

Introduction to Transformers

Transformers

LLMs are built out of transformers

Transformer: a specific kind of network architecture, like a fancier feedforward network, but based on attention

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaizer@google.com

Illia Polosukhin* †
illia.polosukhin@gmail.com

A very approximate timeline

1990 Static Word Embeddings

2003 Neural Language Model

2008 Multi-Task Learning

2015 Attention

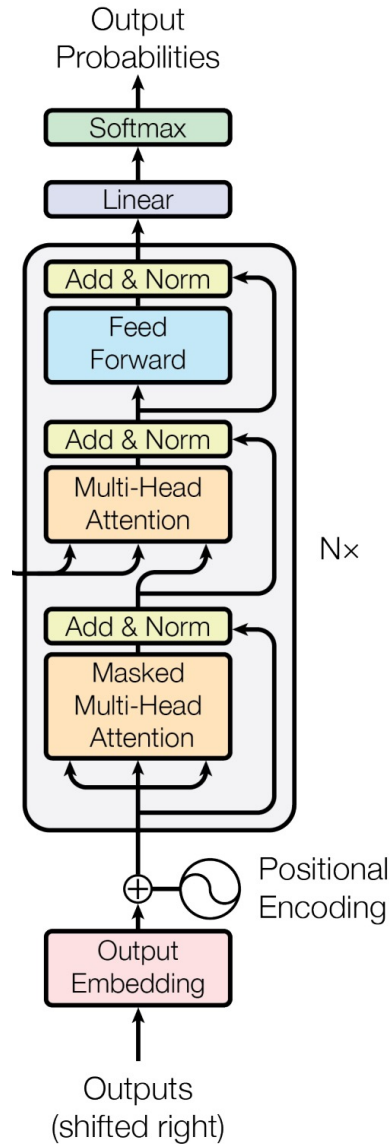
2017 Transformer

2018 Contextual Word Embeddings and Pretraining

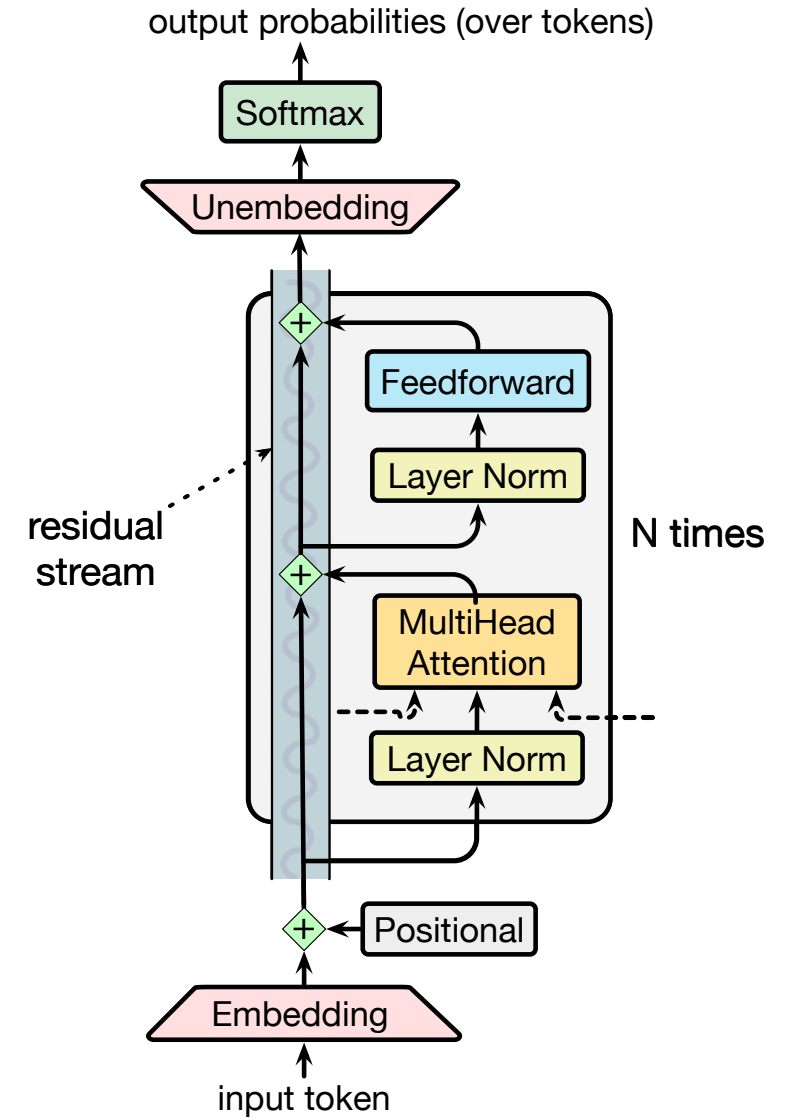
2019 Prompting

The transformer

Postnorm



Prenorm

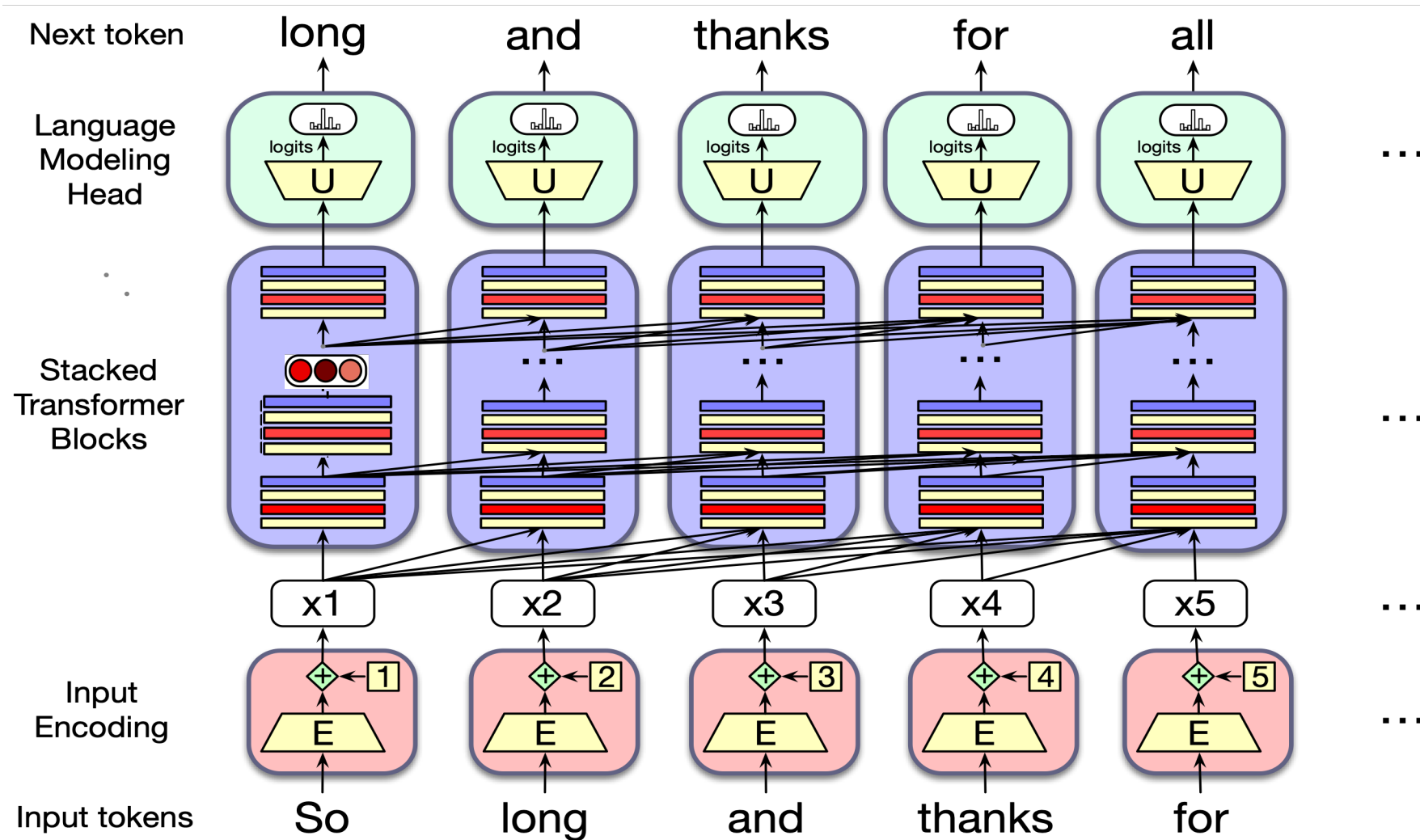


Attention

Transformers

Instead of starting with the big picture

Let's consider the embeddings for an individual word from a particular layer



Problem with static embeddings (word2vec)

They are static! The embedding for a word doesn't reflect how its meaning changes in context.

The chicken didn't cross the road because  it was too tired

What is the meaning represented in the static embedding for "it"?

Contextual Embeddings

- Intuition: a representation of meaning of a word should be different in different contexts!
- **Contextual Embedding:** each word has a different vector that expresses different meanings depending on the surrounding words
- How to compute contextual embeddings?
 - **Attention**

Contextual Embeddings

The chicken didn't cross the road because it

What should be the properties of "it"?

