Transformers

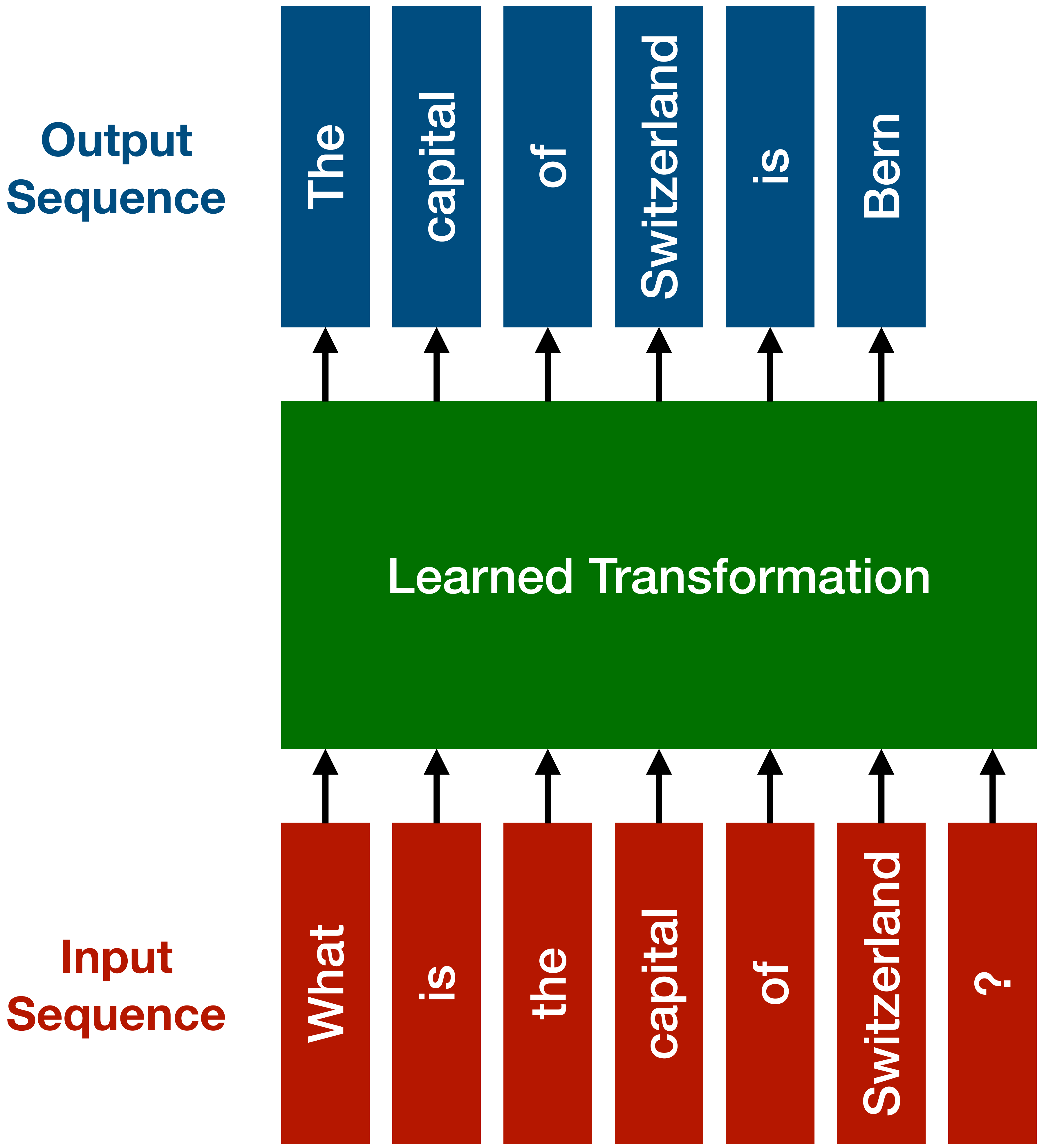
Machine Learning Course - CS-433
Nov 12, 2024
Nicolas Flammarion

EPFL

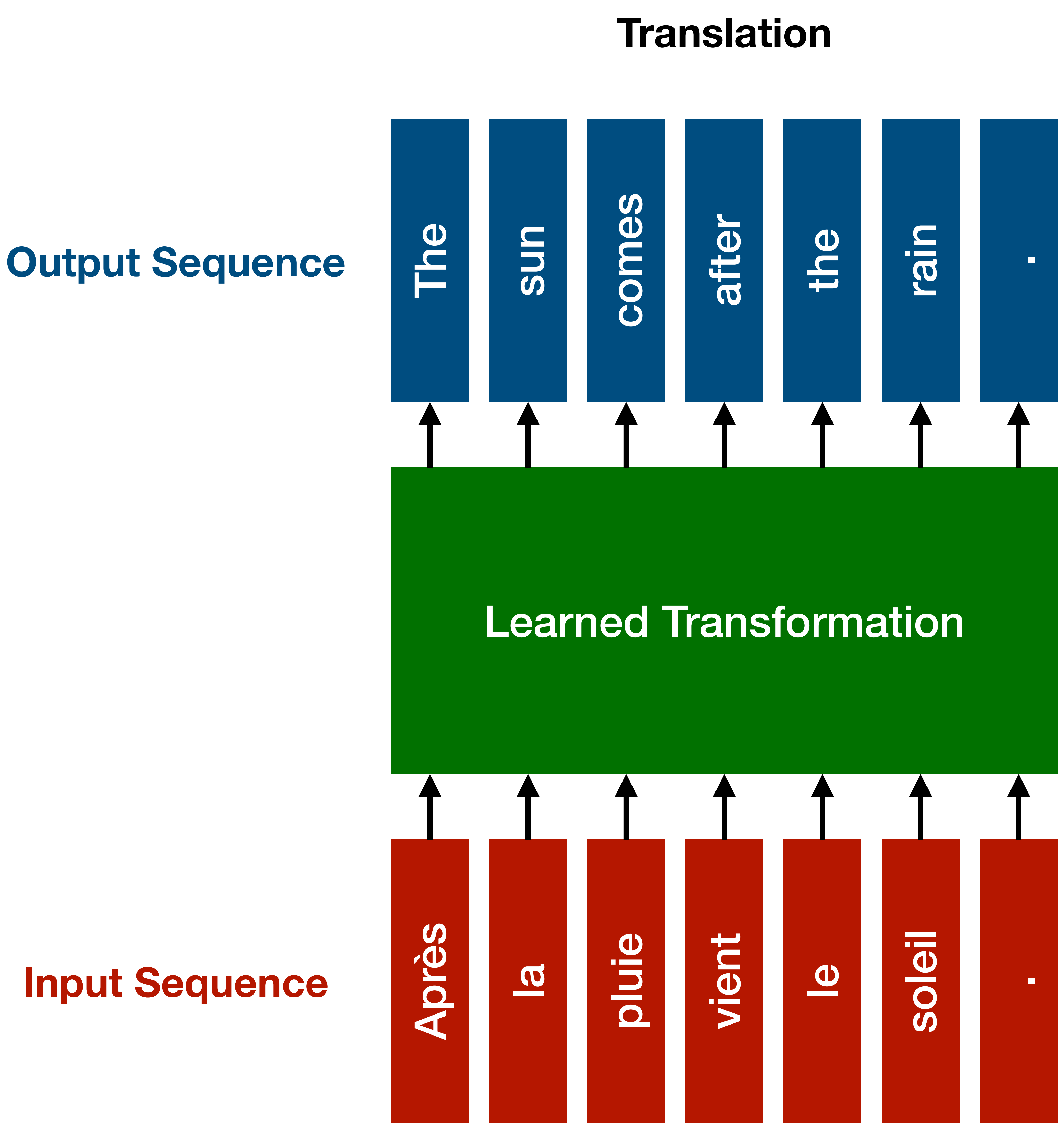
Sequence-to-Sequence Transformations

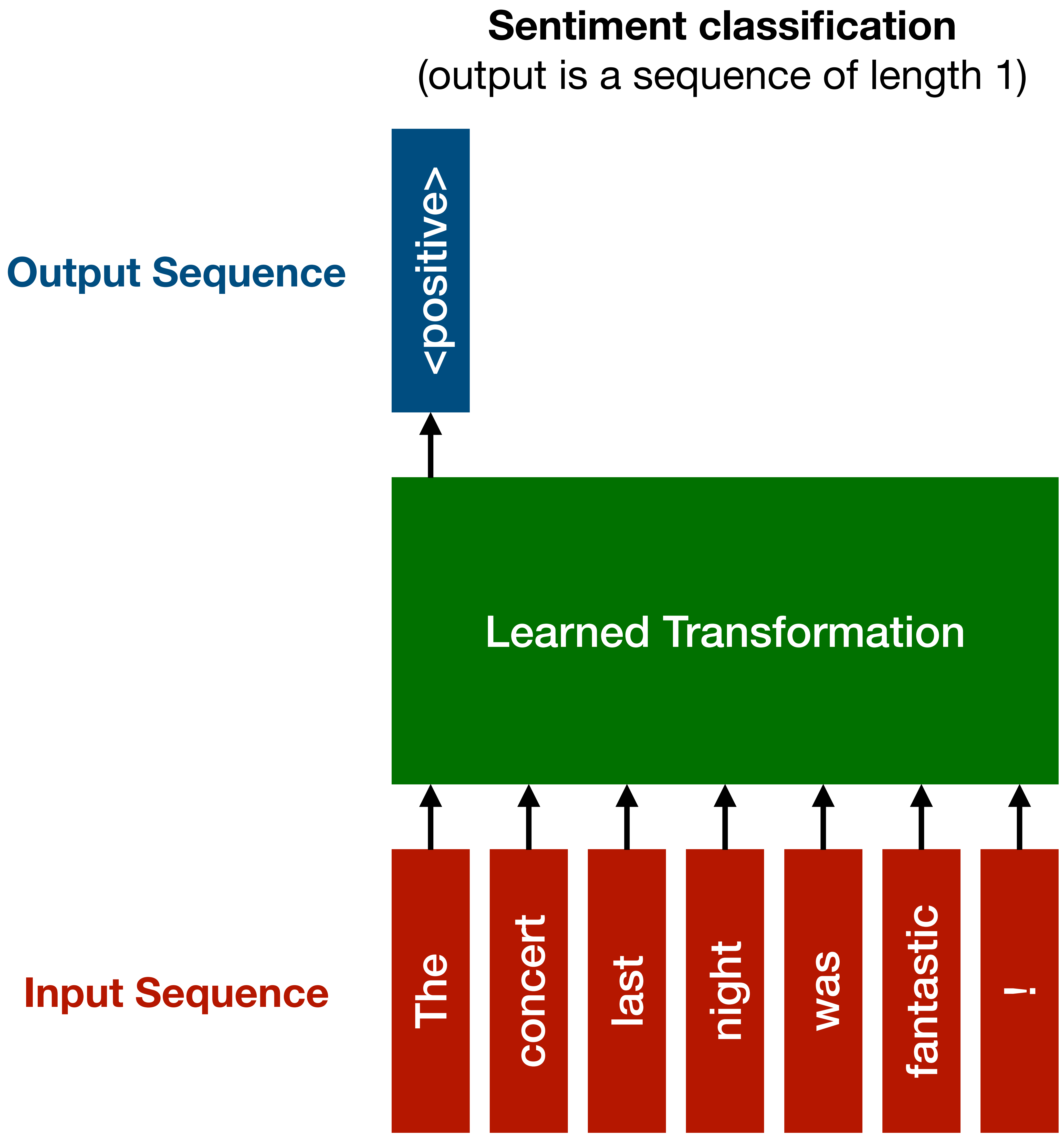
Sequence-to-Sequence Transformations

- Many interesting problems in ML can be expressed as **mapping one sequence to another**
- **Example**: chatbots like ChatGPT
  - Input: question (word sequence)
  - Output: answer (word sequence)
- Input and output sequences can represent various types of data: words, images, speech, proteins, time series, etc.

