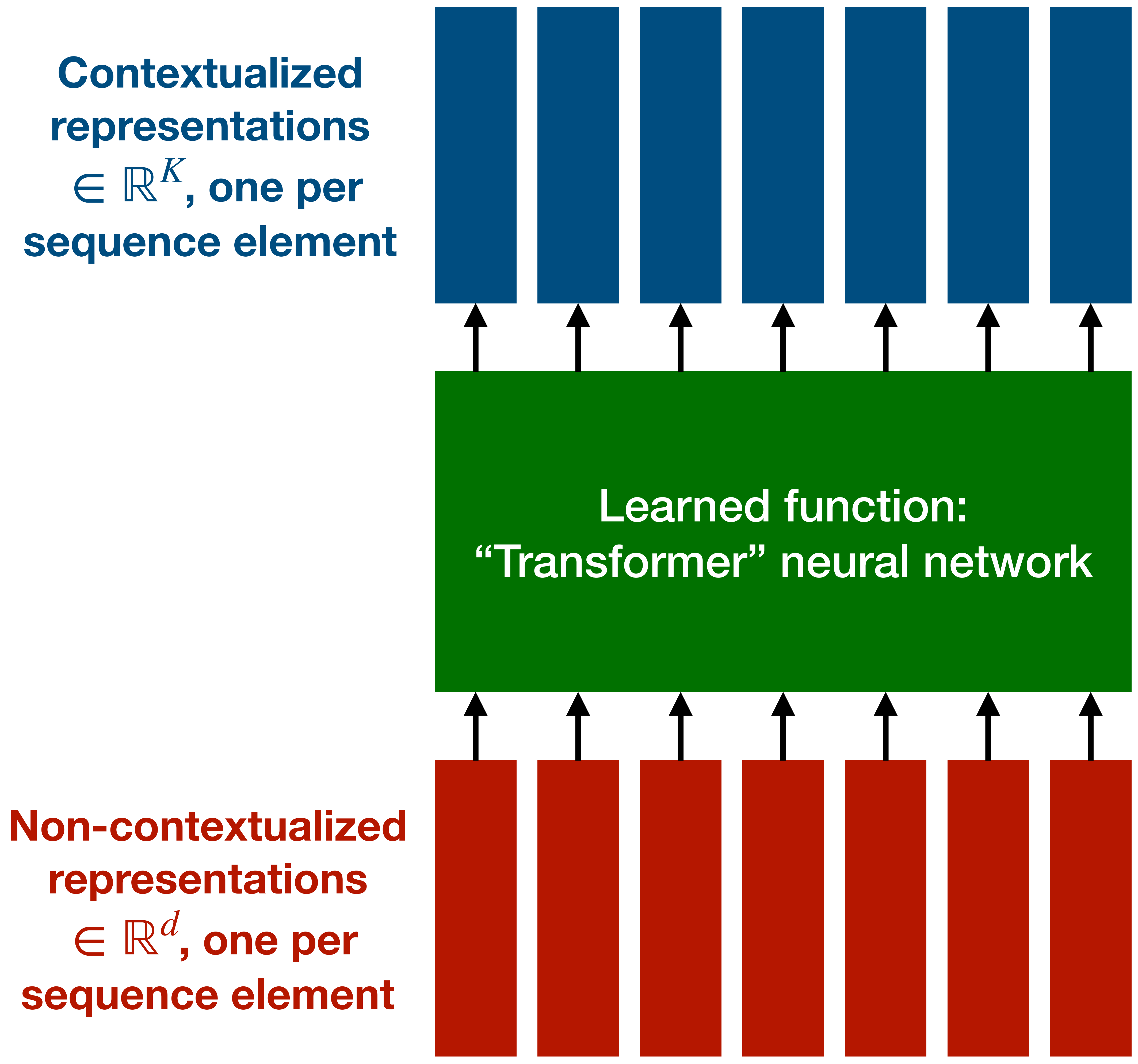
Sequences can be multimodal (image + text)



Transformers



What is a Transformer?

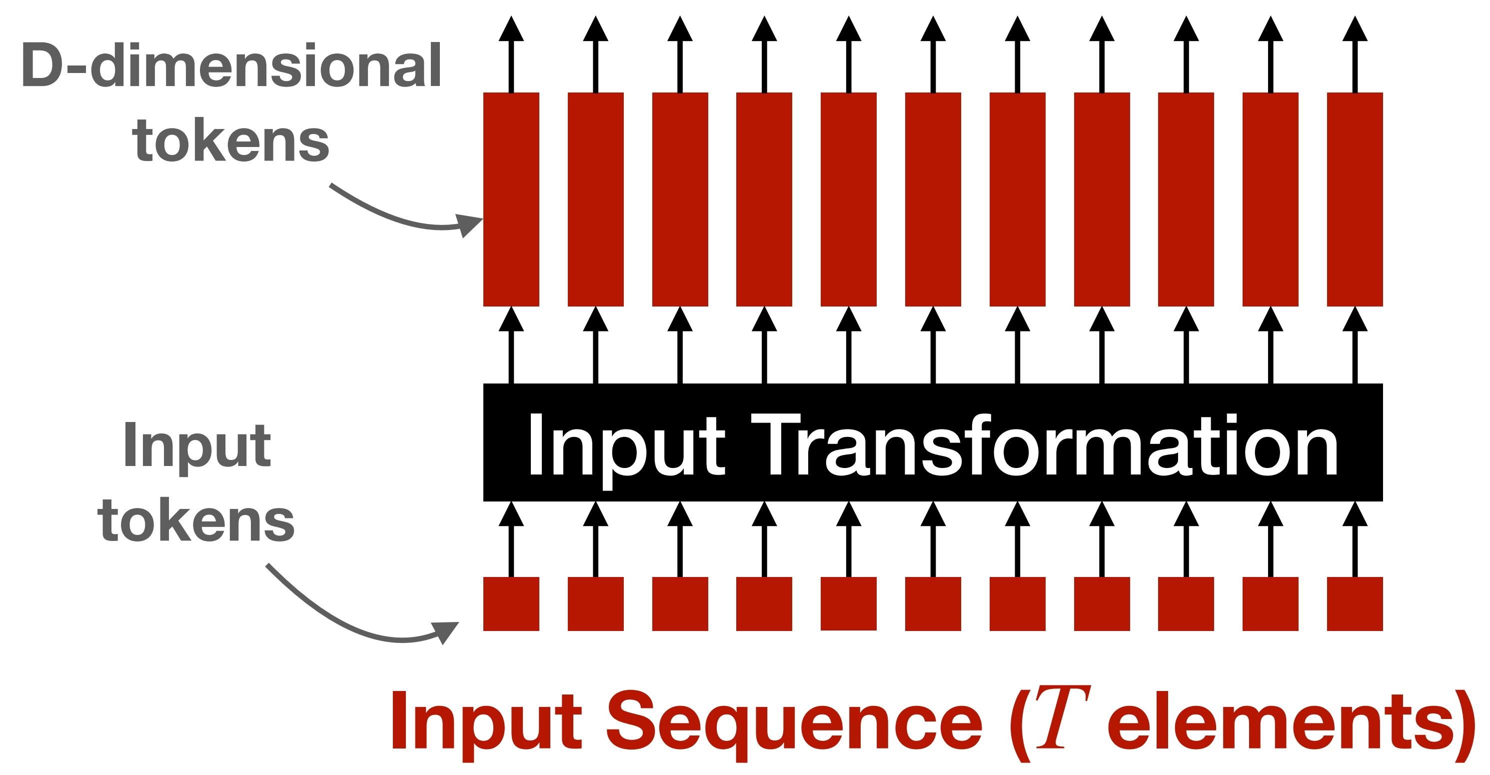


- Iteratively transforms sequence of **non-contextualized** representations into a sequence of **contextualized** representations
- Mixes information between sequence elements via **self-attention**

Overview of Transformer architecture

Input transformation: converts raw input sequence elements ("**tokens**") into real-valued vector representations:

- **Text:** every word is replaced with a (fixed, non-contextualized) vector representing the word
- **Images:** each image patch is flattened into a vector

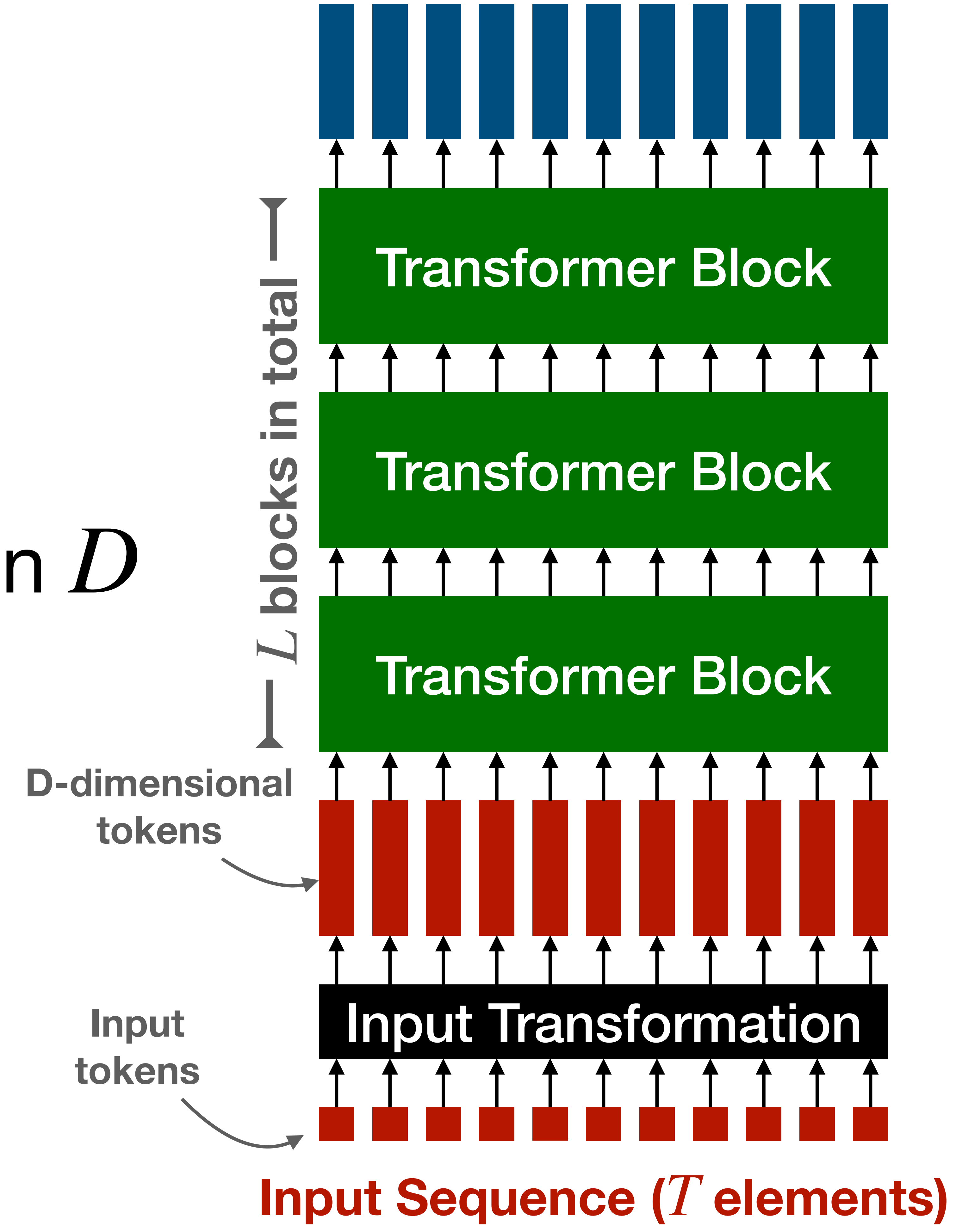


Overview of Transformer architecture

Input transformation: converts raw input sequence elements ("**tokens**") into real-valued vector representations:

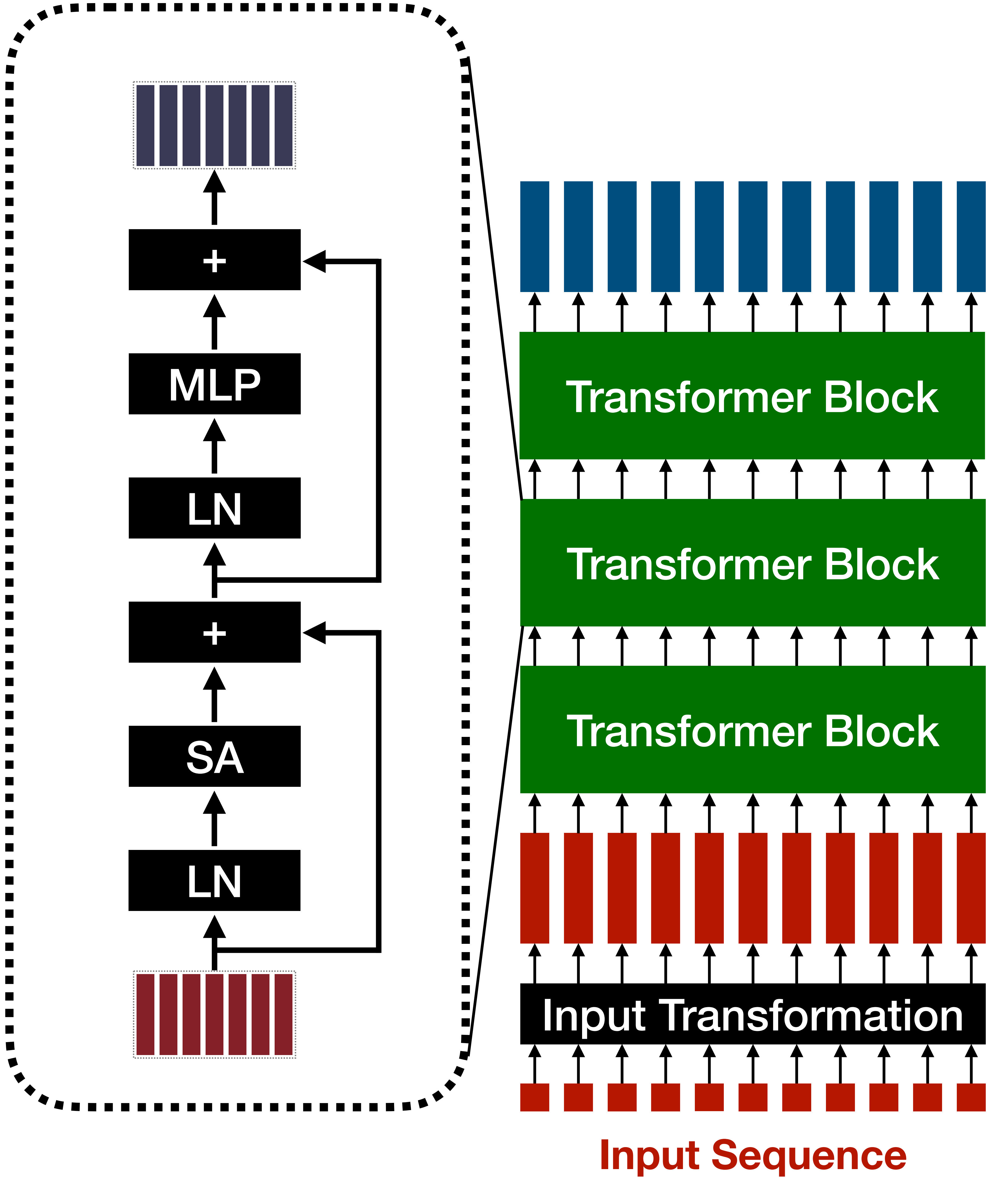
- **Text:** every word is replaced with a (fixed, non-contextualized) vector representing the word
- **Images:** each image patch is flattened into a vector

Transformer block: transforms a sequence of T vectors of dimension D into a new sequence of T vectors of dimension D using **self-attention** and **MLP sub-blocks**



Transformer block

- **Self-Attention (SA):** mixes information **between** tokens
- **Multi-Layer Perceptron (MLP):** mixes information **within** each token
- Other standard components:
  - Skip connections are widely used
  - Layer normalization (LN) is usually placed at the start of a residual branch



Input transformations

Text token embeddings

Tokenization: split the input text into a sequence of *input tokens* (typically word fragments + some special symbols) according to some predefined *tokenizer procedure*:

- Text: `"Transformers are awesome!"`
- Tokens: `["Trans", "form", "ers_", "are_", "awe", "some", "!"]`
- Token IDs: `[5124, 1029, 645, 3001, 6931, 7330, 10]` (each token corresponds to some number $i \in \{1,\ldots,N_{vocab}\}$)

Token embedding: maps each token ID $i \in \{1,\ldots,N_{vocab}\}$ into a real-valued vector $\mathbf{w}_i \in \mathbb{R}^D$:

- Token embeddings: $[\mathbf{w}_{5124}, \mathbf{w}_{1029}, \mathbf{w}_{645}, \mathbf{w}_{3001}, \mathbf{w}_{6931}, \mathbf{w}_{7330}, \mathbf{w}_{10}]$

➡The whole input sequence of T tokens leads to an input matrix $X = \begin{bmatrix} \mathbf{w}_{5124} \\ \mathbf{w}_{1029} \\ \vdots \\ \mathbf{w}_{10} \end{bmatrix} \in \mathbb{R}^{T \times D}$

Notation: Throughout this lecture, all vectors will be treated as row vectors.

Text token embeddings: learning

- The matrix $W_{\text{emb}} = \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_{N_{vocab}} \end{bmatrix} \in \mathbb{R}^{N_{vocab} \times D}$ is learned via backpropagation, along with all other transformer parameters

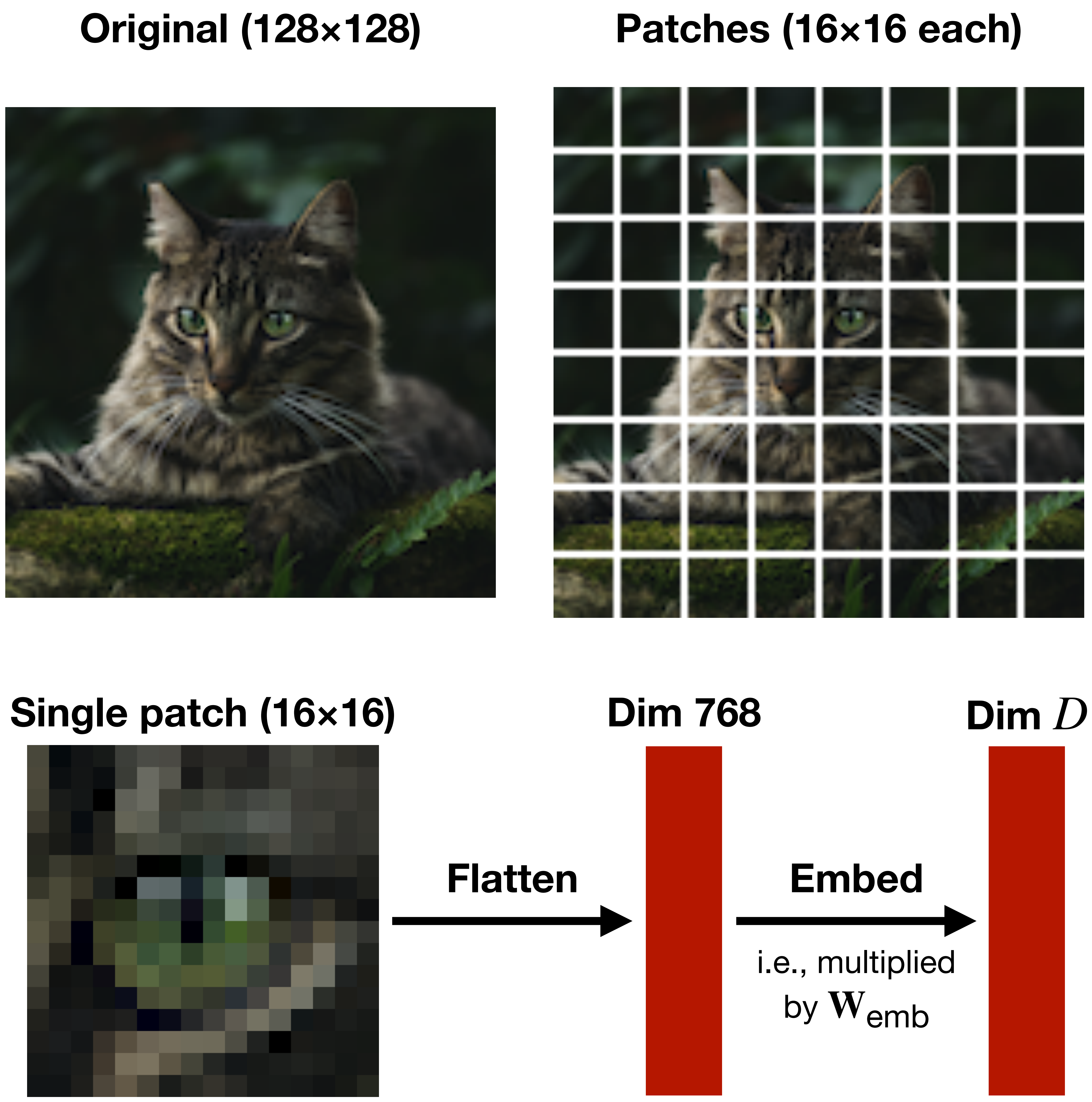
- Token embeddings of concrete input sequence obtained via matrix multiplication:

$$X = \begin{bmatrix} \mathbf{e}_{i_1} \\ \vdots \\ \mathbf{e}_{i_T} \end{bmatrix} W_{\text{emb}} \quad \text{(since } \mathbf{e}_i W_{\text{emb}} = (W_{\text{emb}})_{i,:} = \mathbf{w}_i\text{)}$$

