Advanced Deep Learning

DATA.ML.230

Transformers

From: Simon J. D. Prince, Chapter 12 – Understanding Deep Learning, MIT Press (19 May 2025)

Motivation

Transformers were initially proposed for text processing tasks.

Example: Restaurant review

- “The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread.”

Goal: design a network to process this text into a representation for downstream tasks:

- Classification to positive or negative
- Question answering: “Des the restaurant serve steak?

Motivation

Transformers were initially proposed for text processing tasks.

Example: Restaurant review

- “The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread.”

Parameter sharing → Varying size of input (or even sentences)

Goal: design a network to process this text into a representation for downstream tasks:

- Classification to positive or negative
- Question answering: “Des the restaurant serve steak?|

Motivation

Transformers were initially proposed for text processing tasks.

Example: Restaurant review

- "The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread."

Language can be ambiguous

Goal: design a network to process this text into a representation for downstream tasks:

- Classification to positive or negative
- Question answering: "Des the restaurant serve steak?|

Motivation

Transformers were initially proposed for text processing tasks.

Example: Restaurant review

- "The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread."

Attention → Language can be ambiguous

Goal: design a network to process this text into a representation for downstream tasks:

- Classification to positive or negative
- Question answering: "Des the restaurant serve steak?|

Dot-product self-attention

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

Dot-product self-attention

A self-attention block **sa**[•] takes N inputs $\mathbf{x}_1$, …, $\mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

Dot-product self-attention

A self-attention block **sa**[•] takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

Parameter sharing:
The same parameters are applied to all inputs

This computation scales linearly with the sequence length N

Dot-product self-attention

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

- Then, the n^{th} output $\mathbf{sa}_n[\mathbf{x}_1,\ldots,\mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1,\ldots,\mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m$$

Dot-product self-attention

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

- Then, the n^{th} output $\mathbf{sa}_n[\mathbf{x}_1,\ldots,\mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1,\ldots,\mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m$$

Attention weights:

- Combine the values from different inputs
- N^2 in total (quadratic dependence on the sequence length N)
- Independent of the dimension of $\mathbf{x}_n$

Dot-product self-attention

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

- Then, the n^{th} output $\mathbf{sa}_n[\mathbf{x}_1,\ldots,\mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1,\ldots,\mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m$$

- The scalar weight $a[x_m, x_n]$ is the attention that the n^{th} output pays to input $\mathbf{x}_m$
- The N weights $a[\bullet, x_n]$ are non-negative and sum to one

Dot-product self-attention

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and returns N outputs:

- A set of *values* are computed for each input:

$$\mathbf{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_m$$

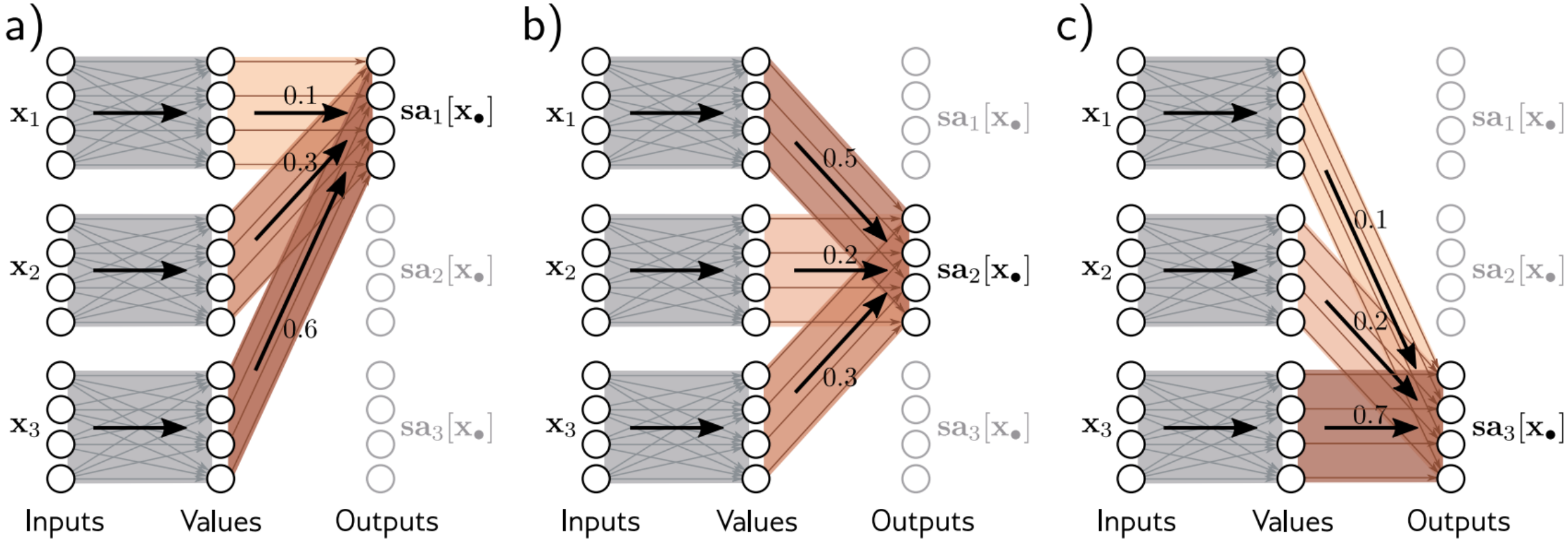
- Then, the n^{th} output $\mathbf{sa}_n[\mathbf{x}_1,\ldots,\mathbf{x}_N]$ is a weighted sum of all the values $\mathbf{v}_1,\ldots,\mathbf{v}_N$:

$$\mathbf{sa}_n[\mathbf{x}_1, \ldots, \mathbf{x}_N] = \sum_{m=1}^{N} a[\mathbf{x}_m, \mathbf{x}_n]\mathbf{v}_m$$

- The scalar weight $a[x_m, x_n]$ is the attention that the n^{th} output pays to input $\mathbf{x}_m$
- The N weights $a[\bullet, x_n]$ are non-negative and sum to one ← Leads to non-linear mapping

Dot-product self-attention

A self-attention as routing



Attention weights

The overall self-attention computation is nonlinear:

- Apply two more linear transformations:

Queries $\quad \mathbf{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q \mathbf{x}_n$

Keys $\quad \mathbf{k}_m = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{x}_m$

Attention weights

The overall self-attention computation is nonlinear:

- Apply two more linear transformations:

$$\begin{aligned} \text{Queries} \quad \mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q \mathbf{x}_n \\ \text{Keys} \quad \mathbf{k}_m &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{x}_m \end{aligned}$$

- Compute the dot products between the queries and keys and pass the results through a softmax function:

$$a[\mathbf{x}_m, \mathbf{x}_n] = \text{softmax}_m\left[\mathbf{k}_\bullet^T \mathbf{q}_n\right] = \frac{\exp\left[\mathbf{k}_m^T \mathbf{q}_n\right]}{\sum_{m'=1}^{N} \exp\left[\mathbf{k}_{m'}^T \mathbf{q}_n\right]}$$

Attention weights

The overall self-attention computation is nonlinear:

- Apply two more linear transformations:

$$\begin{aligned} \text{Queries} \quad \mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q \mathbf{x}_n \\ \text{Keys} \quad \mathbf{k}_m &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{x}_m \end{aligned}$$

- Compute the dot products between the queries and keys and pass the results through a softmax function:

$$a[\mathbf{x}_m, \mathbf{x}_n] = \text{softmax}_m\left[\mathbf{k}_\bullet^T \mathbf{q}_n\right] = \frac{\exp\left[\mathbf{k}_m^T \mathbf{q}_n\right]}{\sum_{m'=1}^{N} \exp\left[\mathbf{k}_{m'}^T \mathbf{q}_n\right]}$$

The weights $a[\mathbf{x}_\bullet, \mathbf{x}_n]$ sum to 1

Attention weights

The overall self-attention computation is nonlinear:

- Apply two more linear transformations:

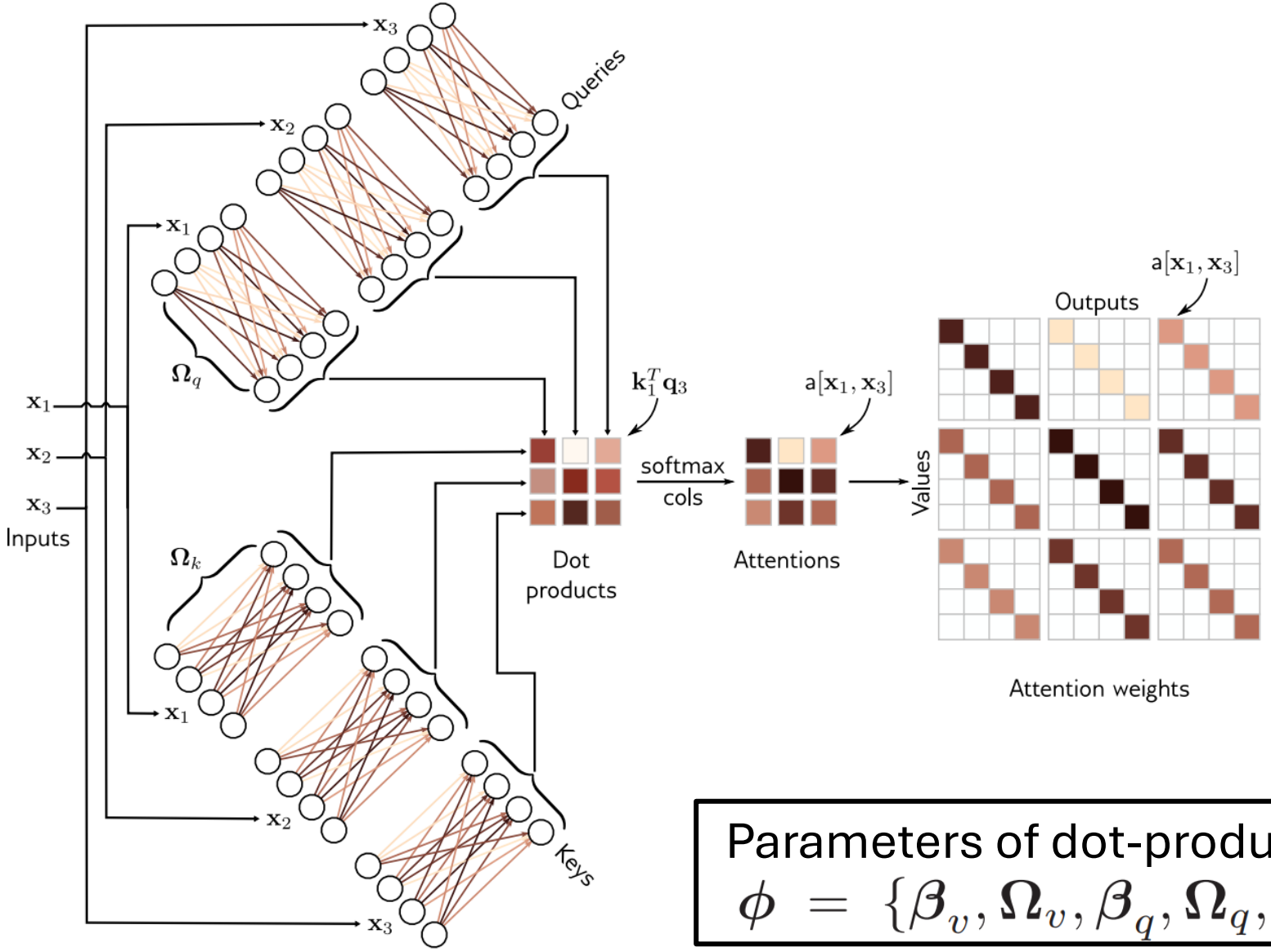
$$\begin{aligned} \text{Queries} \quad \mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q \mathbf{x}_n \\ \text{Keys} \quad \mathbf{k}_m &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{x}_m \end{aligned}$$

- Compute the dot products between the queries and keys and pass the results through a softmax function:

$$a[\mathbf{x}_m, \mathbf{x}_n] = \text{softmax}_m\left[\mathbf{k}_\bullet^T \mathbf{q}_n\right] = \frac{\exp\left[\mathbf{k}_m^T \mathbf{q}_n\right]}{\sum_{m'=1}^{N} \exp\left[\mathbf{k}_{m'}^T \mathbf{q}_n\right]}$$

Keys compete to contribute to the result → The weights $a[\mathbf{x}_\bullet, \mathbf{x}_n]$ sum to 1

Dot-product self-attention



Parameters of dot-product self-attention:

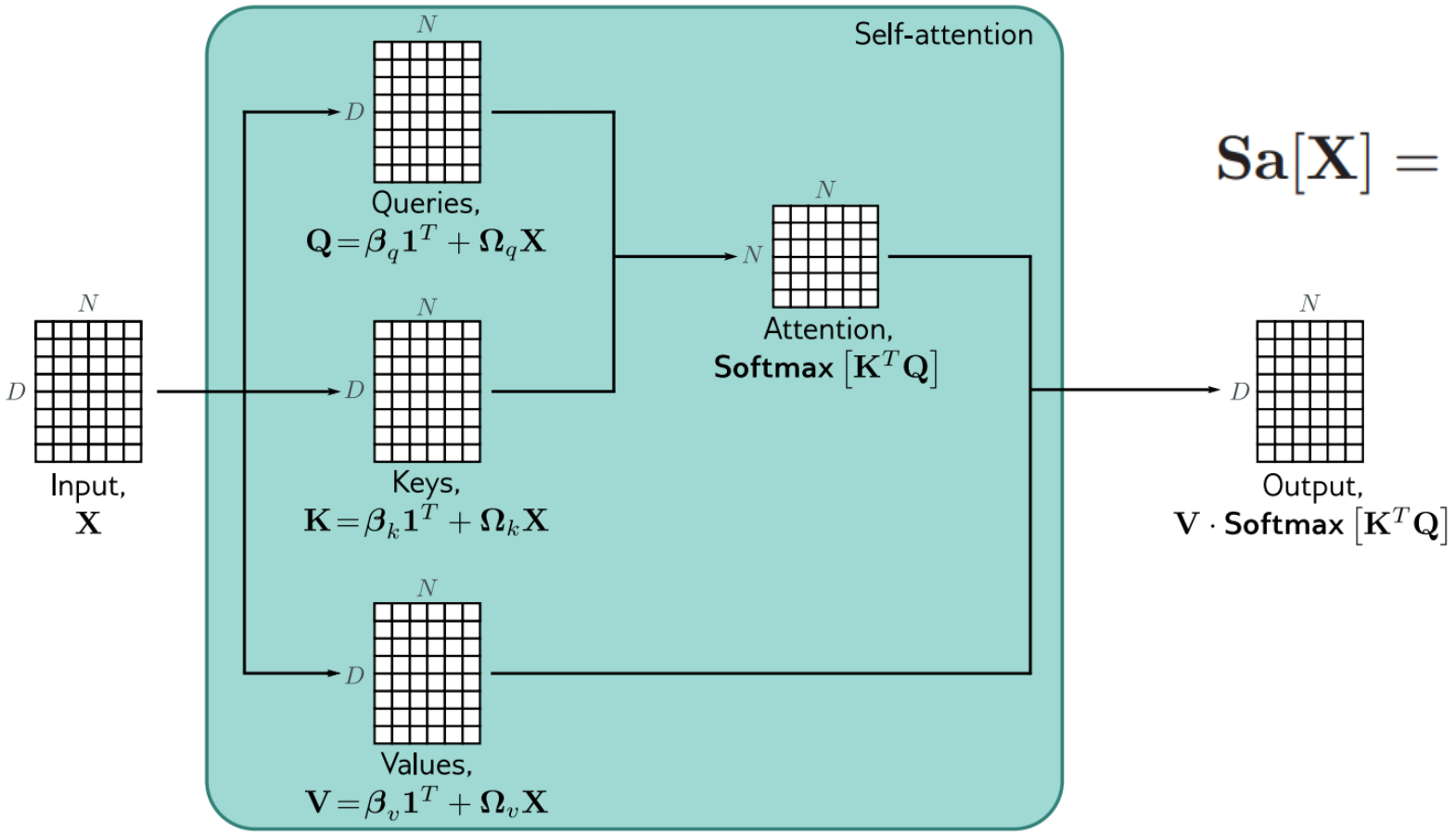
$$\boldsymbol{\phi} = \{\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$$