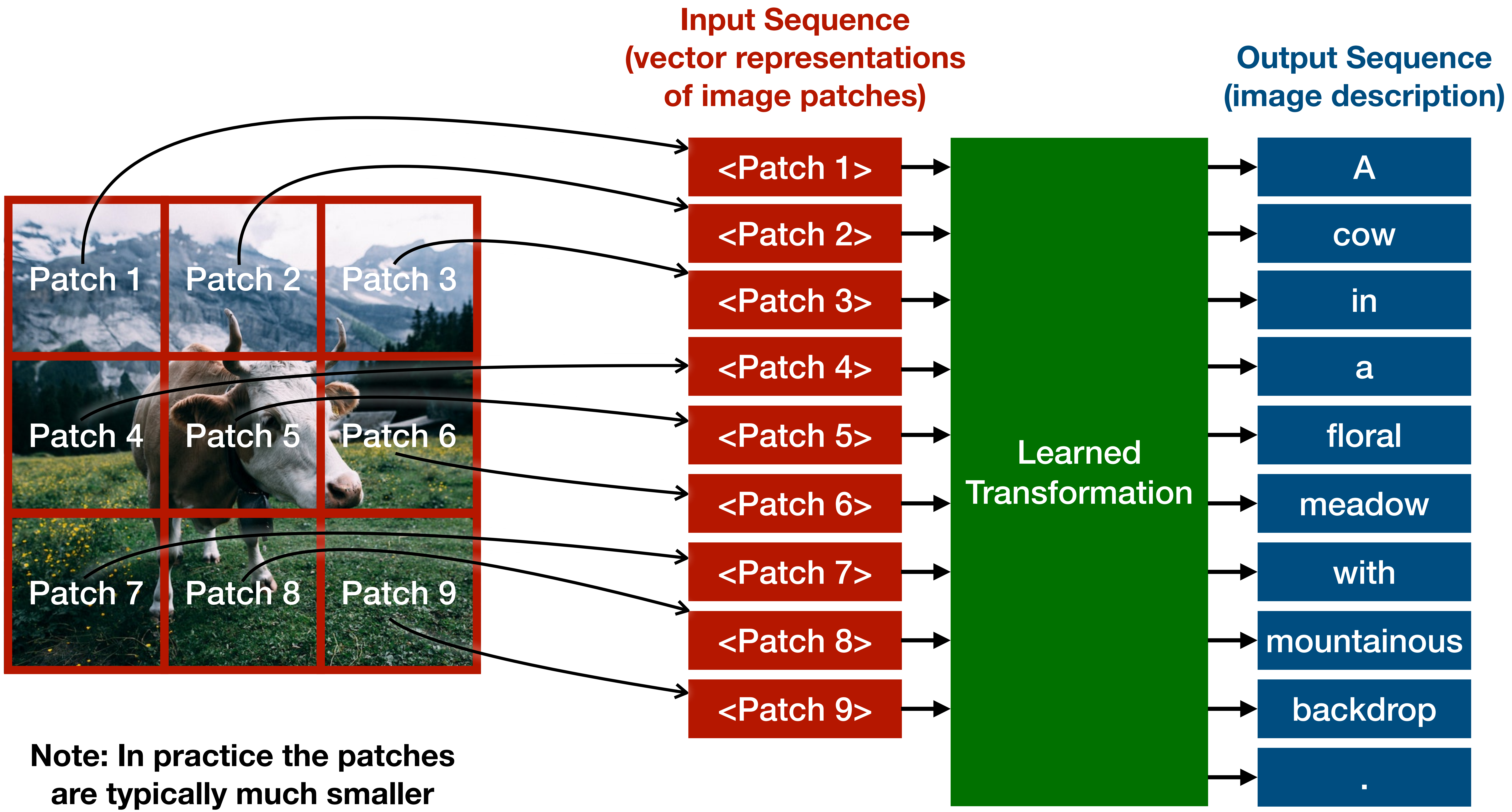


The sequence-to-sequence framework is very general



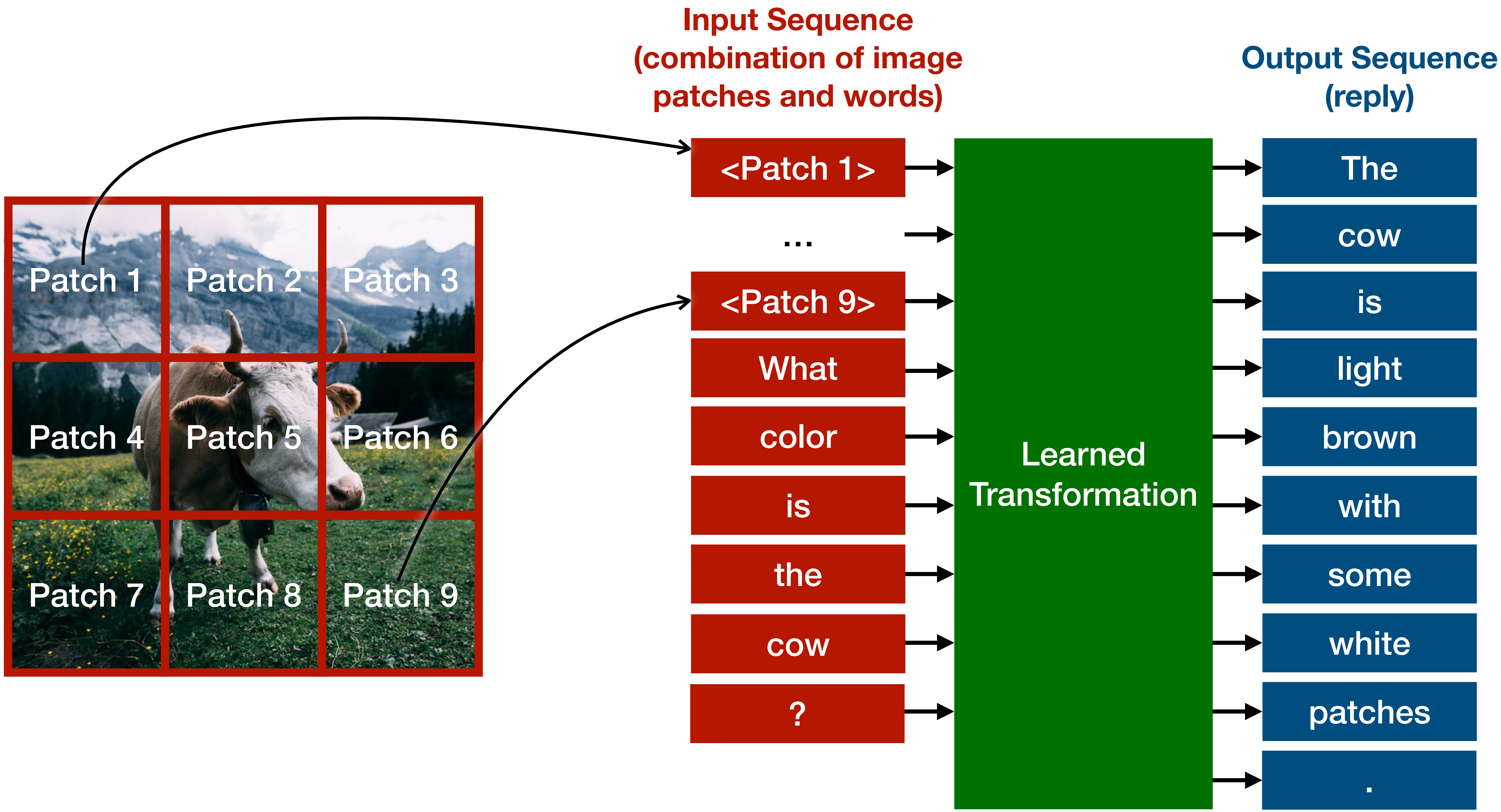


Images can also be represented as sequences



Note: In practice the patches are typically much smaller

Sequences can be multimodal (image + text)



Transformers

What is a Transformer?

$$\boxed{f : sequence \rightarrow sequence}$$

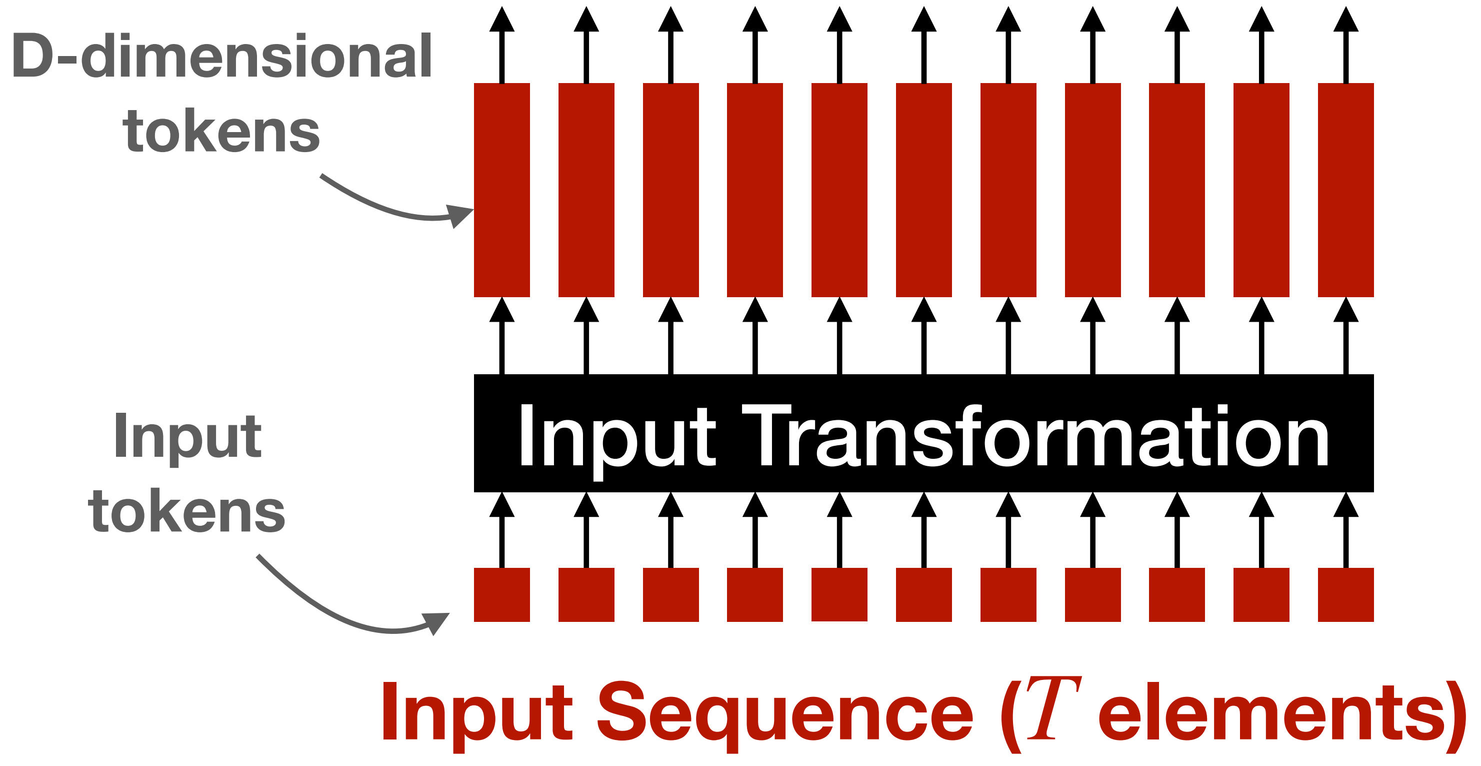
(using **self-attention**)

Transformer is a neural network f that iteratively transforms a sequence to another sequence and mixes the information between the sequence elements via **self-attention**

Overview of Transformer Architecture

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

- maps a one-hot word vector to a real-valued vector
- extracts an image patch and flattens it into a vector

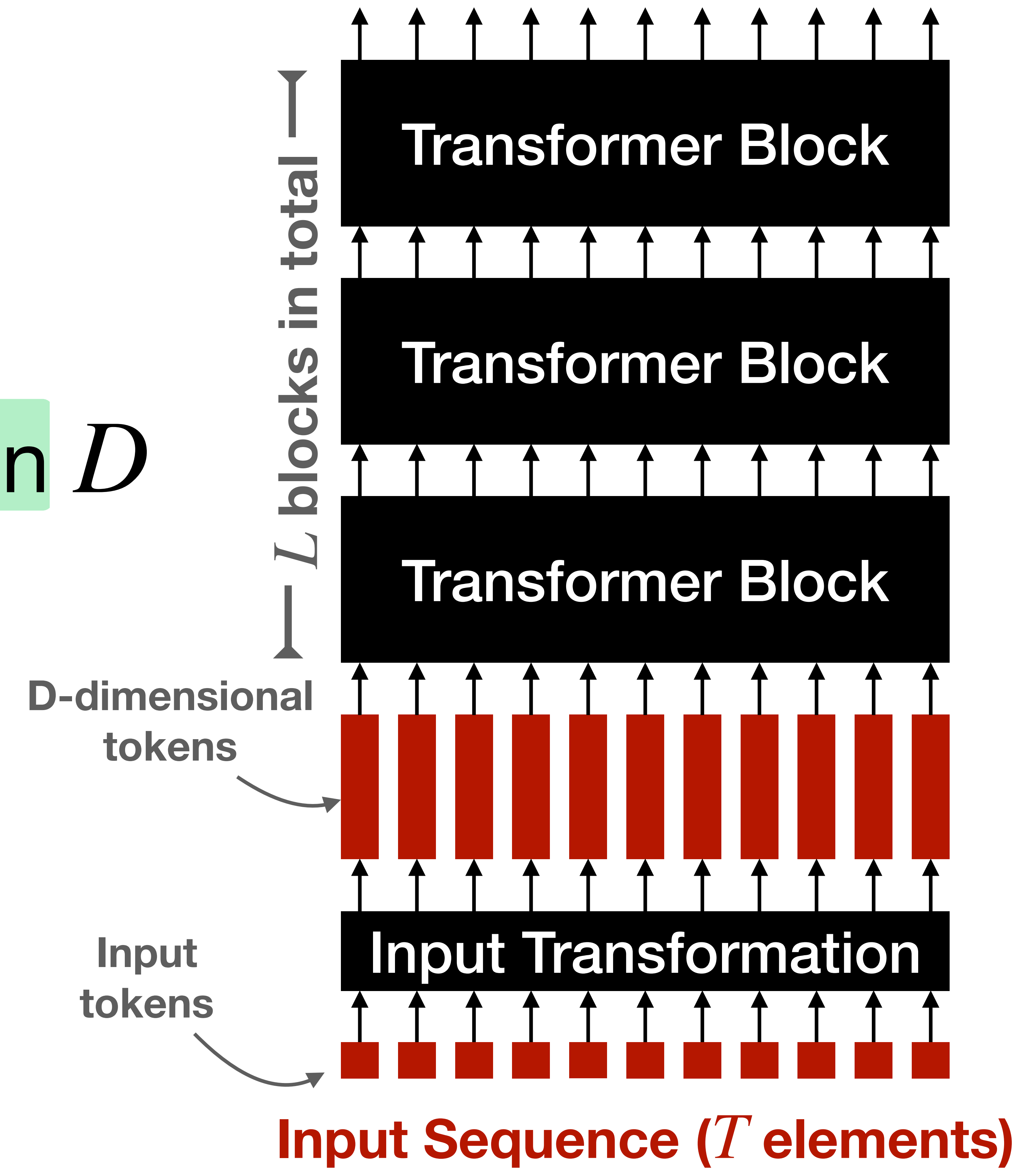


Overview of Transformer Architecture

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

- maps a one-hot word vector to a real-valued vector
- extracts an image patch and flattens it 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**



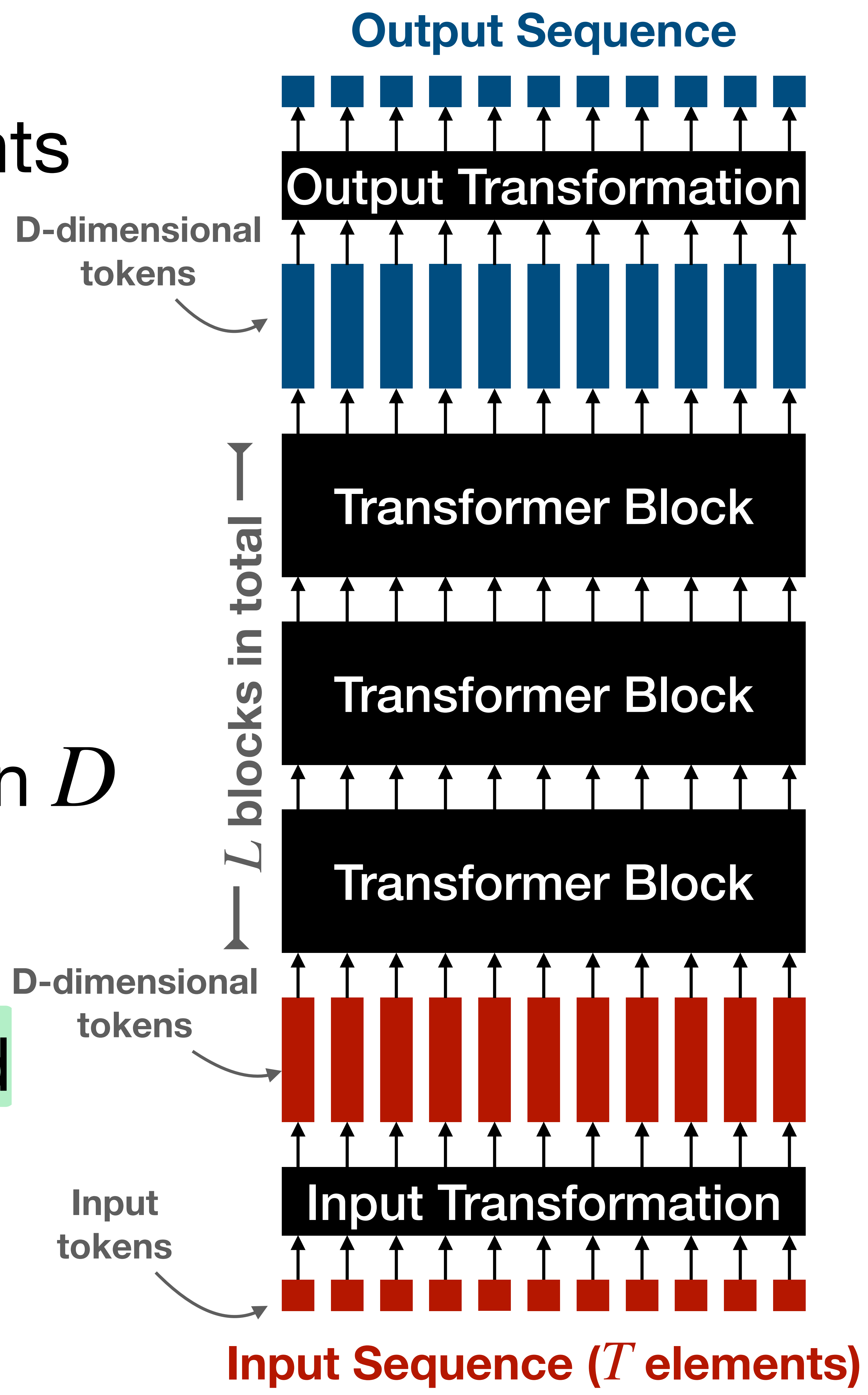
Overview of Transformer Architecture

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

- maps a one-hot word vector to a real-valued vector
- extracts an image patch and flattens it into a vector