The chicken didn't cross the road because it was too **tired**

The chicken didn't cross the road because it was too **wide**

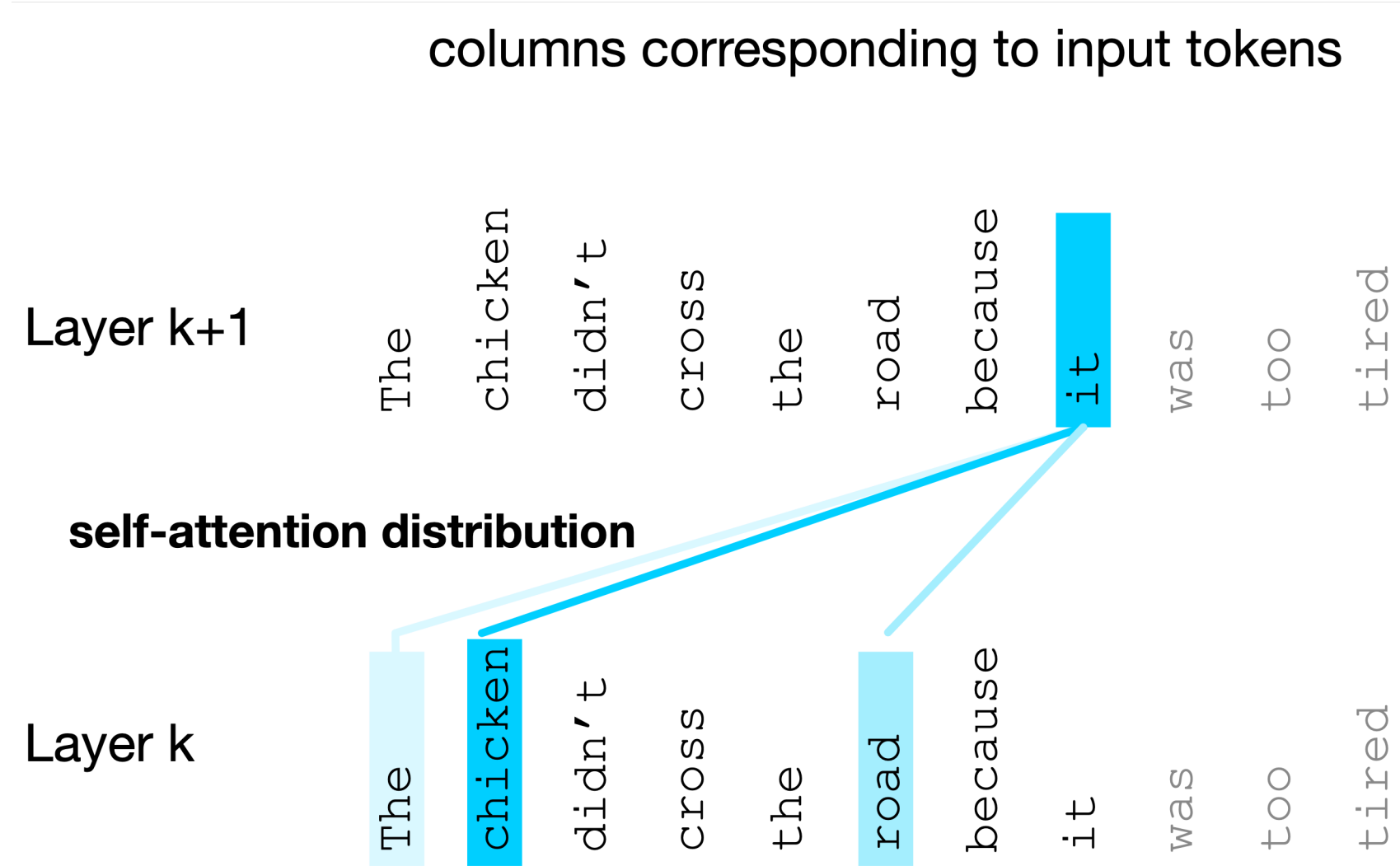
At this point in the sentence, it's probably referring to either the chicken or the street

Intuition of attention

Build up the contextual embedding from a word by selectively integrating information from all the neighboring words

We say that a word "attends to" some neighboring words more than others

Intuition of attention:

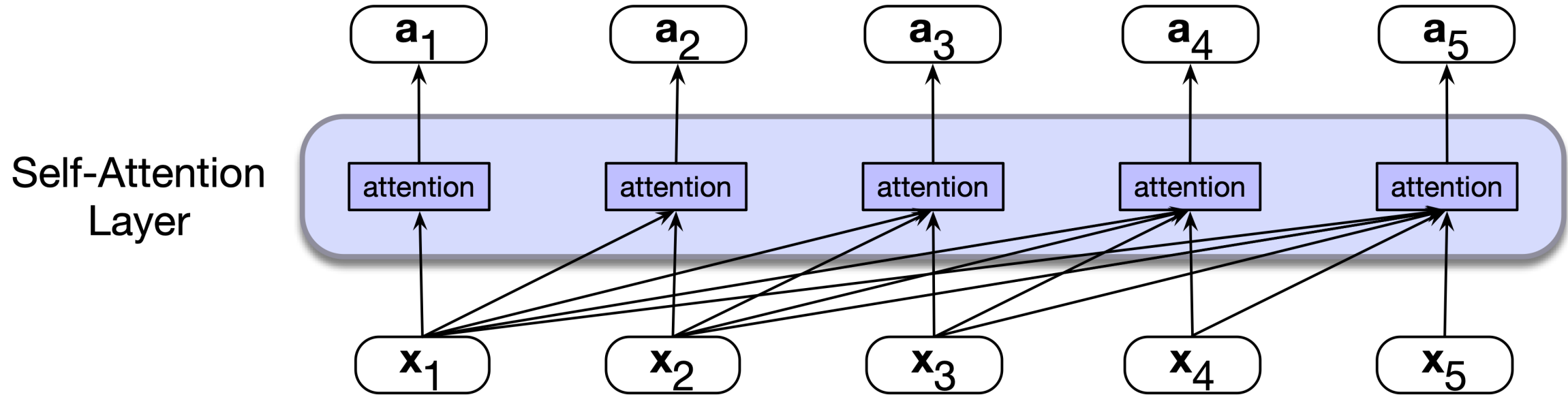


Attention definition

A mechanism for helping compute the embedding for a token by selectively attending to and integrating information from surrounding tokens (at the previous layer).

More formally: a method for doing a weighted sum of vectors.

Attention is left-to-right



Simplified version of attention: a sum of prior words weighted by their similarity with the current word

Given a sequence of token embeddings:

$$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \mathbf{x}_4 \quad \mathbf{x}_5 \quad \mathbf{x}_6 \quad \mathbf{x}_7 \quad \mathbf{x}_i$$

Produce: \mathbf{a}_i = a weighted sum of \mathbf{x}_1 through \mathbf{x}_7 (and \mathbf{x}_i)

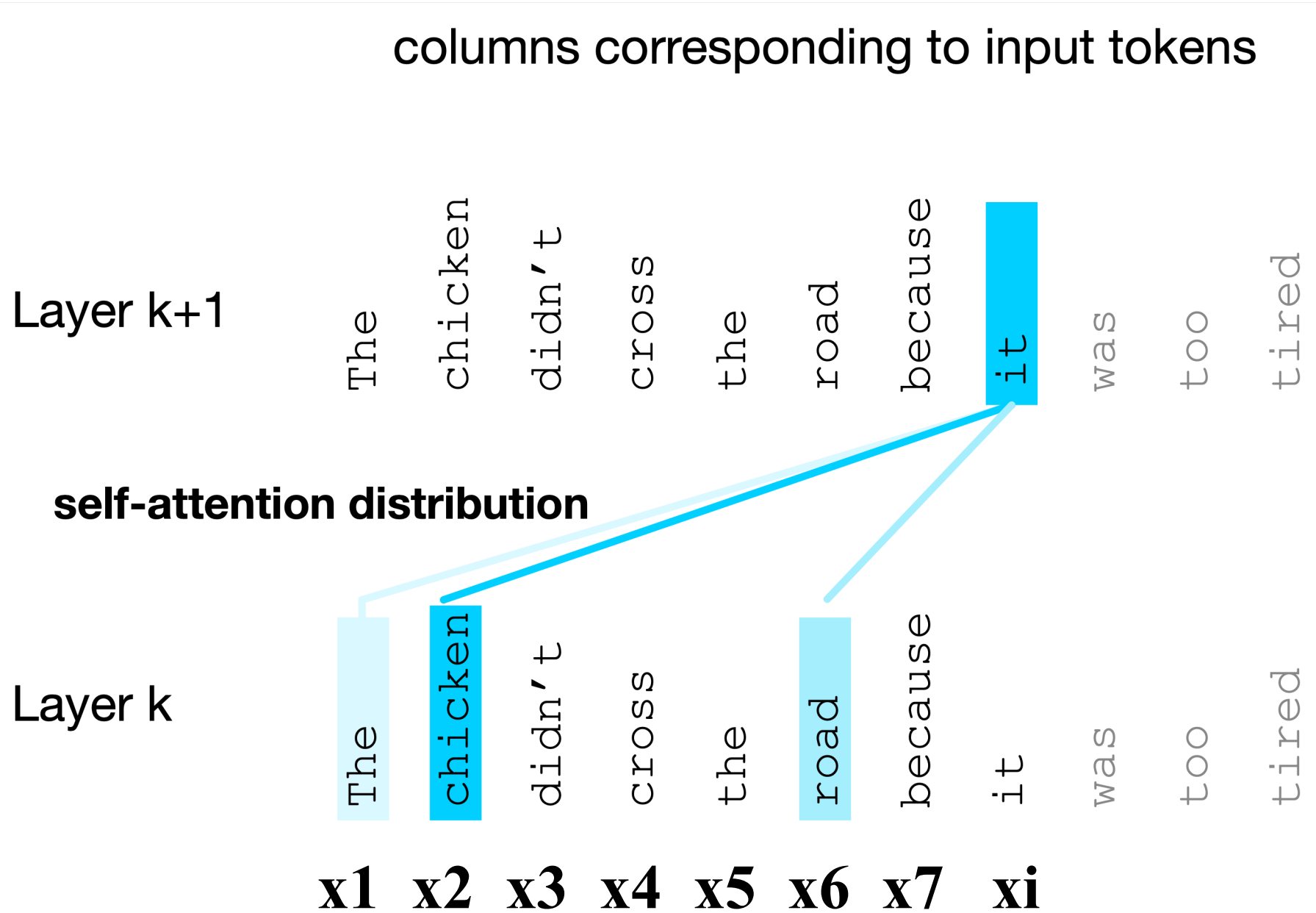
Weighted by their similarity to \mathbf{x}_i

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \quad \forall j \leq i$$

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$$

Intuition of attention:

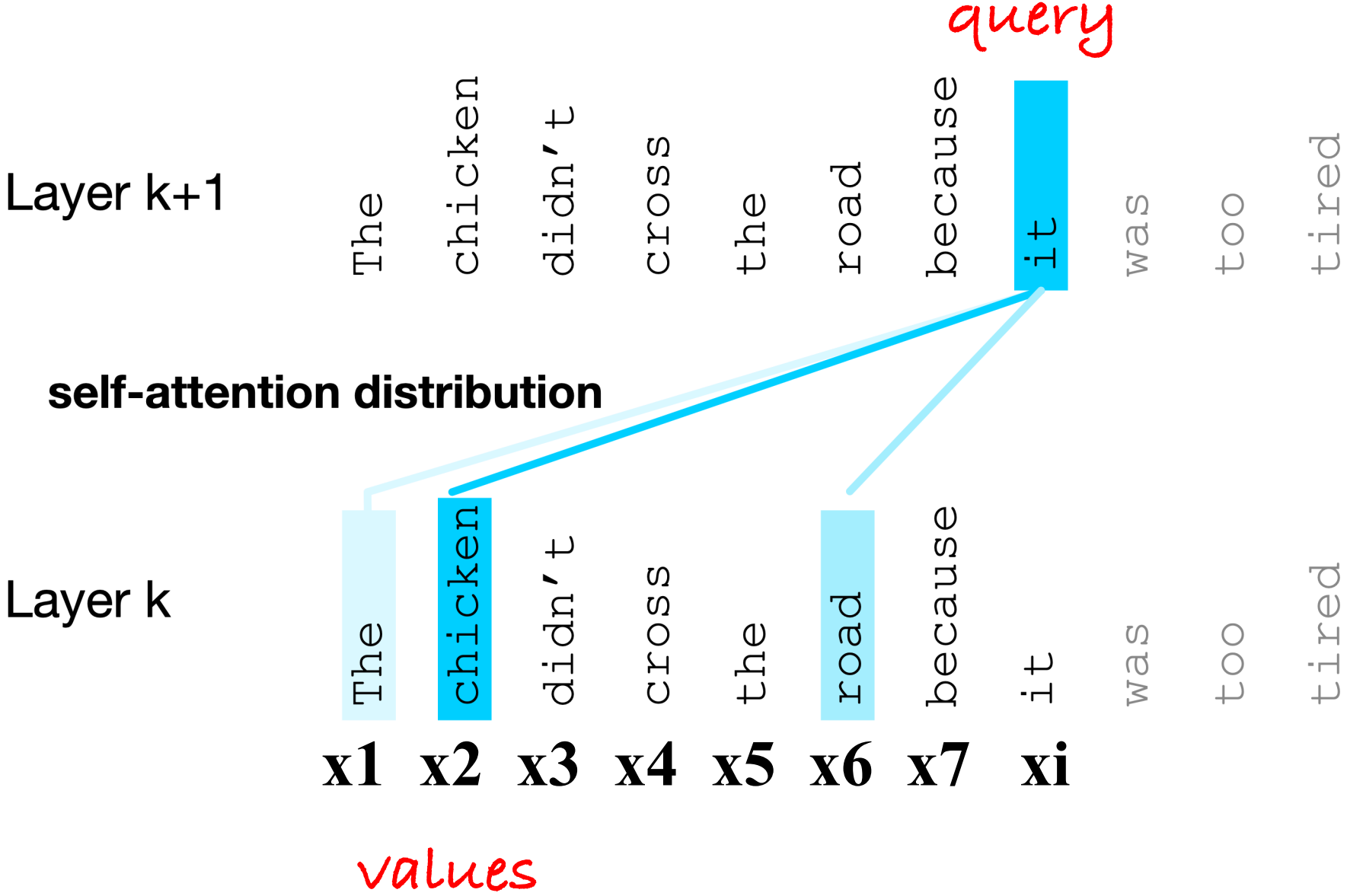


An Actual Attention Head: slightly more complicated

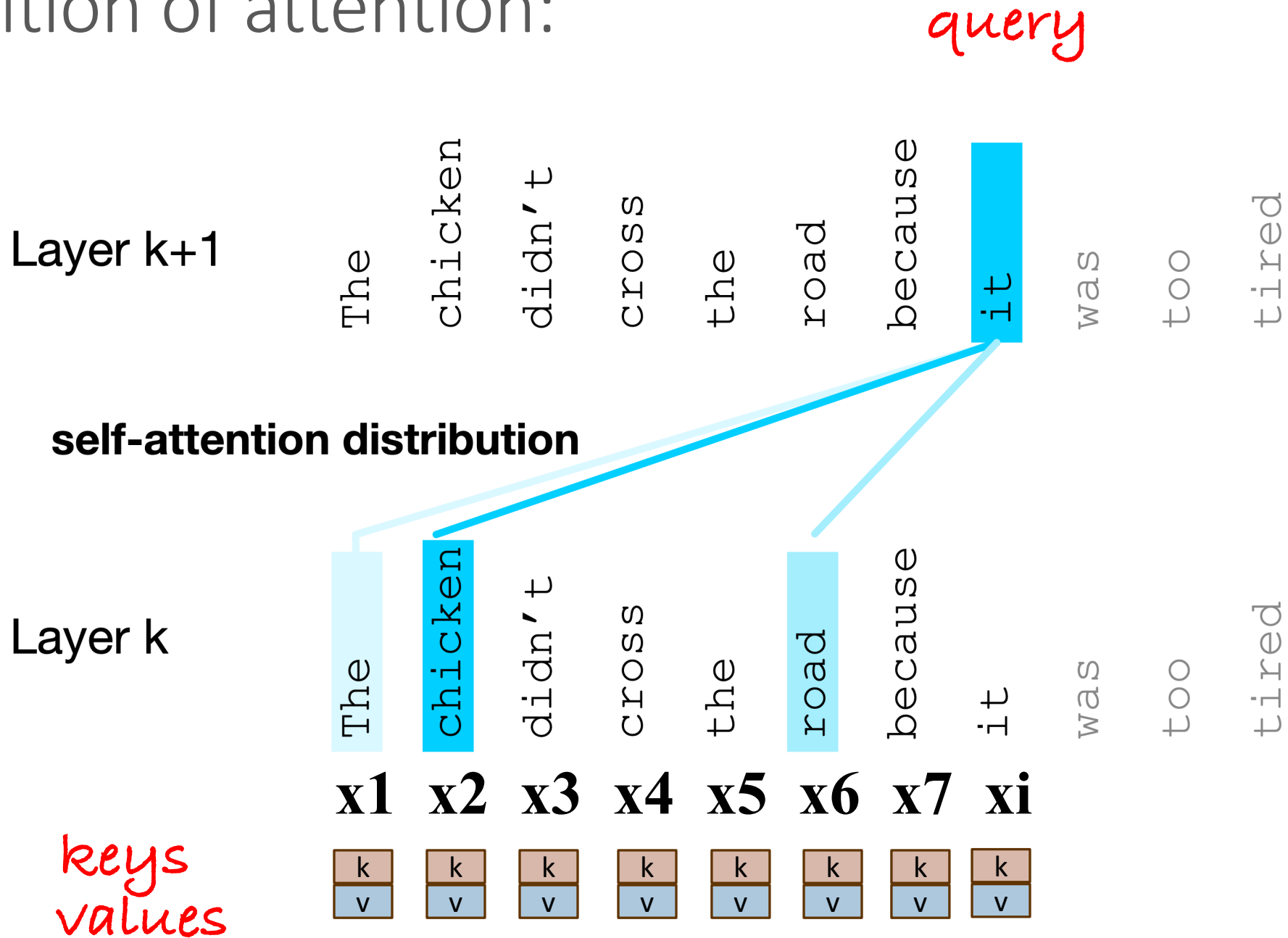
High-level idea: instead of using vectors (like x_i and x_4) directly, we'll represent 3 separate roles each vector x_i plays:

- **query**: *As the current element* being compared to the preceding inputs.
- **key**: *as a preceding input* that is being compared to the current element to determine a similarity
- **value**: a value of a preceding element that gets weighted and summed

Attention intuition



Intuition of attention:



An Actual Attention Head: slightly more complicated

We'll use matrices to project each vector \mathbf{x}_i into a representation of its role as query, key, value:

- **query:** \mathbf{W}^Q
- **key:** \mathbf{W}^K
- **value:** \mathbf{W}^V

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q; \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K; \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V$$

An Actual Attention Head: slightly more complicated

Given these 3 representation of \mathbf{x}_i

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q; \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K; \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V$$