Attention weights

Matrix form:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \boldsymbol{\beta}_v\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_v\mathbf{X} \\ \mathbf{Q}[\mathbf{X}] &= \boldsymbol{\beta}_q\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_q\mathbf{X} \\ \mathbf{K}[\mathbf{X}] &= \boldsymbol{\beta}_k\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_k\mathbf{X}\end{aligned}$$

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}] \cdot \mathbf{Softmax}\left[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\right]$$

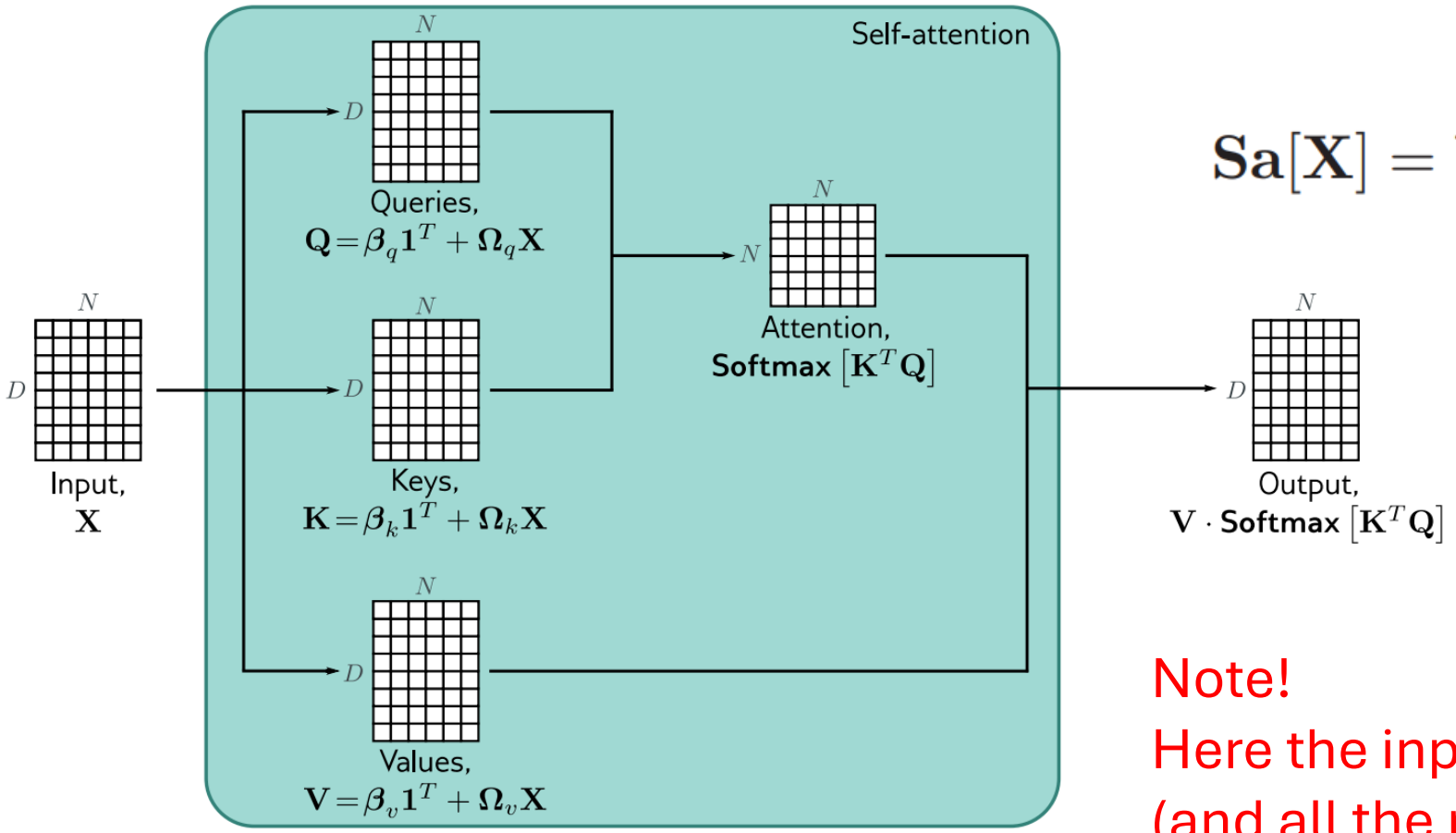


Attention weights

Matrix form:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \boldsymbol{\beta}_v\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_v\mathbf{X} \\ \mathbf{Q}[\mathbf{X}] &= \boldsymbol{\beta}_q\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_q\mathbf{X} \\ \mathbf{K}[\mathbf{X}] &= \boldsymbol{\beta}_k\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_k\mathbf{X}\end{aligned}$$

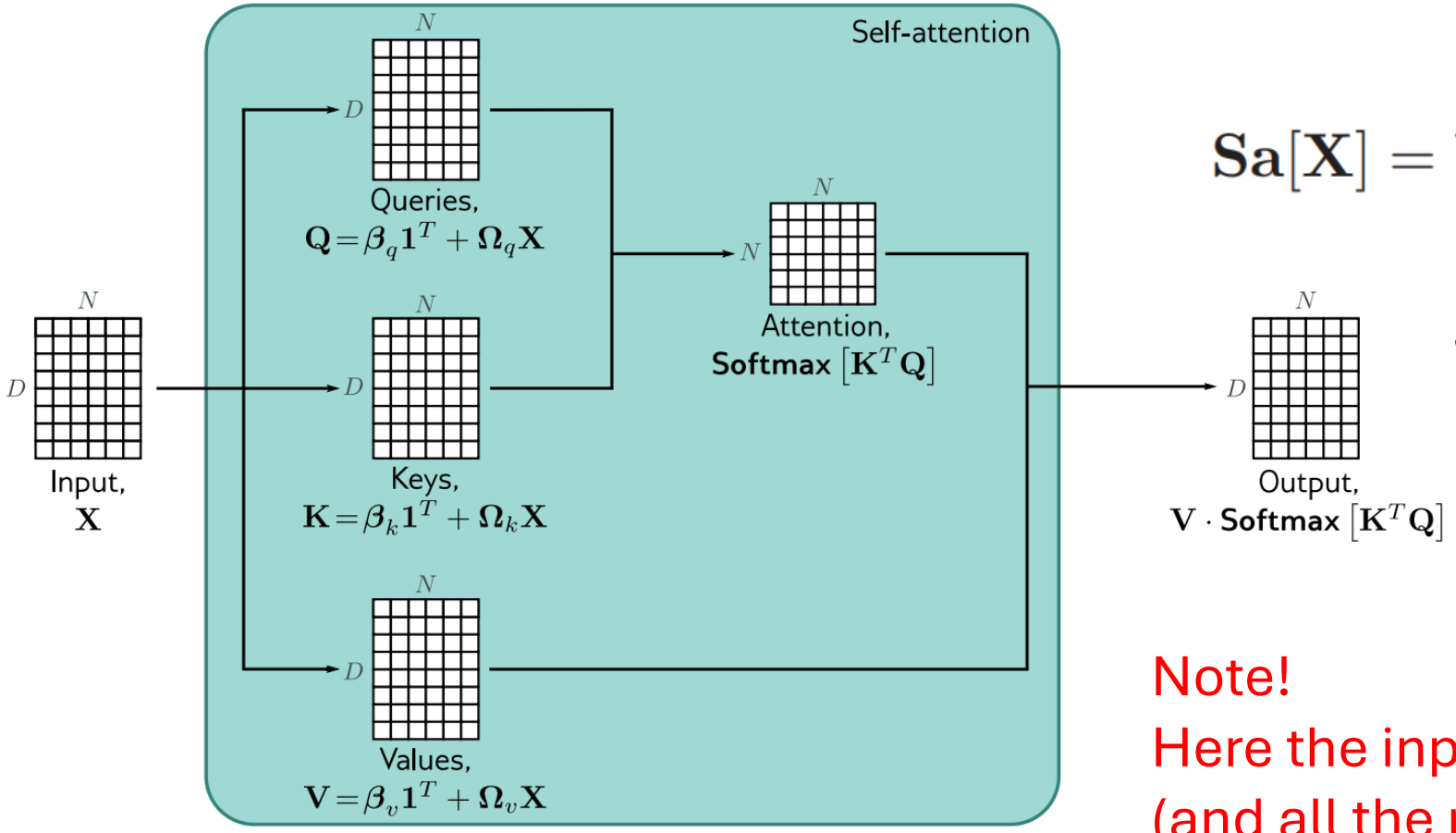
$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}] \cdot \mathbf{Softmax}\left[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\right]$$



Note!
Here the input X is a column-matrix. Often, X (and all the rest matrices) are row-matrices

Attention weights

Matrix form:



$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \beta_v\mathbf{1}^\mathbf{T}+\Omega_v\mathbf{X}\\ \mathbf{Q}[\mathbf{X}] &= \beta_q\mathbf{1}^\mathbf{T}+\Omega_q\mathbf{X}\\ \mathbf{K}[\mathbf{X}] &= \beta_k\mathbf{1}^\mathbf{T}+\Omega_k\mathbf{X}\end{aligned}$$

$$\mathbf{Sa}[\mathbf{X}]=\mathbf{V}[\mathbf{X}]\cdot\mathbf{Softmax}\left[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\right]$$

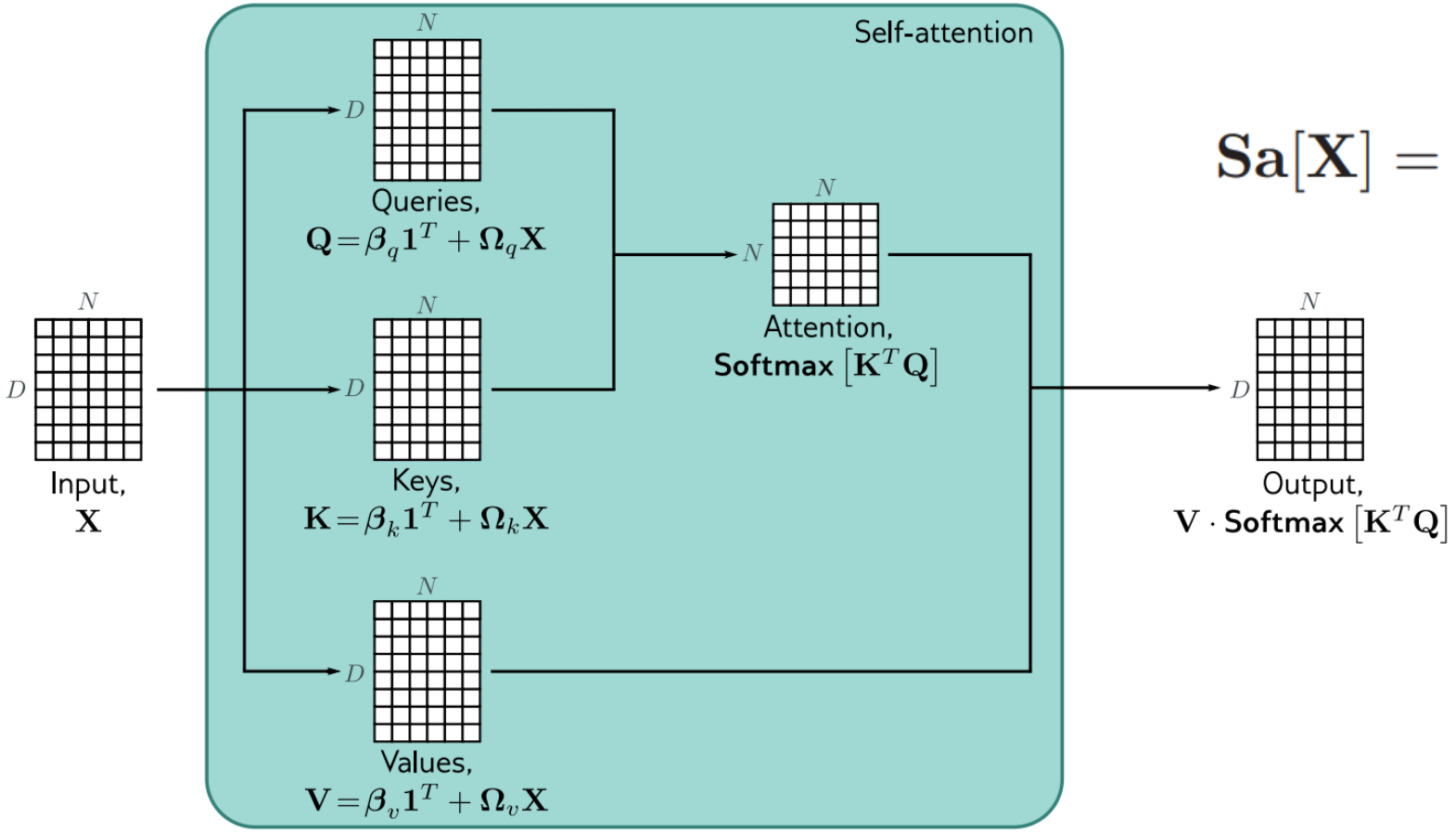
Scaled dot-product attention

$$\mathbf{Sa}[\mathbf{X}]=\mathbf{V}\cdot\mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]$$

Note!
Here the input X is a column-matrix. Often, X (and all the rest matrices) are row-matrices

Attention weights

Matrix form:



$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \beta_v\mathbf{1}^\mathbf{T}+\mathbf{\Omega}_v\mathbf{X}\\ \mathbf{Q}[\mathbf{X}] &= \beta_q\mathbf{1}^\mathbf{T}+\mathbf{\Omega}_q\mathbf{X}\\ \mathbf{K}[\mathbf{X}] &= \beta_k\mathbf{1}^\mathbf{T}+\mathbf{\Omega}_k\mathbf{X}\end{aligned}$$

$$\mathbf{Sa}[\mathbf{X}]=\mathbf{V}[\mathbf{X}]\cdot\mathbf{Softmax}\left[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\right]$$

Scaled dot-product attention

$$\mathbf{Sa}[\mathbf{X}]=\mathbf{V}\cdot\mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]$$

Self-attention:

Keys and Queries (defining the attention) depend on the same input as the Values

Positional encodings

The self-attention mechanism does not consider the order of the inputs x_n:

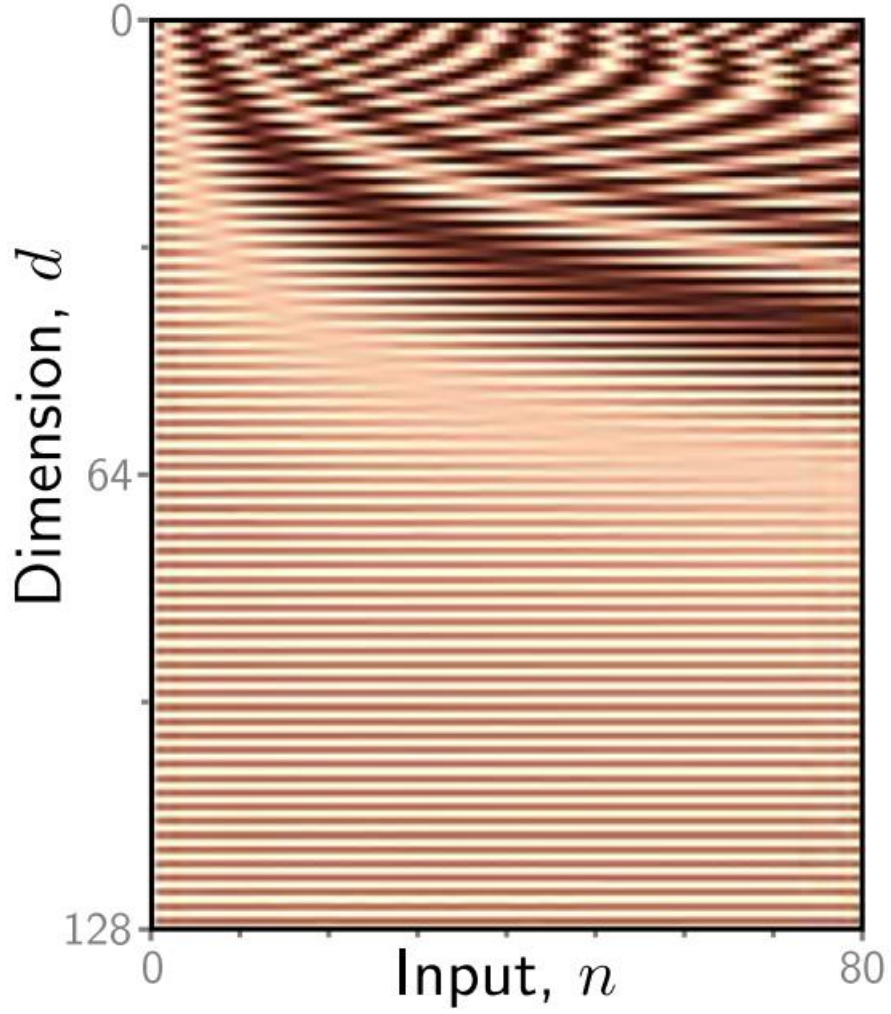
- Multiple equivalent attentions for different permutations of the data
- Order of the inputs is important, e.g., order of words in a sentence:
  - The woman ate the racoon
  - The racoon ate the woman