- The tokenizer procedure is typically fixed in advance and not learned

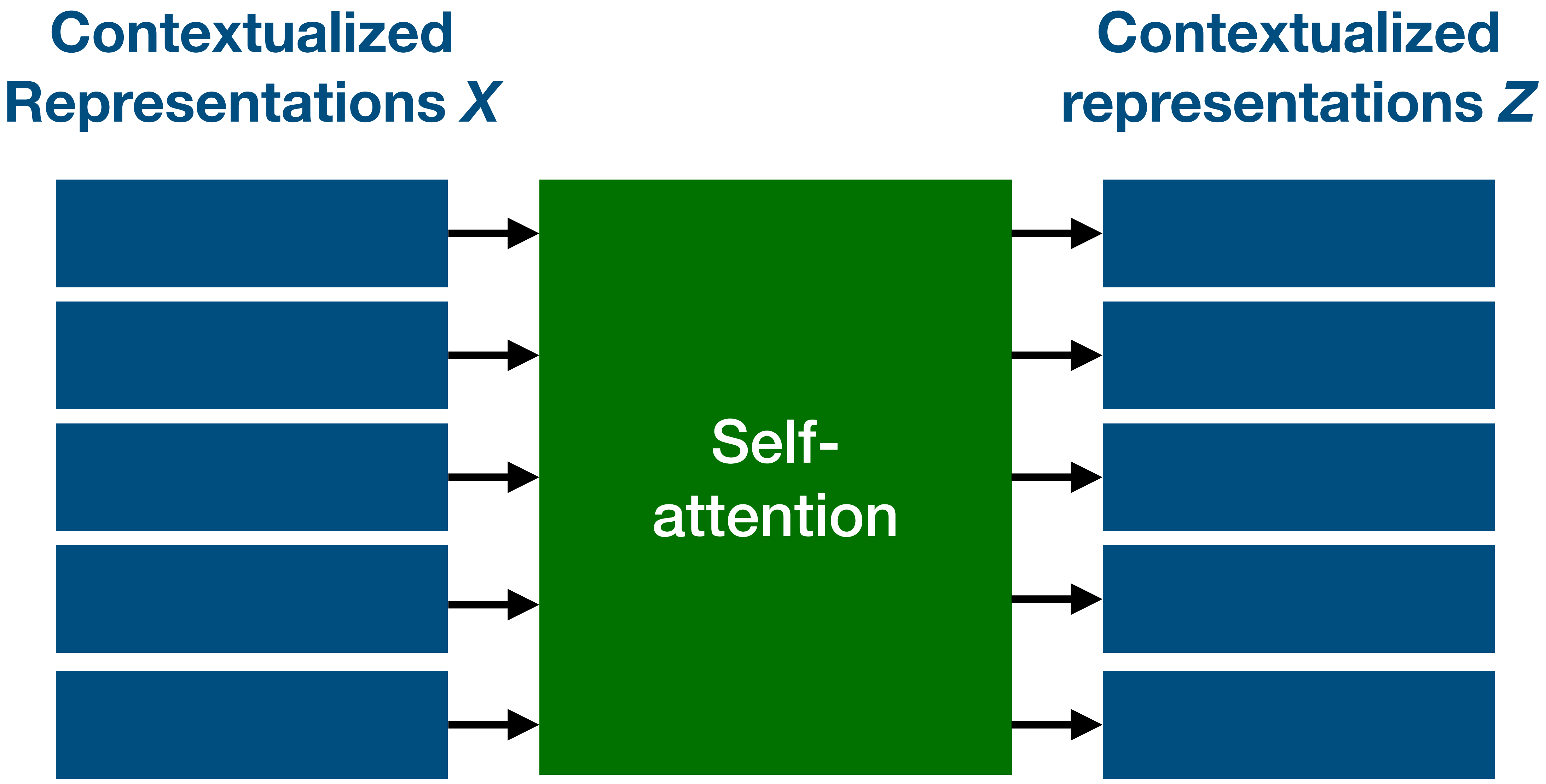
Image patch embeddings

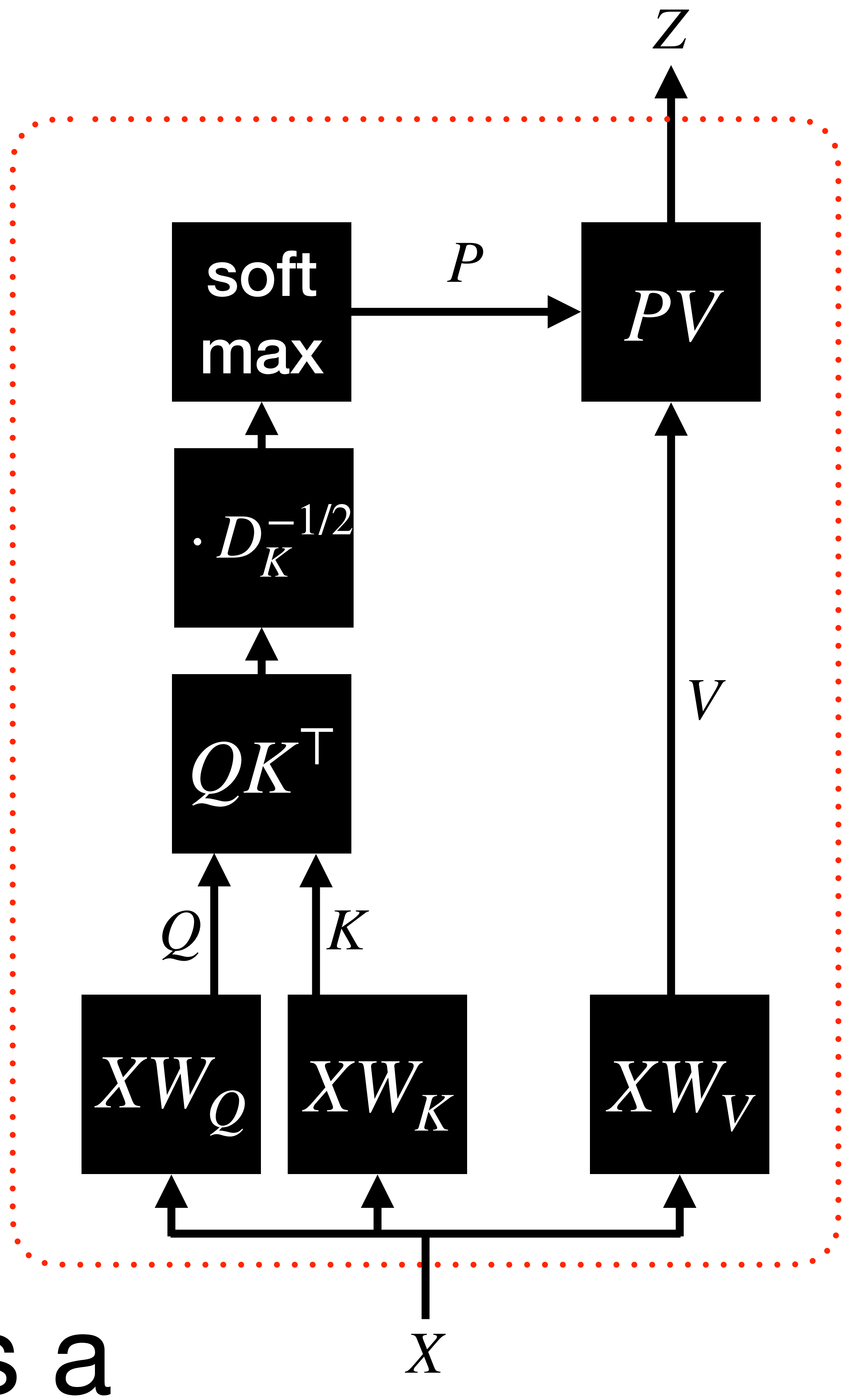
- Divide image into patches of a given size (typical choice: 16×16 pixels each)
- Flatten each patch into a vector of size $16 \cdot 16 \cdot 3 = 768$ (height*width*color channels)
- Multiply each resulting vector by an embedding matrix $W_{\text{emb}} \in \mathbb{R}^{768 \times D}$ which is shared for all inputs
- Learn W_{emb} through backpropagation, along with all other transformer parameters
- The whole input sequence of T embedded patches leads to an input matrix $X \in \mathbb{R}^{T \times D}$



Self-attention

What is self-attention?





Self-attention is a function that transforms a sequence of contextualized representations (one per input token) into a new sequence of contextualized representations using **learned input-dependent weighted averages**

Attention as a weighted average

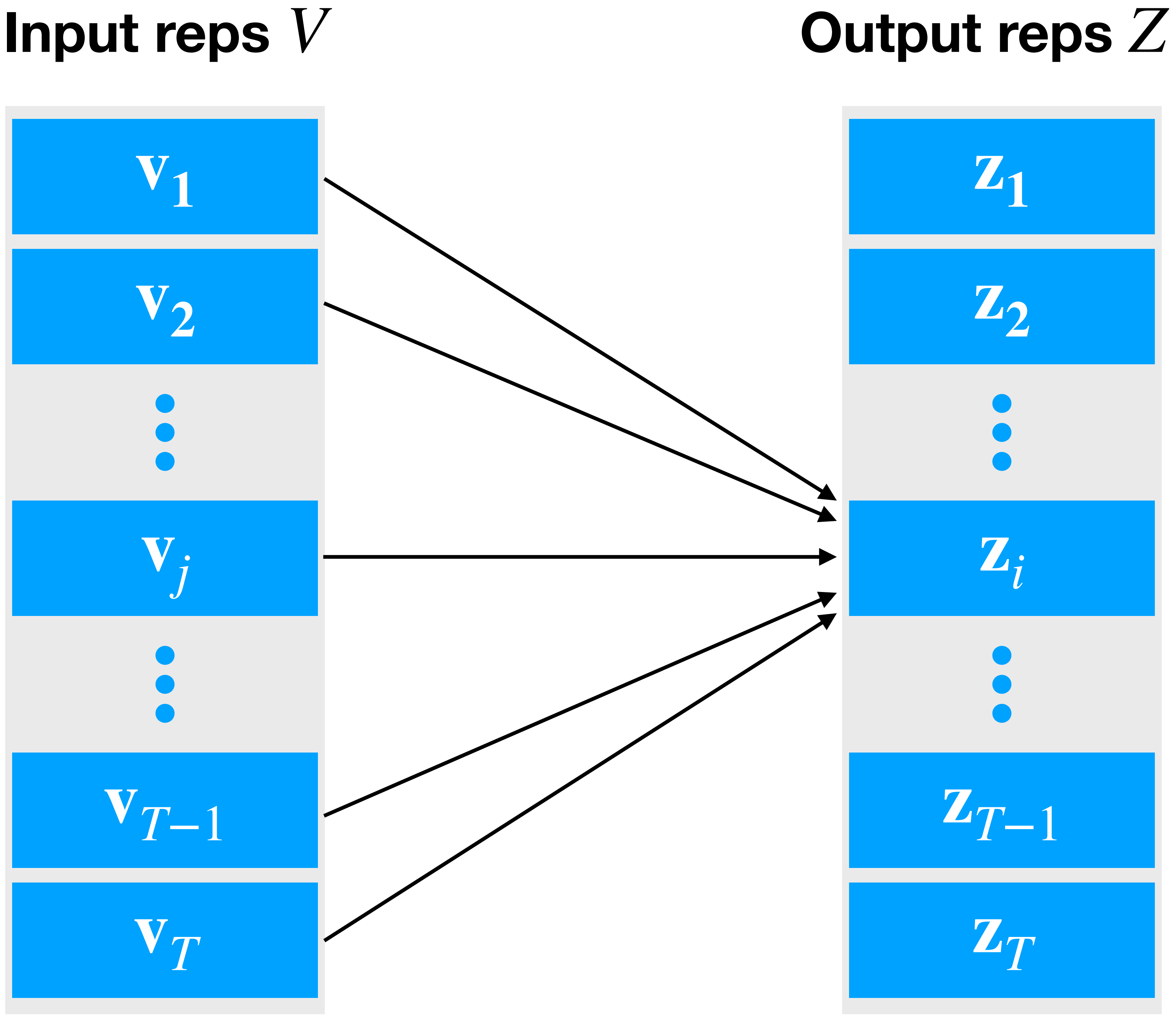
- T **input and output tokens**: $V \in \mathbb{R}^{T \times D_V}, Z \in \mathbb{R}^{T \times D_V}$
- Each output is a **weighted average** of the inputs:

$$\mathbf{z}_i = \sum_{j=1}^{T} p_{i,j} \mathbf{v}_j \quad \text{or in matrix form} \quad Z = PV$$

- Weighting coefficients $P \in [0,1]^{T \times T}$ form valid probability distributions over the input tokens:

➡ $\sum_{j=1}^{T} p_{i,j} = 1$ (i.e., each row sums to one)

Notation: throughout this lecture, the j-th **rows** of V and Z are denoted by $\mathbf{v}_j$ and $\mathbf{z}_j$



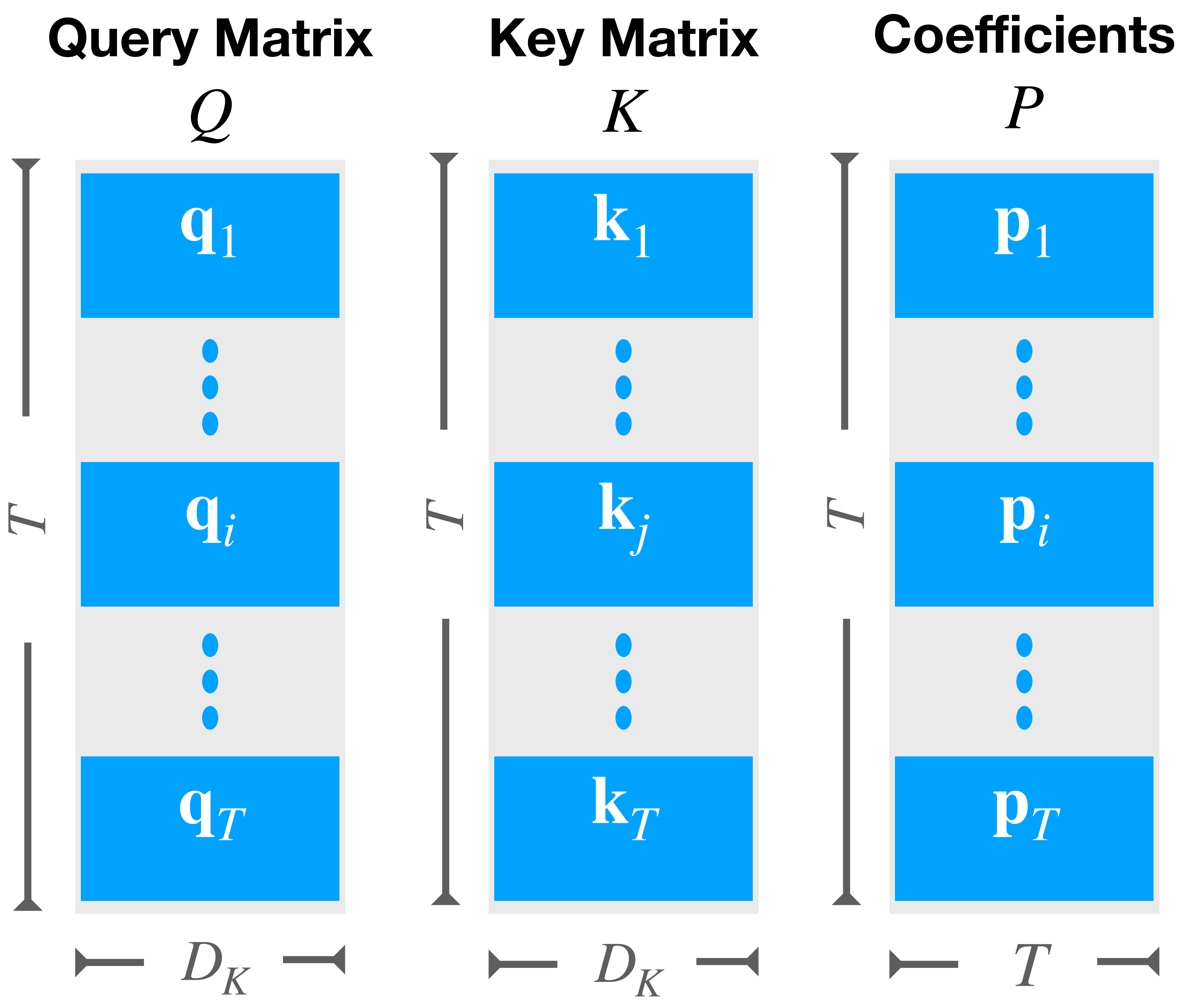
The weighting coefficients P

- **Query tokens** $Q \in \mathbb{R}^{T \times D_K}$ (one query per output token)
- **Key tokens** $K \in \mathbb{R}^{T \times D_K}$ (one key per input token)
- Determine weight $p_{i,j}$ based on **how similar $\mathbf{q}_i$ and $\mathbf{k}_j$** are
  - Use inner product to obtain raw similarity scores
  - Normalize with **softmax** (scaled with "temperature" $\sqrt{D_K}$) to obtain a probability distribution
- This can be expressed as:

Element-wise: $p_{i,j} = \dfrac{\exp\left(\mathbf{q}_i \mathbf{k}_j^\top / \sqrt{D_K}\right)}{\sum_{t=1}^{T} \exp\left(\mathbf{q}_i \mathbf{k}_t^\top / \sqrt{D_K}\right)}$

Matrix form: $P = \text{softmax}\left(\dfrac{QK^\top}{\sqrt{D_K}}\right)$

The softmax
is applied on each
row *independently*



Computation complexity:
$O(T \times T)$

The weighting coefficients P

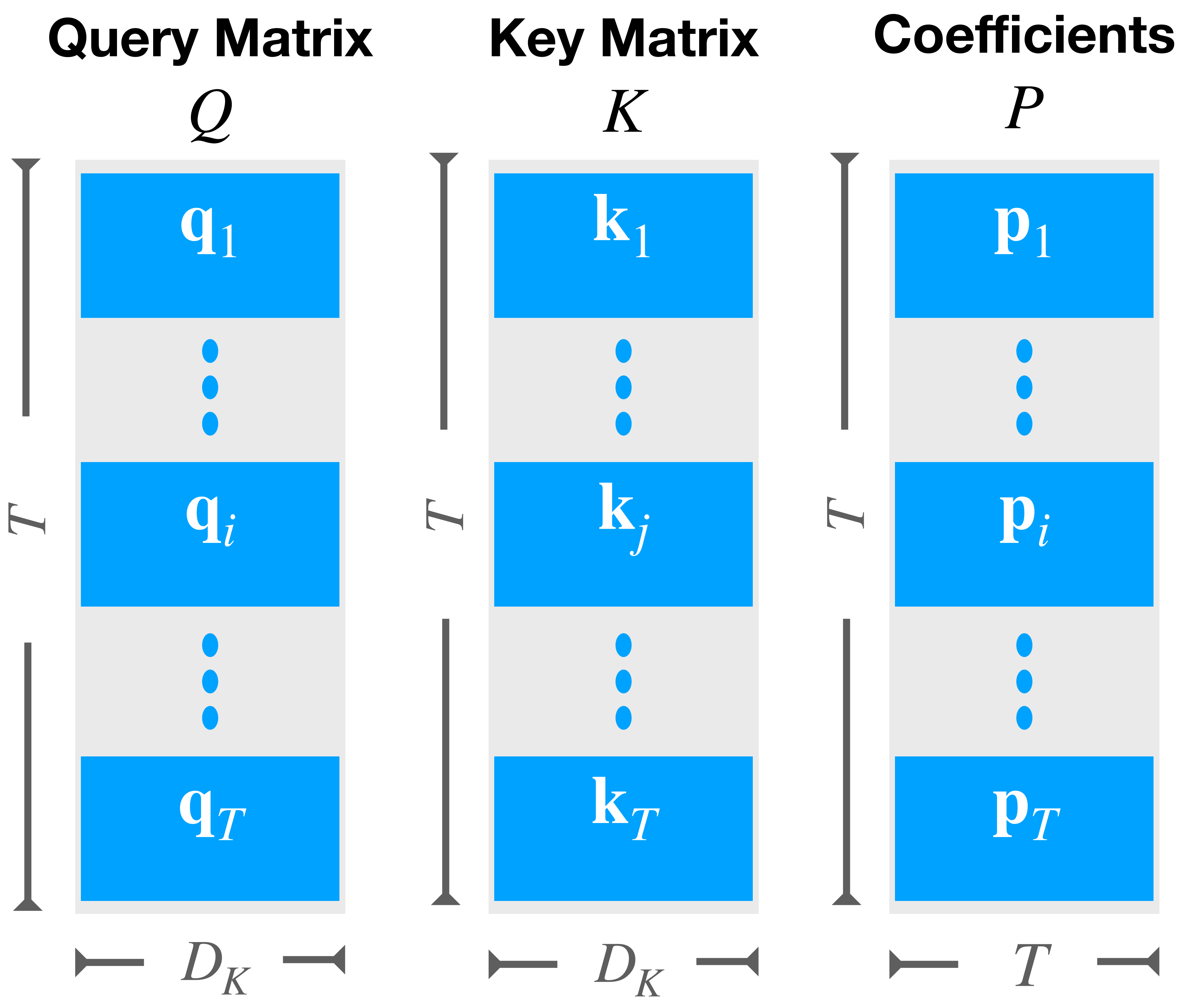
- **Query tokens** $Q \in \mathbb{R}^{T \times D_K}$ (one query per output token)
- **Key tokens** $K \in \mathbb{R}^{T \times D_K}$ (one key per input token)
- Determine weight $p_{i,j}$ based on **how similar $\mathbf{q}_i$ and $\mathbf{k}_j$** are
  - Use inner product to obtain raw similarity scores
  - Normalize with softmax (scaled the temperature by $\sqrt{D_K}$) to obtain a probability distribution

In some applications, **causal masking** is used:

Sum until position i: $p_{i,j} = \dfrac{\exp\left(\mathbf{q}_i \mathbf{k}_j^\top / \sqrt{D_K}\right)}{\sum_{t=1}^{i} \exp\left(\mathbf{q}_i \mathbf{k}_t^\top / \sqrt{D_K}\right)}$ for $j \leq i$ and $p_{i,j} = 0$ otherwise

Mask before softmax: $P = \mathrm{softmax}\left(M + \dfrac{QK^\top}{\sqrt{D_K}}\right)$

where $M \in \mathbb{R}^{T \times T}$ is the matrix $M_{ij} = -\infty$ for $j > i$ and $M_{i,j} = 0$ otherwise



Computation complexity:

$O(T \times T)$

Computing *K*, *Q*, *V*

Define K, Q, V from the **same** input sequence $X \in \mathbb{R}^{T\times D}$

- **Keys:** $K = XW_K \in \mathbb{R}^{T\times D_K}$
- **Queries:** $Q = XW_Q \in \mathbb{R}^{T\times D_K}$
- **Values:** $V = XW_V \in \mathbb{R}^{T\times D_V}$

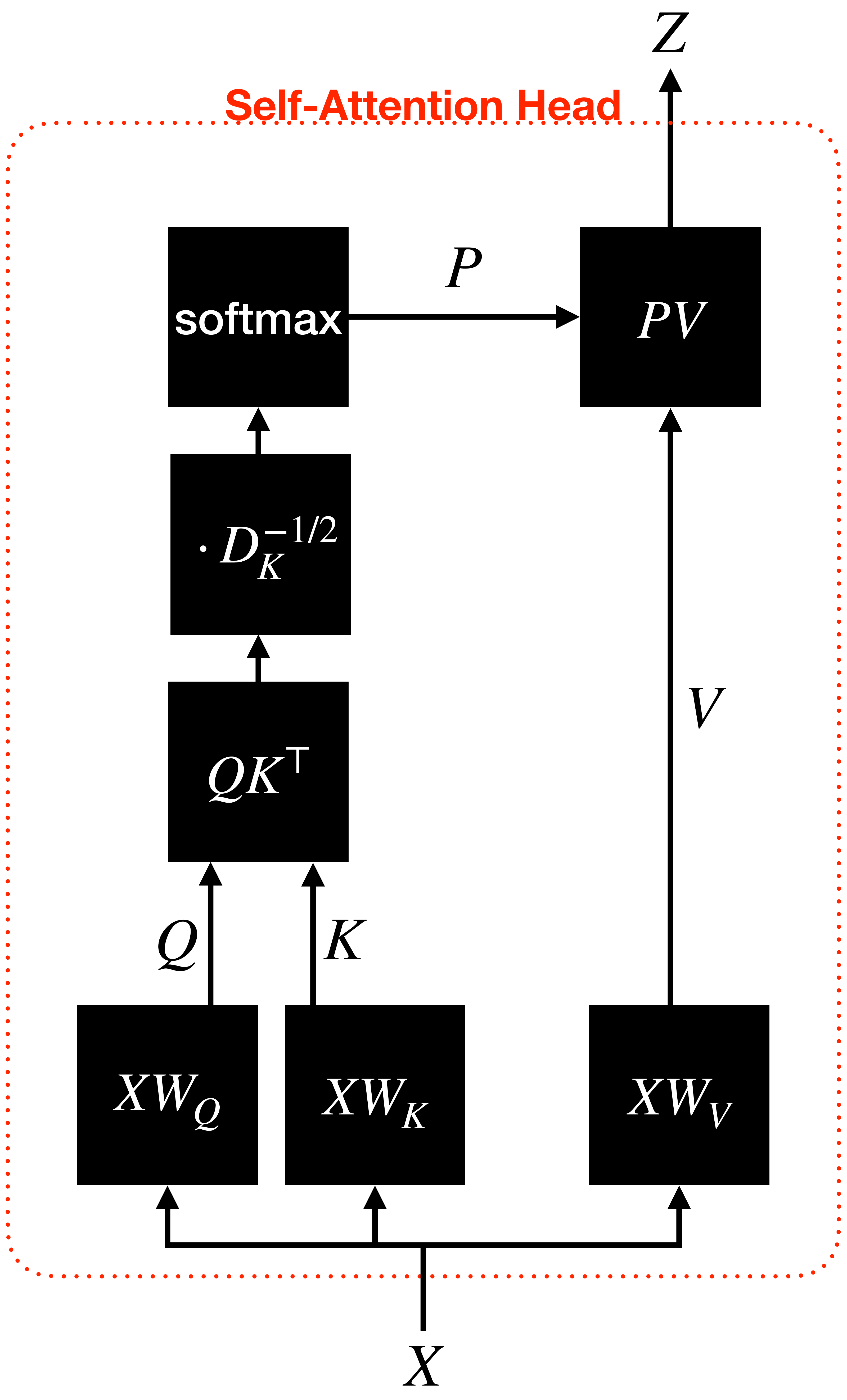
➡ $W_K, W_Q \in \mathbb{R}^{D\times D_K}$, $W_V \in \mathbb{R}^{D\times D_V}$ are parameters

The output of self-attention is then given by:

$$Z = \text{softmax}\left(\frac{QK^\top}{\sqrt{D_K}}\right)V \qquad \text{softmax}(x_1, \ldots, x_T) = \left(\frac{e^{x_i}}{\sum_{t=1}^{T} e^{x_t}}\right)_{i=1}^{T}$$

➡ softmax$(\,\cdot\,)$ is applied row-wise

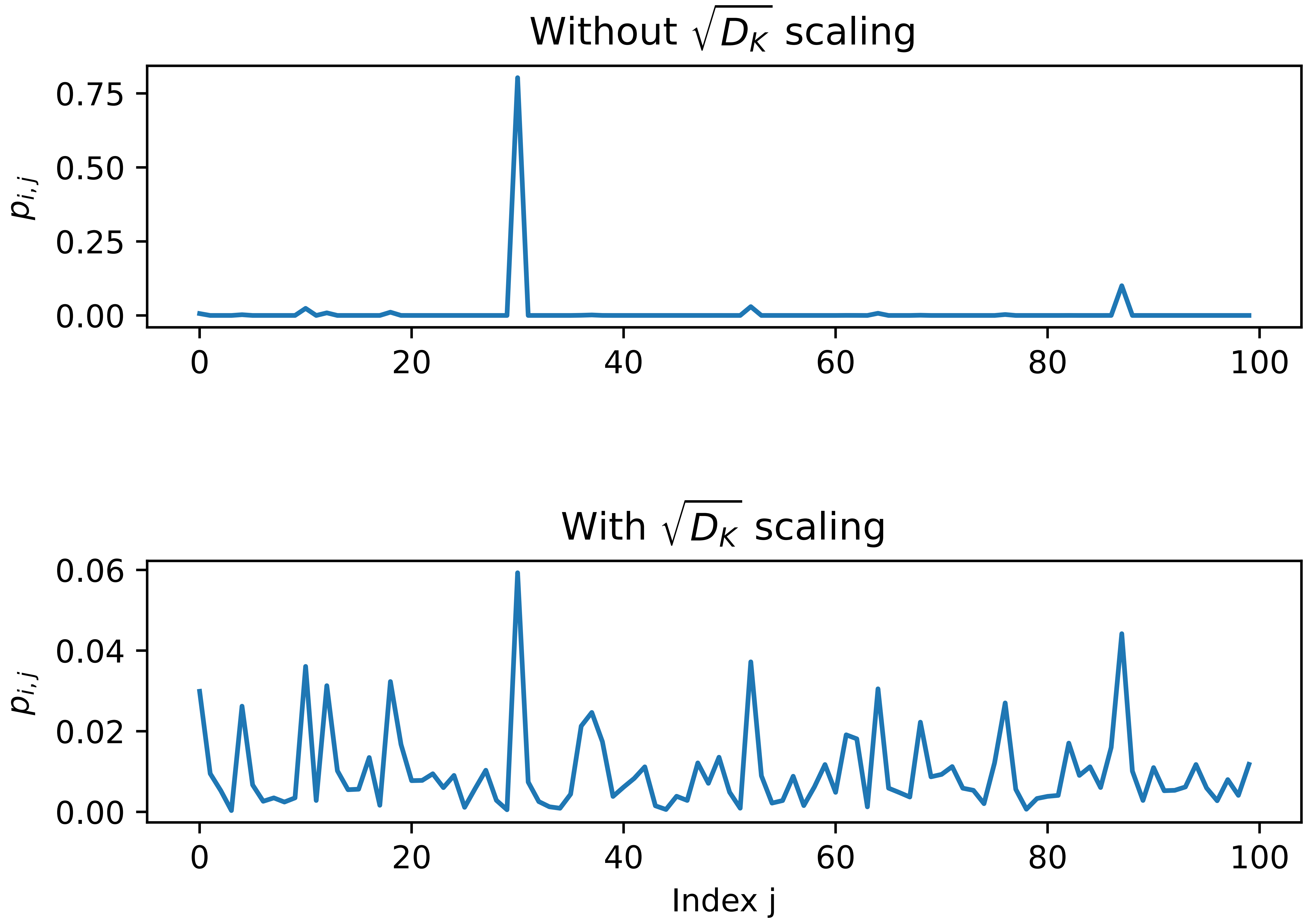
➡ Quadratic computational complexity $O(T^2)$