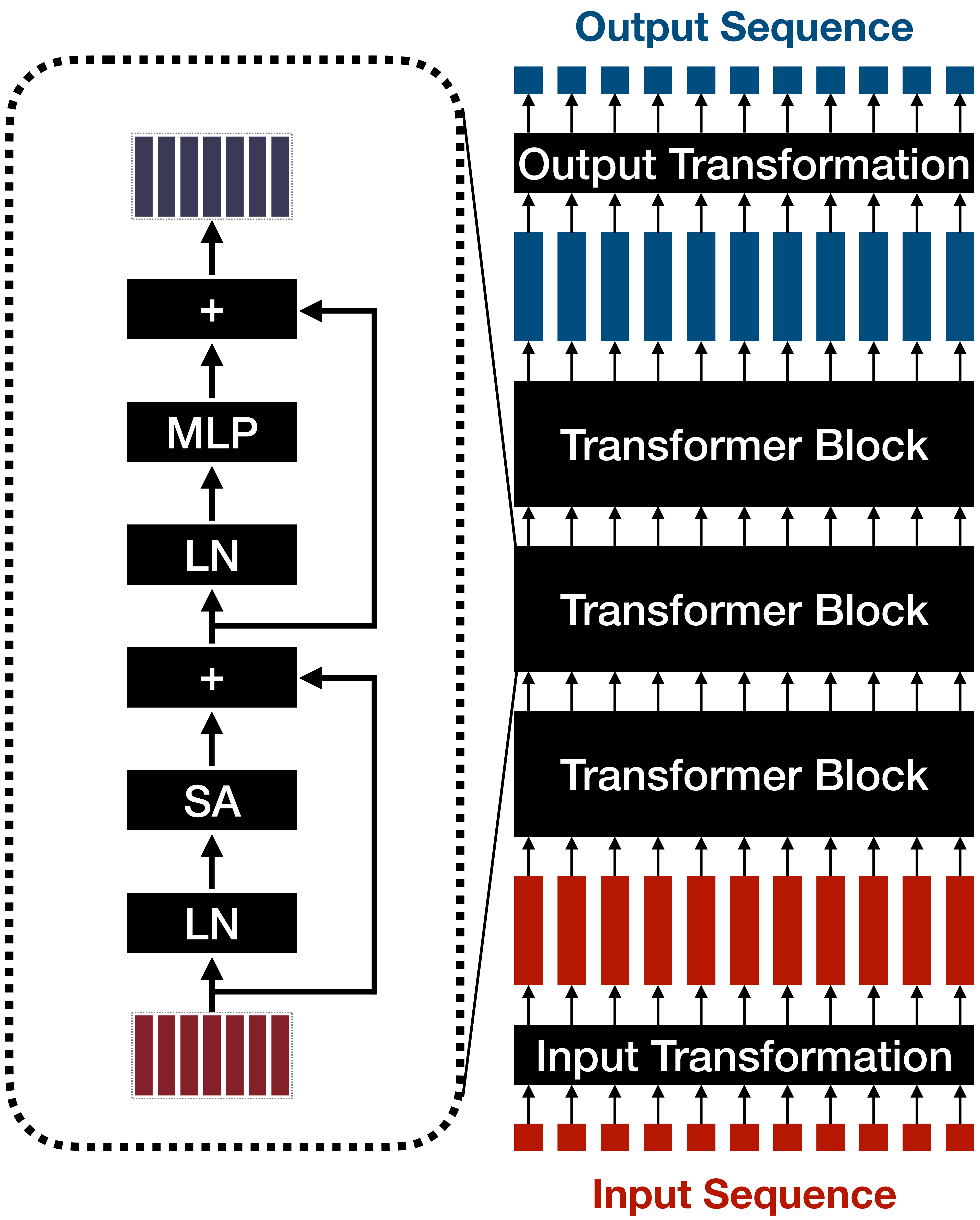
Transformer block: transforms a sequence of T vectors of dimension D into a new sequence of T vectors of dimension D using **self-attention** and **MLP sub-blocks**

Output transformation: converts the vectors to the desired output format (e.g., single-element sequence for classification, multiple-element sequence of words)



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: "<User:>Transformers are awesome!"`
- `Tokens: [<User token>, "Trans", "form", "ers_", "are_", "awe", "some", "!"]`
- `Token IDs: [0, 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:` $[w_0, w_{5124}, w_{1029}, w_{645}, w_{3001}, w_{6931}, w_{7330}, w_{10}]$

➡The whole input sequence of T tokens leads to an input matrix $X = \begin{bmatrix} w_0^\top \\ w_{5124}^\top \\ \vdots \\ w_{10}^\top \end{bmatrix} \in \mathbb{R}^{T \times D}$

Text Token Embeddings - Learning

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

- This can be seen as a matrix multiplication:

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

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

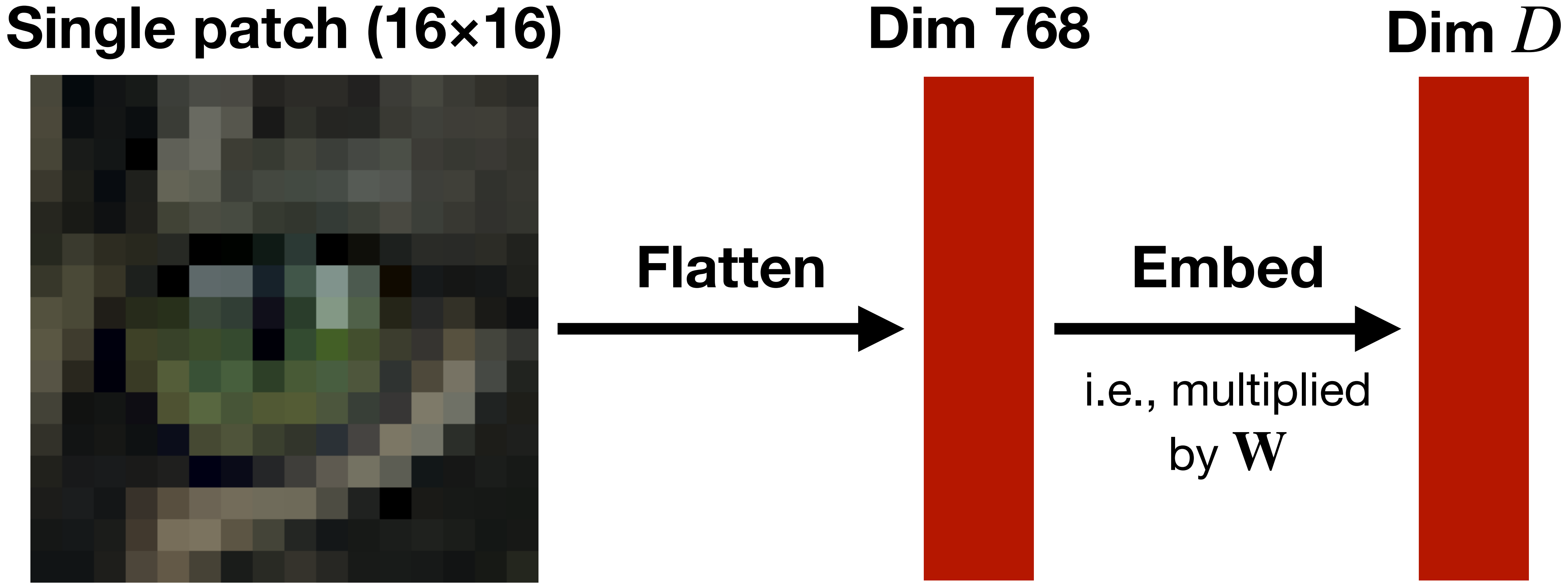
Image Patch Embeddings

- 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 $\mathbf{W}_{\text{emb}} \in \mathbb{R}^{D \times 768}$ which is shared for all inputs
- Learn $\mathbf{W}_{\text{emb}}$ through backpropagation, along with all other transformer parameters
- The whole input sequence of T embedded patches leads to an input matrix $X \in \mathbb{R}^{T \times D}$

Original (128×128)

Patches (16×16 each)

Single patch (16×16)



Self-attention

What is Self-attention?

$$A : tokens \rightarrow tokens$$

(using a **weighted average**)

Reminder: a token is simply a real-valued vector

Self-attention is a function that transforms a sequence of tokens to a new sequence of tokens using a **learned input-dependent weighted average**

Self-Attention

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**: $V = XW_V \in \mathbb{R}^{T \times D_V}$

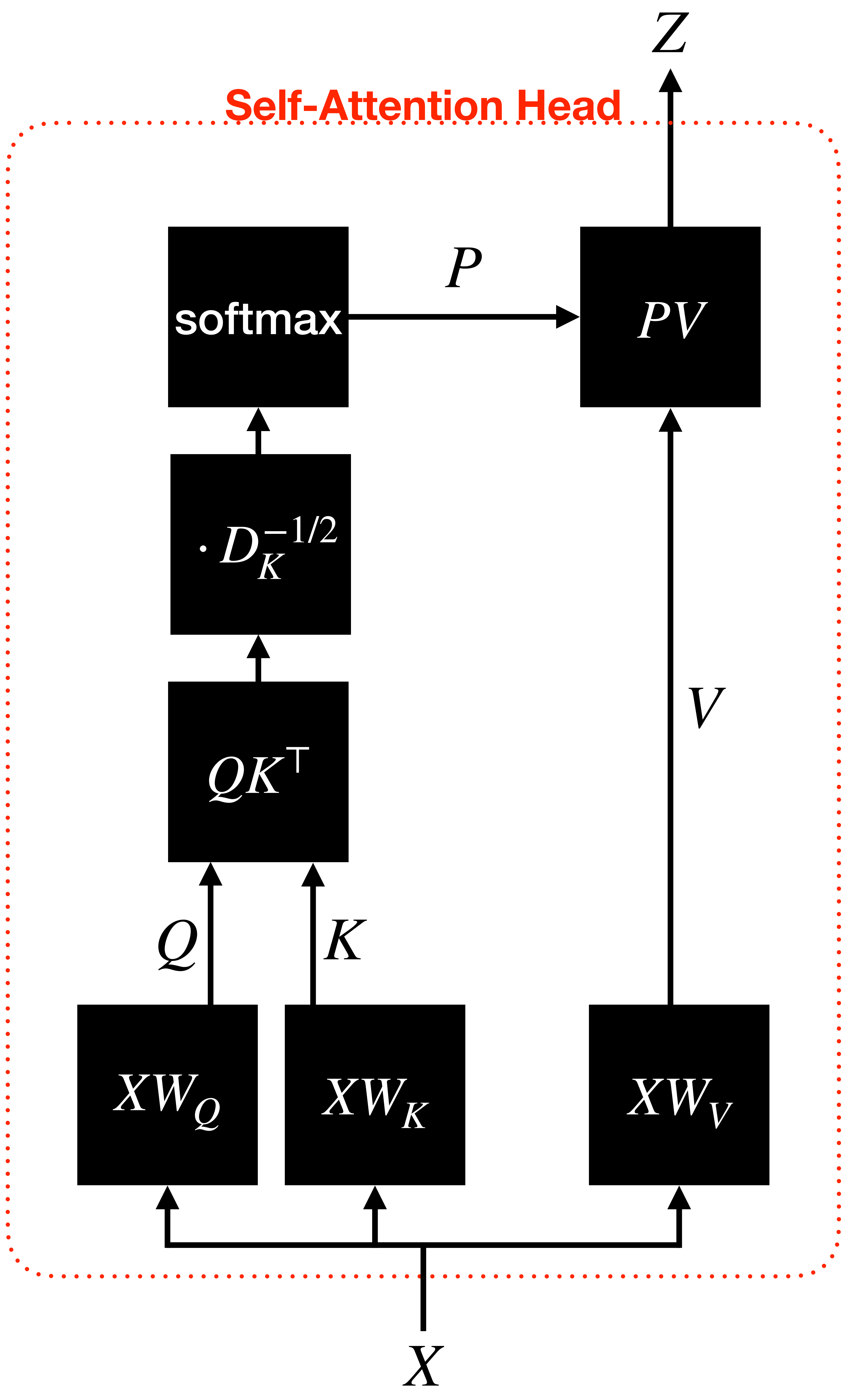
➡ $W_K, W_Q \in \mathbb{R}^{T \times D_K}$, $W_V \in \mathbb{R}^{T \times D_V}$ are parameters

The output of self-attention is then given by:

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

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

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



Attention as a Weighted Average

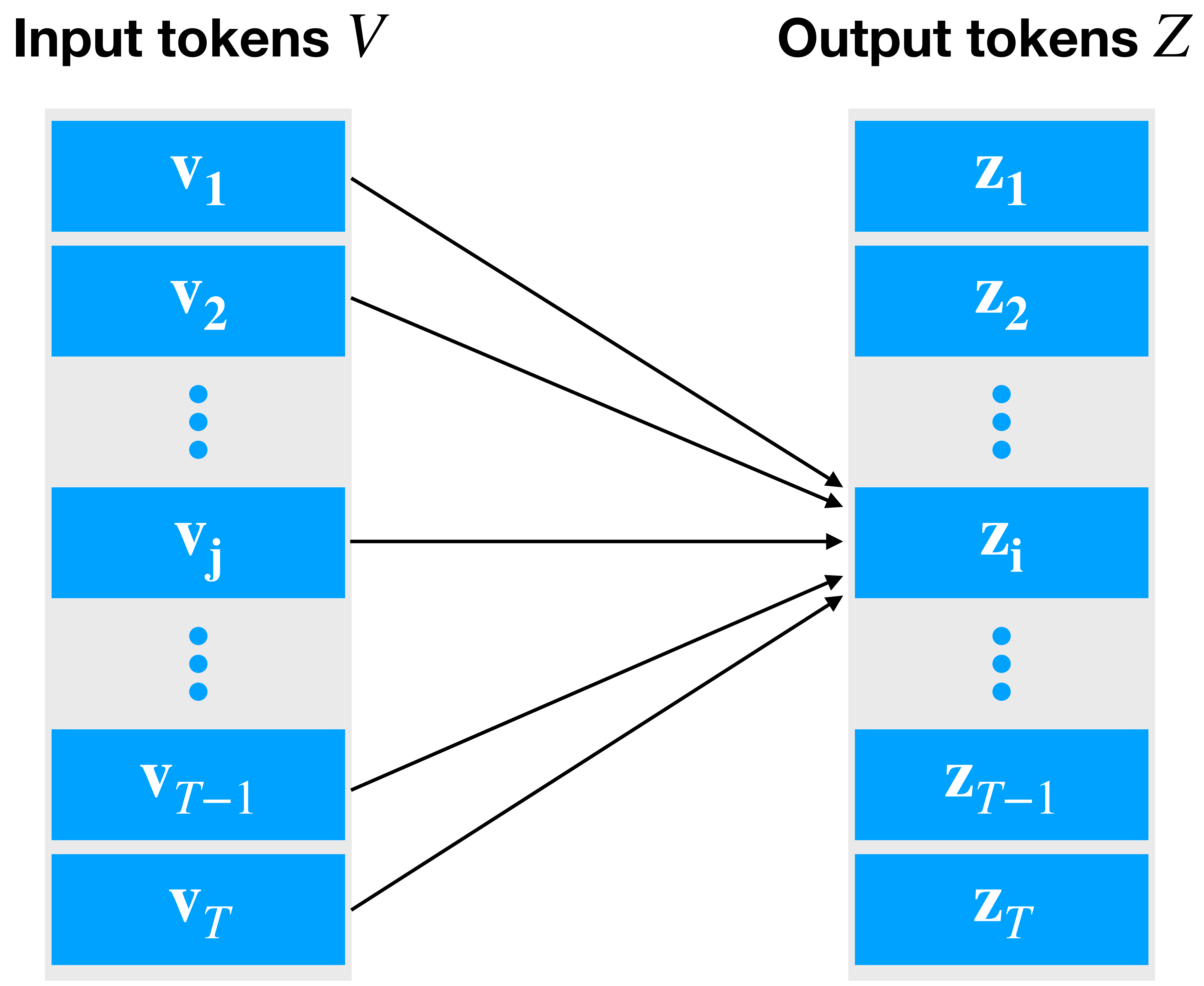
- T **input and output tokens:** $V \in \mathbb{R}^{T \times D_V}, Z \in \mathbb{R}^{T \times D}$
- Outputs are a **weighted average** of the inputs:

$$\mathbf{z_i} = \sum_{j=1}^{T} p_{i,j} \mathbf{v_j} \quad \text{or in matrix form} \quad Z = PV$$