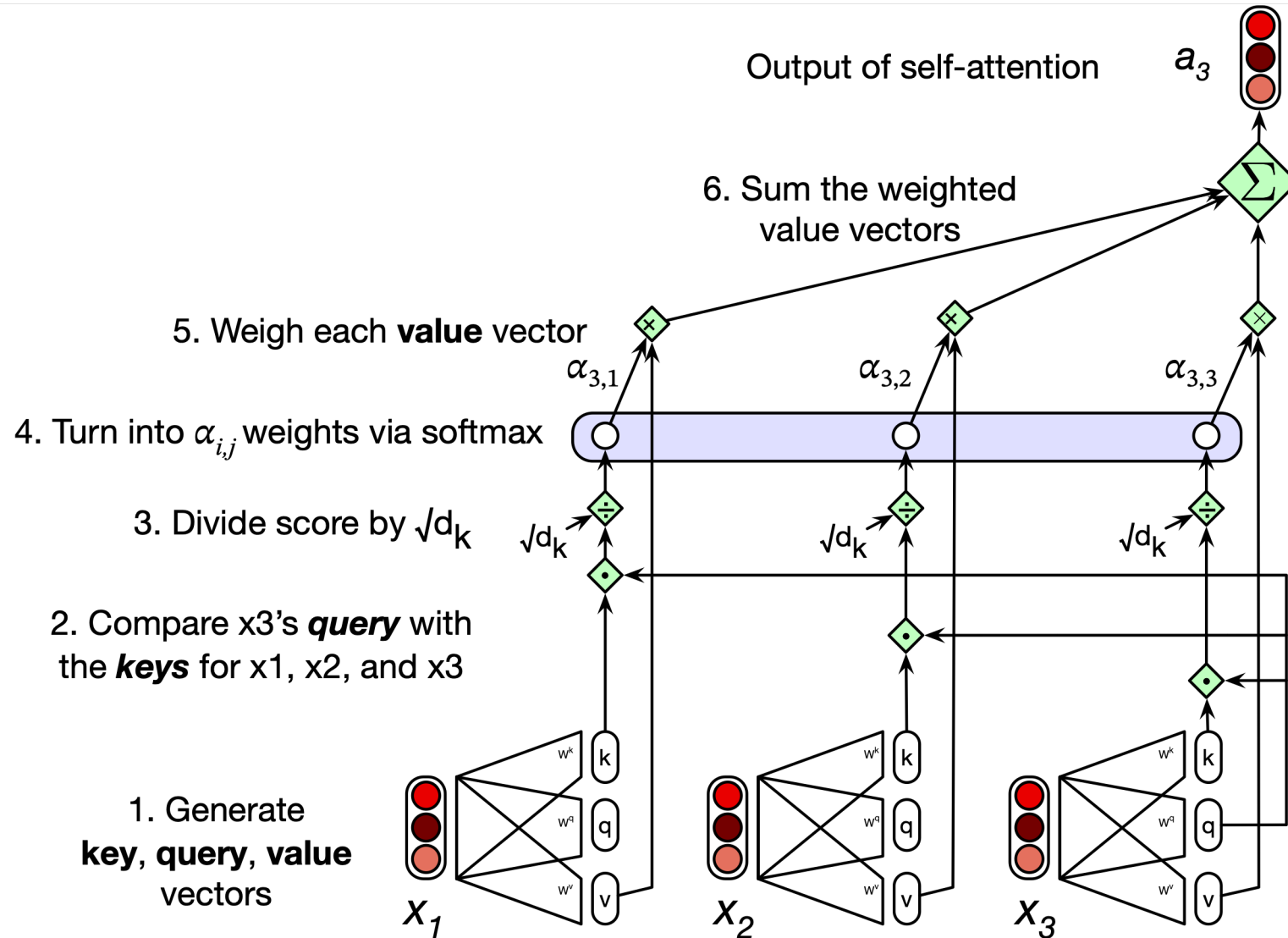
To compute similarity of current element \mathbf{x}_i with some prior element \mathbf{x}_j

We'll use dot product between \mathbf{q}_i and \mathbf{k}_j .

And instead of summing up \mathbf{x}_j , we'll sum up \mathbf{v}_j

Final equations for one attention head

Calculating the value of a_3



Actual Attention: slightly more complicated

- Instead of one attention head, we'll have lots of them!
- Intuition: each head might be attending to the context for different purposes
 - Different linguistic relationships or patterns in the context

$$\mathbf{q}_i^c = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}c}; \quad \mathbf{k}_j^c = \mathbf{x}_j \mathbf{W}^{\mathbf{K}c}; \quad \mathbf{v}_j^c = \mathbf{x}_j \mathbf{W}^{\mathbf{V}c}; \quad \forall c \quad 1 \leq c \leq h$$

$$\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}}$$

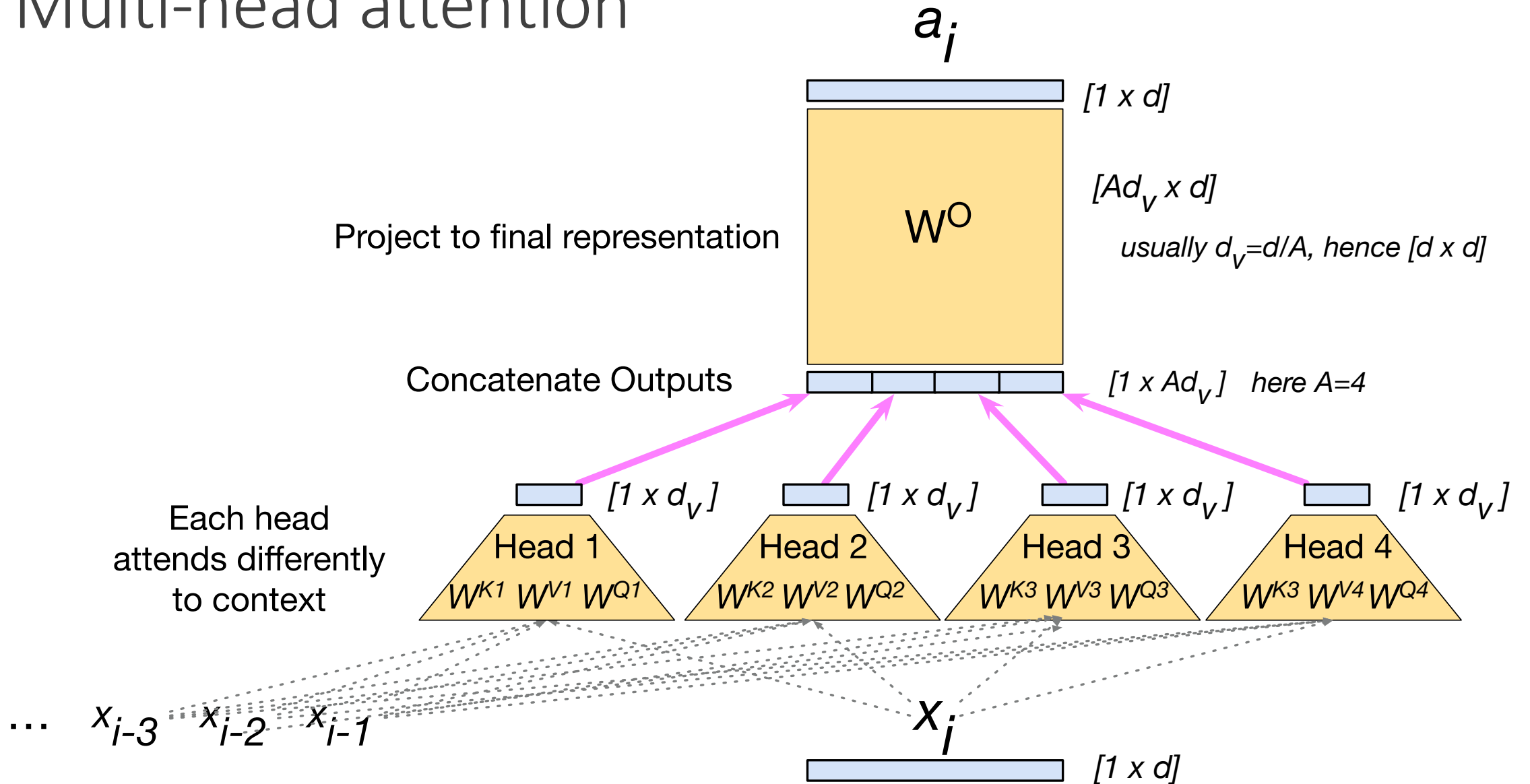
$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j)) \quad \forall j \leq i$$

$$\text{head}_i^c = \sum_{j \leq i} \alpha_{ij}^c \mathbf{v}_j^c$$

$$\mathbf{a}_i = (\text{head}^1 \oplus \text{head}^2 \dots \oplus \text{head}^h) \mathbf{W}^O$$

$$\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \dots, \mathbf{x}_N]) = \mathbf{a}_i$$

Multi-head attention



Summary

Attention is a method for enriching the representation of a token by incorporating contextual information

The result: the embedding for each word will be different in different contexts!

Contextual embeddings: a representation of word meaning in its context.

We'll see in the next lecture that attention can also be viewed as a way to move information from one token to another.