Positional encodings

Absolute positional encoding:

- Add a matrix **Π** encoding the positional information to the input **X**
- **Π** can be predefined or learned
- **Π** can be added in the input or at every network layer
- Sometimes **Π** is added to **X** in the computation of queries and keys, but not to the values

Sinusoidal pattern positional encodings



Positional encodings

Relative positional encoding:

- Sometimes the absolute position of an input is less important compared to the relative position with respect to other inputs
- Each element of the attention matrix:
  - A particular offset between key position a and query position b

Process:

- Learn a parameter $\pi_{a,b}$ for each offset
- Use it to modify the attention matrix by altering it (e.g., addition, multiplication)

Scaled dot-product self-attention

The dot products in the attention computation can have large magnitudes:

- the largest value completely dominates the softmax function calculation
- Normalize the input to the softmax function

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]$$

Multiple heads

- Calculate multiple self-attentions in parallel:

$$\begin{aligned}\mathbf{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{vh}\mathbf{X} \\ \mathbf{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{qh}\mathbf{X} \\ \mathbf{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{kh}\mathbf{X}\end{aligned}$$

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}_h^T\mathbf{Q}_h}{\sqrt{D_q}}\right]$$

Each self-attention is called *head*, and learns different parameters $\{\boldsymbol{\Omega}_{\bullet h}, \boldsymbol{\beta}_{\bullet h}\}$

Multiple heads

- Calculate multiple self-attentions in parallel:

$$\begin{aligned}\mathbf{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{vh}\mathbf{X} \\ \mathbf{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{qh}\mathbf{X} \\ \mathbf{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{kh}\mathbf{X}\end{aligned}$$

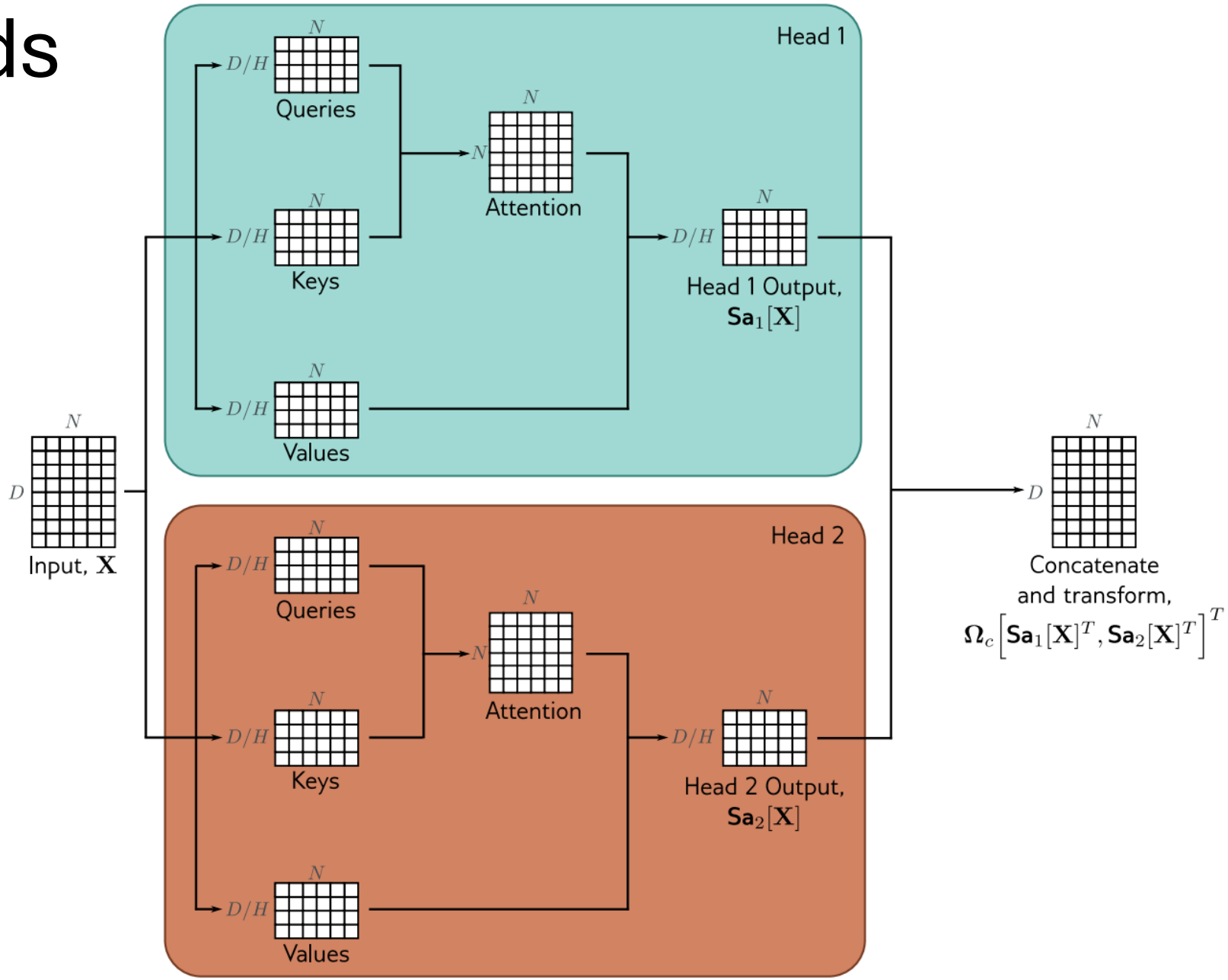
Each self-attention is called *head*, and learns different parameters $\{\boldsymbol{\Omega}_{\bullet h}, \boldsymbol{\beta}_{\bullet h}\}$

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}_h^T\mathbf{Q}_h}{\sqrt{D_q}}\right]$$

- Concatenate the outputs and perform a linear transformation for fusing the information:

$$\mathbf{MhSa}[\mathbf{X}] = \boldsymbol{\Omega}_c\left[\mathbf{Sa}_1[\mathbf{X}]^T, \mathbf{Sa}_2[\mathbf{X}]^T, \ldots, \mathbf{Sa}_H[\mathbf{X}]^T\right]^T$$

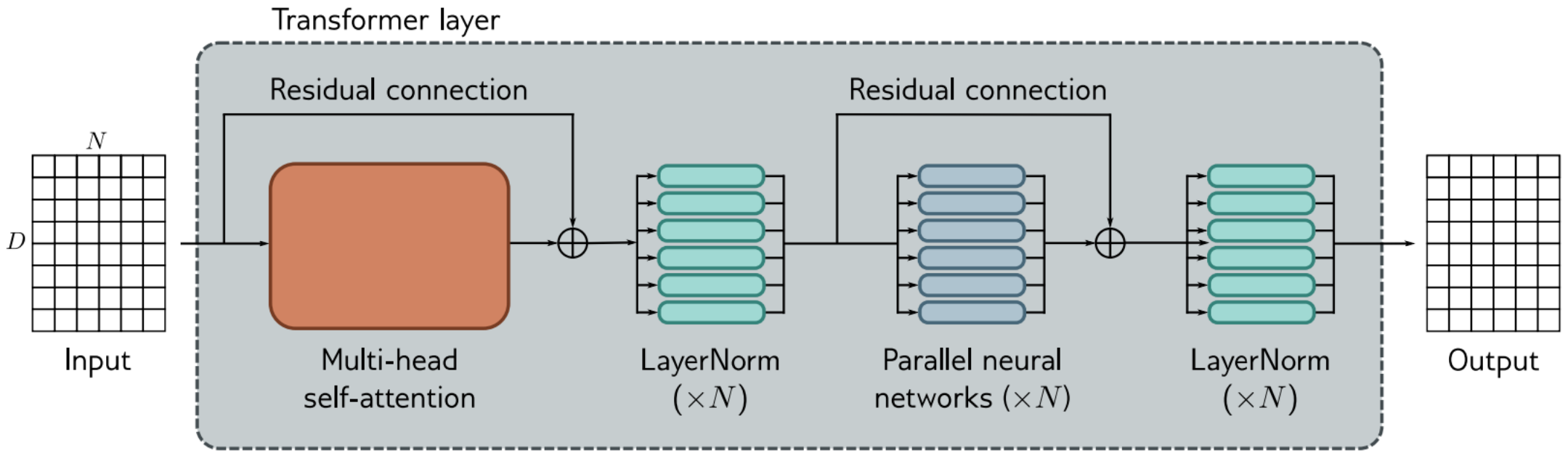
Multiple heads



Transformer layer

Calculations:

$$\begin{aligned}
\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\
\mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}]
\end{aligned}$$

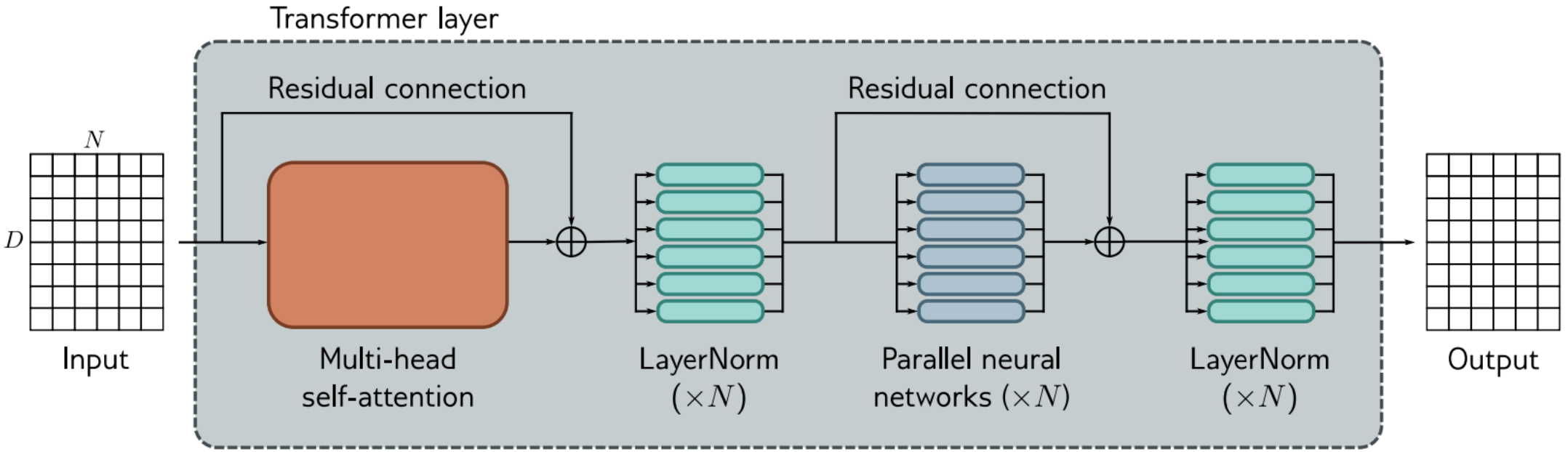


Transformer layer

LayerNorm is similar to BatchNorm but normalizes each embedding in each batch element separately using statistics calculated across its D embedding dimensions

Calculations:

$$\begin{aligned}
\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\
\mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}]
\end{aligned}$$



Transformers for Natural Language Processing

Tokenization:

- Splits the text into smaller constituent units (called *tokens*) from a vocabulary of possible tokens

Transformers for Natural Language Processing

Tokenization:

- Splits the text into smaller constituent units (called *tokens*) from a vocabulary of possible tokens
- Difficulties:
  - To include all possible words, an enormous (infinite…) vocabulary is needed
  - Unclear how to handle punctuation (e.g., question mark)
  - Unclear how to clarify/encode different versions of a word (e.g., a different token?)

Transformers for Natural Language Processing

Tokenization:

- Splits the text into smaller constituent units (called *tokens*) from a vocabulary of possible tokens
- Difficulties:
  - To include all possible words, an enormous (infinite…) vocabulary is needed
  - Unclear how to handle punctuation (e.g., question mark)
  - Unclear how to clarify/encode different versions of a word (e.g., a different token?)
- Compromise between letters and full words
  - Include both common words and word fragments
  - Those can be used to create larger and less frequent words

Transformers for Natural Language Processing

Example sub-word tokenizer:

a) The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table.

b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of tokens (in this case, se) and merges them

c) At the second iteration, the algorithm merges e and the whitespace character_.

d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words.

e) If we continue this process indefinitely, the tokens eventually represent the full words

In a real situation, the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value

a)

a_sailor_went_to_sea_sea_sea_
to_see_what_he_could_see_see_see_
but_all_that_he_could_see_see_see_
was_the_bottom_of_the_deep_blue_sea_sea_sea_

_	e	s	a	t	o	h	l	u	b	d	w	c	f	i	m	n	p	r
33	28	15	12	11	8	6	6	4	3	3	3	2	1	1	1	1	1	1

b)

a_sailor_went_to_sea_sea_sea_
to_see_what_he_could_see_see_see_
but_all_that_he_could_see_see_see_
was_the_bottom_of_the_deep_blue_sea_sea_sea_

_	e	se	a	t	o	h	l	u	b	d	w	c	s	f	i	m	n	p	r
33	15	13	12	11	8	6	6	4	3	3	3	2	2	1	1	1	1	1	1

c)

a_sailor_went_to_sea_sea_sea_
to_see_what_he_could_see_see_see_
but_all_that_he_could_see_see_see_
was_the_bottom_of_the_deep_blue_sea_sea_sea_

_	se	a	e_	t	o	h	l	u	b	d	e	w	c	s	f	i	m	n	p	r
21	13	12	12	11	8	6	6	4	3	3	3	3	2	2	1	1	1	1	1	1

⋮ ⋮

d)

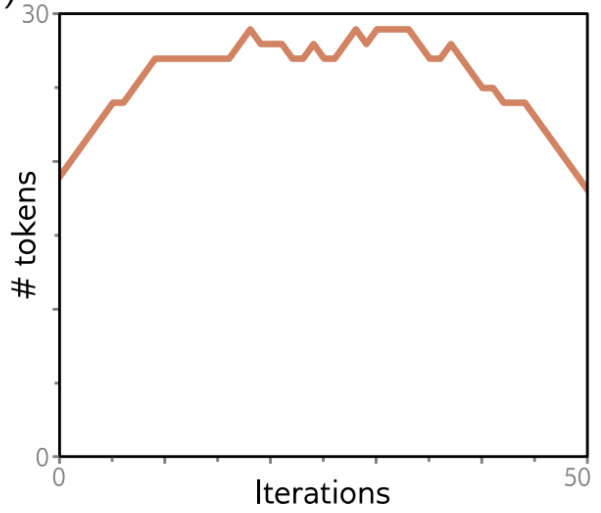
see_	sea_	e	b	l	w	a	could_	hat_	he_	o	t	t_	the_	to_	u	a_	d	f	m	n	p	s	sailor_	to
7	6	4	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1