- 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 $\mathbf{z}_j$



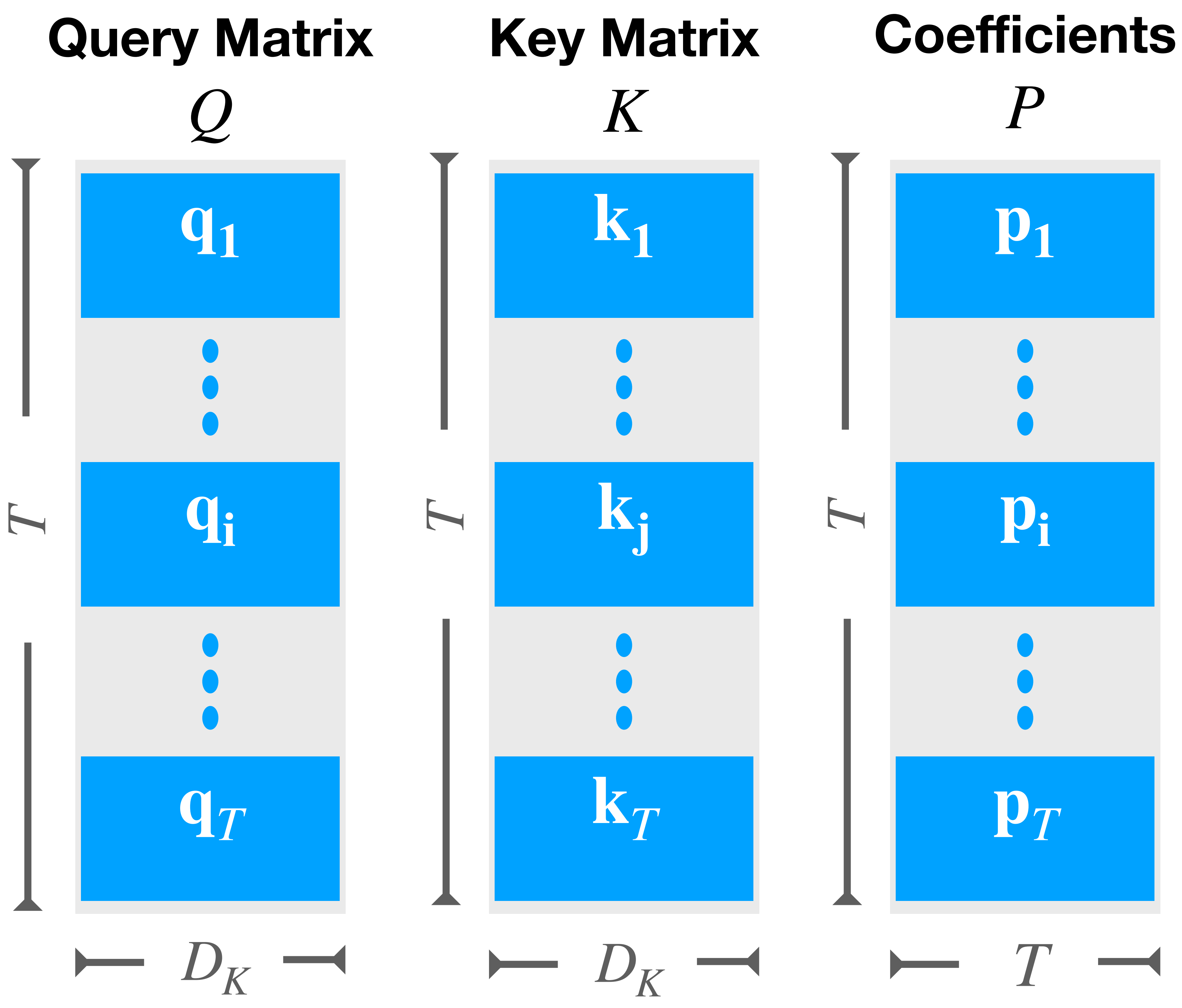
The Weighting Coefficients P

- **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 the temperature by $\sqrt{D_K}$) to obtain a probability distribution
- This can be expressed as:

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

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

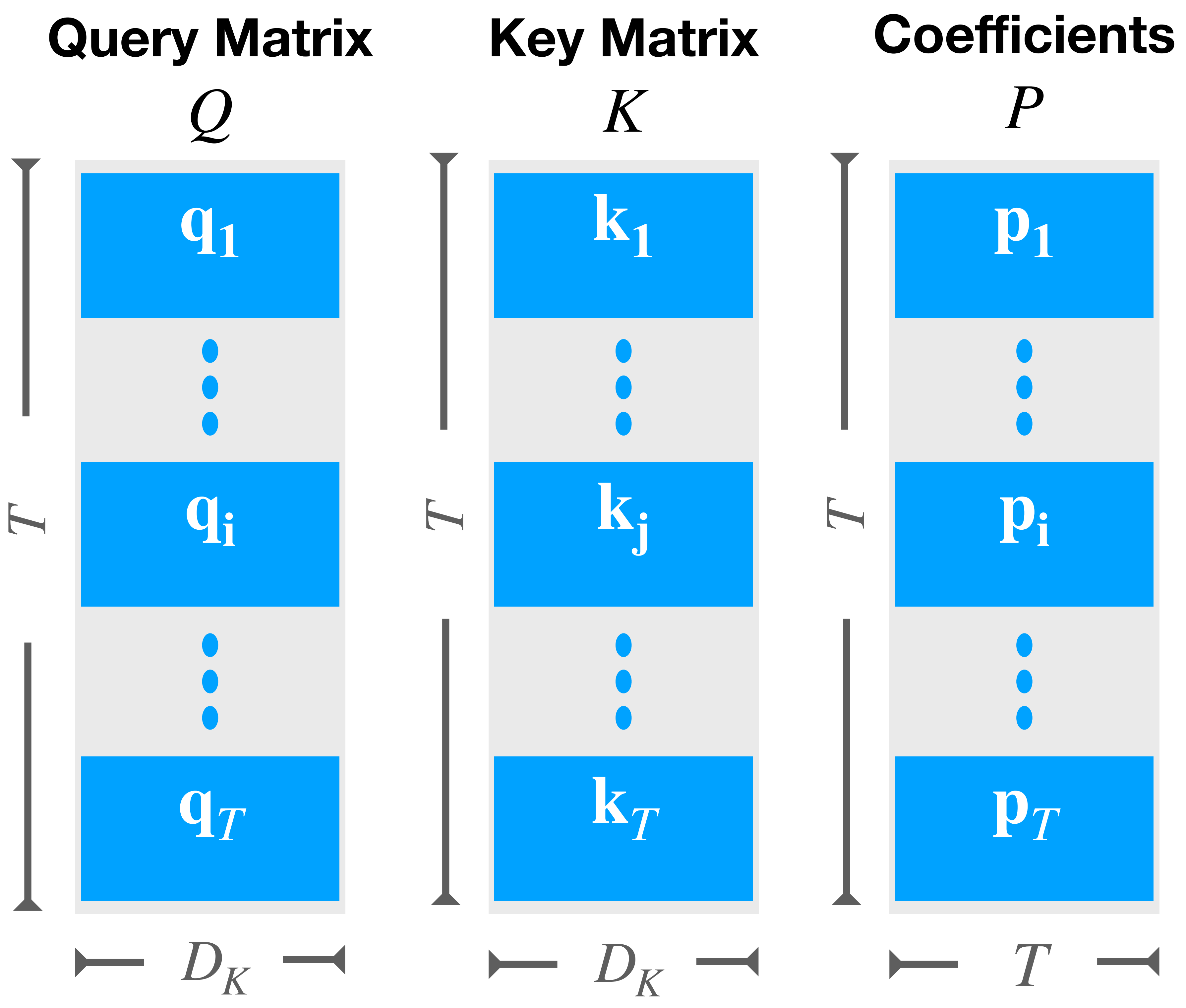
The softmax is applied on each row *independently*



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

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:

$O(T \times T)$

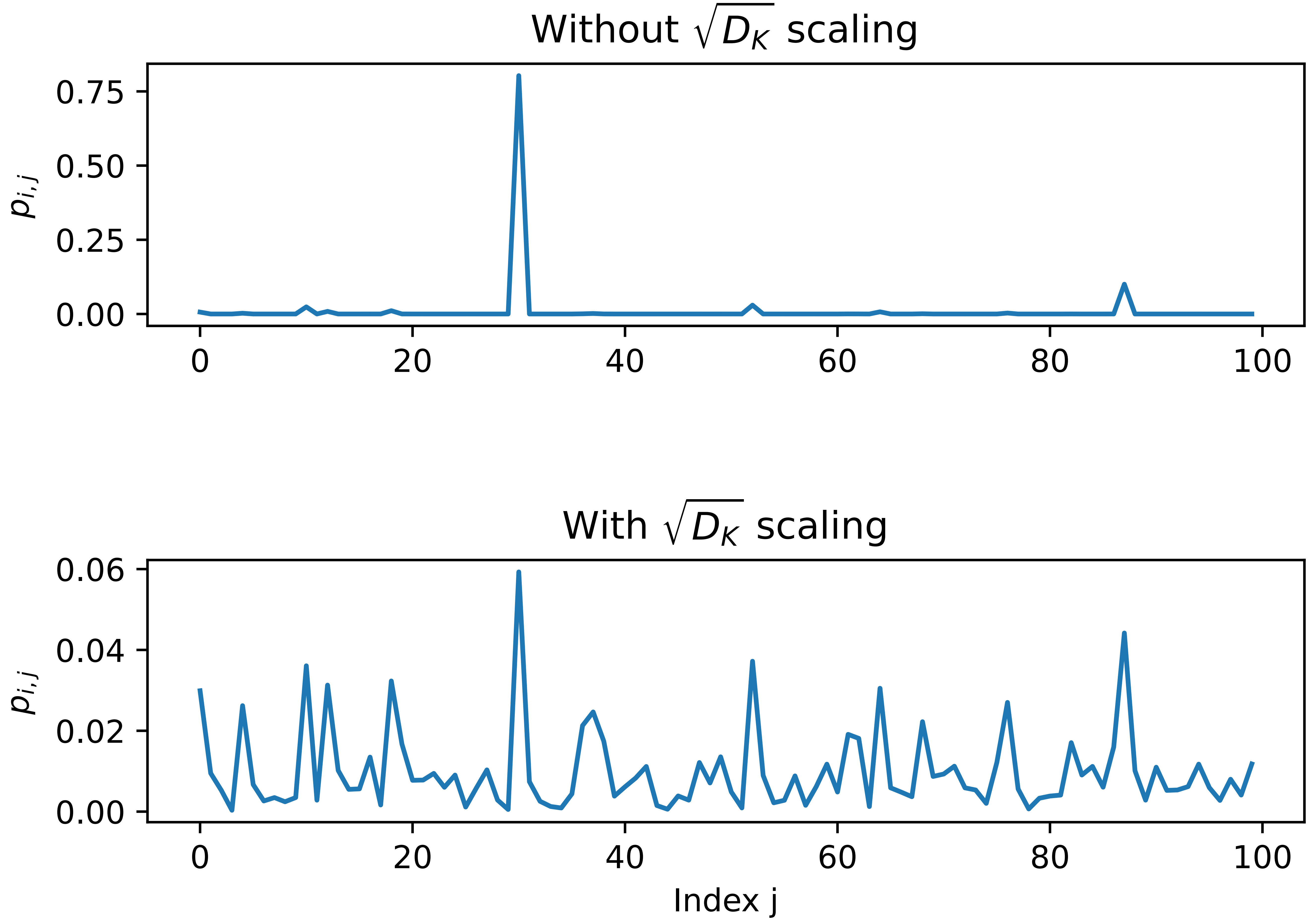
Why Use the $1/\sqrt{D_K}$ Scaling?

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

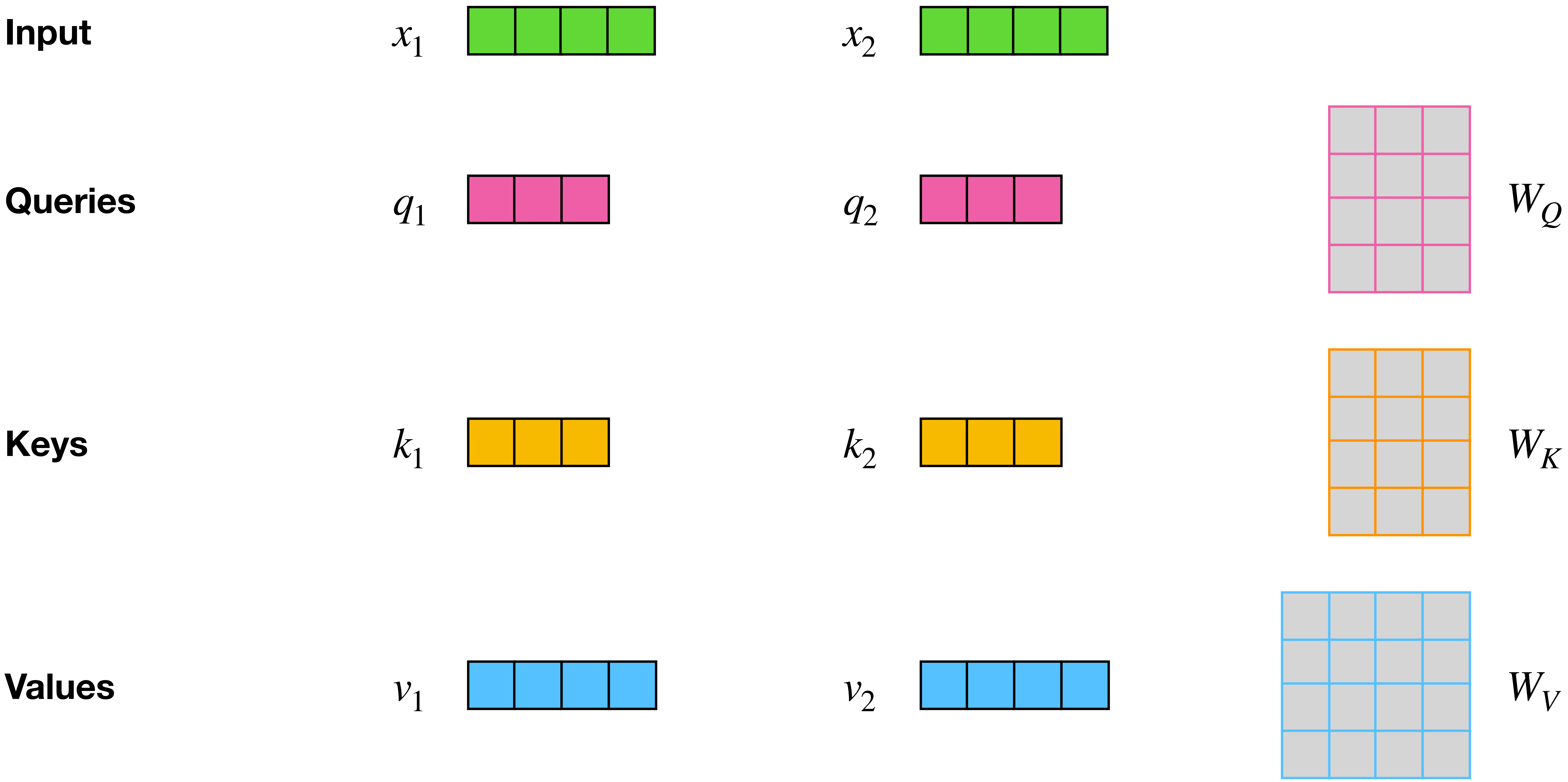
- **Without scaling**: sharp distribution of the attention weights $p_{i,j}$ at random initialization