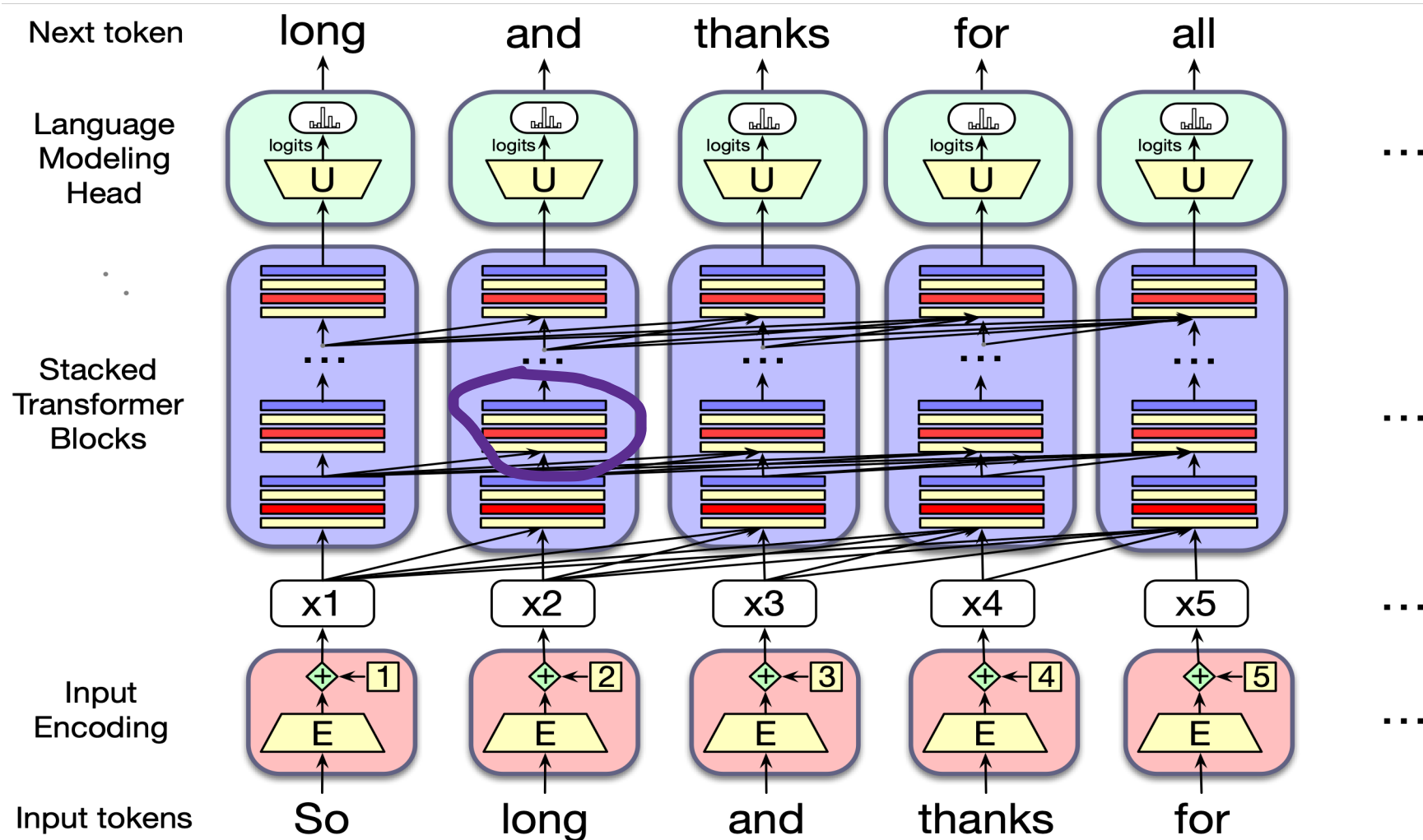
Attention

Transformers

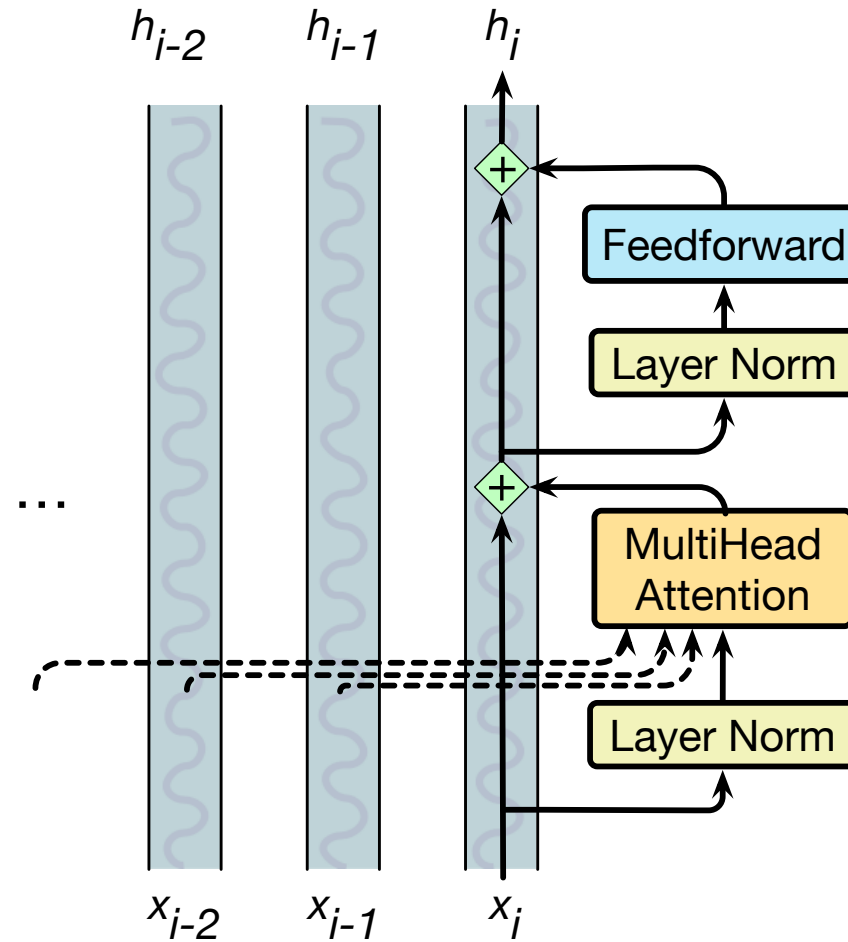
The Transformer Block

Transformers

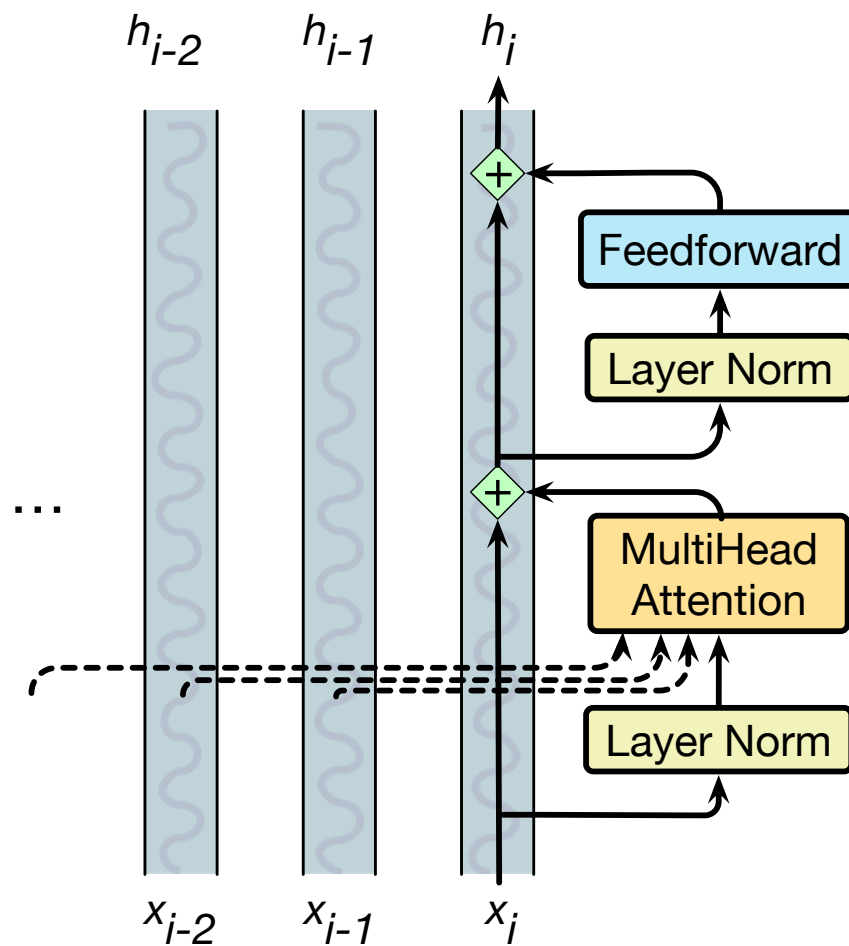
Reminder: transformer language model



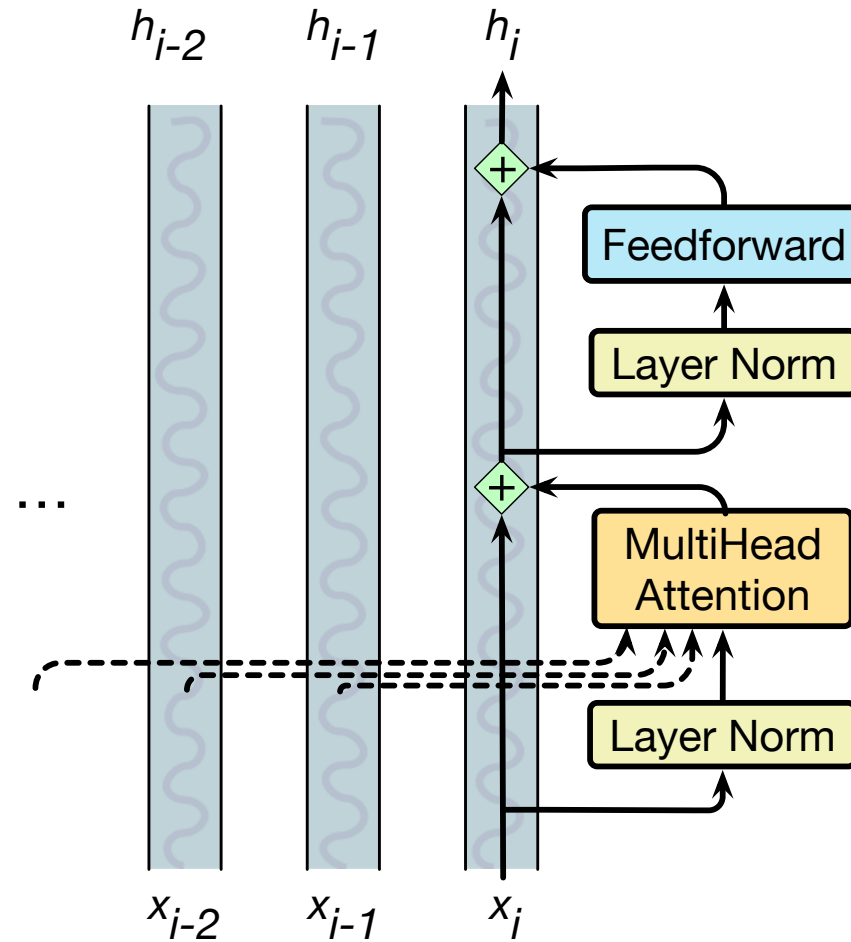
The residual stream: each token gets passed up and modified



We'll need nonlinearities, so a feedforward layer



Layer norm: the vector x_i is normalized twice



Layer Norm

Layer norm is a variation of the z-score from statistics, applied to a single vector in a hidden layer