⋮ ⋮ ⋮

e)

see_	sea_	could_	he_	the_	a_	all_	blue_	bottom_	but_	deep_	of_	sailor_	that_	to_	was_	went_	what_
7	6	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1

f)



Transformers for Natural Language Processing

Embeddings:

- Each token in the vocabulary *V* is mapped to a unique *word embedding* stored in a matrix $\boldsymbol{\Omega}_e \in \mathbb{R}^{D \times |\mathcal{V}|}$

Transformers for Natural Language Processing

Embeddings:

- Each token in the vocabulary *V* is mapped to a unique *word embedding* stored in a matrix $\Omega_e \in \mathbb{R}^{D \times |\mathcal{V}|}$

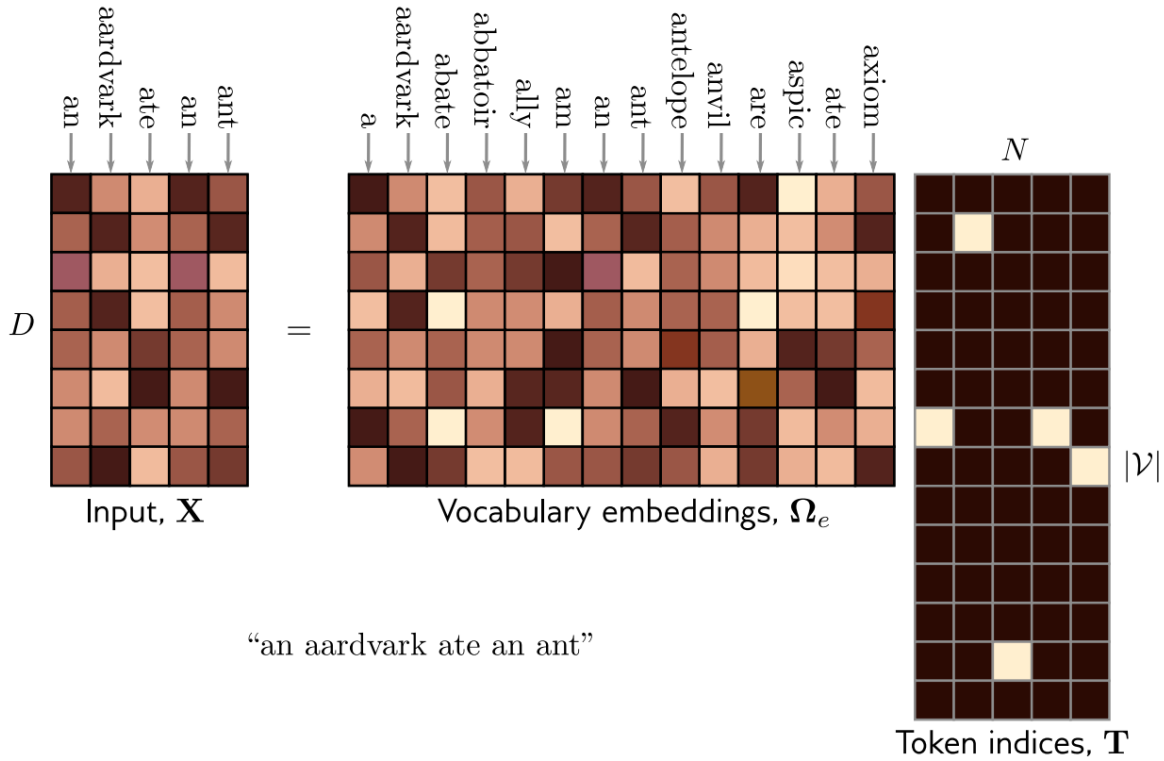
Transformers for Natural Language Processing

Embeddings:

- Each token in the vocabulary *V* is mapped to a unique *word embedding* stored in a matrix $\mathbf{\Omega}_e \in \mathbb{R}^{D \times |\mathcal{V}|}$

Learning embeddings:

- Create the one-hot vector-based matrix $\mathbf{T} \in \mathbb{R}^{|\mathcal{V}| \times N}$
- Learn the word embedding $\mathbf{X} = \mathbf{\Omega}_e \mathbf{T}$ like any other network parameter



Transformers for Natural Language Processing

Transformer models:

- Encoder: transforms the text embeddings to a representation that can support different tasks
- Decoder: predicts the next token to continue the input text
- Encoder-decoder: used in sequence-to-sequence tasks (e.g., machine translation)

Encoder model example (BERT)

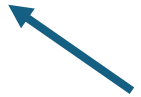
The Bidirectional Encoder Representations from Transformers (BERT):

- Vocabulary: 30,000 tokens
- Word embeddings: 1024 dimensions
- Structure: 24 Transformer layers, each with 16 heads
  64-dimensional queries, keys, values
  MLP layer of 4096 units

Encoder model example (BERT)

The Bidirectional Encoder Representations from Transformers (BERT):

- Vocabulary: 30,000 tokens
- Word embeddings: 1024 dimensions
- Structure: 24 Transformer layers, each with 16 heads
  64-dimensional queries, keys, values
  MLP layer of 4096 units



~340 million parameters

Encoder model example (BERT)

The Bidirectional Encoder Representations from Transformers (BERT):

- Vocabulary: 30,000 tokens
- Word embeddings: 1024 dimensions
- Structure: 24 Transformer layers, each with 16 heads
  64-dimensional queries, keys, values
  MLP layer of 4096 units

- Training using *Transfer Learning*:
  - *Pre-training* using *self-supervision* on a large dataset
  - *Fine-tuning* on a smaller labelled dataset to adapt to a downstream task

Encoder model example (BERT)

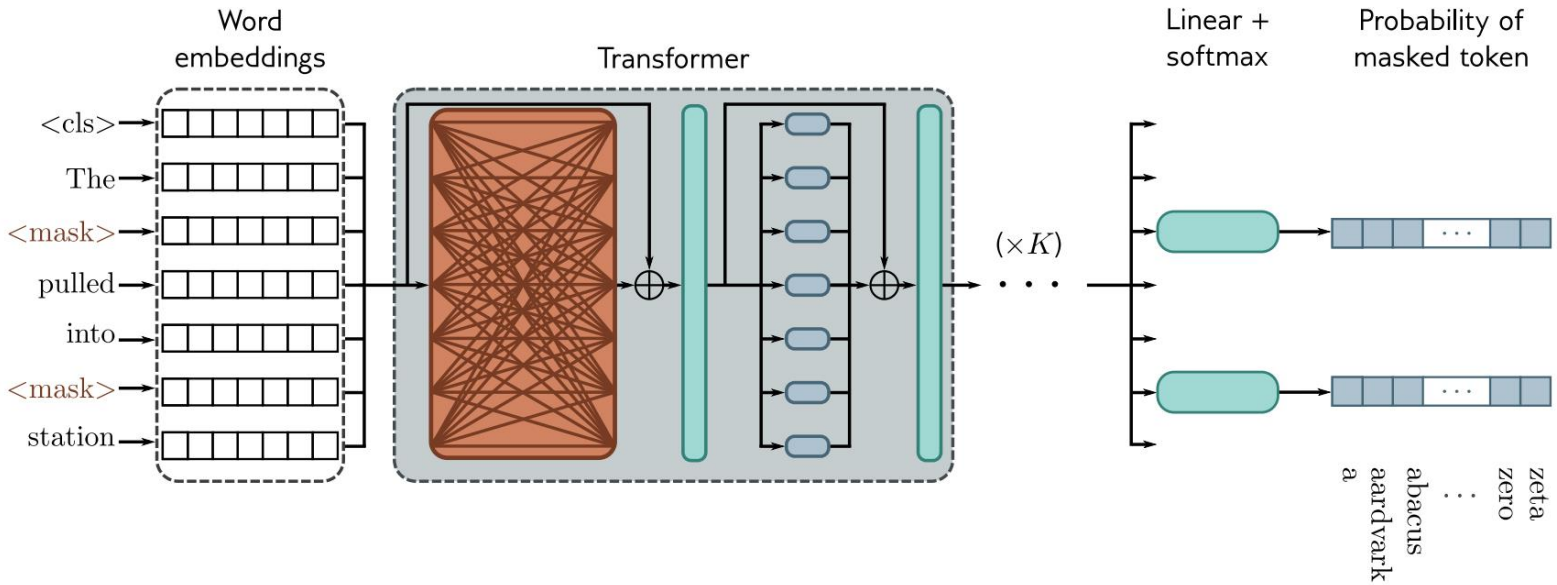
<u>Pre-training using self-supervision:</u>

- Allows for using enormous amounts of data without needing labels

Tasks: 1) Predict missing words from sentences

2) Predict whether two sentences were originally adjacent

- Maximum input length of 512 tokens
- Batch size of 256
- Dataset size: 3.3-billion-word corpus
- 1 million training updates (~50 epochs)



Encoder model example (BERT)

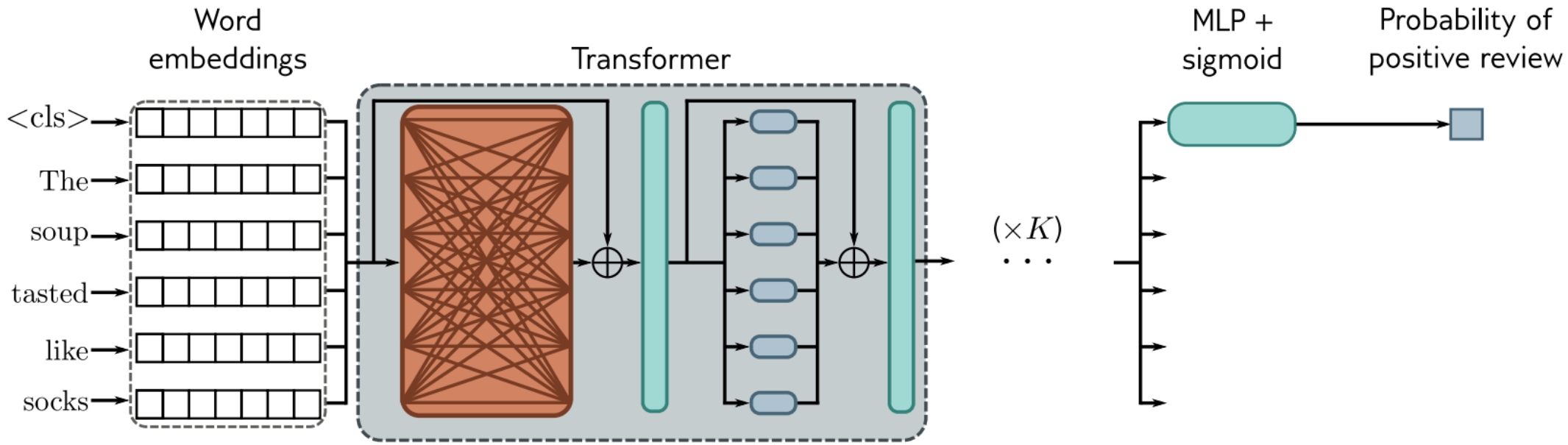
Fine-tuning:

- Initialize the model with the parameters obtained by the pre-training
- Add an extra layer at the end to fit the type and output of the downstream task
- Adjust/update the network parameters with a labeled dataset (some parameters may be frozen / not updated)

Encoder model example (BERT)

Downstream tasks:

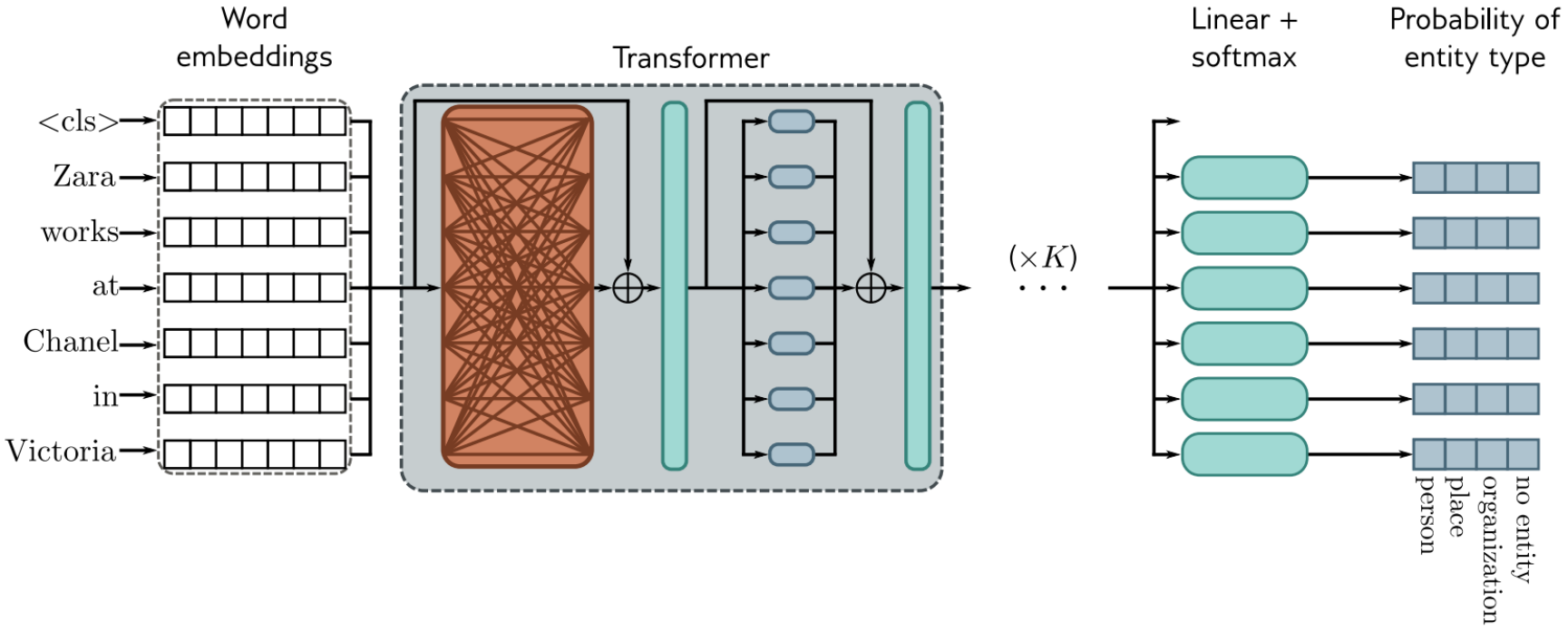
- Text classification: Use a special output <cls> token of the final Transformer layer as input to a classification layer



Encoder model example (BERT)

Downstream tasks:

- Word classification (*named entity recognition*): Classify each output word embedding from the last Transformer layer using a softmax layer