- 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

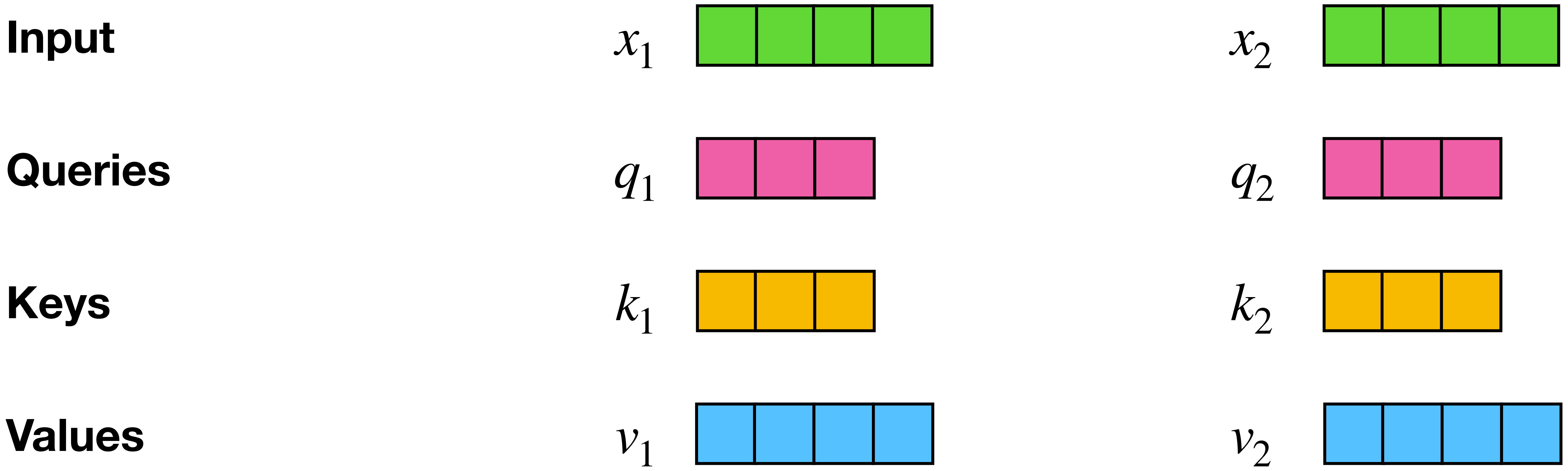


$$\boxed{\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \end{aligned}}$$

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

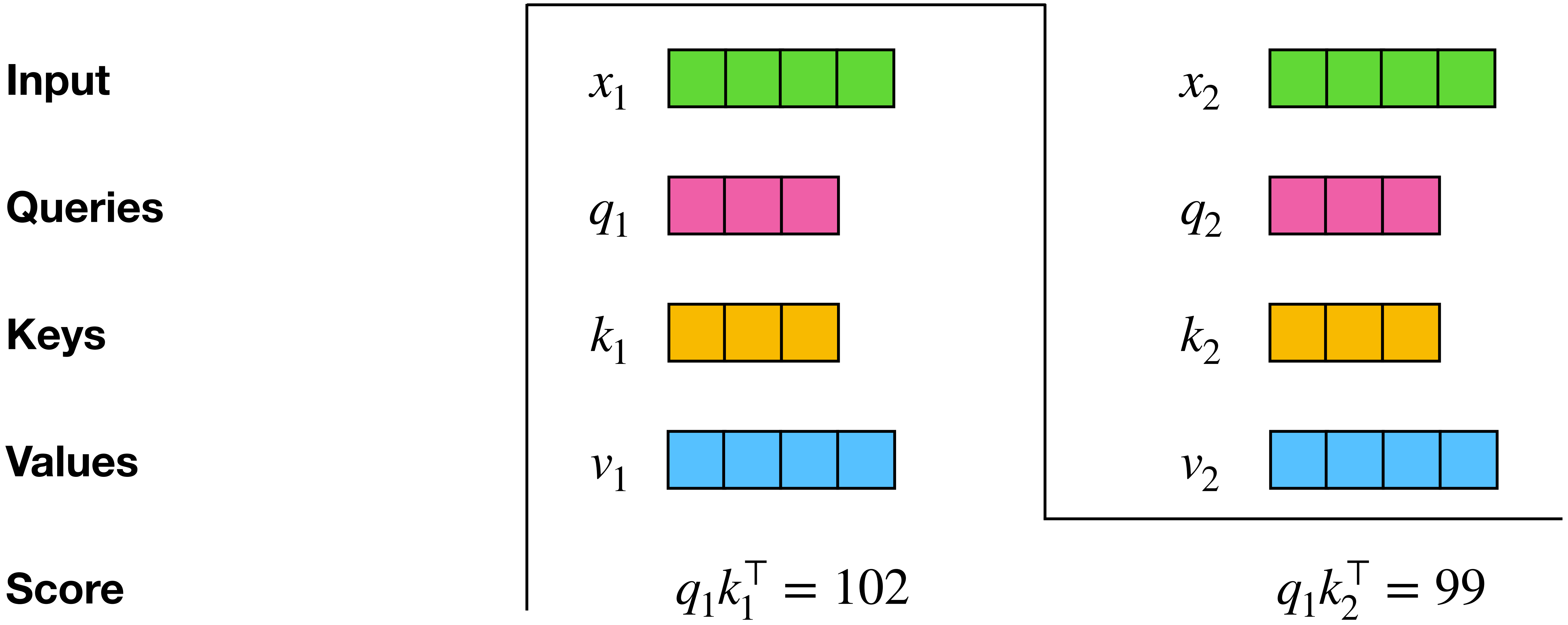


Step 1: create query, key and value vectors for each input token

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

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

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

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step

Input	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 k_1^\top = 102$	$q_1 k_2^\top = 99$
Divide by $\sqrt{D_K}$	$\frac{q_1 k_1^\top}{\sqrt{D_K}} = 58.9$	$\frac{q_1 k_2^\top}{\sqrt{D_K}} = 57.2$
Softmax	$p_{1,1} = 0.85$	$p_{1,2} = 0.15$

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

Step 4: Compute the softmax of these values

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

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step

Input	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 k_1^\top = 102$	$q_1 k_2^\top = 99$
Divide by $\sqrt{D_K}$	$\frac{q_1 k_1^\top}{\sqrt{D_K}} = 58.9$	$\frac{q_1 k_2^\top}{\sqrt{D_K}} = 57.2$
Softmax	$p_{1,1} = 0.85$	$p_{1,2} = 0.15$
Softmax*Value	$p_{1,1} v_1$	$p_{1,2} v_2$
Sum	z_1	z_2

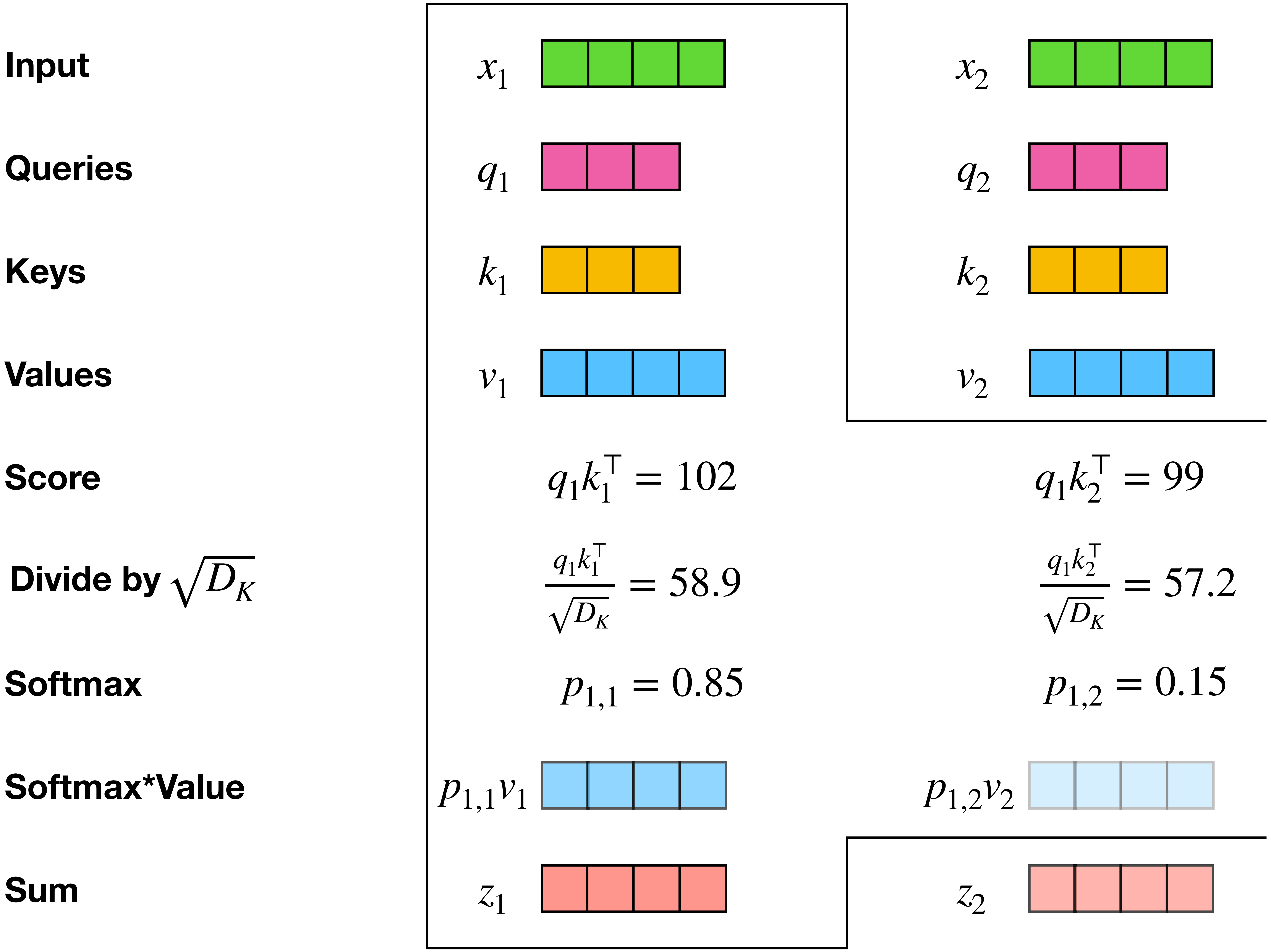
Step 5: Multiply each value vector by the softmax score

Step 6: Sum up the weighted value vectors

$$\boxed{Z = PV}$$

Illustrations borrowed from The Illustrated Transformer

Self-Attention: Step-by-Step



$$Z = \text{softmax}\left(\frac{X W_Q W_K^\top X^\top}{\sqrt{D_K}}\right) X W_V$$

Illustrations borrowed from The Illustrated Transformer

Multi-Head Self-Attention

- It is desirable to have multiple attention patterns per layer, similar to having multiple convolutions in a convolutional layer

    ➡ Run H Self-Attention “heads” in parallel

- The output of head h is given by:

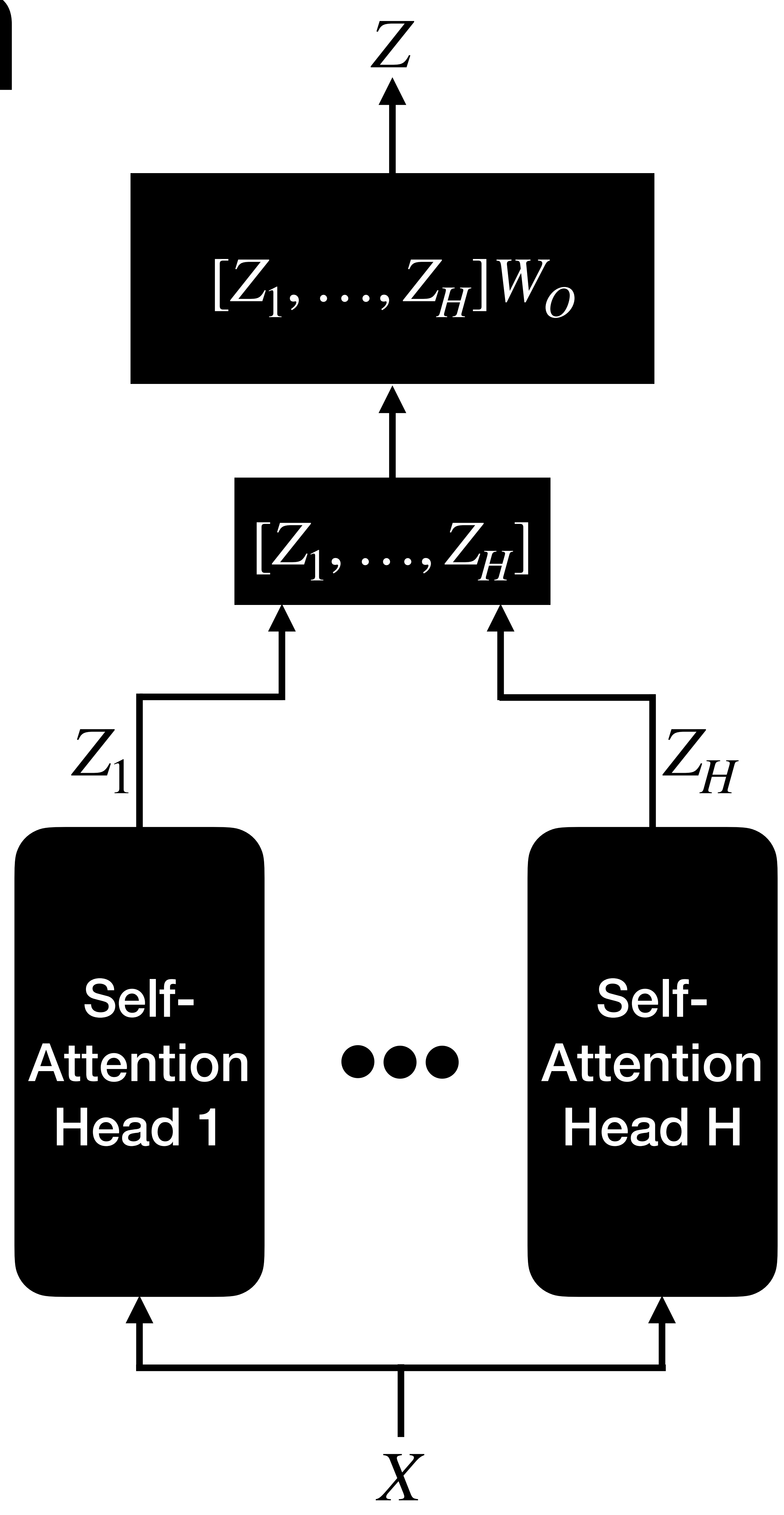
$$Z_h = \text{softmax}\left(\frac{XW_{Q,h}W_{K,h}^\top X^\top}{\sqrt{D_K}}\right)XW_{V,h}$$