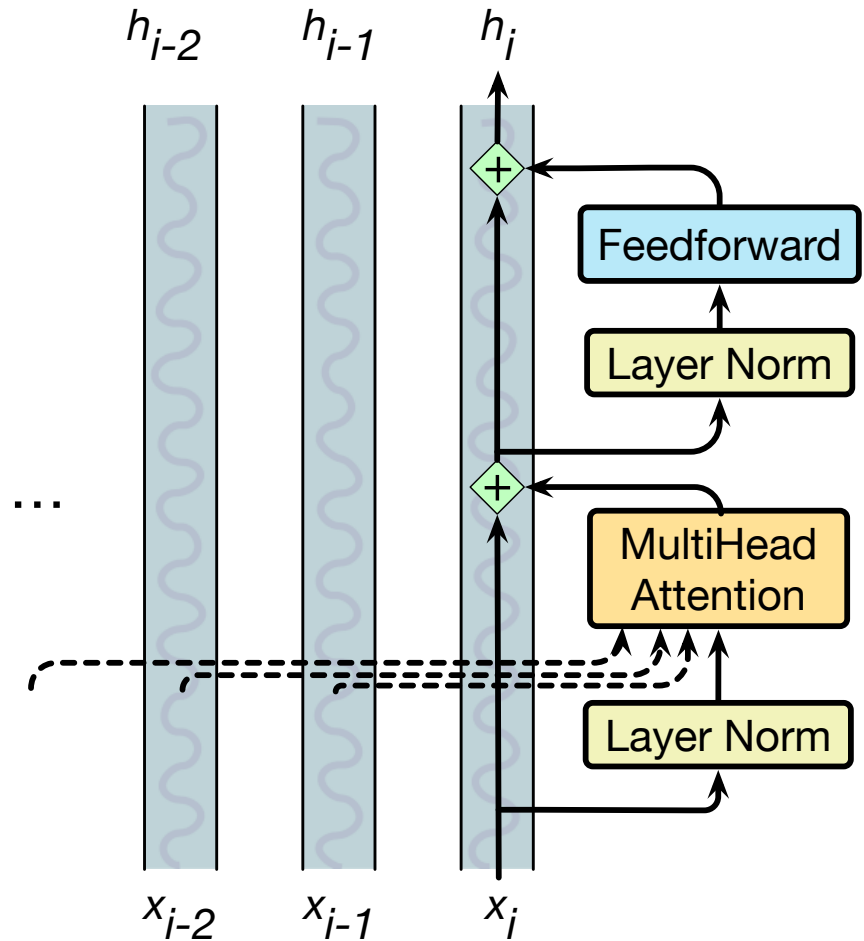
$$\mu = \frac{1}{d} \sum_{i=1}^d x_i$$

$$\sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}$$

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma}$$

$$\text{LayerNorm}(\mathbf{x}) = \gamma \frac{(\mathbf{x} - \mu)}{\sigma} + \beta$$

PreNorm Transformer block, Version 1: LayerNorm only on current token before Attention



$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i)$$

$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{x}_1^1, \dots, \mathbf{x}_N^1])$$

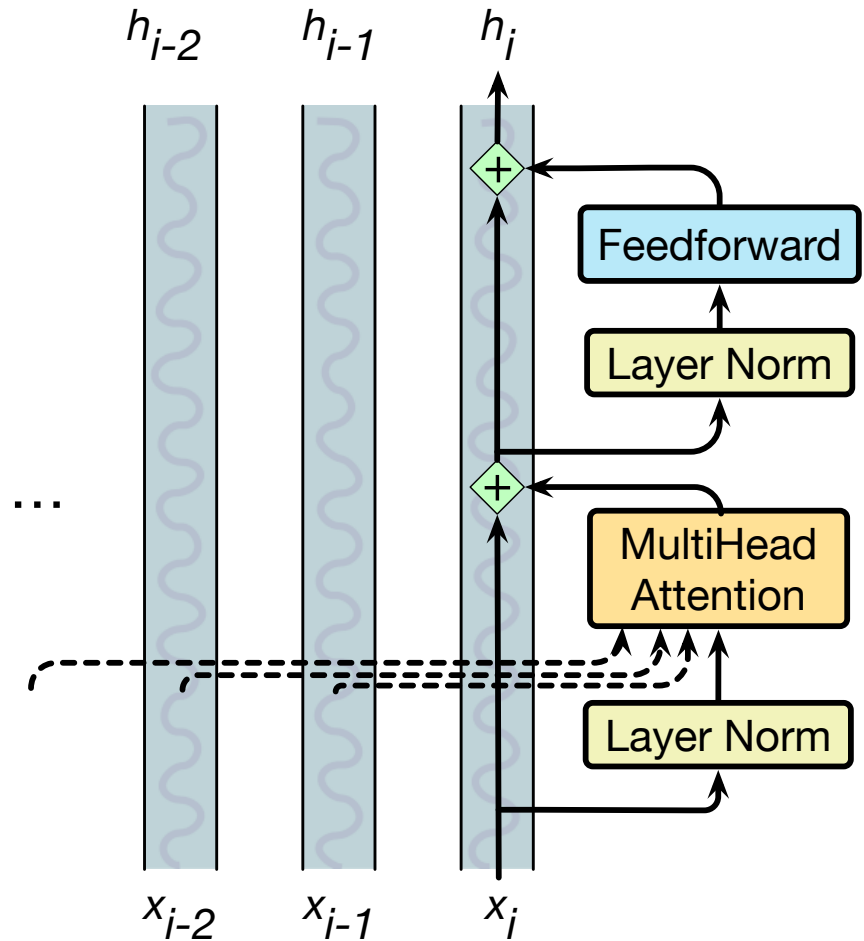
$$\mathbf{t}_i^3 = \mathbf{t}_i^2 + \mathbf{x}_i$$

$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3)$$

$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4)$$

$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3$$

PreNorm Transformer block, Version 2: LayerNorm on all tokens before Attention



$$\mathbf{t}_i^1 = \text{LayerNorm}(\mathbf{x}_i)$$

$$\mathbf{t}_i^2 = \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{t}_1^1, \dots, \mathbf{t}_N^1])$$

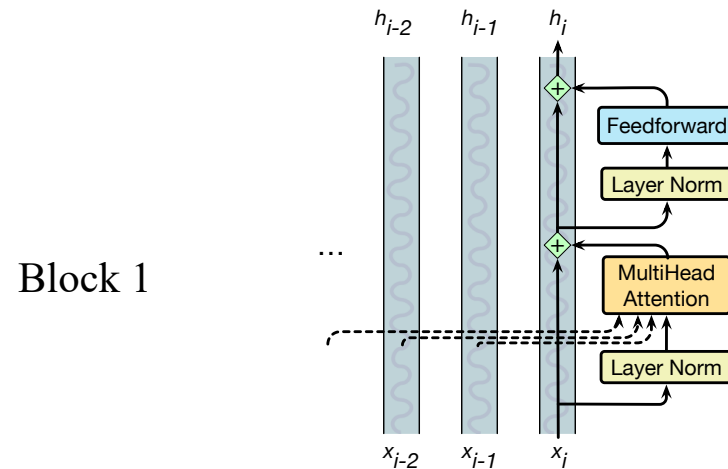
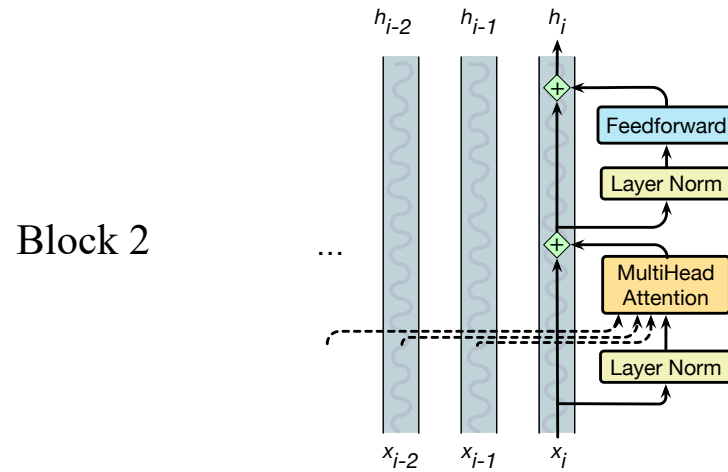
$$\mathbf{t}_i^3 = \mathbf{t}_i^2 + \mathbf{x}_i$$

$$\mathbf{t}_i^4 = \text{LayerNorm}(\mathbf{t}_i^3)$$

$$\mathbf{t}_i^5 = \text{FFN}(\mathbf{t}_i^4)$$

$$\mathbf{h}_i = \mathbf{t}_i^5 + \mathbf{t}_i^3$$

A transformer is a stack of these blocks
so all the vectors are of the same dimensionality d

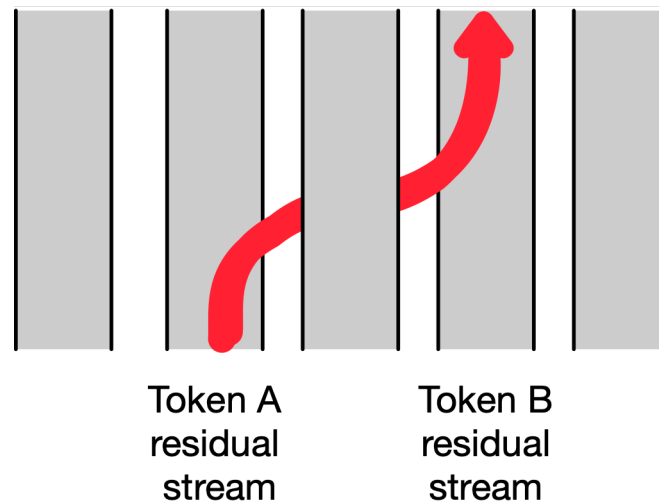


Residual streams and attention

Notice that all parts of the transformer block apply to 1 residual stream (1 token).

Except attention, which takes information from other tokens

[Elhage et al. \(2021\)](#) show that we can view attention heads as literally moving information from the residual stream of a neighboring token into the current stream



Transformers

Parallelizing Attention Computation

Parallelizing computation using X

For attention/transformer block we've been computing a **single** output at a **single** time step i in a **single** residual stream.

But we can pack the N tokens of the input sequence into a single matrix X of size $[N \times d]$.

Each row of X is the embedding of one token of the input.