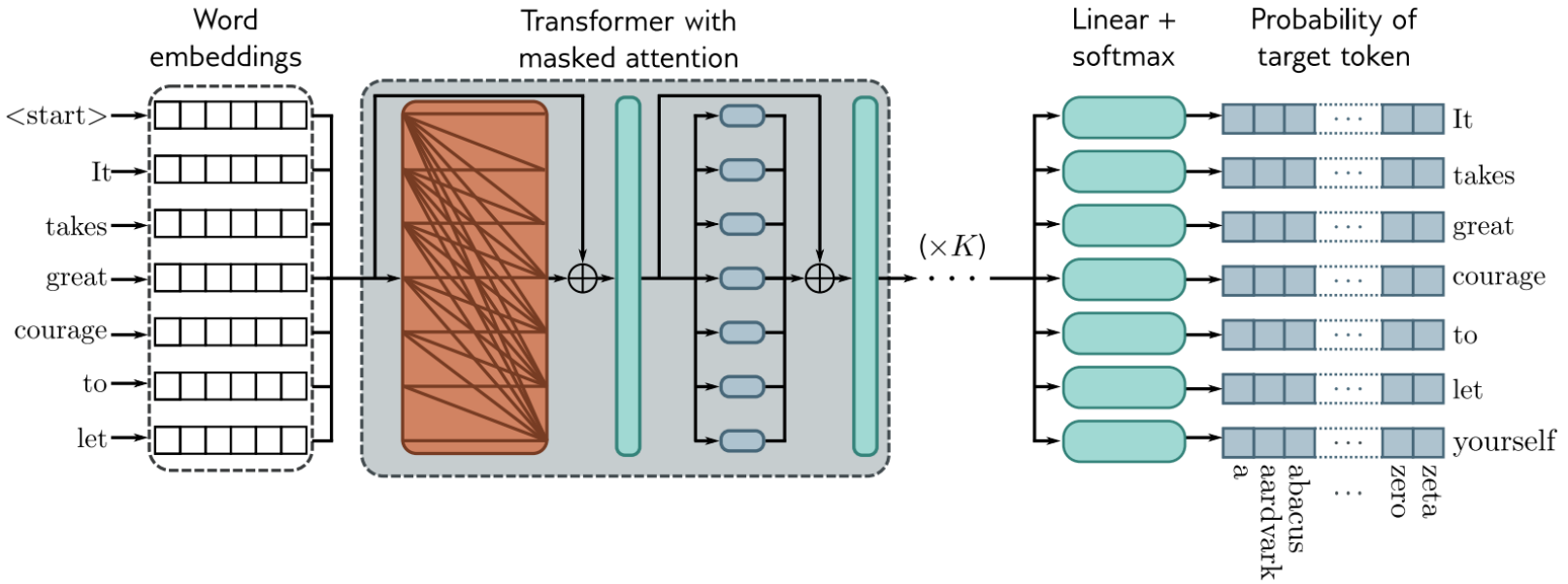


Decoder model example (GPT3)

The Generative Pre-trained Transformer version 3 (GPT3):

- Generates the next token in a sequence

- Sequence length: 2048 tokens
- Total batch size: 3.2 million tokens
- Architecture:
  - 96 Transformer layers
  - word embedding size 12,288
  - 96 heads in the self-attention
  - Value/Key dimension: 128
- Parameter count: 176 billion
- Dataset size: 300-billion-word corpus



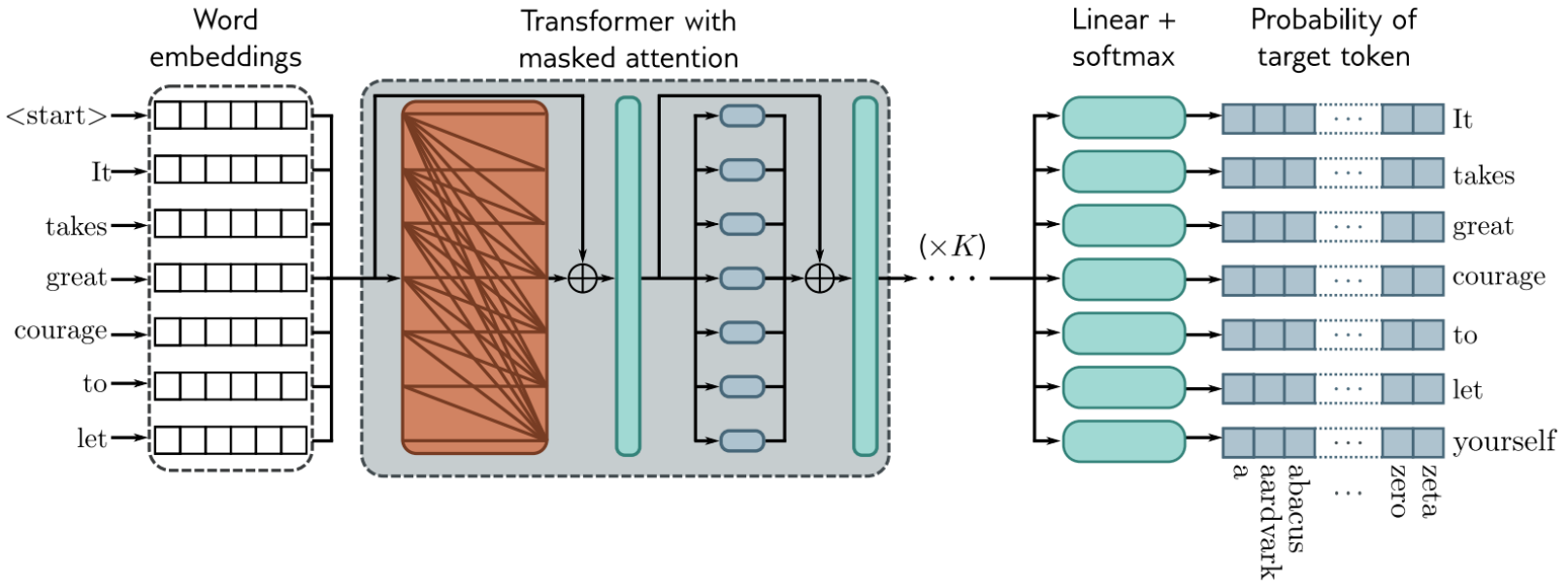
Decoder model example (GPT3)

The Generative Pre-trained Transformer version 3 (GPT3):

- Generates the next token in a sequence

It can generate a coherent text passage by feeding the extended sequence back into the model (*autoregressive model*)

- Sequence length: 2048 tokens
- Total batch size: 3.2 million tokens
- Architecture:
  - 96 Transformer layers
  - word embedding size 12,288
  - 96 heads in the self-attention
  - Value/Key dimension: 128
- Parameter count: 176 billion
- Dataset size: 300-billion-word corpus



Decoder model example (GPT3)

Example: Sentence It takes great courage to let yourself appear weak

$$
\begin{aligned}
&Pr(\text{It takes great courage to let yourself appear weak}) \quad = \\
&\qquad Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\
&\qquad Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\
&\qquad Pr(\text{yourself}|\text{It takes great courage to let}) \times \\
&\qquad Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\
&\qquad Pr(\text{weak}|\text{It takes great courage to let yourself appear}).
\end{aligned}
$$

Decoder model example (GPT3)

Example: Sentence It takes great courage to let yourself appear weak

$$
\begin{aligned}
&Pr(\text{It takes great courage to let yourself appear weak}) \quad = \\
&\qquad Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\
&\qquad Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\
&\qquad Pr(\text{yourself}|\text{It takes great courage to let}) \times \\
&\qquad Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\
&\qquad Pr(\text{weak}|\text{It takes great courage to let yourself appear}).
\end{aligned}
$$

- Predict the conditional probabilities $Pr(t_n|t_1,\ldots,t_{n-1})$ of each token given all the prior tokens.
- Thus, it can calculate the joint probability:

$$Pr(t_1, t_2, \ldots, t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n|t_1, \ldots, t_{n-1})$$

Decoder model example (GPT3)

Masked self-attention:

- Using the full sentence for learning the probability of each token is not appropriate

  E.g., computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ would have access to both:

  - The answer great
  - The right context (rest of the sentence) courage to let yourself appear weak

Decoder model example (GPT3)

Masked self-attention:

- Using the full sentence for learning the probability of each token is not appropriate

  E.g., computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ would have access to both:

  - The answer great
  - The right context (rest of the sentence) courage to let yourself appear weak

- Observation → Tokens interact only in the self-attention layers

Make the attention to the answer and the context equal to zero

Decoder model example (GPT3)

Masked self-attention:

- Using the full sentence for learning the probability of each token is not appropriate

  E.g., computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ would have access to both:
  - The answer great
  - The right context (rest of the sentence) courage to let yourself appear weak
- Observation → Tokens interact only in the self-attention layers

↓

Make the attention to the answer and the context equal to zero

↓

Set the corresponding elements in the self-attention to -∞

Decoder model example (GPT3)

Masked self-attention:

- Using the full sentence for learning the probability of each token is not appropriate

  E.g., computing $\log\left[Pr(\text{great}|\text{It takes})\right]$ would have access to both:
  - The answer great
  - The right context (rest of the sentence) courage to let yourself appear weak
- Observation → Tokens interact only in the self-attention layers

Masked self-attention

Each token attends only to the previous tokens

Make the attention to the answer and the context equal to zero

Set the corresponding elements in the self-attention to -∞

Decoder model example (GPT3)

Generating text:

- Feed with an input sequence text (e.g., a special <start> token)
- The network outputs the probabilities over possible subsequent tokens

Decoder model example (GPT3)

Generating text:

- Feed with an input sequence text (e.g., a special <start> token)
- The network outputs the probabilities over possible subsequent tokens
- Pick the next token of the sequence:
  - Pick the most likely token, or
  - Sample from the output probability distribution
- Introduce the extended sequence to the network

Decoder model example (GPT3)

Generating text:

- Feed with an input sequence text (e.g., a special <start> token)
- The network outputs the probabilities over possible subsequent tokens
- Pick the next token of the sequence:
  - Pick the most likely token, or
  - Sample from the output probability distribution
- Introduce the extended sequence to the network

Prior embeddings do not depend on subsequent ones due to the masked self-attention

Earlier computations can be recycled as we generate subsequent tokens (KV-cache)

Encoder-decoder model example

Machine translation from English to French:

- The Encoder receives the sentence in English and outputs a representation for each input token

Encoder-decoder model example

Machine translation from English to French:

- The Encoder receives the sentence in English and outputs a representation for each input token
- The Decoder:
  - receives the ground truth translation in French and passes it through a series of Transformer layers (that use masked self-attention) to predict the following word at each position
  - each of its layers attends to the output of the Encoder

Encoder-decoder model example

Machine translation from English to French:

- The Encoder receives the sentence in English and outputs a representation for each input token
- The Decoder:
  - receives the ground truth translation in French and passes it through a series of Transformer layers (that use masked self-attention) to predict the following word at each position
  - each of its layers attends to the output of the Encoder

Each French word is conditioned on:

- the previous output words
- the source English sentence

Cross-Attention

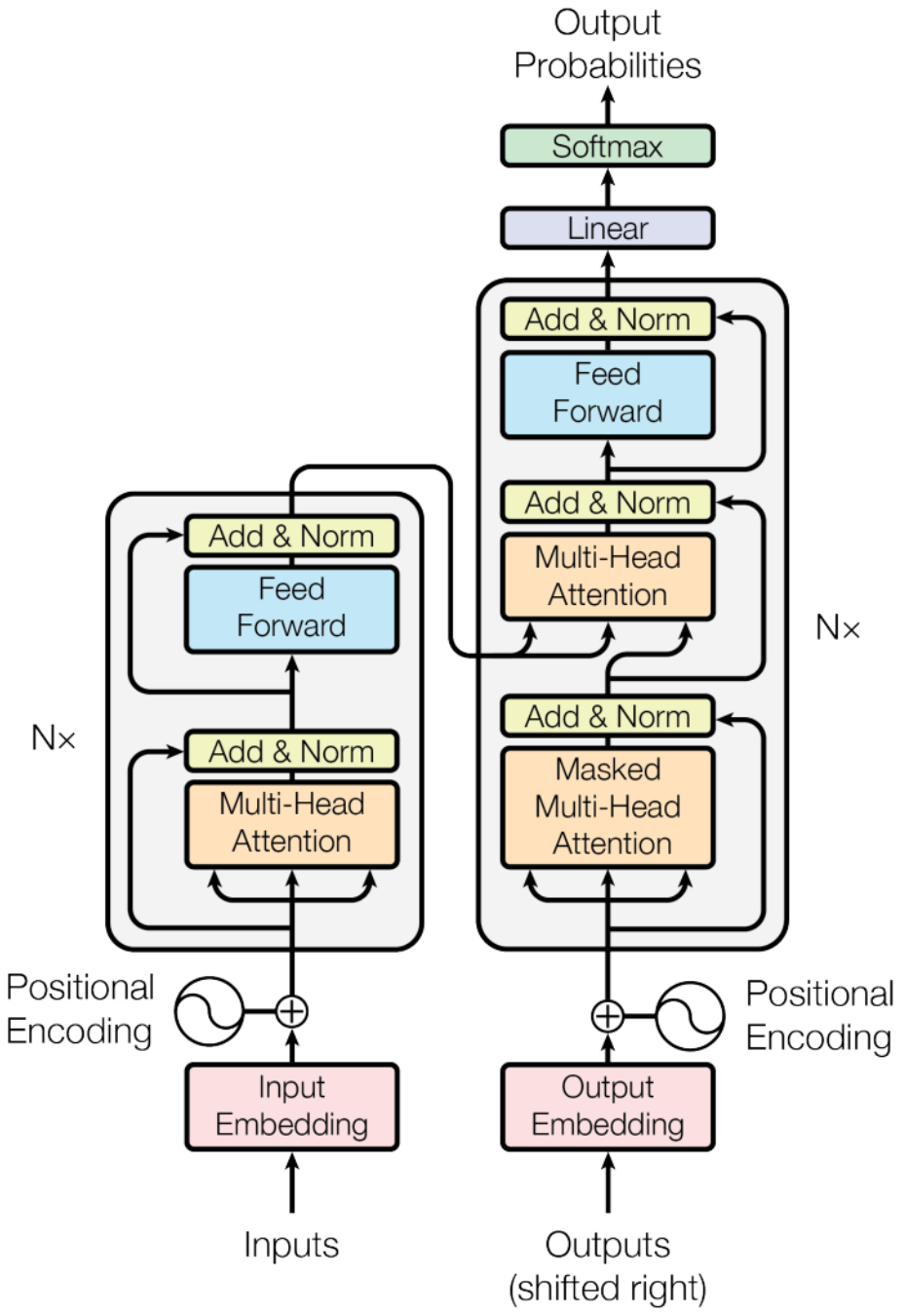
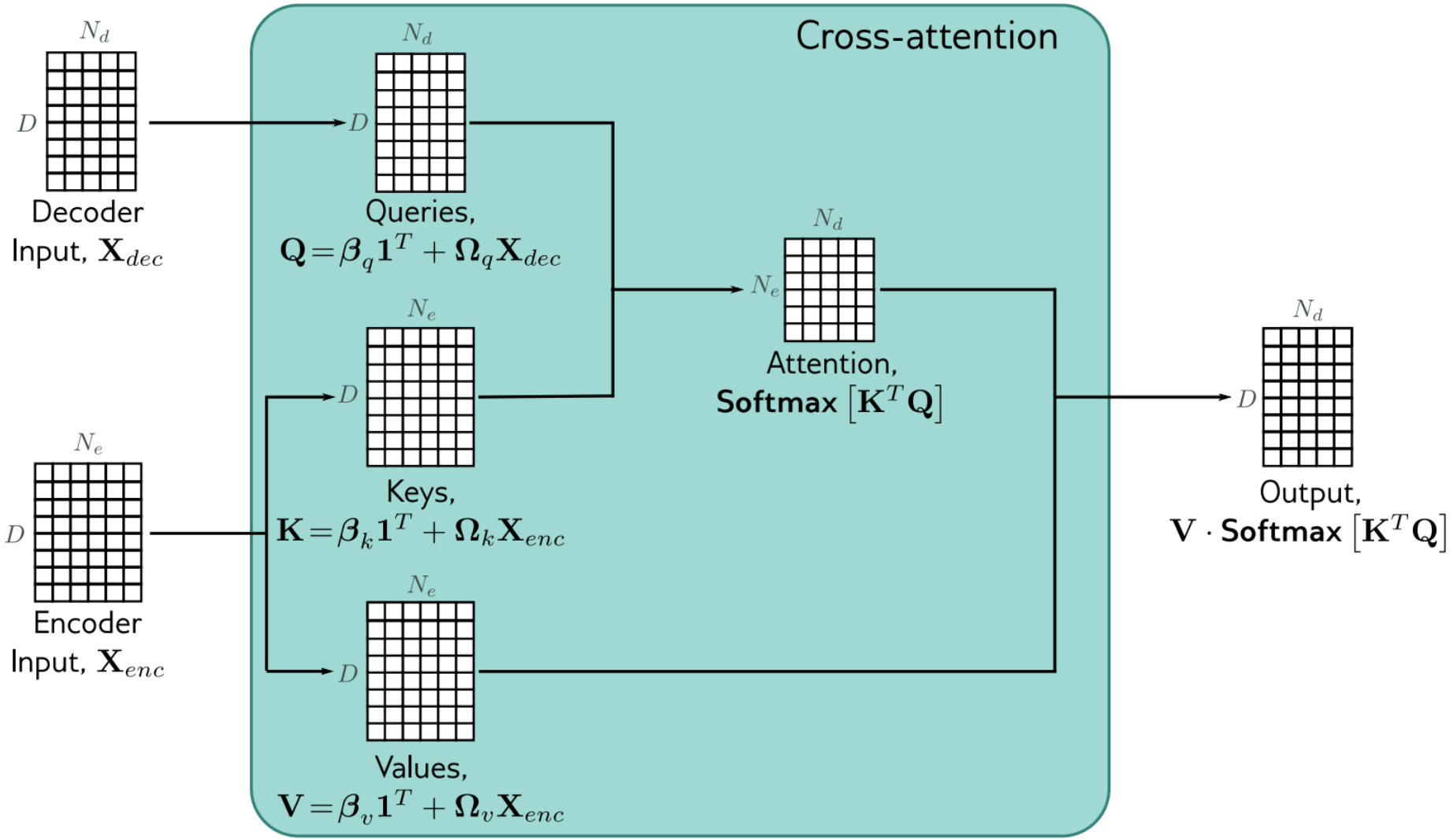


Figure 1: The Transformer - model architecture.