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

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

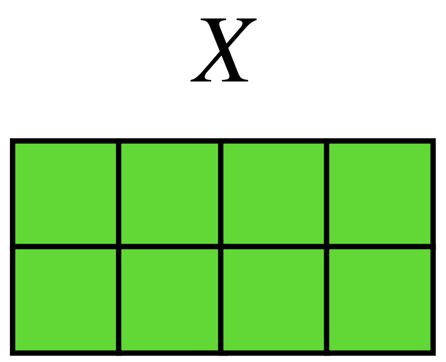
$$Z = [Z_1, \ldots, Z_H]W_O$$

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

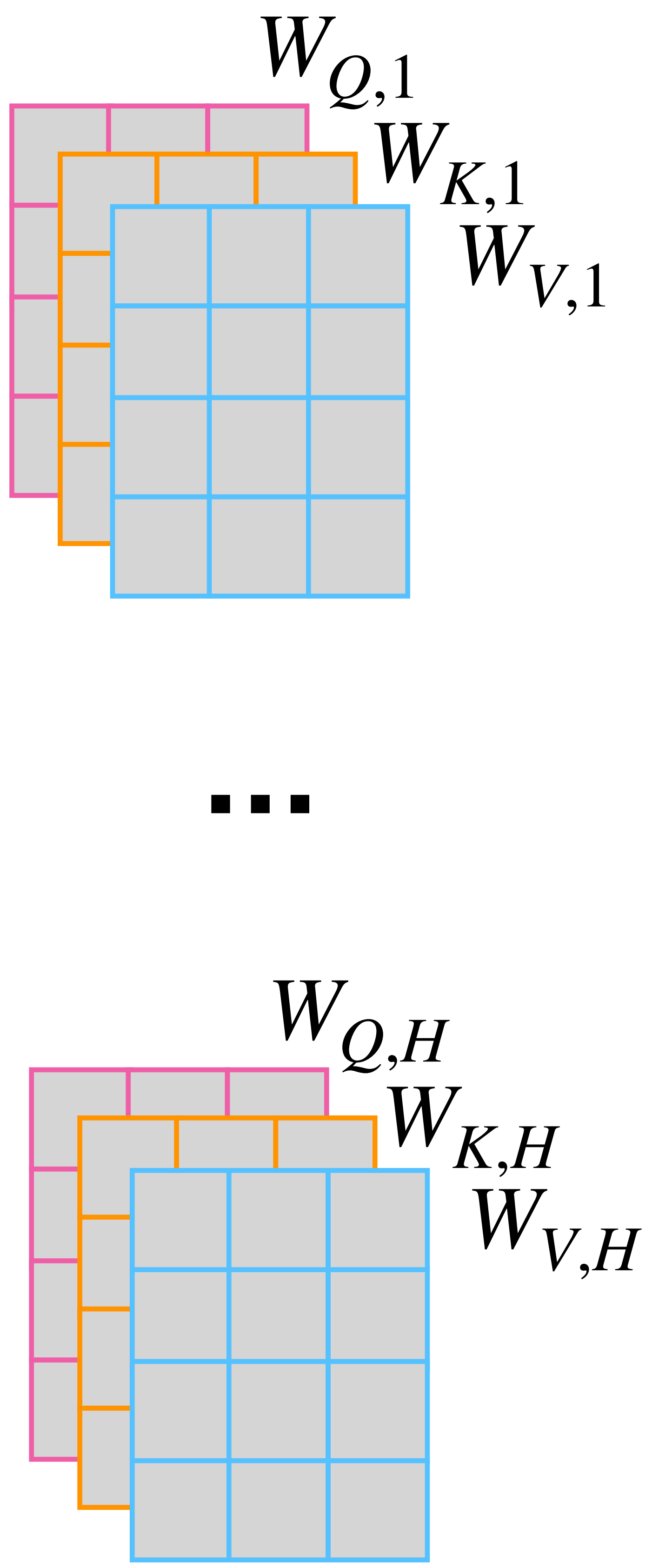


Multi-Head Self-Attention: recap

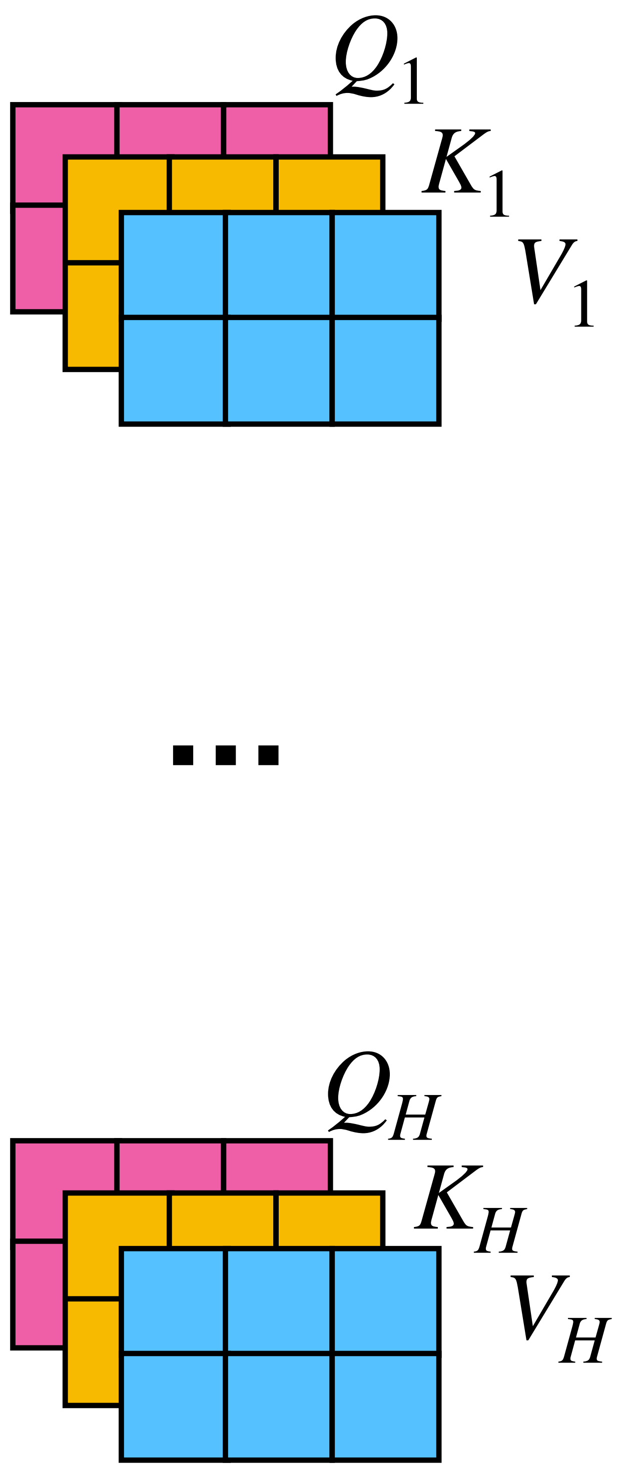
1) Input



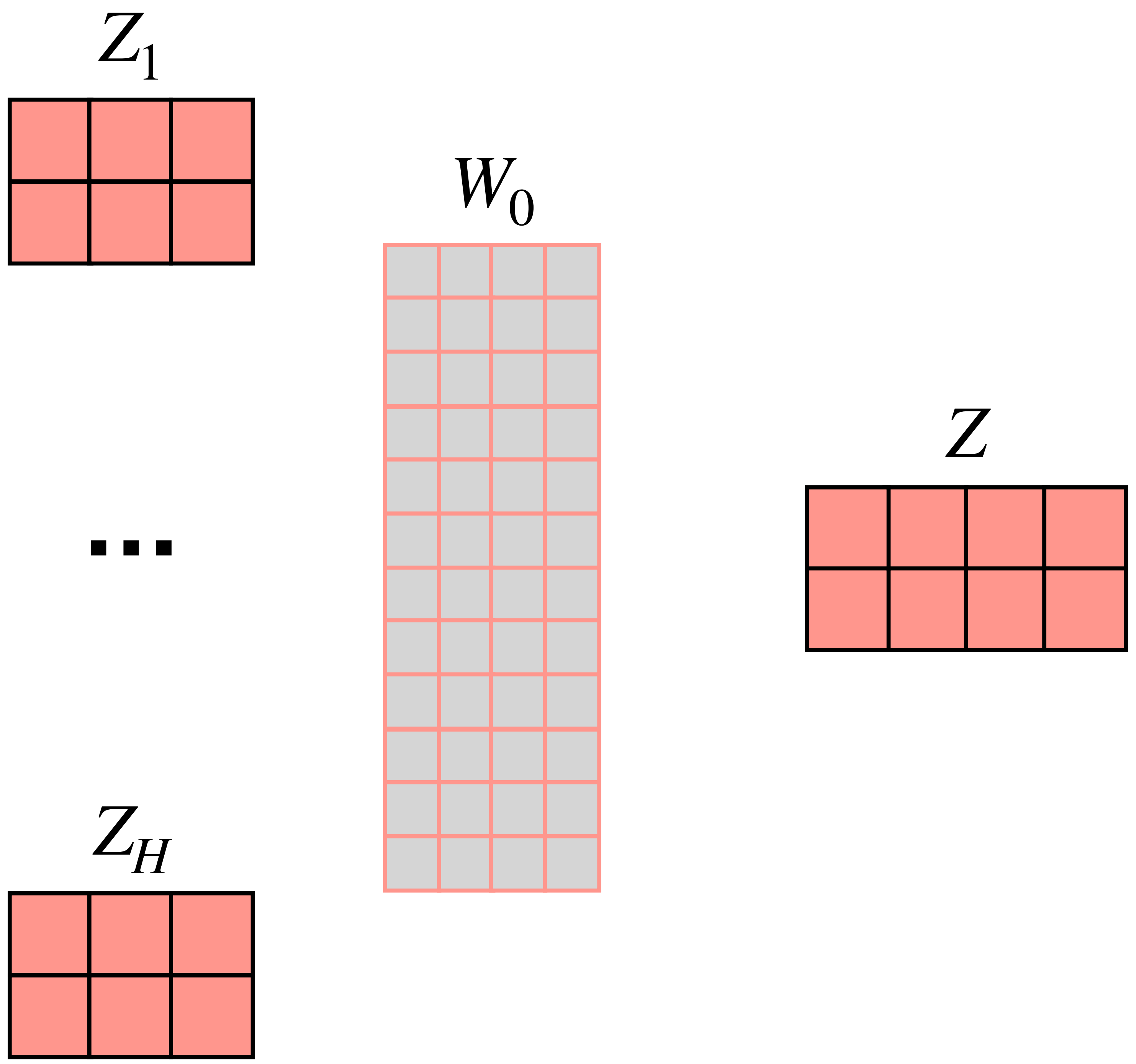
2) Split into H heads
We multiply X by weight matrices



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:

$$
\begin{aligned}
Z_R &= \text{softmax}\left(\frac{RXW_Q W_K^\top X^\top R^\top}{\sqrt{D_K}}\right) RXW_V && \textbf{Permute every X in original formula} \\
&= R\text{softmax}\left(\frac{XW_Q W_K^\top X^\top R^\top}{\sqrt{D_K}}\right) RXW_V && \textbf{Since softmax is computed row-wise} \\
&= R\text{softmax}\left(\frac{XW_Q W_K^\top X^\top}{\sqrt{D_K}}\right) R^\top RXW_V && \textbf{Reordering the terms in the softmax sum does not affect the output} \\
&= RPR^{-1}RXW_V && \textbf{For a permutation matrix: transpose=inverse} \\
&= RPXW_V
\end{aligned}
$$

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 $\text{pos} : \{1, \ldots, T\} \rightarrow \mathbb{R}^D$
- **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:

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_{i_1}^\top \\ \vdots \\ \mathbf{e}_{i_T}^\top \end{bmatrix} \mathbf{W}_{\text{emb}} + \begin{bmatrix} \mathbf{e}_{1}^\top \\ \vdots \\ \mathbf{e}_{T}^\top \end{bmatrix} \mathbf{W}_{\text{pos}}$$

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

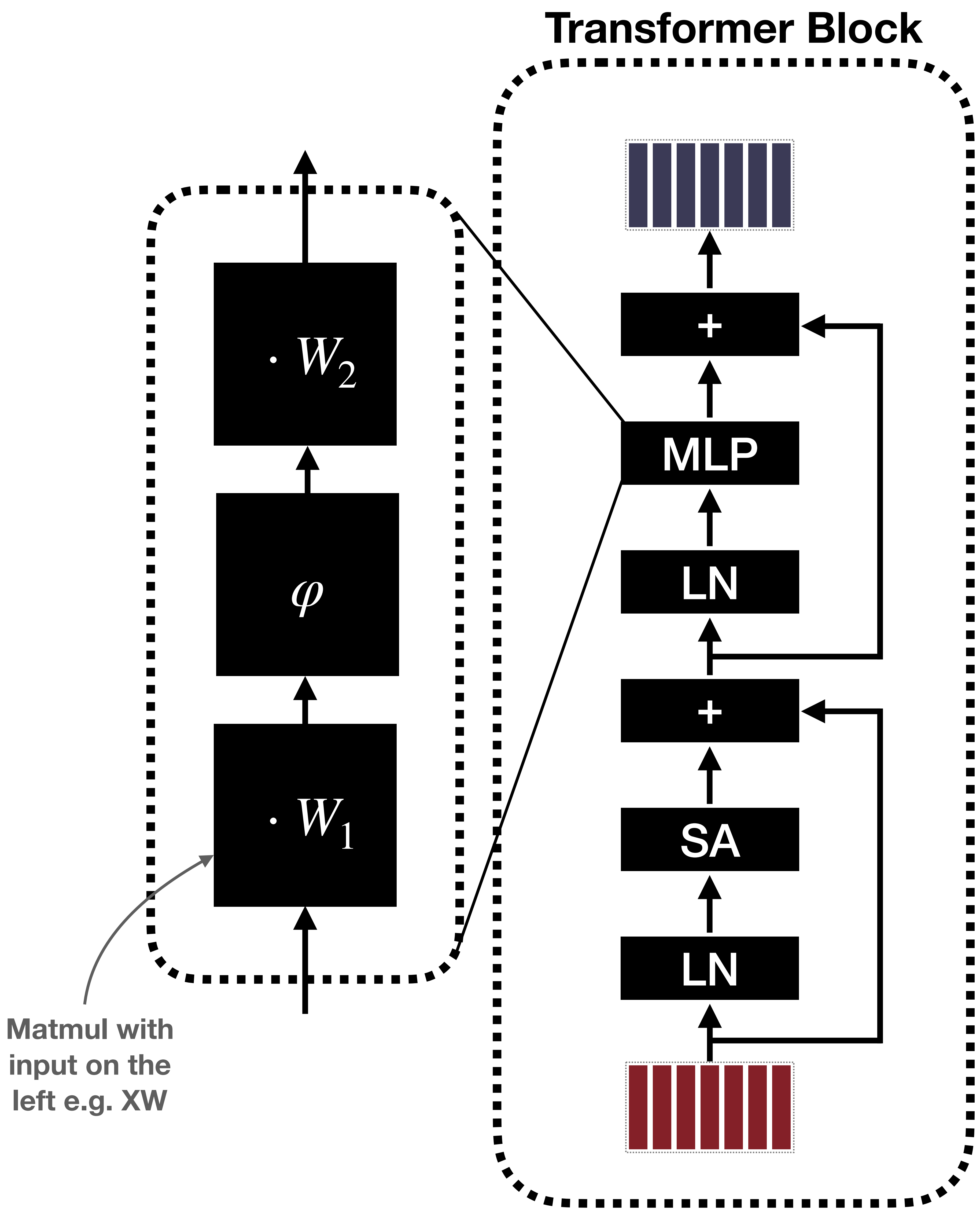
MLP

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:

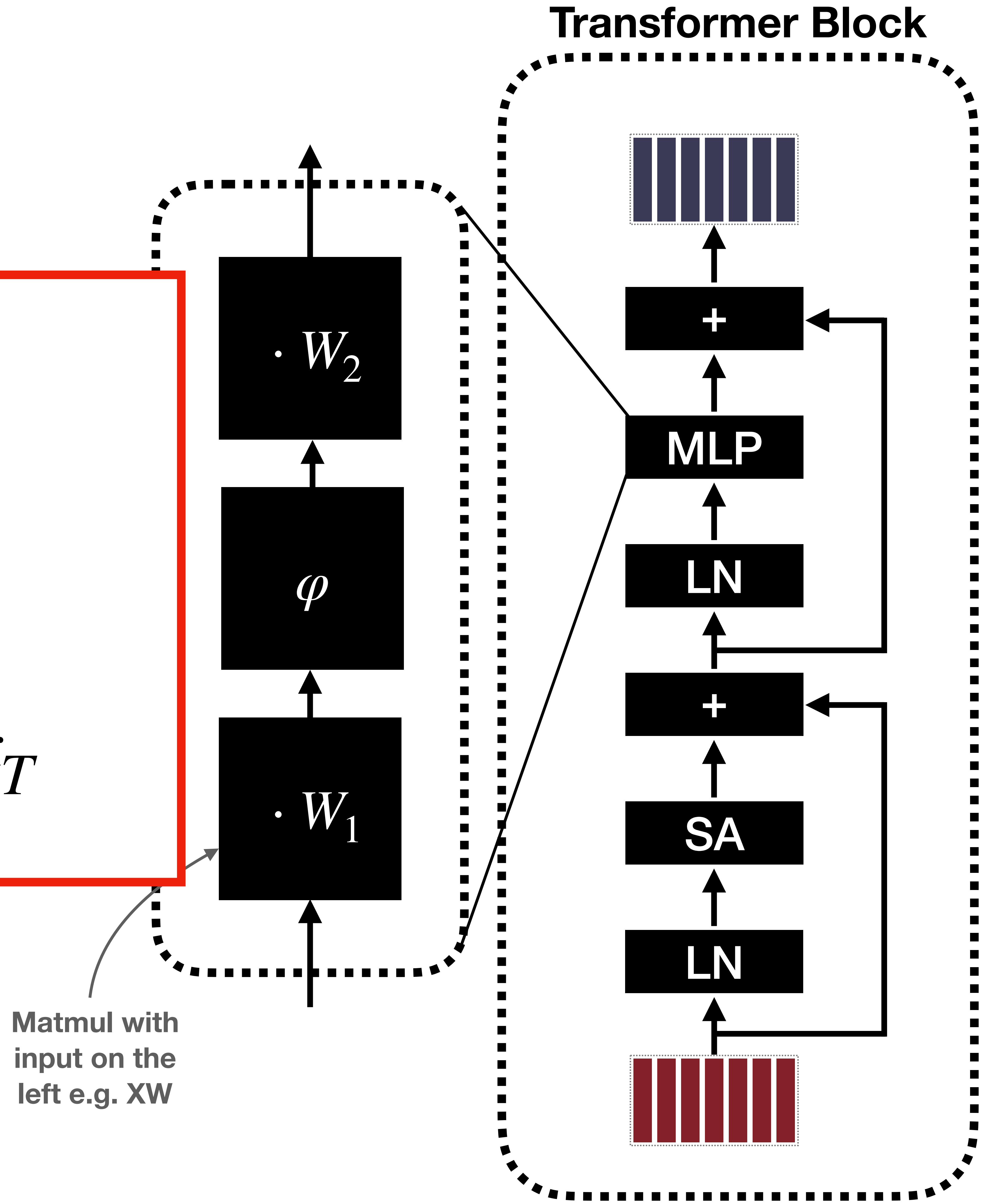
$$MLP(X) = \varphi(XW_1)W_2$$

$$\iff$$

$$MLP(x_i) = \varphi(x_i W_1)W_2, \text{ for each token } x_1, \ldots, x_T$$

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

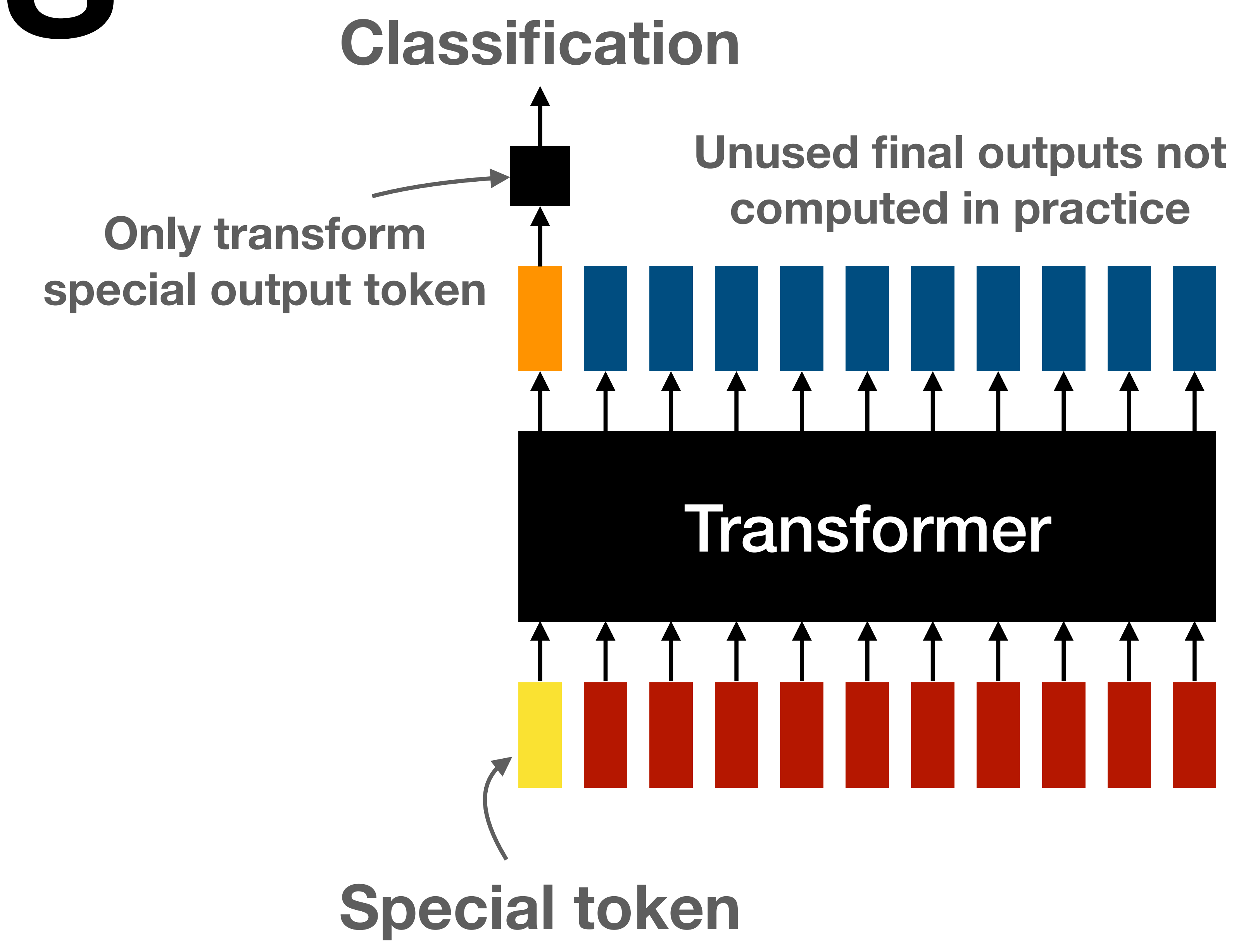
- The model may also include learned bias terms

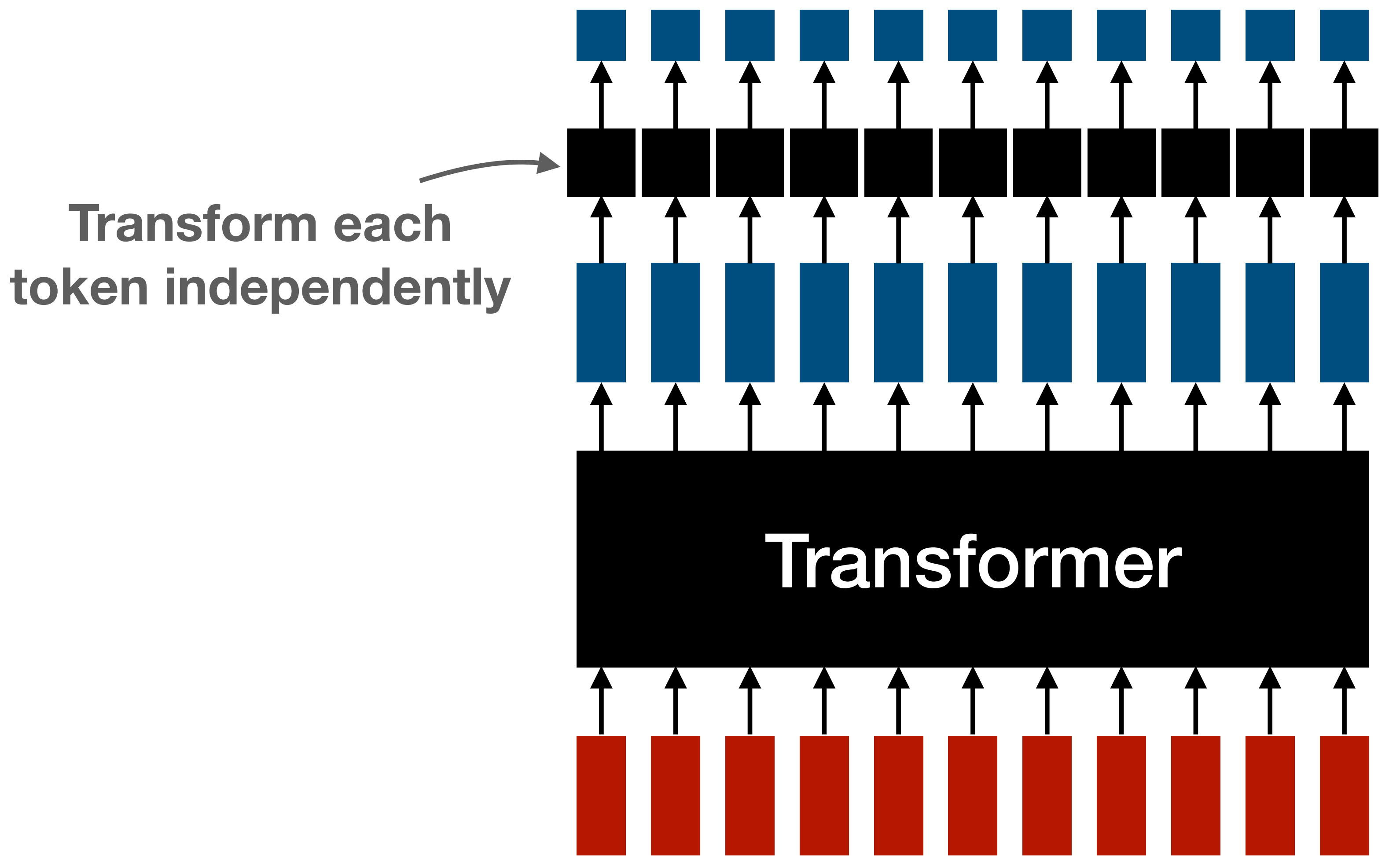


Output Transformations

Output Transformations

- We obtain the output from the final transformer block
- Output transformation is typically simple: linear transformation or a small MLP
- The specifics are highly dependent on the task:
  - **Single output** (e.g., sequence-level classification): apply an output transformation to a special task-specific input token or to the average of all tokens
  - **Multiple outputs** (e.g., per-token classification): apply an output transformation to each token independently

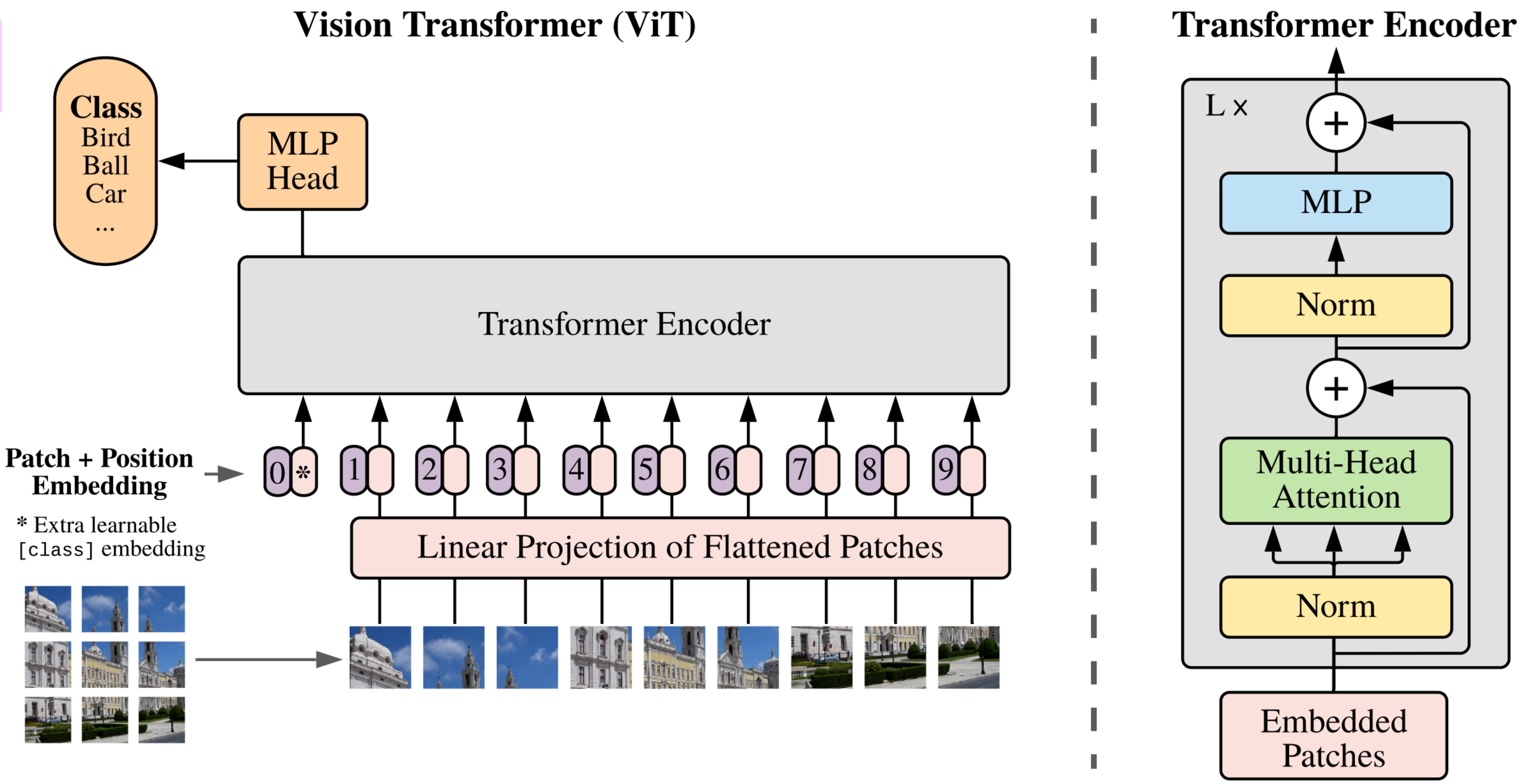




Putting the pieces together: Vision Transformers

Vision Transformer Architecture

- **Simple architecture**: number of features D is constant across all layers. There is no use of padding, pooling, or strides.
- Self-attention is **more general** than convolution and can express it
- The receptive field is the **whole image** after just one self-attention layer
- ViTs require more data than CNNs due to their reduced inductive bias in extracting local features
- However, ViTs become competitive with CNNs after **large-scale pretraining**