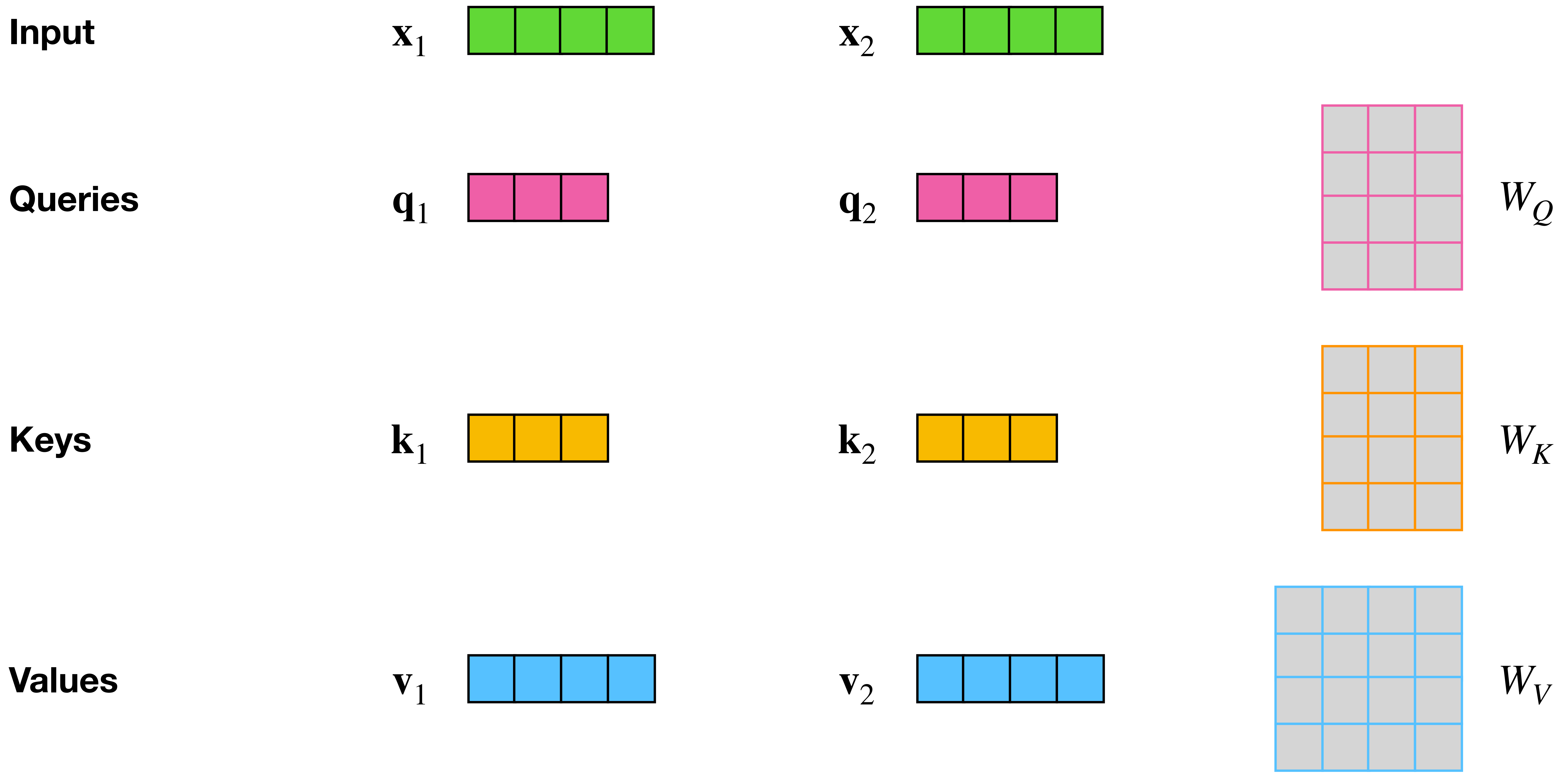
Why use $1/\sqrt{D_K}$ scaling?

$$P = \text{softmax}\left(\frac{QK^\top}{\sqrt{D_K}}\right)$$

- **Without scaling**: sharp distribution of the attention weights $p_{i,j}$ at random initialization
- The model takes much more time to adjust from the initial peak due to vanishing gradients
- The $1/\sqrt{D_K}$ scaling ensures uniformity at initialization and faster convergence



Self-Attention: Step-by-Step



$$\boxed{\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \end{aligned}}$$

Multiplying the input by the Q/K/V weight matrices, we create a query, a key and a value projection of each input of the input sequence

Illustrations borrowed from The Illustrated Transformer

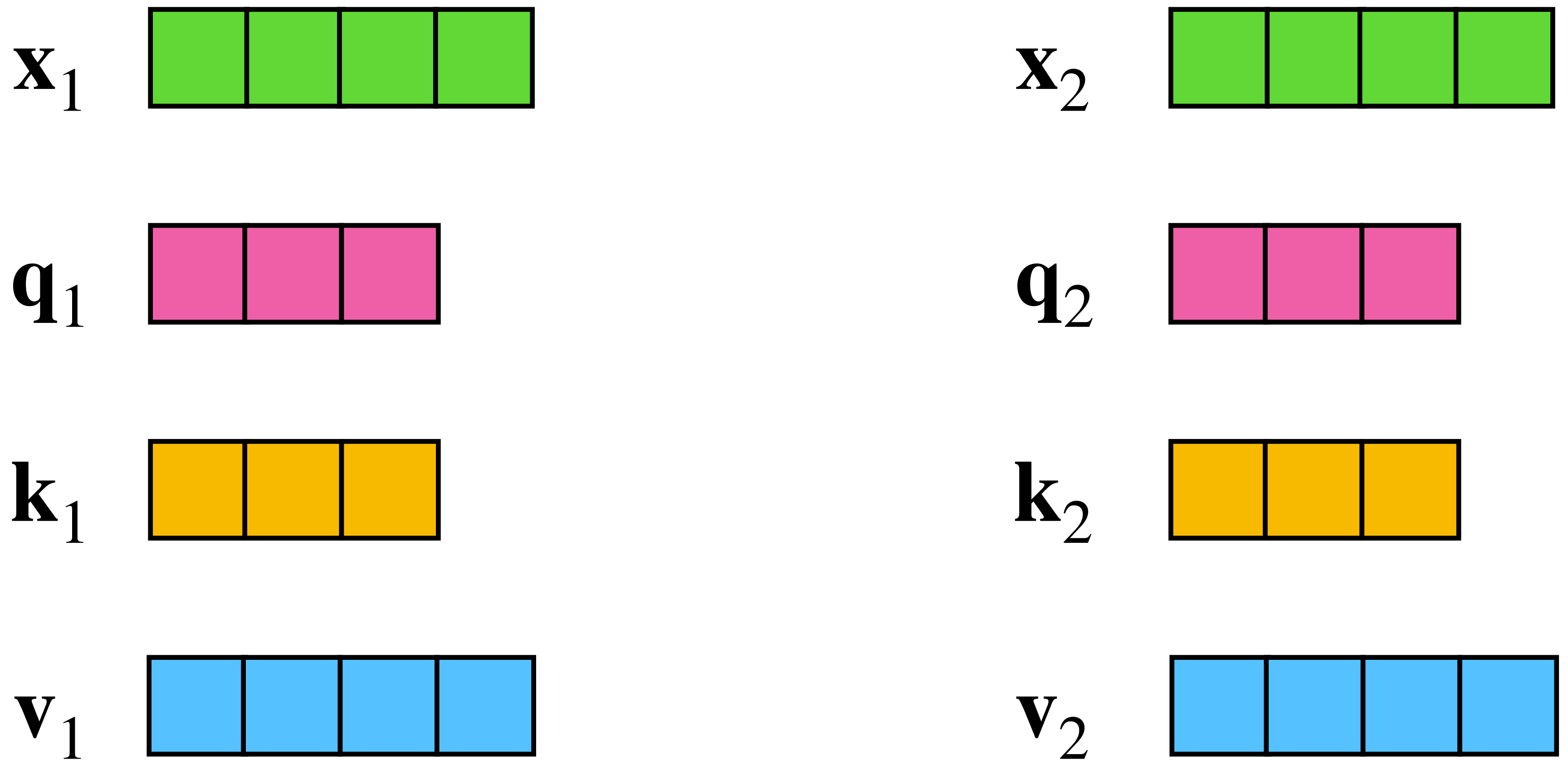
Self-Attention: Step-by-Step

Input

Queries

Keys

Values

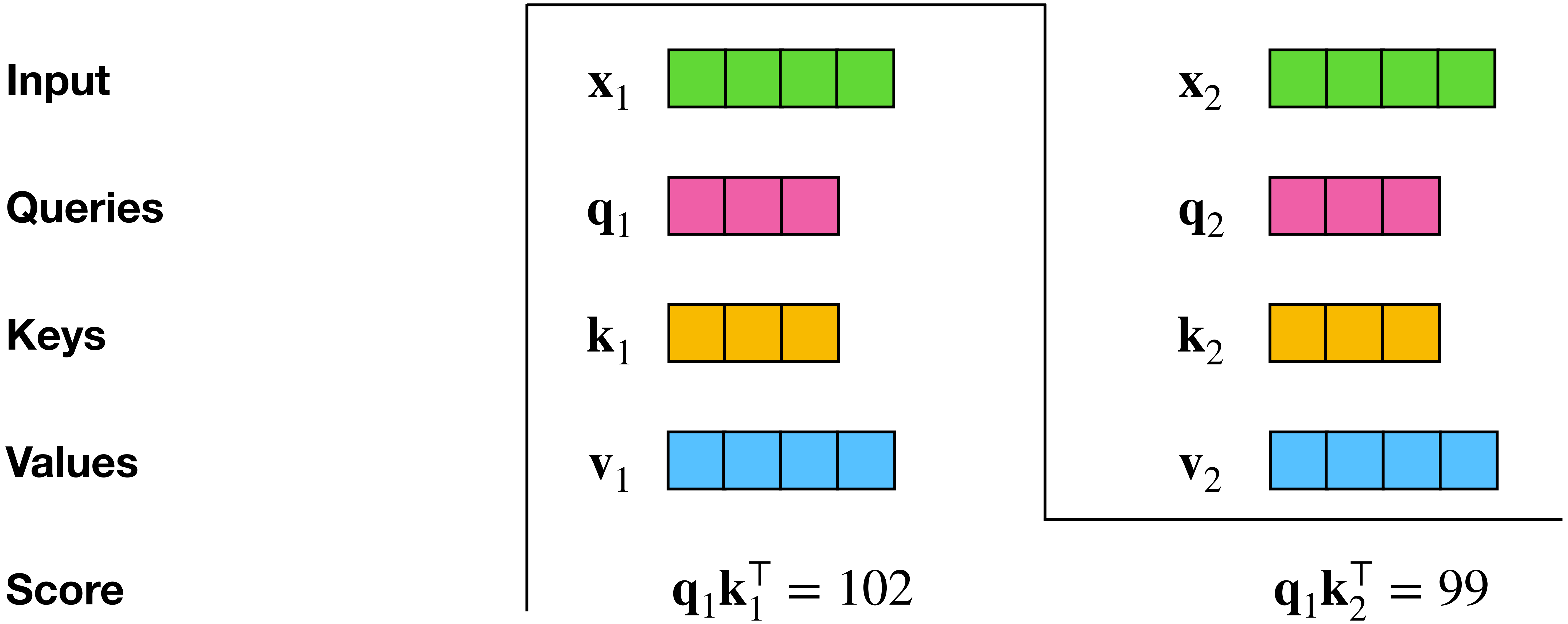


Step 1: create query, key and value vectors for each input token

$$Q = XW_Q$$
$$K = XW_K$$
$$V = XW_V$$

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step



Step 2: calculate the scores by taking scalar product of the query and key vectors