Vaswani et al., “Attention is all you need”, NIPS 2017

Cross-Attention



Processing long sequences

Computational complexity of self-attention scales quadratically with the length of the sequence

→ impractical for long sequences

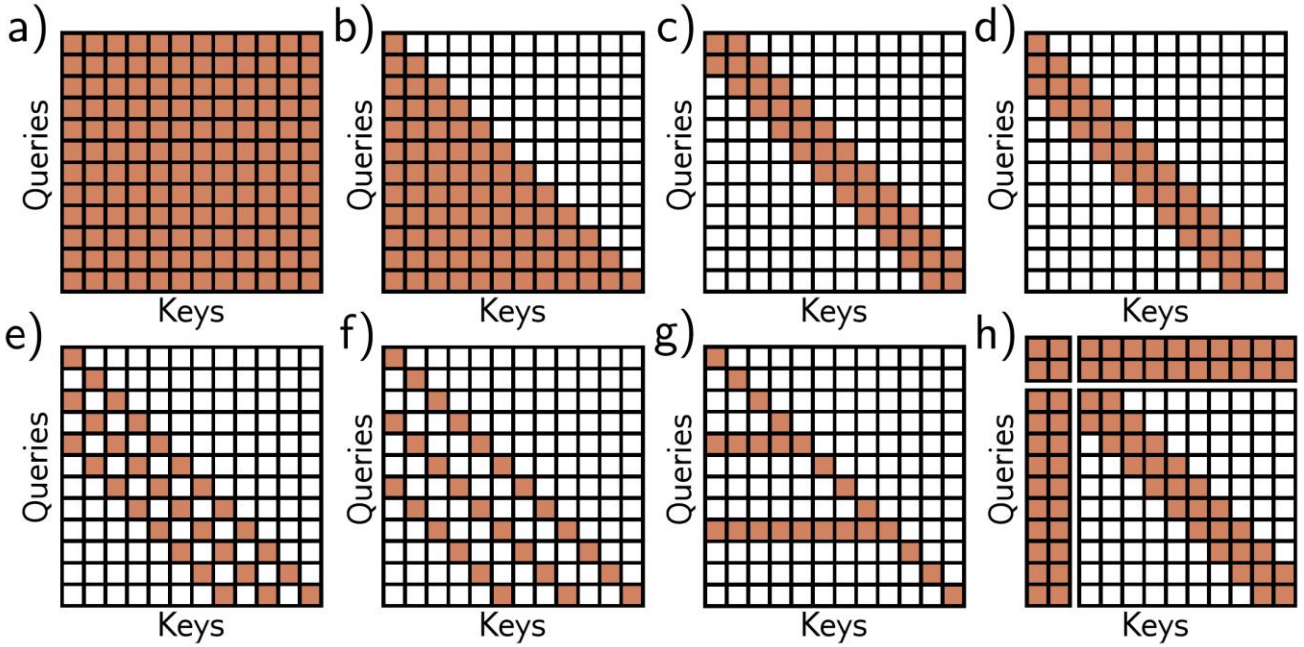


Figure 12.15 Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token, and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case). d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case). g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case pictured). h) Alternatively, global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

Processing long sequences

Note!
Here the input (and all the rest matrices) is row-matrix

Example model: Nyströmformer approximates the Scaled Dot-Product Attention with low-rank approximation:

$$\mathrm{s}\left(\frac{QK^T}{\sqrt{d}}\right) \approx \mathrm{s}\left(\frac{Q\tilde{K}^T}{\sqrt{d}}\right) \mathrm{s}\left(\frac{\tilde{Q}\tilde{K}^T}{\sqrt{d}}\right)^{\dagger} \mathrm{s}\left(\frac{\tilde{Q}K^T}{\sqrt{d}}\right)$$

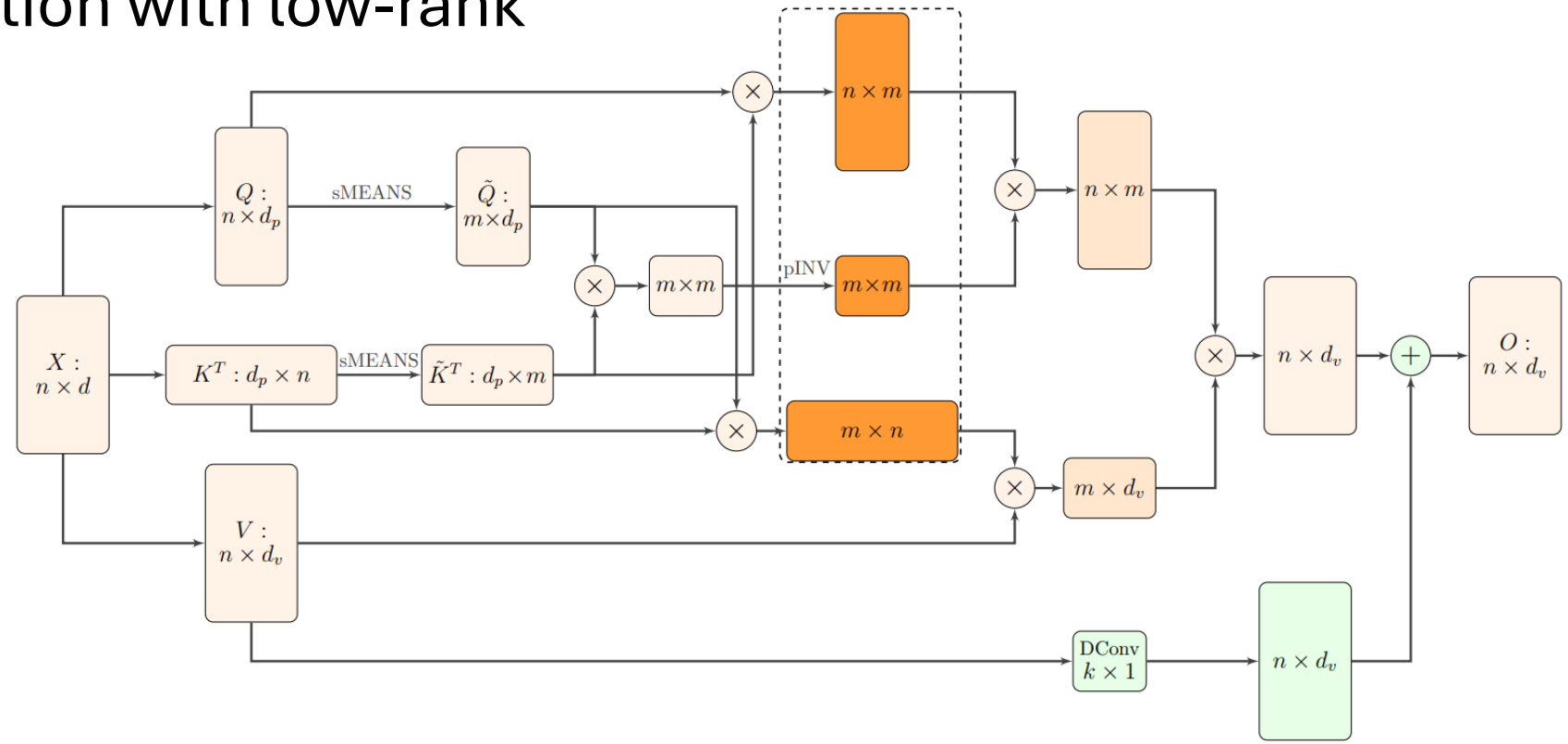
where $\Omega^{\dagger}$ is the Moore-Penrose pseudo-inverse of matrix Ω, and $\tilde{Q}$ and $\tilde{K}$ are matrices formed by sets of m landmarks, computed as the segment means of the matrices Q and K, respectivel

Xiong et al., “Nyströmformer: A Nyström-based Algorithm for Approximating Self-Attention”, AAAI 2021

Processing long sequences

Note!
Here the input (and all the rest matrices) is row-matrix

Example model: Nyströmformer approximates the Scaled Dot-Product Attention with low-rank



Xiong et al., “Nyströmformer: A Nyström-based Algorithm for Approximating Self-Attention”, AAAI 2021

Transformers for images

ImageGPT is a Transformer decoder:

- Builds an autoregressive model:
  - Input: partial image
  - Output: next pixel value
- Largest model (6.8 billion parameters) can operate on 64x64 pixel images and nine-bit color space quantization, so the system ingests (and predicts) one of 512 possible tokens at each position

Transformers for images

ImageGPT

Image generation

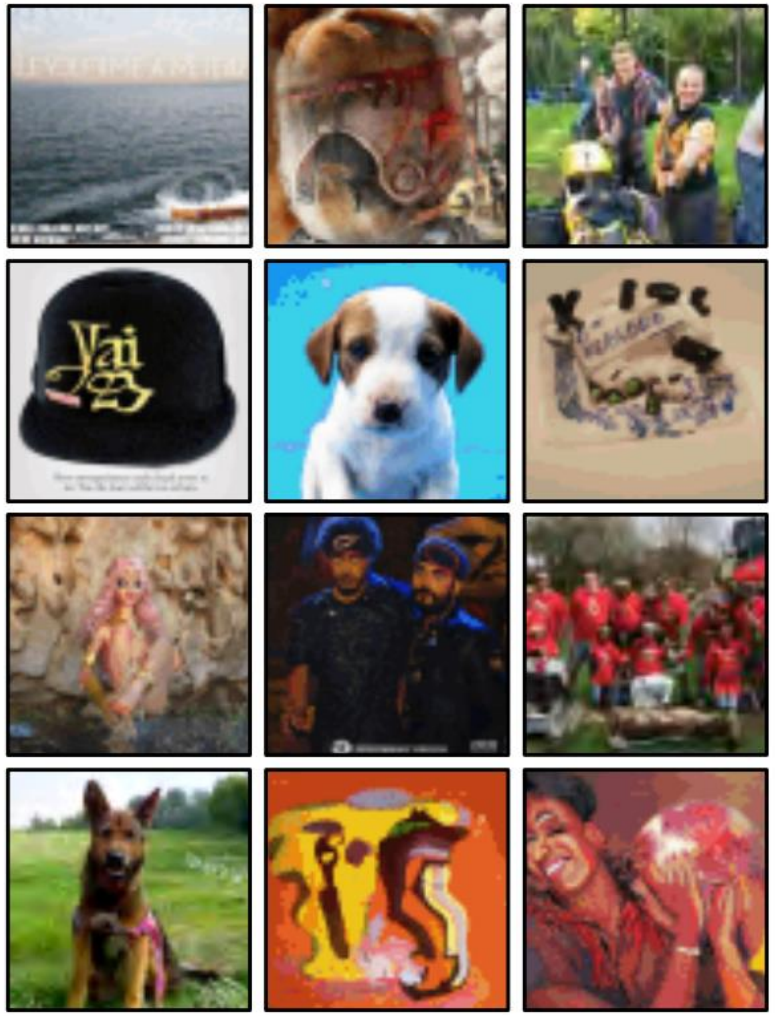
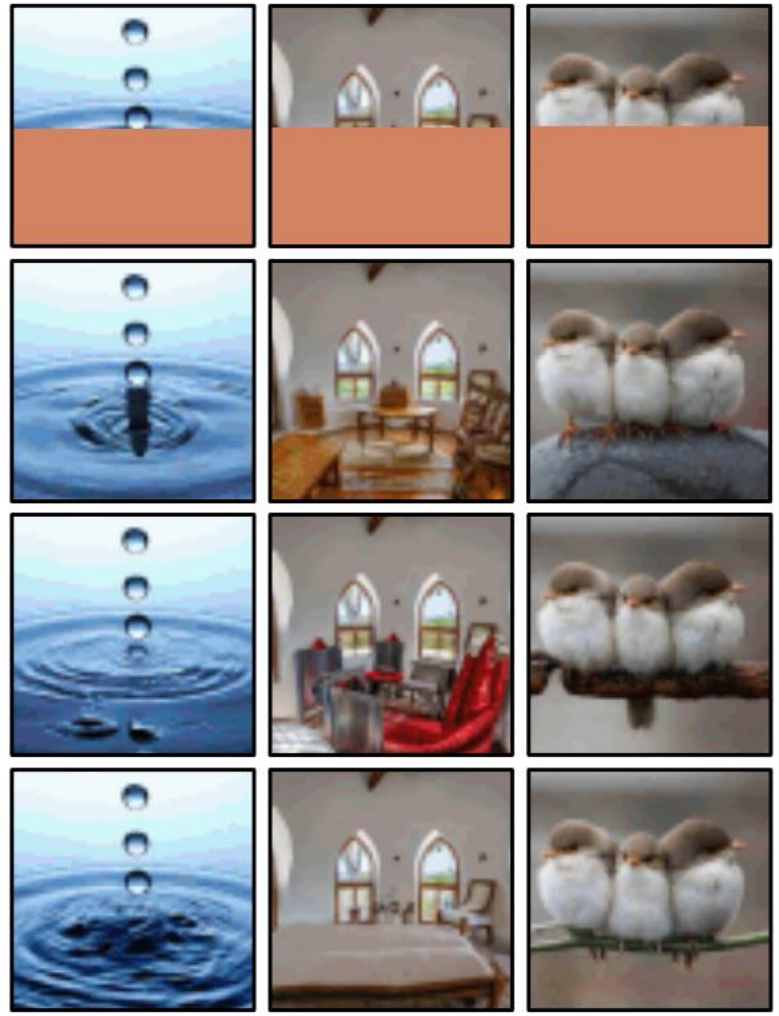


Image completion



Transformers for images

Vision Transformer (ViT) is a Transformer encoder:

- Divides the image into 16x16 patches
- Each patch is mapped to an input embedding (learnable)
- Resulting embeddings are introduced to the Transformer network
- Standard 1D positional encodings are learned

Transformers for images

Vision Transformer (ViT) is a Transformer encoder:

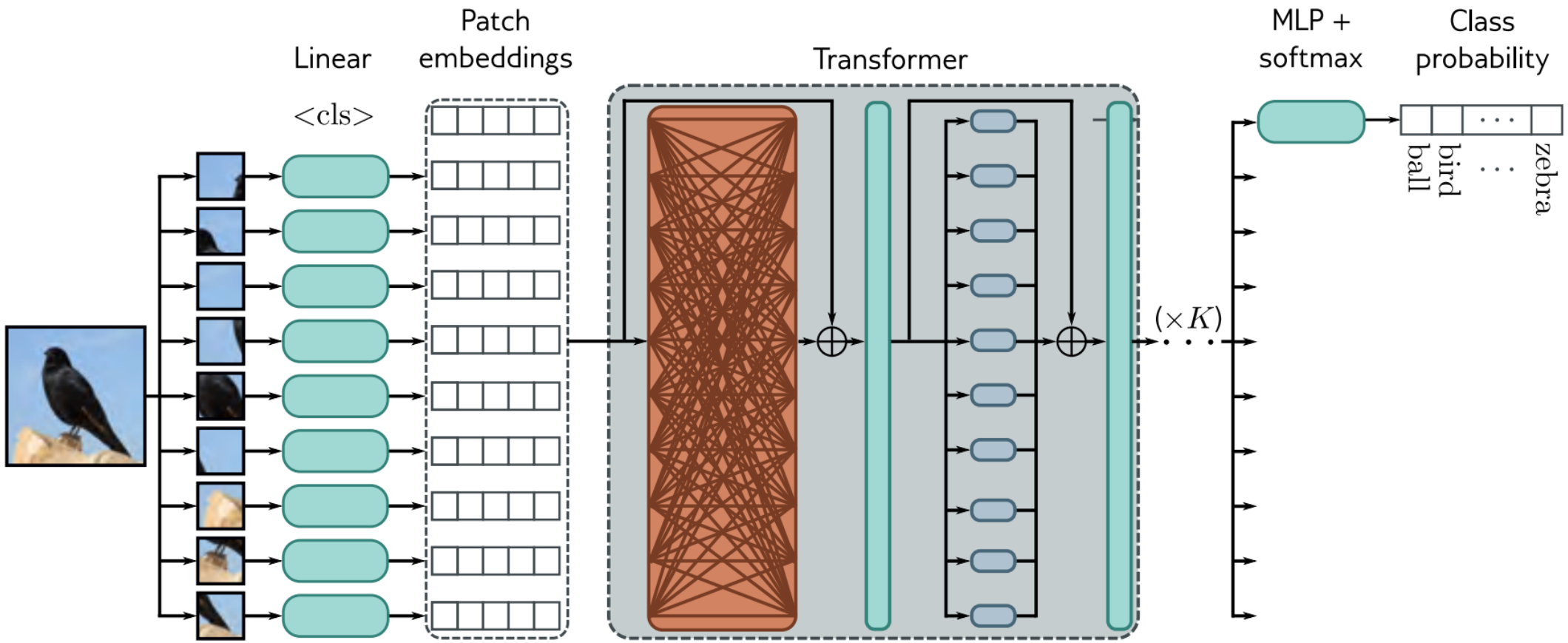
- Divides the image into 16x16 patches
- Each patch is mapped to an input embedding (learnable)
- Resulting embeddings are introduced to the Transformer network
- Standard 1D positional encodings are learned

Used supervised pre-training on a dataset with

- 303 million labeled images
- 18,000 classes

Transformers for images

Vision Transformer (ViT)

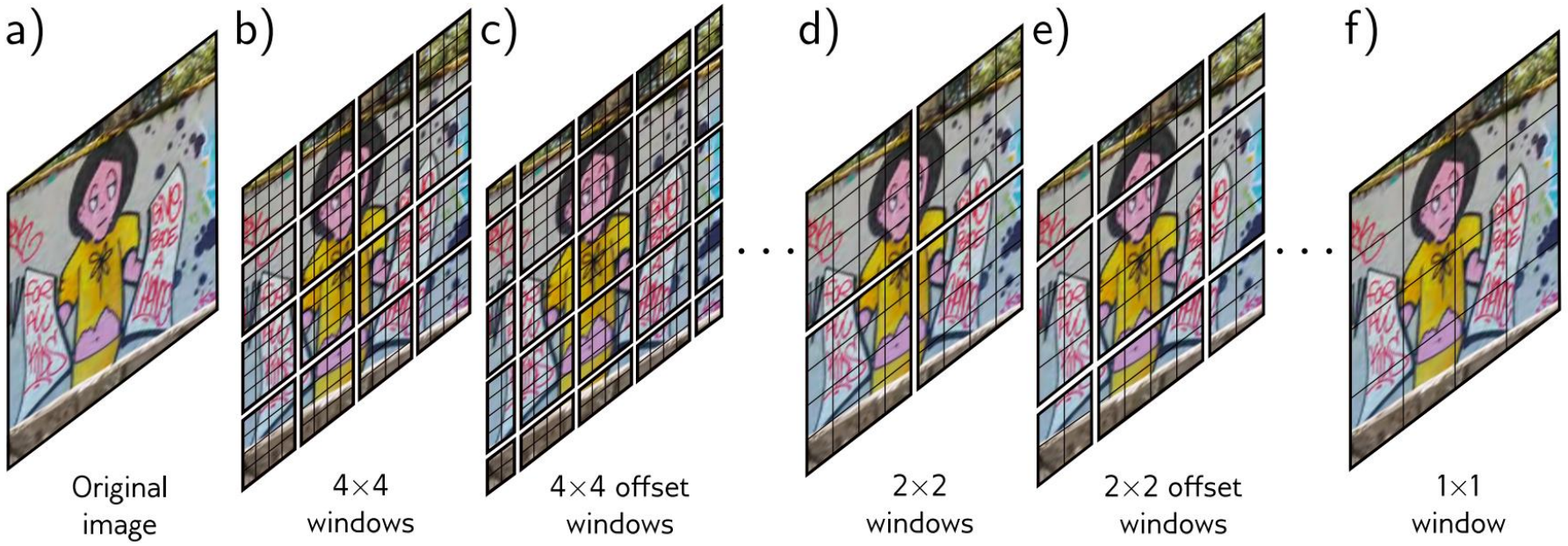


Transformers for images

Multi-scale Vision Transformer:

- Start with small high-resolution patches
- Gradually enlarge the receptive field, decrease spatial resolution

Example: Shifted-window Transformer (Swin Transformer)



Transformers for videos

Video Vision Transformer (ViViT)

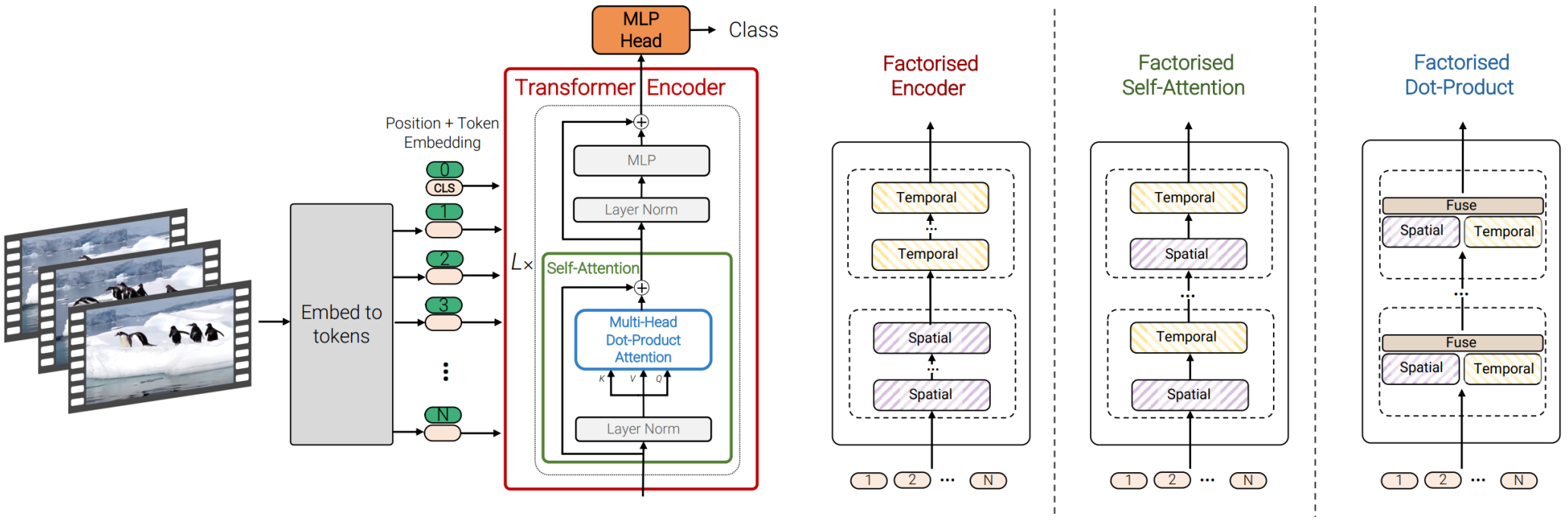


Figure 1: We propose a pure-transformer architecture for video classification, inspired by the recent success of such models for images [18]. To effectively process a large number of spatio-temporal tokens, we develop several model variants which factorise different components of the transformer encoder over the spatial- and temporal-dimensions. As shown on the right, these factorisations correspond to different attention patterns over space and time.

Arnab et al., "ViViT: A Video Vision Transformer", ICCV 2021

Transformers for videos

Video Vision Transformer (ViViT)

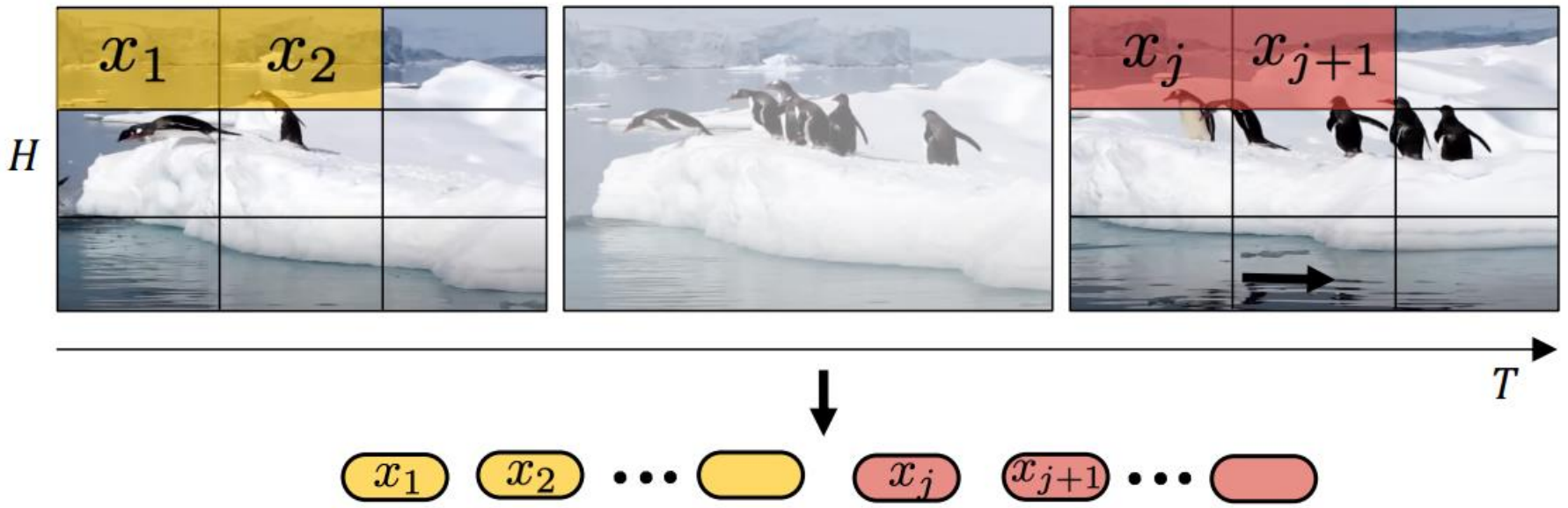


Figure 2: Uniform frame sampling: We simply sample n_t frames, and embed each 2D frame independently following ViT [17].

Arnab et al., “ViViT: A Video Vision Transformer”, ICCV 2021

Transformers for videos

Video Vision Transformer (ViViT)

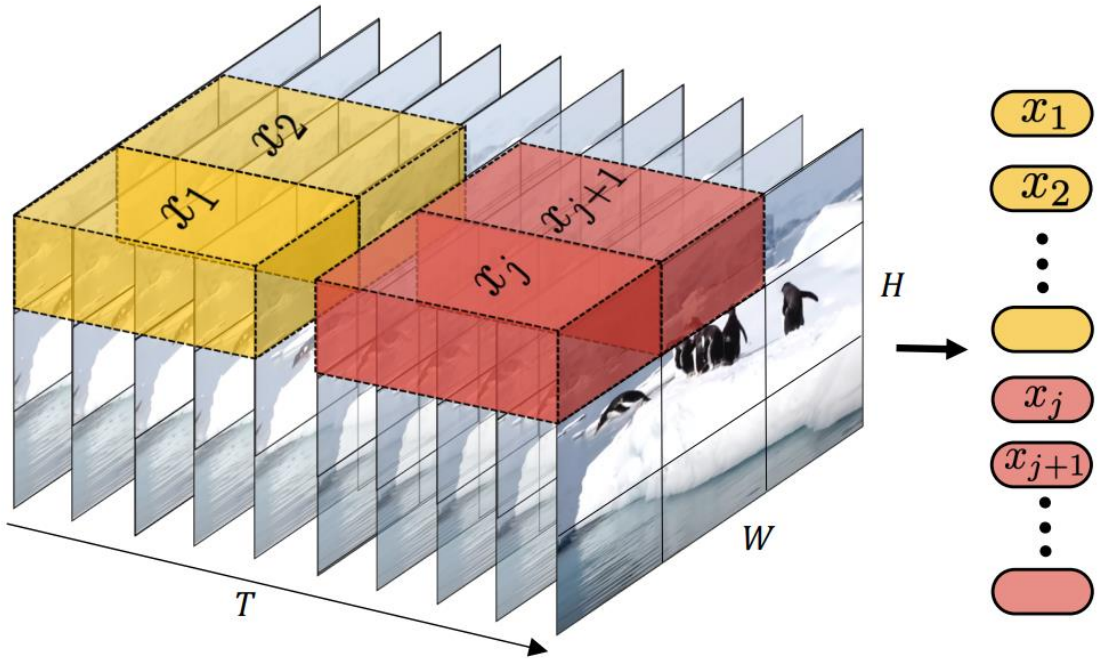


Figure 3: Tubelet embedding. We extract and linearly embed non-overlapping tubelets that span the spatio-temporal input volume.

Arnab et al., “ViViT: A Video Vision Transformer”, ICCV 2021

Transformers for videos

Video Vision Transformer (ViViT)

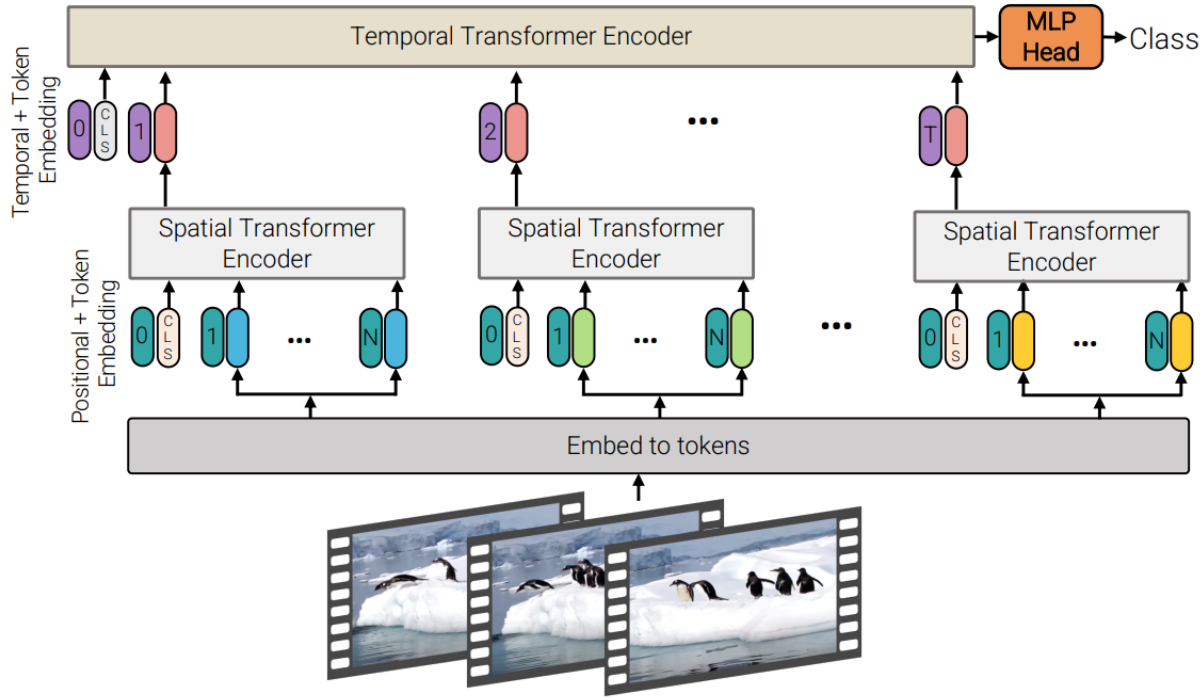


Figure 4: Factorised encoder (Model 2). This model consists of two transformer encoders in series: the first models interactions between tokens extracted from the same temporal index to produce a latent representation per time-index. The second transformer models interactions between time steps. It thus corresponds to a "late fusion" of spatial- and temporal information.