X can have 1K-32K rows, each of the dimensionality of the embedding d (the **model dimension**)

$$Q = XW^Q; \quad K = XW^K; \quad V = XW^V$$

$$QK^T$$

Now can do a single matrix multiply to combine Q and K^T

	q1·k1	q1·k2	q1·k3	q1·k4
	q2·k1	q2·k2	q2·k3	q2·k4
N	q3·k1	q3·k2	q3·k3	q3·k4
	q4·k1	q4·k2	q4·k3	q4·k4
				N

Parallelizing attention

- Scale the scores, *mask*, take the *softmax*, and multiply the result by *V* resulting in a matrix of shape $N \times d$
 - An attention vector for each input token

$$\mathbf{A} = \text{softmax} \left(\text{mask} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \right) \right) \mathbf{V}$$

Masking out the future

$$\mathbf{A} = \text{softmax} \left(\text{mask} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \right) \right) \mathbf{V}$$

- What is this **mask** function?
 - $\mathbf{Q}\mathbf{K}^T$ has a score for each query dot every key, *including those that follow the query.*
 - Guessing the next word is pretty simple if you already know it!

Masking out the future

$$\mathbf{A} = \text{softmax} \left(\text{mask} \left(\frac{\mathbf{QK}^T}{\sqrt{d_k}} \right) \right) \mathbf{V}$$

Add $-\infty$ to cells in upper triangle

\Rightarrow the *softmax* will turn it to 0.

	q1·k1	$-\infty$	$-\infty$	$-\infty$
	q2·k1	q2·k2	$-\infty$	$-\infty$
	q3·k1	q3·k2	q3·k3	$-\infty$
	q4·k1	q4·k2	q4·k3	q4·k4

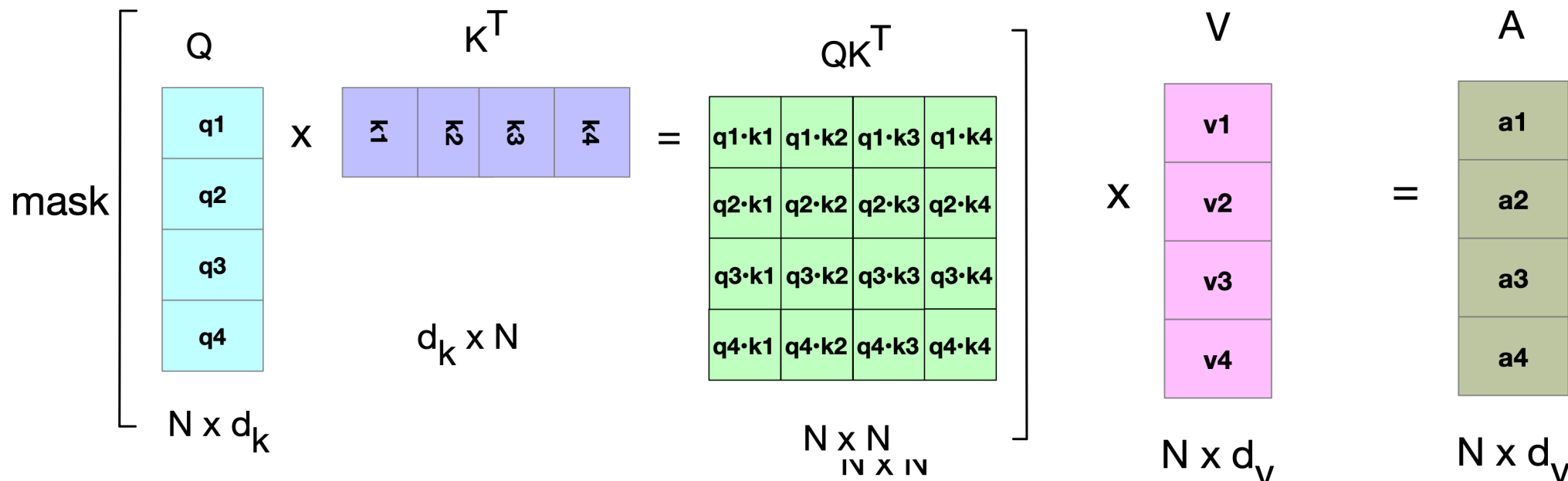
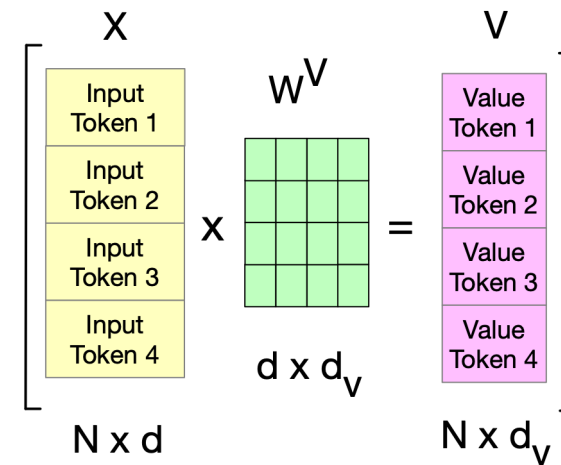
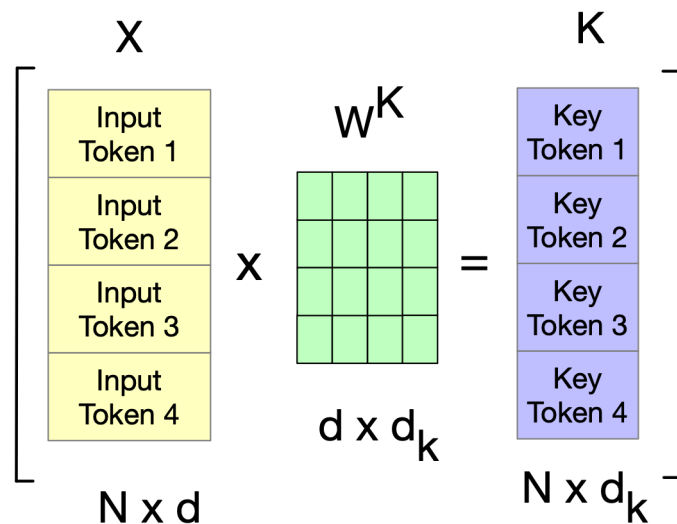
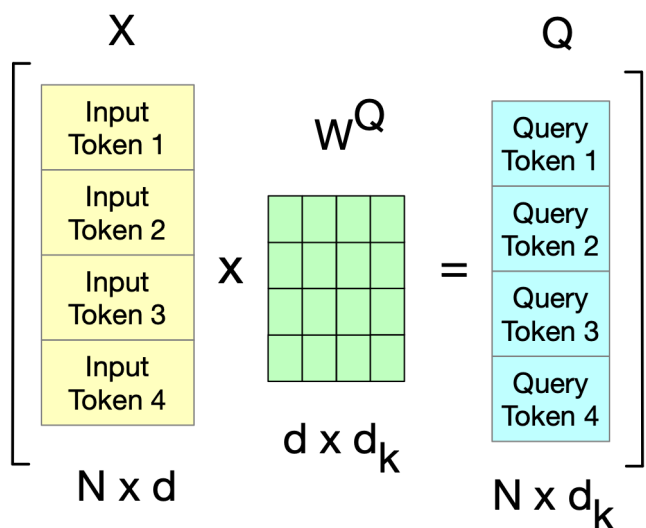
N

N

Another point: Attention is quadratic in length

	$q_1 \cdot k_1$	$-\infty$	$-\infty$	$-\infty$
N	$q_2 \cdot k_1$	$q_2 \cdot k_2$	$-\infty$	$-\infty$
	$q_3 \cdot k_1$	$q_3 \cdot k_2$	$q_3 \cdot k_3$	$-\infty$
	$q_4 \cdot k_1$	$q_4 \cdot k_2$	$q_4 \cdot k_3$	$q_4 \cdot k_4$
				N

Attention again



Parallelizing Multi-head Attention

$$\mathbf{Q}^i = \mathbf{XW}^{\mathbf{Q}^i}; \quad \mathbf{K}^i = \mathbf{XW}^{\mathbf{K}^i}; \quad \mathbf{V}^i = \mathbf{XW}^{\mathbf{V}^i}$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}^i, \mathbf{K}^i, \mathbf{V}^i) = \text{softmax} \left(\frac{\mathbf{Q}^i \mathbf{K}^{i\top}}{\sqrt{d_k}} \right) \mathbf{V}^i$$

$$\text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \dots \oplus \mathbf{head}_h) \mathbf{W}^0$$

PostNorm: Parallelizing Multi-head Attention

$$\mathbf{O} = \text{LayerNorm}(\mathbf{X} + \text{MultiHeadAttention}(\mathbf{X}))$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{O} + \text{FFN}(\mathbf{O}))$$

or

$$\mathbf{T}^1 = \text{MultiHeadAttention}(\mathbf{X})$$

$$\mathbf{T}^2 = \mathbf{X} + \mathbf{T}^1$$

$$\mathbf{T}^3 = \text{LayerNorm}(\mathbf{T}^2)$$

$$\mathbf{T}^4 = \text{FFN}(\mathbf{T}^3)$$

$$\mathbf{T}^5 = \mathbf{T}^4 + \mathbf{T}^3$$

$$\mathbf{H} = \text{LayerNorm}(\mathbf{T}^5)$$

Transformers

Input and output: Position embeddings and the Language Model Head

Token and Position Embeddings

Problem:

- Attention is **permutation invariant**.
- But positional information is important for building contextual representations!

Solution:

- The matrix X (of shape $[N \times d]$) has an embedding for each word in the context.
- This embedding is created by adding two distinct embeddings for each input
 - **token embedding** + **positional embedding**.

Token Embeddings

Embedding matrix \mathbf{E} has shape $[|V| \times d]$.