$$\boxed{QK^\top = XW_Q W_K^\top X^\top}$$

Self-Attention: Step-by-Step

Step 3: divide the scores by $\sqrt{D_K}$

Step 4: Compute the softmax of these values

Input	$\mathbf{x}_1$	$\mathbf{x}_2$
Queries	$\mathbf{q}_1$	$\mathbf{q}_2$
Keys	$\mathbf{k}_1$	$\mathbf{k}_2$
Values	$\mathbf{v}_1$	$\mathbf{v}_2$
Score	$\mathbf{q}_1\mathbf{k}_1^\top = 102$	$\mathbf{q}_1\mathbf{k}_2^\top = 99$
Divide by $\sqrt{D_K}$	$\frac{\mathbf{q}_1\mathbf{k}_1^\top}{\sqrt{D_K}} = 58.9$	$\frac{\mathbf{q}_1\mathbf{k}_2^\top}{\sqrt{D_K}} = 57.2$
Softmax	$p_{1,1} = 0.85$	$p_{1,2} = 0.15$

$$P = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{D_K}}\right)$$

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step

Input	$\mathbf{x}_1$	$\mathbf{x}_2$
Queries	$\mathbf{q}_1$	$\mathbf{q}_2$
Keys	$\mathbf{k}_1$	$\mathbf{k}_2$
Values	$\mathbf{v}_1$	$\mathbf{v}_2$
Score	$\mathbf{q}_1\mathbf{k}_1^\top = 102$	$\mathbf{q}_1\mathbf{k}_2^\top = 99$
Divide by $\sqrt{D_K}$	$\frac{\mathbf{q}_1\mathbf{k}_1^\top}{\sqrt{D_K}} = 58.9$	$\frac{\mathbf{q}_1\mathbf{k}_2^\top}{\sqrt{D_K}} = 57.2$
Softmax	$p_{1,1} = 0.85$	$p_{1,2} = 0.15$
Softmax*Value	$p_{1,1}\mathbf{v}_1$	$p_{1,2}\mathbf{v}_2$
Sum	$\mathbf{z}_1$	$\mathbf{z}_2$

Step 5: Multiply each value vector by the softmax score

Step 6: Sum up the weighted value vectors

$$\boxed{Z = PV}$$

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step

Input	$\mathbf{x}_1$	$\mathbf{x}_2$
Queries	$\mathbf{q}_1$	$\mathbf{q}_2$
Keys	$\mathbf{k}_1$	$\mathbf{k}_2$
Values	$\mathbf{v}_1$	$\mathbf{v}_2$
Score	$\mathbf{q}_1\mathbf{k}_1^\top = 102$	$\mathbf{q}_1\mathbf{k}_2^\top = 99$
Divide by $\sqrt{D_K}$	$\frac{\mathbf{q}_1\mathbf{k}_1^\top}{\sqrt{D_K}} = 58.9$	$\frac{\mathbf{q}_1\mathbf{k}_2^\top}{\sqrt{D_K}} = 57.2$
Softmax	$p_{1,1} = 0.85$	$p_{1,2} = 0.15$
Softmax*Value	$p_{1,1}\mathbf{v}_1$	$p_{1,2}\mathbf{v}_2$
Sum	$\mathbf{z}_1$	$\mathbf{z}_2$

$$\boxed{Z = \mathrm{softmax}\left(\frac{XW_QW_K^\top X^\top}{\sqrt{D_K}}\right)XW_V}$$

Illustrations borrowed from The Illustrated Transformer

Multi-Head Self-Attention

- It is desirable to have multiple attention patterns per layer, similar to having multiple output channels in a convolution layer

  ➡ Run H self-attention "heads" in parallel

- The output of head h is given by:

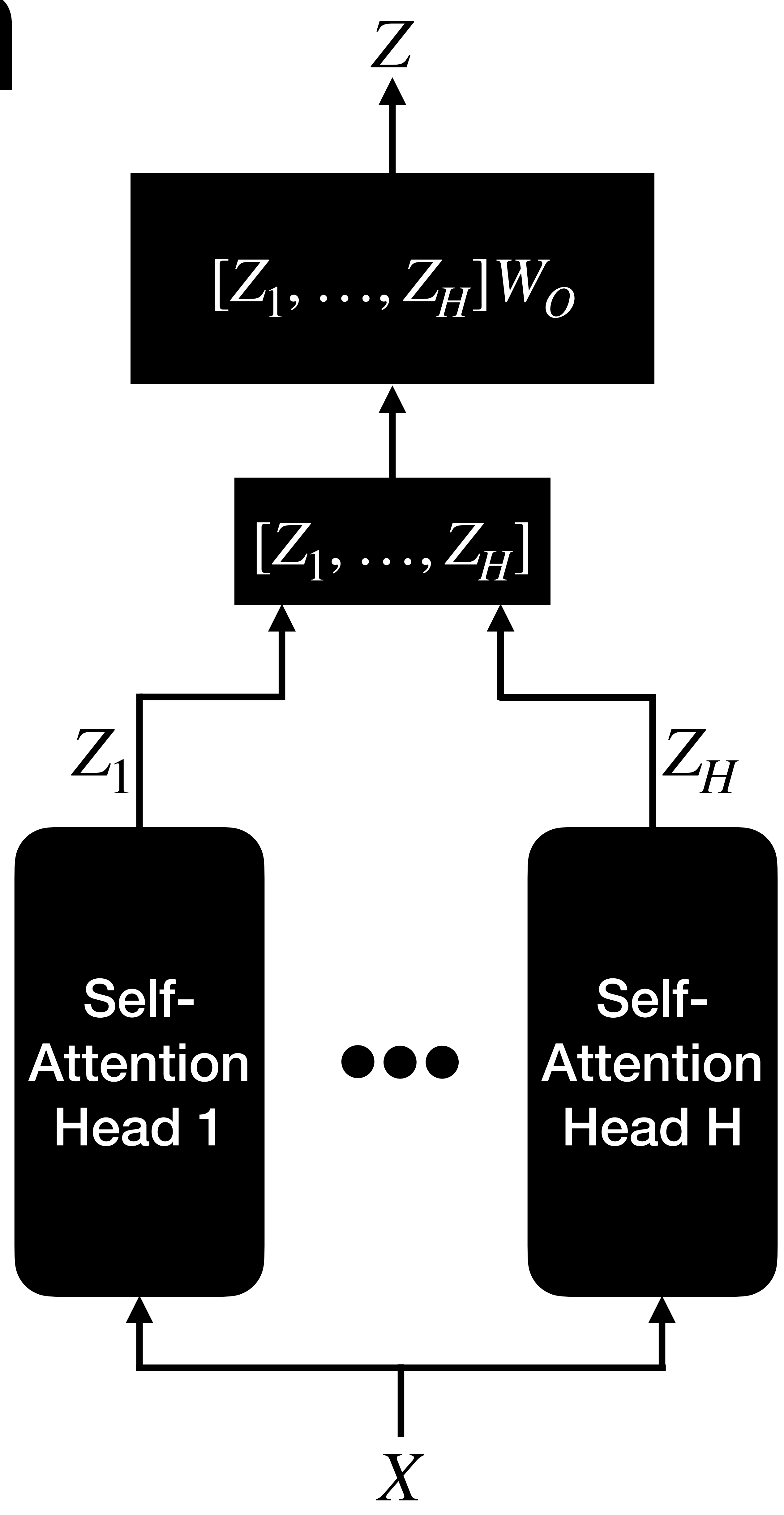
$$Z_h = \text{softmax}\left(\frac{XW_{Q,h}W_{K,h}^\top X^\top}{\sqrt{D_K}}\right)XW_{V,h}$$

$$W_{V,h} \in \mathbb{R}^{D\times D_V},\ W_{K,h} \in \mathbb{R}^{D\times D_K},\ W_{Q,h} \in \mathbb{R}^{D\times D_K}$$

- The final output is obtained by concatenating head-outputs and applying a linear transformation