$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\, \cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D}$$

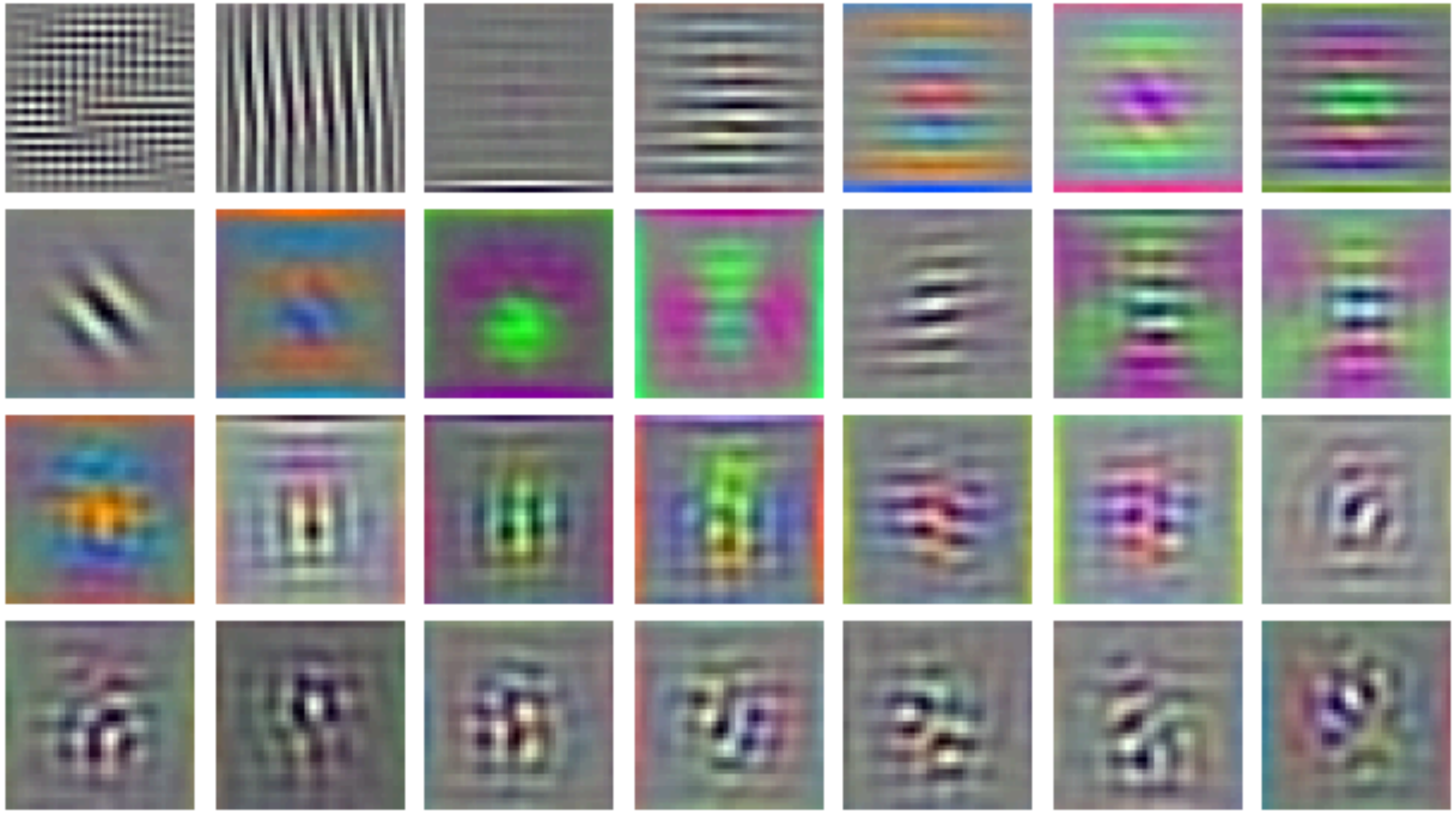
$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1\ldots L$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0)$$

Source: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (ICLR 2020)

What do ViTs learn: embedding layer



The first 28 principal components of the embedding layer applied on patches

Source: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (ICLR 2020)

- The embedding layer: edge/color detectors similar to first-layer convolutions

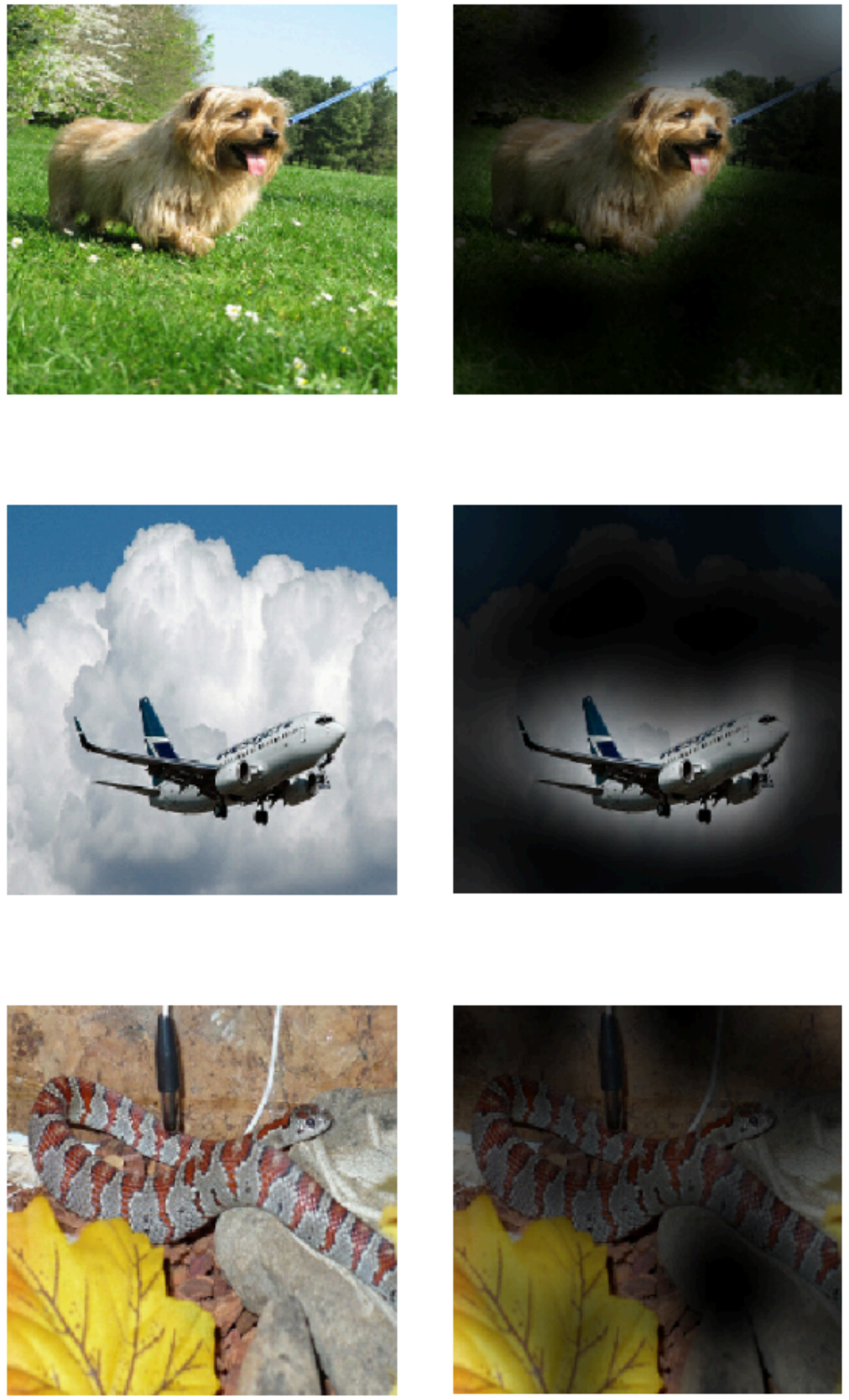
What do ViTs learn: attention

- The input-dependent attention weights can be visualized and manually inspected

- We show here one particular method known as <u>Attention Rollout</u>: where the attention weights are averaged across all heads and the resulting weight matrices of all layers are multiplied together

- This accounts for the mixing of attention across tokens through all layers

- In many cases, the model attends to image regions that are **semantically relevant** for classification

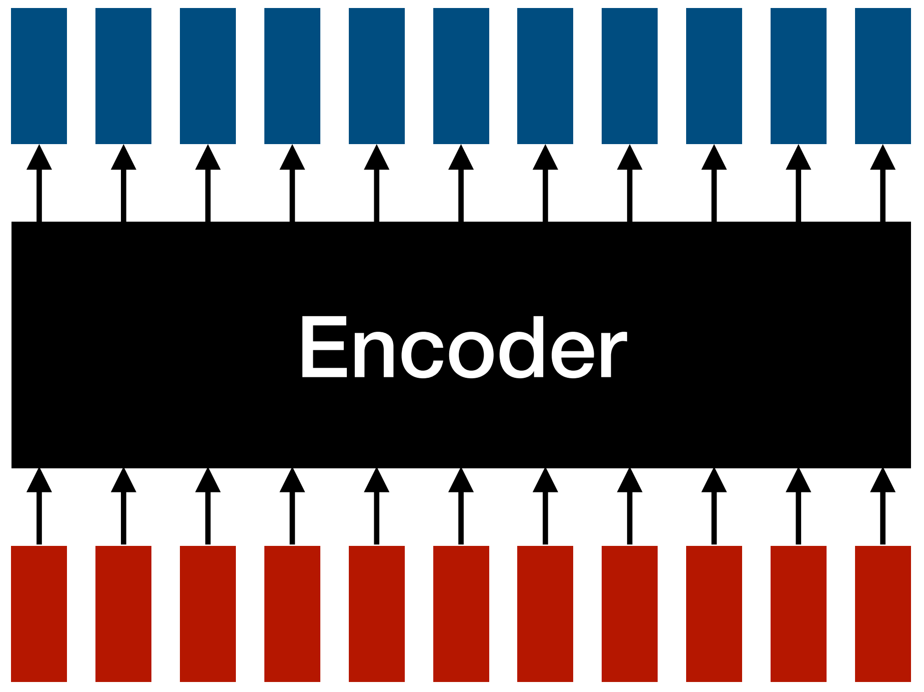


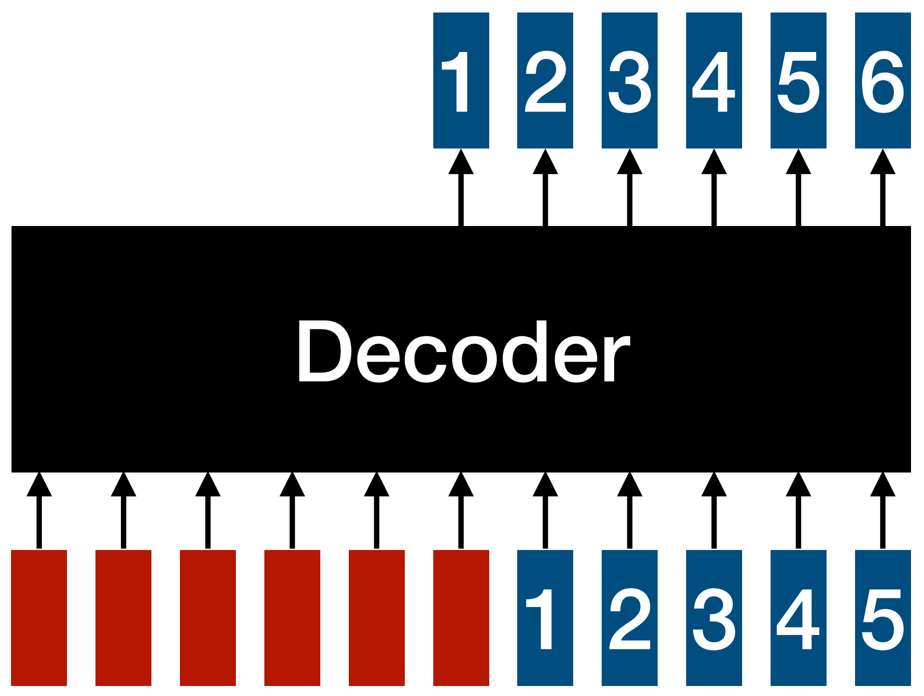
Source: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (ICLR 2020)

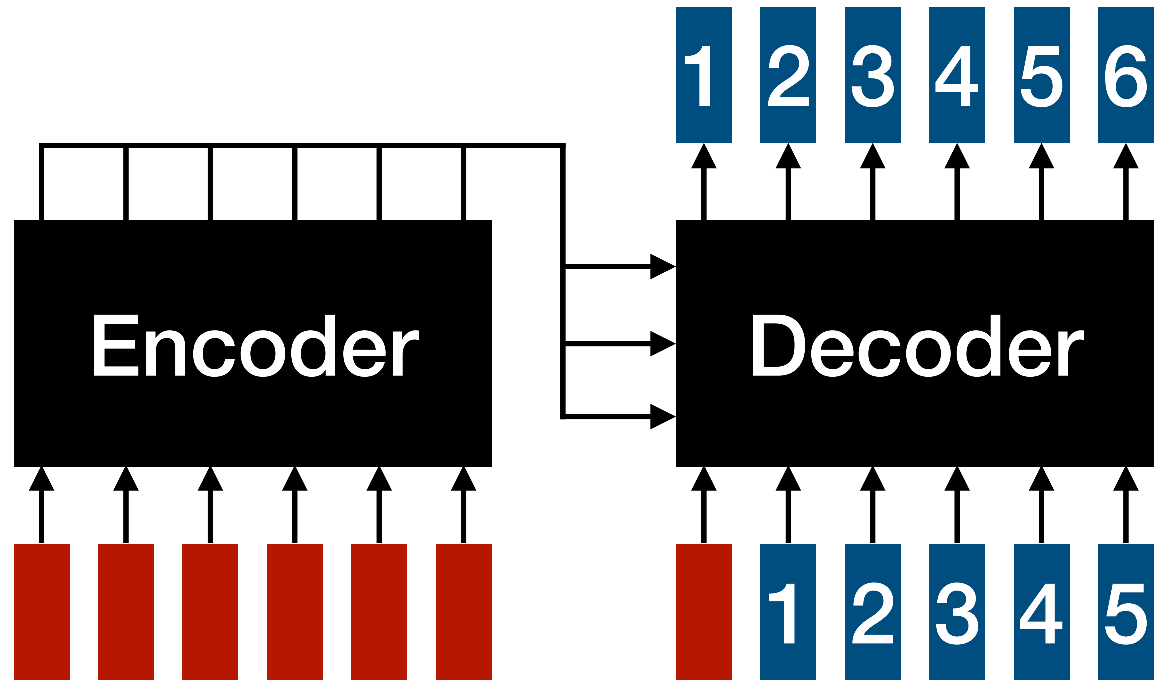
The Big Picture and Takeaways

The transformer architecture can be used in different ways

- **Encoders** (e.g., classification):
  - They produce a fixed output size and process all inputs simultaneously
- **Decoders** (e.g., ChatGPT):
  - **Auto-regressively sample** the next token as $x_{t+1} \sim softmax(f(x_1, \ldots, x_t))$ and use it as **new input token**
  - 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 to token by token (e.g., in a different language)







Transformers: Big Picture

- **Everything can be seen as a token, 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)