Arnab et al., "ViViT: A Video Vision Transformer", ICCV 2021

Transformers for videos

Video Vision Transformer (ViViT)

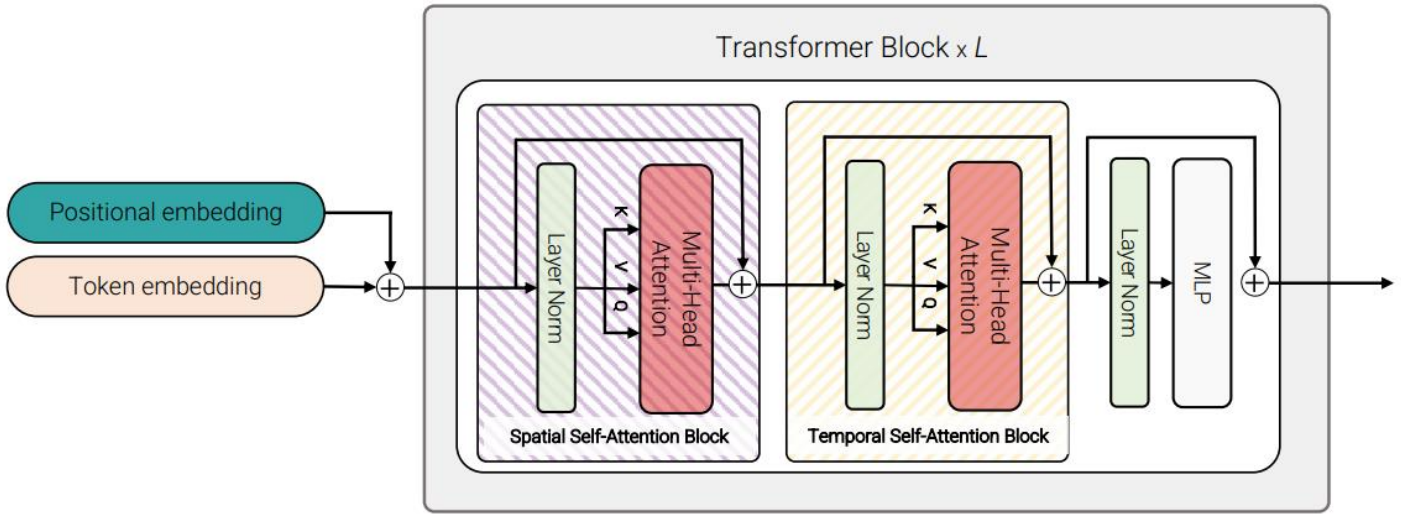


Figure 5: Factorised self-attention (Model 3). Within each transformer block, the multi-headed self-attention operation is factorised into two operations (indicated by striped boxes) that first only compute self-attention spatially, and then temporally.

Arnab et al., “ViViT: A Video Vision Transformer”, ICCV 2021

Continual Inference Networks

A deep neural network, which:

- is capable of continual step inference without computational redundancy,
- is capable of batch inference corresponding to a non-continual neural network,
- produces identical outputs for batch- and step inference given identical receptive fields,
- uses one set of trainable parameters for both batch and step inference

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023

Continual Transformer Encoder

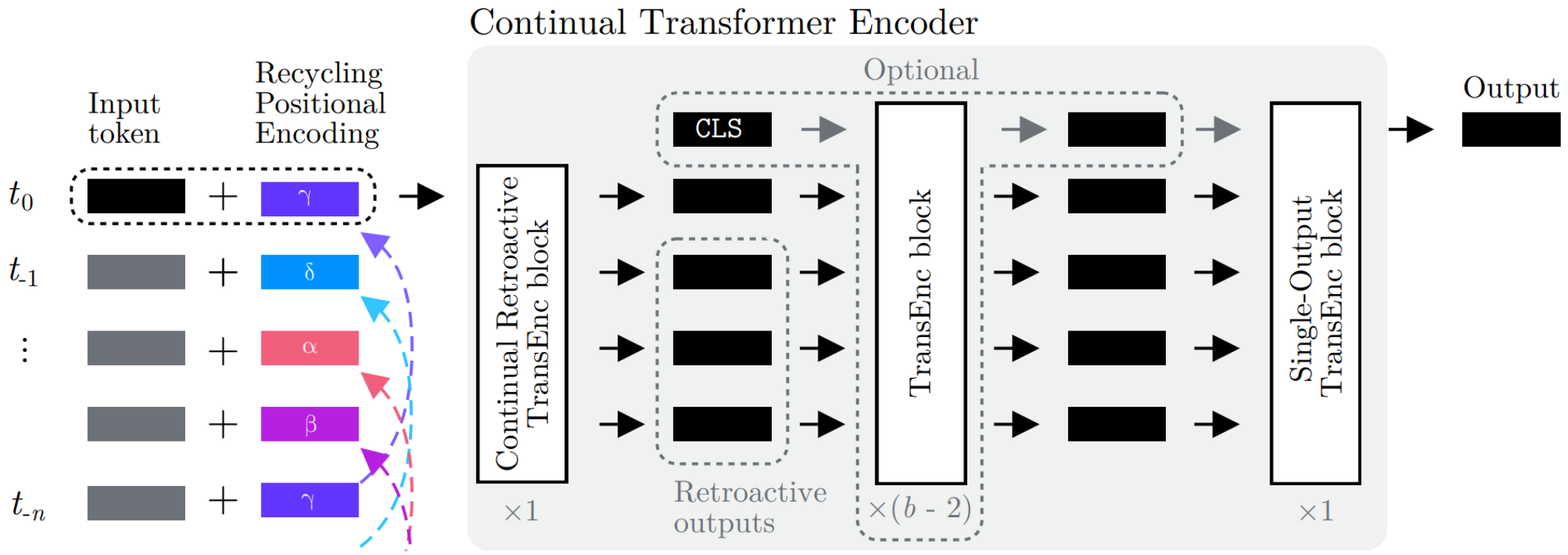


Figure 1: **Multi-block Continual Transformer Encoder with Recycling Positional Encoding.** For $b > 2$ blocks, regular Transformer Encoder blocks can be added between an initial Continual Retroactive block and a final Single-Output block. A class-token may be used after the initial block.

Hedegaard, Bakhtiarnia, Iosifidis, "Continual Transformers: Redundancy-Free Attention for Online Inference", ICLR 2023

Continual Transformer Encoder

Regular scaled dot-product attention:

- Given query, key, and value matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{n \times d}$, it can be defined as:

$$\mathrm{Att}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{V} \qquad \boldsymbol{A} = \exp\left(\boldsymbol{Q}\boldsymbol{K}^{\top}/\sqrt{d}\right) \qquad \boldsymbol{D} = \mathrm{diag}\left(\boldsymbol{A}\mathbb{1}_n^{\top}\right)$$

  where $\boldsymbol{A}, \boldsymbol{D} \in \mathbb{R}^{n \times n}$ and $\mathbb{1}_n$ is a row vector of ones.

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023

Continual Transformer Encoder

Continual retroactive scaled dot-product attention:

We can compute $\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{V}$ in a step-wise manner using the latest query, key, and value steps, $\boldsymbol{q}_{\text{new}}, \boldsymbol{k}_{\text{new}}, \boldsymbol{v}_{\text{new}} \in \mathbb{R}^{1\times d}$, alongside appropriately cached partial results. The softmax normalisation with $\boldsymbol{D}^{-1}$ can be efficiently implemented via column-aligned element-wise multiplications (denoted by $\odot$ hereafter) of a column-vector $\boldsymbol{d} = \boldsymbol{A}\mathbb{1}_n^\top$. If we cache the $n-1$ values for the prior step tokens, i.e. $\boldsymbol{d}_{\text{mem}} = \boldsymbol{A}_{\text{prev}}^{(-n+1:-1)}\mathbb{1}_{n-1}^\top$, alongside $\boldsymbol{Q}$ and $\boldsymbol{K}$, we can define the step update as:

$$\boldsymbol{d}^{(-n+1:-1)} = \boldsymbol{d}_{\text{mem}}^{(-n+2:0)} - \exp\left(\boldsymbol{Q}_{\text{mem}}\boldsymbol{k}_{\text{old}}^\top\right) + \exp\left(\boldsymbol{Q}_{\text{mem}}\boldsymbol{k}_{\text{new}}^\top\right)$$

$$\boldsymbol{d}^{(0)} = \exp\left(\frac{\boldsymbol{q}_{\text{new}}}{\sqrt{d}}\left(\boldsymbol{K}_{\text{mem}} \parallel \boldsymbol{k}_{\text{new}}\right)^\top\right)\mathbb{1}_n^\top,$$

where $\boldsymbol{Q}_{\text{mem}}$ ($\boldsymbol{K}_{\text{mem}}$) are the $n-1$ prior query (key) tokens, $\boldsymbol{k}_{\text{old}}$ is the key from n steps ago, and $\parallel$ denotes concatenation of matrices along the first dimension. Negative indices indicate prior time-steps.

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023

Continual Transformer Encoder

Continual retroactive scaled dot-product attention:

An update for $\boldsymbol{AV}$ can likewise be defined as a function of the $n-1$ prior values $\boldsymbol{AV}_{\text{mem}}$:

$$\boldsymbol{AV}^{(-n+1:-1)} = \boldsymbol{AV}_{\text{mem}}^{(-n+2:0)} - \exp\left(\boldsymbol{Q}_{\text{mem}}\boldsymbol{k}_{\text{old}}^{\top}\right)\boldsymbol{v}_{\text{old}} + \exp\left(\boldsymbol{Q}_{\text{mem}}\boldsymbol{k}_{\text{new}}^{\top}\right)\boldsymbol{v}_{\text{new}}$$

$$\boldsymbol{AV}^{(0)} = \exp\left(\frac{\boldsymbol{q}_{\text{new}}}{\sqrt{d}}\left(\boldsymbol{K}_{\text{mem}} \parallel \boldsymbol{k}_{\text{new}}\right)^{\top}\right)\left(\boldsymbol{V}_{\text{mem}} \parallel \boldsymbol{v}_{\text{new}}\right).$$

Finally, we compute the Continual Retroactive Attention output in the usual manner:

$$CoReAtt(\boldsymbol{q}_{\text{new}}, \boldsymbol{k}_{\text{new}}, \boldsymbol{v}_{\text{new}}) = \boldsymbol{d}^{-1} \odot \boldsymbol{AV}.$$

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023

Continual Transformer Encoder

Continual single-output scaled dot-product attention:

Both the Regular and Continual Retroactive Dot-Product Attentions produce attention outputs for the current step, as well as $n-1$ retroactively updated steps. In cases where retroactive updates are not needed, we can simplify the computation greatly via a Continual Single-Output Dot-Product Attention ($CoSi$Att). In essence, the regular SDA is reused, but prior values of $\mathbf{k}$ and $\mathbf{v}$ are cached between steps (as in (Ott et al., 2019)), and only the attention corresponding to a single query token $\boldsymbol{q}$ is computed:

$$CoSi\text{Att}(\boldsymbol{q}, \boldsymbol{k}_{\text{new}}, \boldsymbol{v}_{\text{new}}) = \mathbf{a}\left(\boldsymbol{V}_{\text{mem}} \parallel \boldsymbol{v}_{\text{new}}\right) / \mathbf{a}\mathbb{1}_n^\top, \qquad \mathbf{a} = \exp\left(\frac{\boldsymbol{q}}{\sqrt{d}}\left(\boldsymbol{K}_{\text{mem}} \parallel \boldsymbol{k}_{\text{new}}\right)^\top\right).$$

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023

Continual Transformer Encoder

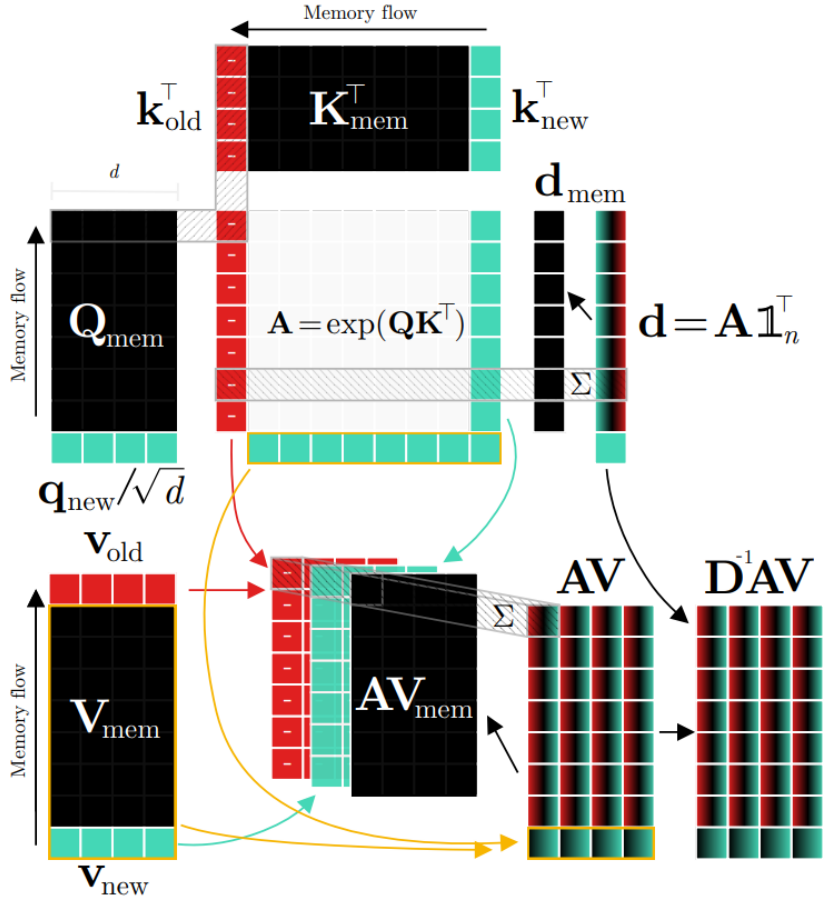


Figure 4: **Continual Retroactive Dot-Product Attention**. The query ($\boldsymbol{Q}$), key ($\boldsymbol{K}$), and value ($\boldsymbol{V}$) matrices are aggregated over time by caching the step vectors $\boldsymbol{q}_{\text{new}}$, $\boldsymbol{k}_{\text{new}}$, and $\boldsymbol{v}_{\text{new}}$ in each their FIFO queue (denoted by $\square_{\text{mem}}$). During each step, only the entries of $\boldsymbol{A}$ associated with $\boldsymbol{q}_{\text{new}}$, $\boldsymbol{k}_{\text{new}}$ and the oldest $\boldsymbol{K}$ step, $\boldsymbol{k}_{\text{old}}$ are computed. The diagonal entries of the row-normalisation matrix $\boldsymbol{D}$ as well as the $\boldsymbol{AV}$ can be updated retroactively by subtracting features corresponding to $\boldsymbol{k}_{\text{old}}$ and adding features related to $\boldsymbol{k}_{\text{new}}$ to the cached outputs of the previous step, $\boldsymbol{D}_{\text{mem}}$ and $\boldsymbol{AV}_{\text{mem}}$, respectively.

Hedegaard, Bakhtiarnia, Iosifidis, “Continual Transformers: Redundancy-Free Attention for Online Inference”, ICLR 2023