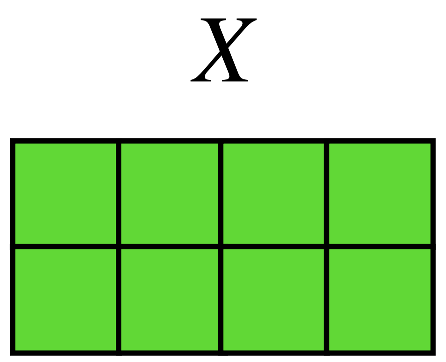
$$Z = [Z_1, \ldots, Z_H]W_O$$

where $W_O \in \mathbb{R}^{HD_V\times D}$ is learned via backpropagation

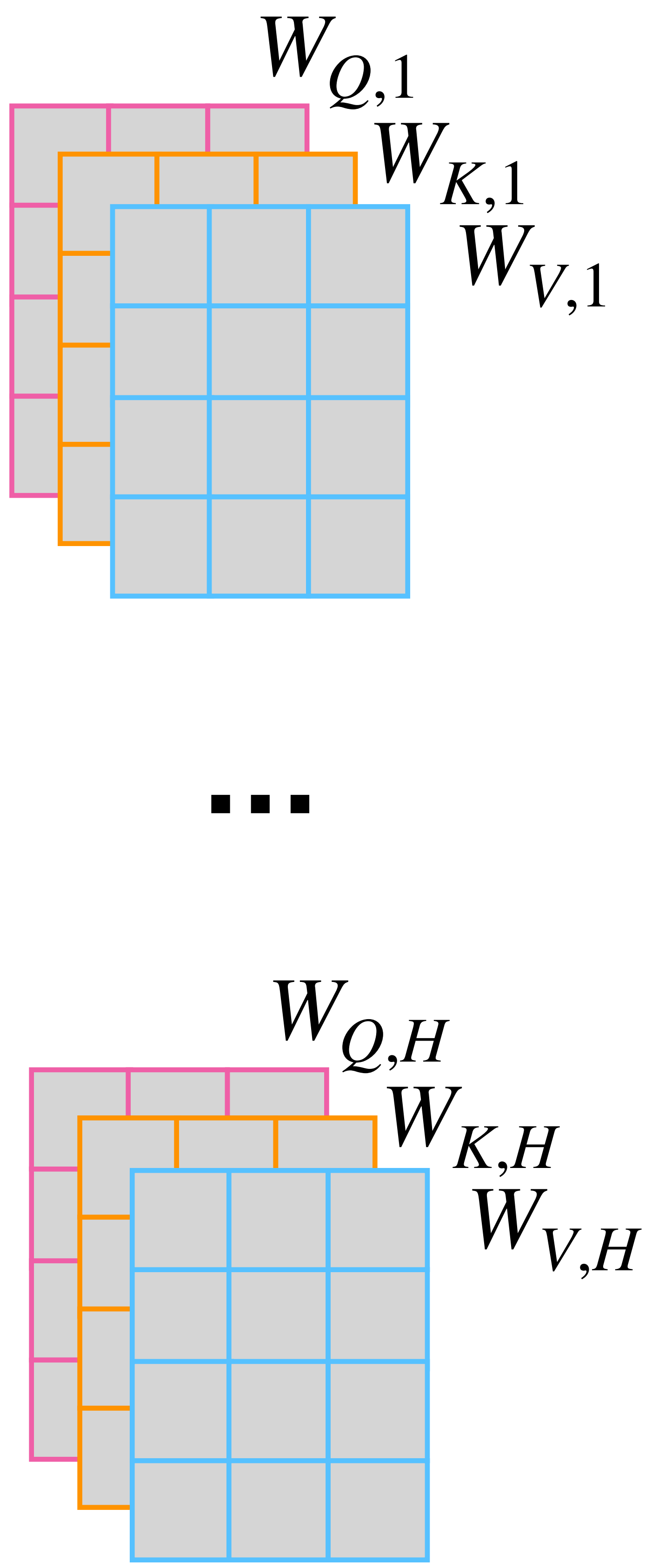


Multi-Head Self-Attention: recap

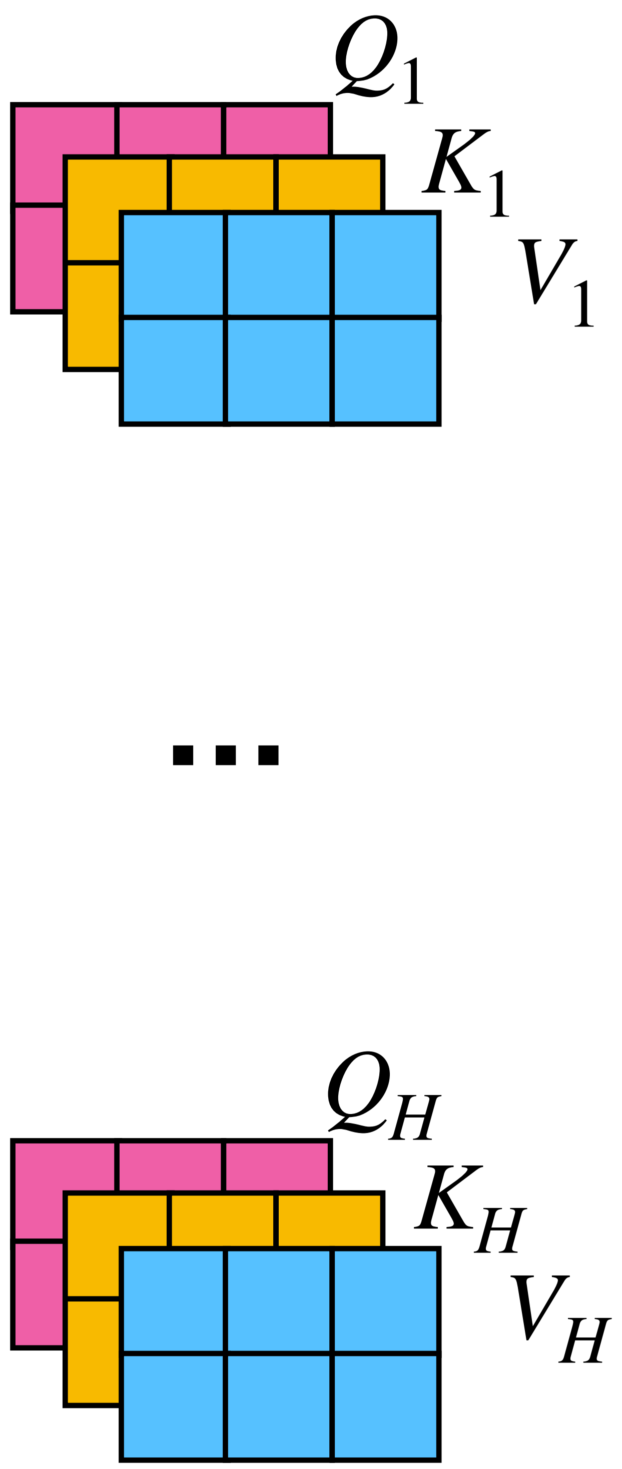
1) Input



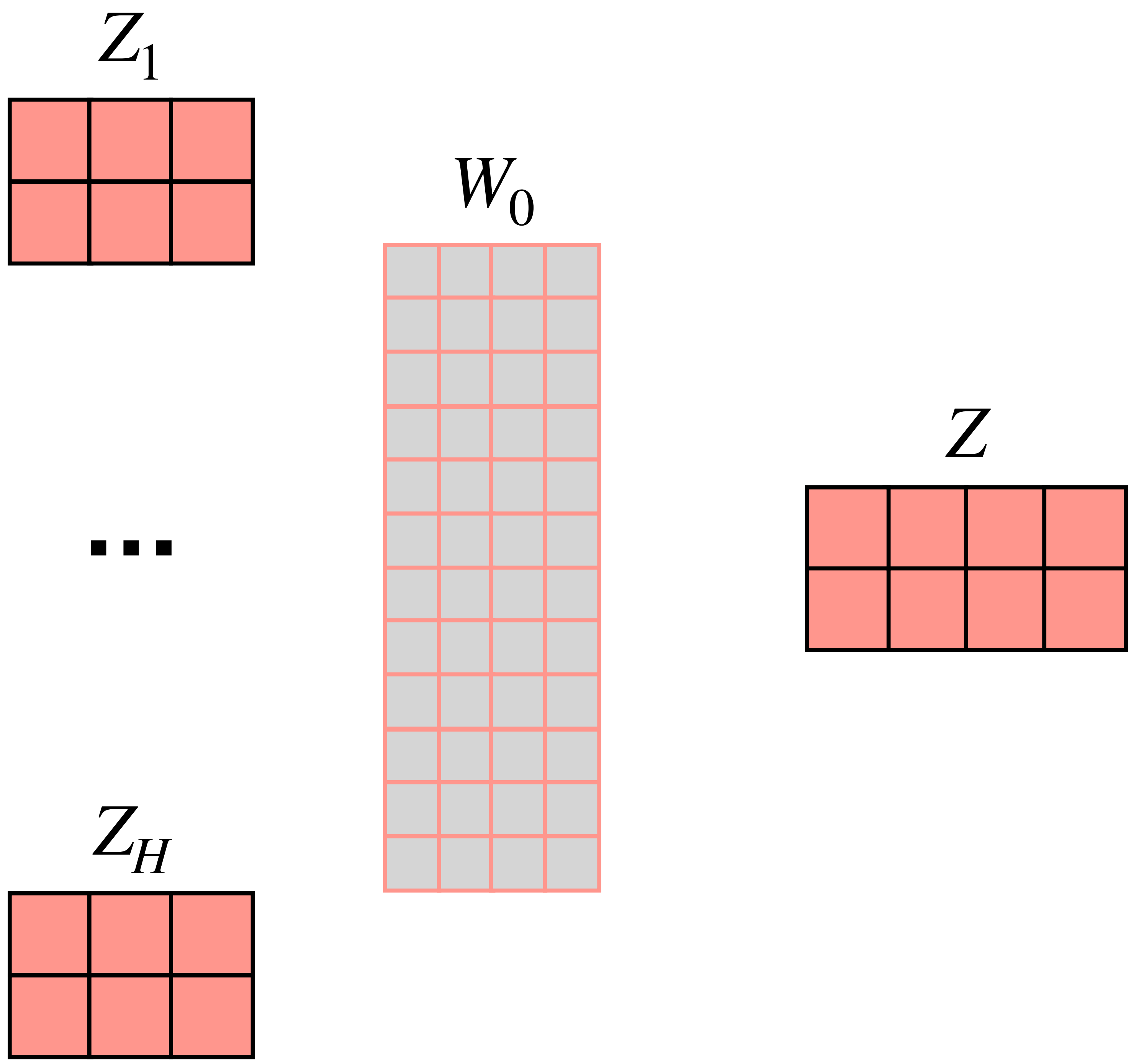
2) Split into H heads
We multiply X by weight matrices



3) Calculate attention using the resulting Q_h, K_h, V_h matrices



4) Concatenate the resulting matrices Z_h and multiply by W_0 to obtain the final output Z of the self-attention layer



Illustrations borrowed from The Illustrated Transformer

Positional information

Attention does not account for the order of input

For a permutation matrix $R \in \{0,1\}^{T\times T}$ we have:

$$Z_R = \text{softmax}\left(\frac{RXW_QW_K^\top X^\top R^\top}{\sqrt{D_K}}\right)RXW_V$$

Permute every X in original formula

$$= R\text{softmax}\left(\frac{XW_QW_K^\top X^\top R^\top}{\sqrt{D_K}}\right)RXW_V$$

Since softmax is computed row-wise

$$= R\text{softmax}\left(\frac{XW_QW_K^\top X^\top}{\sqrt{D_K}}\right)R^\top RXW_V$$

Reordering the terms in the softmax sum does not affect the output

$$= RPR^{-1}RXW_V$$

For a permutation matrix: transpose=inverse

$$= RPXW_V$$

Which is equivalent to a permutation of the original output $Z = PV$

Positional Information in Transformers

- **In practice, the input order matters:**
  "She prefers cats to dogs" $\neq$ "She prefers dogs to cats"

- **Solution**: incorporate a positional encoding in the network which is a function from the position to a feature vector pos : $\{1,\ldots,T\} \rightarrow \mathbb{R}^D$

- **The most basic choice** is to add a positional embedding W_{pos} corresponding to each token's position t to the input embedding. $W_{\text{pos}} \in \mathbb{R}^{T \times D}$ is learned via backpropagation along with the other parameters:

$$X = \begin{bmatrix} \mathbf{e}_{i_1} \\ \vdots \\ \mathbf{e}_{i_T} \end{bmatrix} W_{\text{emb}} + \begin{bmatrix} \mathbf{e}_1 \\ \vdots \\ \mathbf{e}_T \end{bmatrix} W_{\text{pos}}$$

- Numerous hand-crafted positional encodings exist (active area of research!)

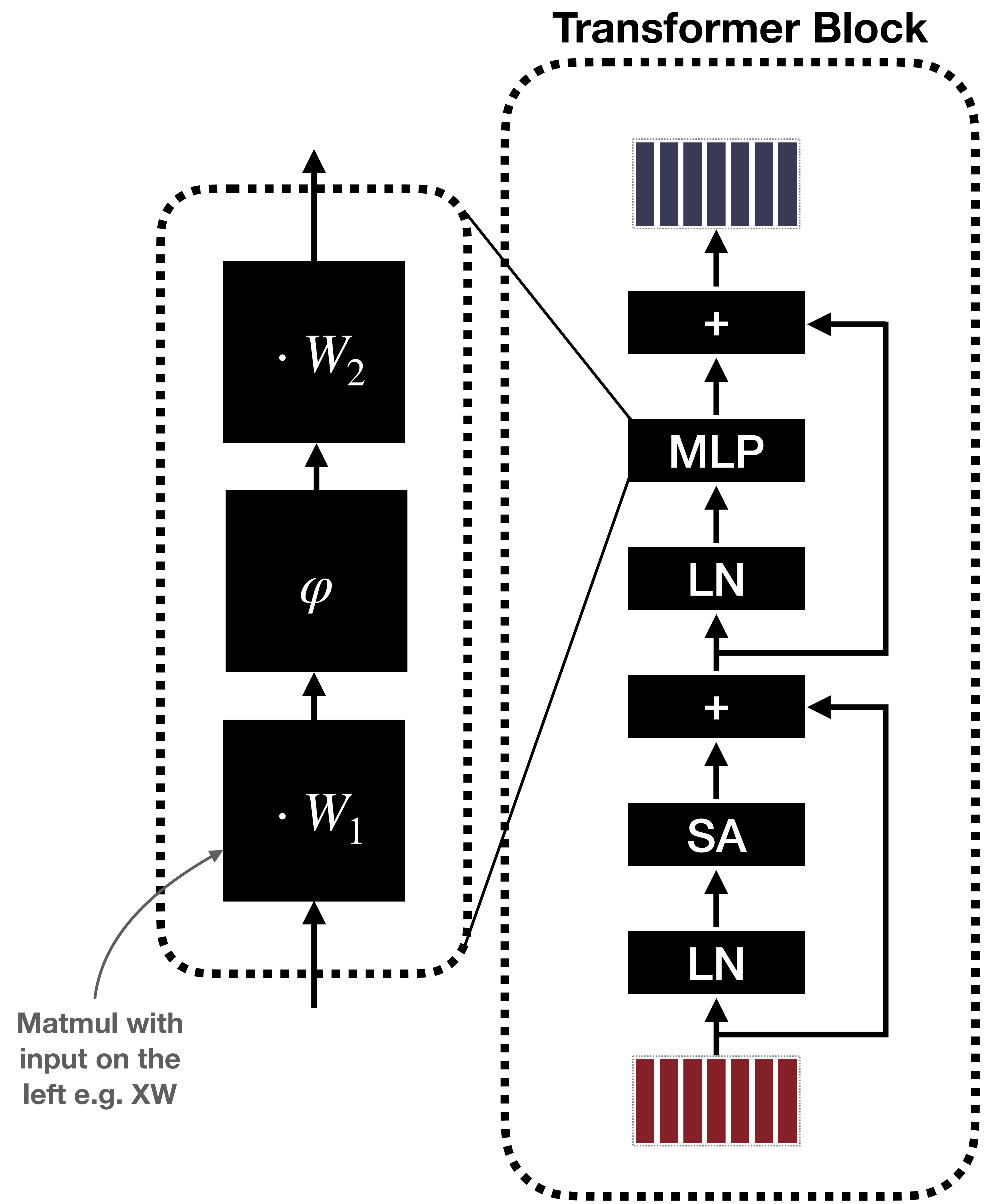
MLP

Mixing Information within Tokens

- **MLP** mixes information within each token
- Apply the same transformation to each token independently:

$$MLP(X) = \varphi(XW_1)W_2$$

- Matrices $W_1, W_2 \in \mathbb{R}^{D \times D}$ learned via backprop
- Non-linearity φ in between (e.g., ReLU or GeLU)
- The model may also include learned bias terms



Mixing Information within Tokens

- **MLP** mixes information within each token

The same MLP is applied to each token:

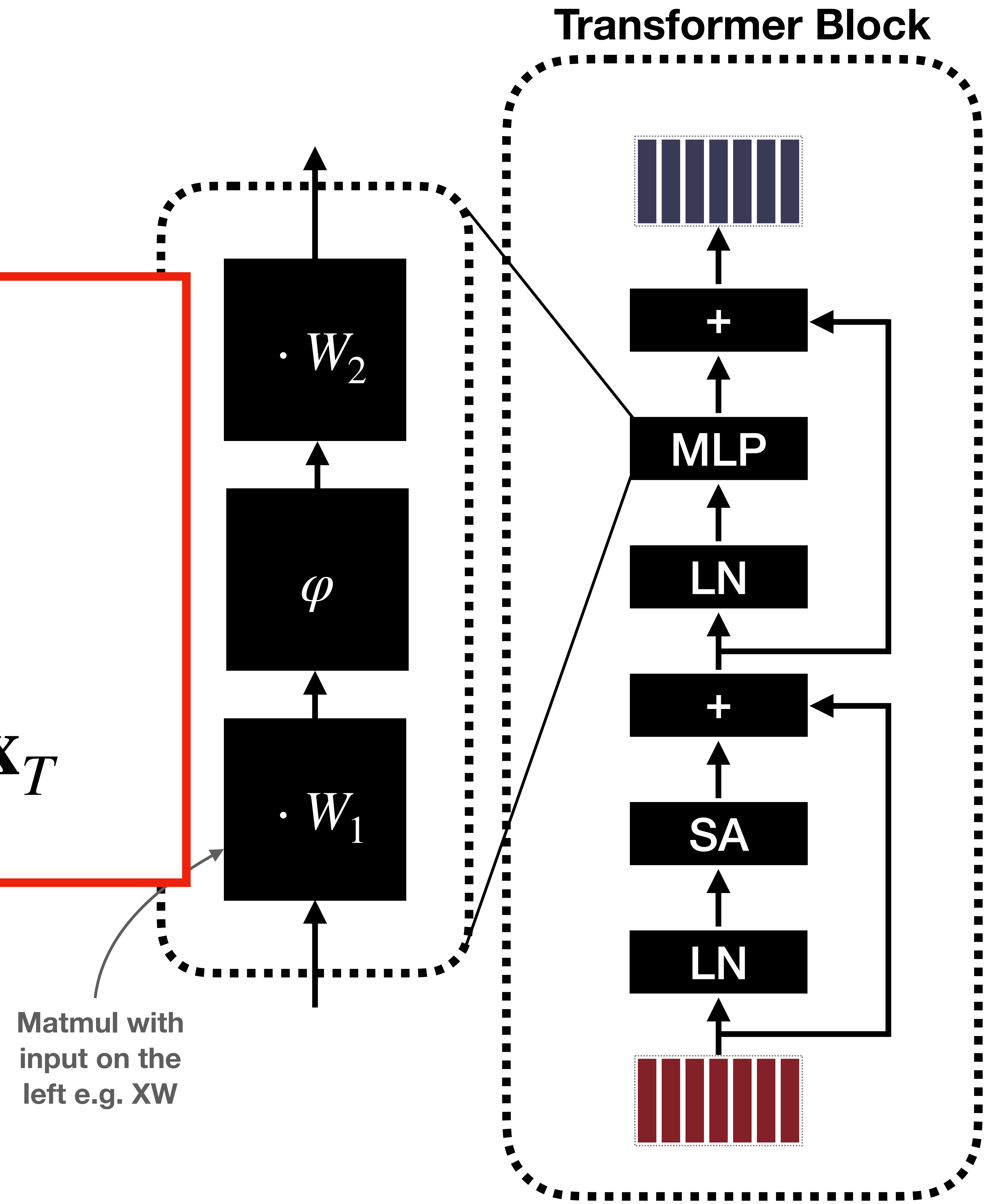
$$MLP(X) = \varphi(XW_1)W_2$$

$$\Longleftrightarrow$$

$$MLP(\mathbf{x}_t) = \varphi(\mathbf{x}_t W_1)W_2, \text{ for each token } \mathbf{x}_1, \ldots, \mathbf{x}_T$$

- Non-linearity φ in between (e.g., ReLU or GeLU)

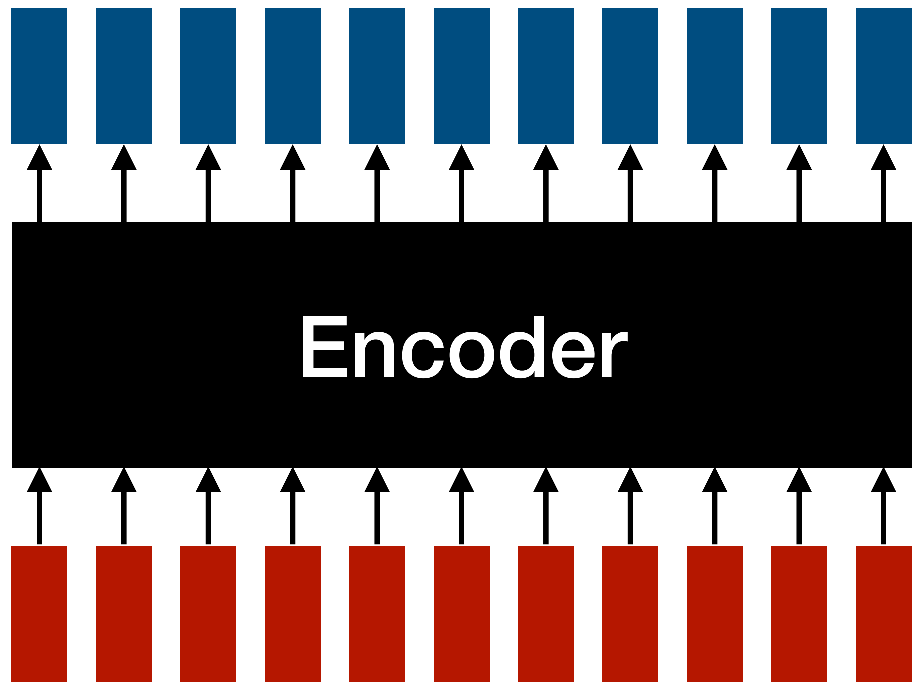
- The model may also include learned bias terms

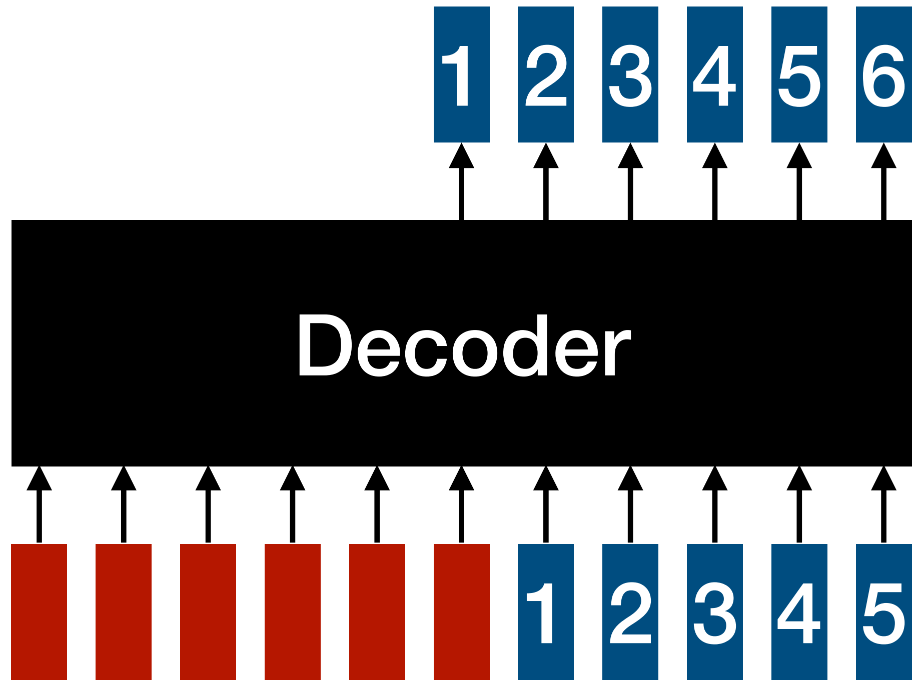


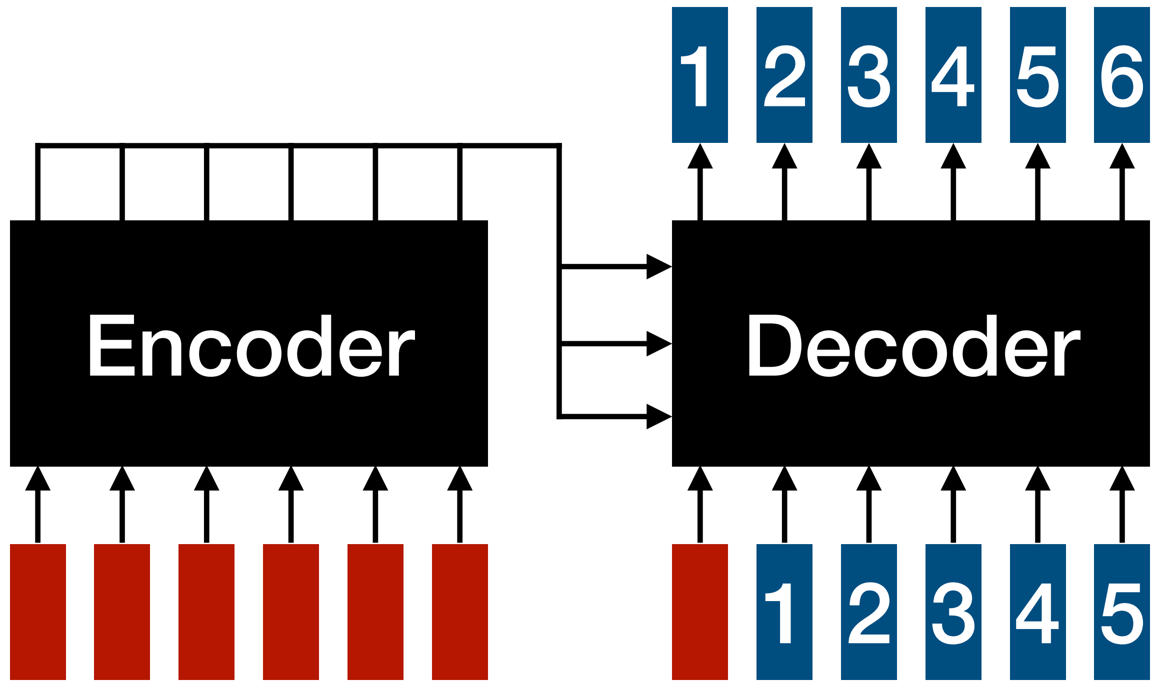
Big picture and takeaways

The transformer architecture can be used in different ways

- **Encoders** (e.g., element-level tagging, sequence-level classification):
  - They produce a fixed output size and process all inputs simultaneously
- **Decoders** (e.g., ChatGPT):
  - **Auto-regressively sample** the next token as $\mathbf{x}_{t+1} \sim \text{softmax}(f(\mathbf{x}_1, \ldots, \mathbf{x}_t))$ and append it to the input as a **new token**; repeat
  - Capable of generating responses of arbitrary length
- **Encoder-decoder** (e.g., translation):
  - First encode the whole input (e.g., in one language) and then decode token by token (e.g., in a different language)







Transformers: big picture

- **“Tokenize everything”:** Everything can be seen as tokens, hence transformers are applicable across any modality
- **CNNs can also be used for text processing, but transformers excel at capturing long-range dependencies** (as an example, the latest GPT-4 model can process up to 128k input tokens, equivalent to ~300 pages of text).
- **Self-attention scales quadratically with sequence length**, making it computationally expensive for large volumes of text or numerous patches—active area of research
- **However, self-attention is highly parallelizable**, which is advantageous for multi-GPU or multi-node training setups
- Transformers are now the **preferred method** for both text and vision applications
- **Emergent abilities at scale**: few-shot learning (aka in-context learning from a few example) and zero-short learning (e.g., you can ask ChatGPT any question without prior training on the task)

Recap

- **Transformers** iteratively map sequences to sequences using the self-attention mechanism
- The whole architecture is remarkably simple:
  - **Self-attention blocks** mix the information **between** tokens
  - **MLP blocks** mix the information **within** each token
- Transformers excel at modeling long-range dependencies
- Different architectures are possible (e.g., ChatGPT is decoder-only, but neural translation typically employs an encoder-decoder)
- Transformers have become a **universal architecture** for almost any type of data modality; they perform exceptionally well when given enough pretraining data

Additional Resources

If you want to learn more about attention and transformers:

- **The Illustrated Transformer**: https://jalammar.github.io/illustrated-transformer/ (a good step-by-step guide with detailed illustrations)

- **The blog of Lilian Weng (OpenAI)**: https://lilianweng.github.io/posts/2018-06-24-attention/ (from 2018 but covers well the history of the attention mechanism and its different versions)

- **CS231n: Deep Learning for Computer Vision (Stanford)**: http://cs231n.stanford.edu/slides/2023/lecture_9.pdf (more on positional encodings, masked self-attention, general attention, discussion of recurrent neural networks)

- **Minimal implementation of GPT-2**: https://github.com/karpathy/nanoGPT/ (some things are just clearer in code)