- One row for each of the $|V|$ tokens in the vocabulary.
- Each word is a row vector of d dimensions

Given: string "*Thanks for all the*"

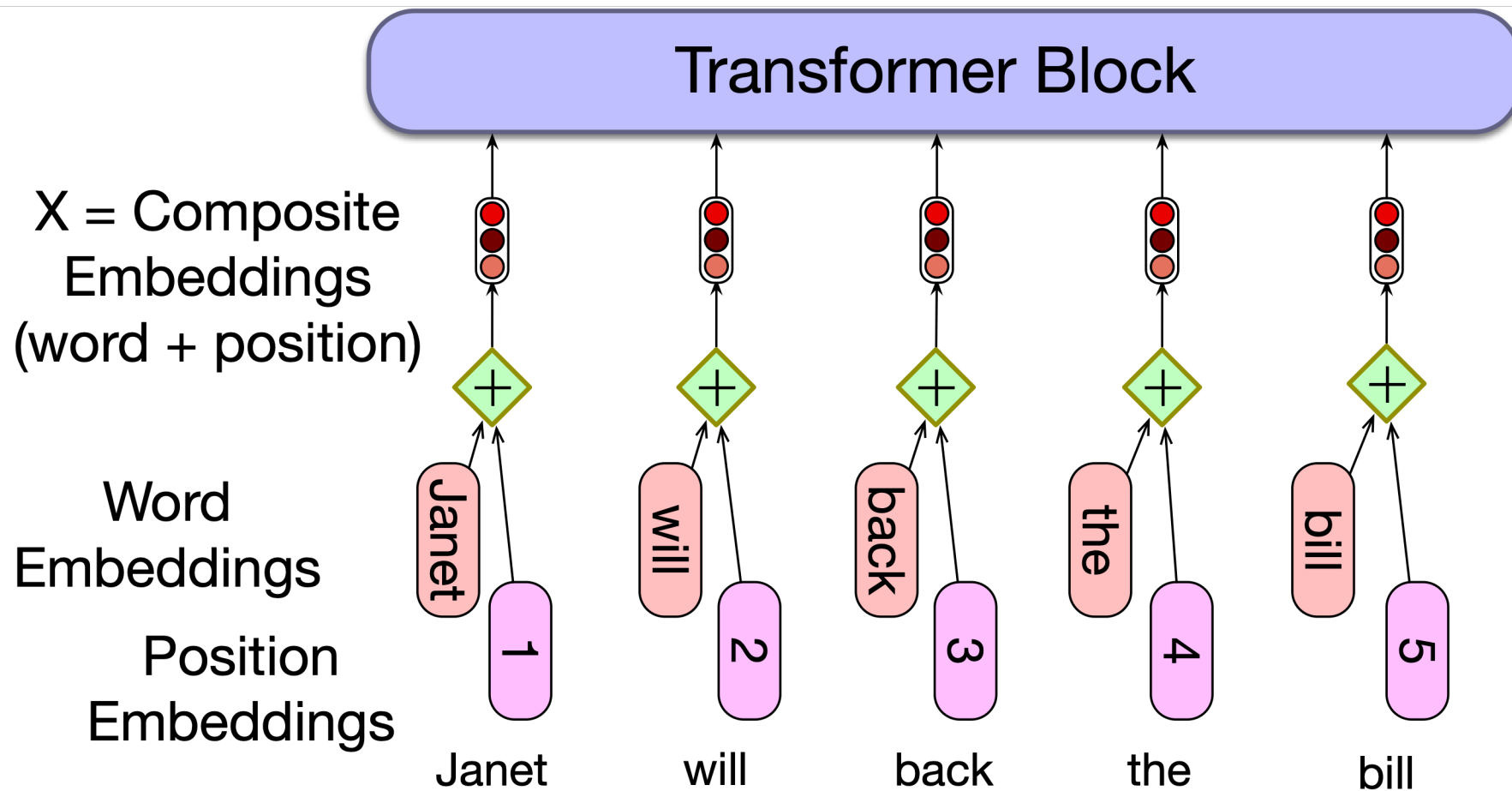
1. Tokenize with BPE and convert into vocab indices
[5, 4000, 10532, 2224]
2. Select the corresponding rows from \mathbf{E} , each row an embedding
(row 5, row 4000, row 10532, row 2224)

Position Embeddings

There are many methods:

- **Sine/cosine** embeddings.
 - Predefined embeddings computed with sin/cos at position-dependent frequencies.
- **Absolute position** embeddings.
 - Learned embeddings, one for each integer, up to some maximum length N .
 - Stored in a position embedding matrix **E_{pos}** of shape $[d \times N]$.
 - As with token embeddings **E**, these position embeddings **E_{pos}** are learned along with other parameters during training.

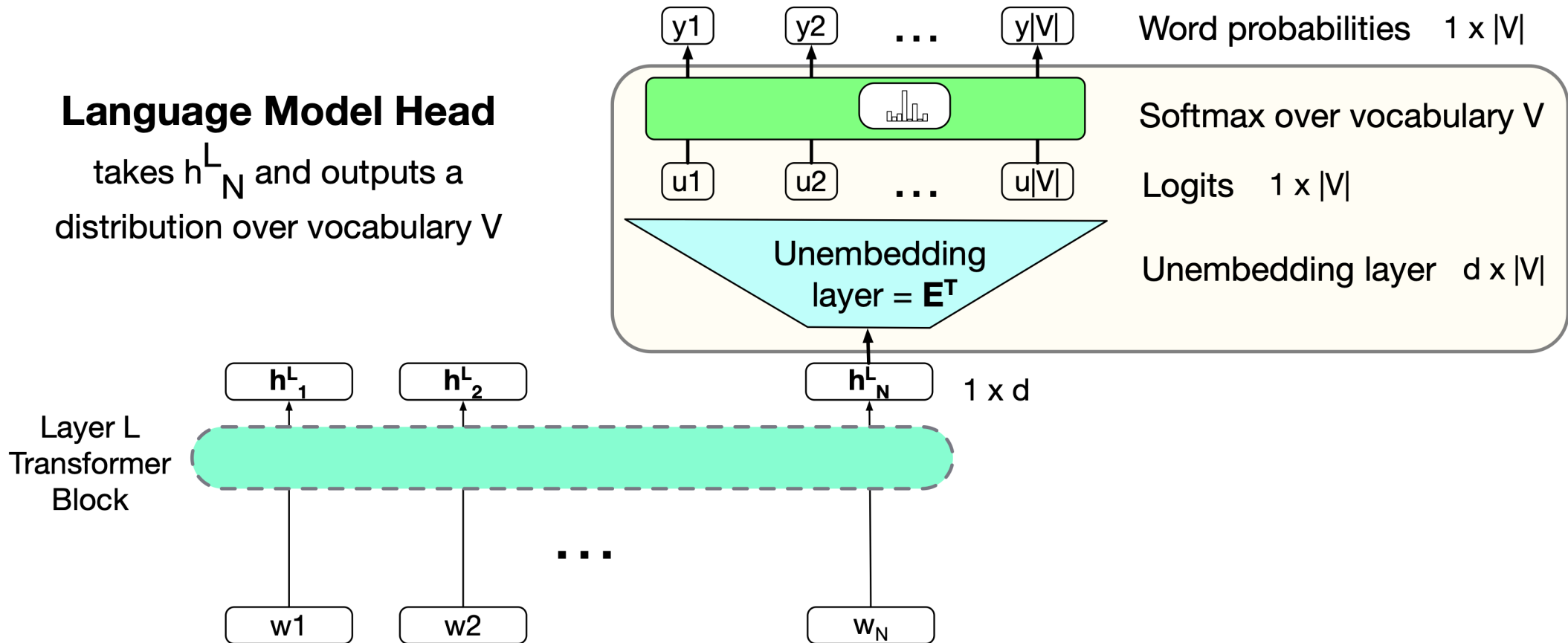
Each x is just the sum of word and position embeddings



Language modeling head

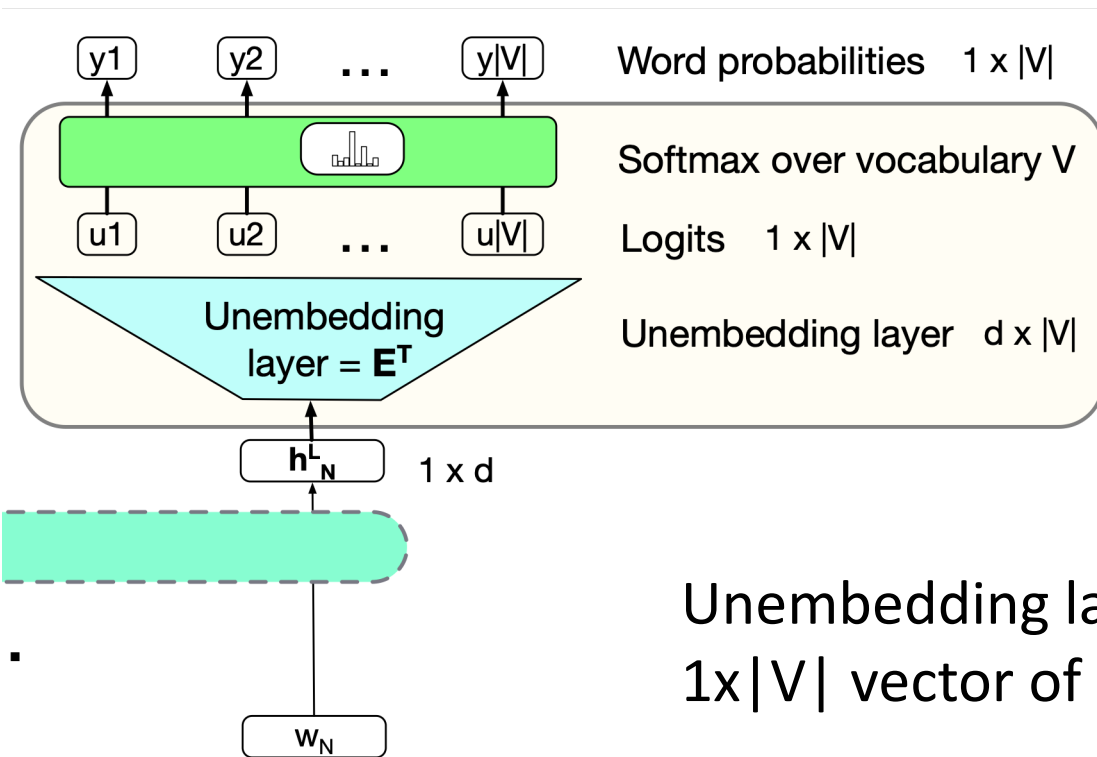
Language Model Head

takes h^L_N and outputs a distribution over vocabulary V



Language modeling head

Unembedding layer: linear layer projects from h_N^L (shape $[1 \times d]$) to logit vector

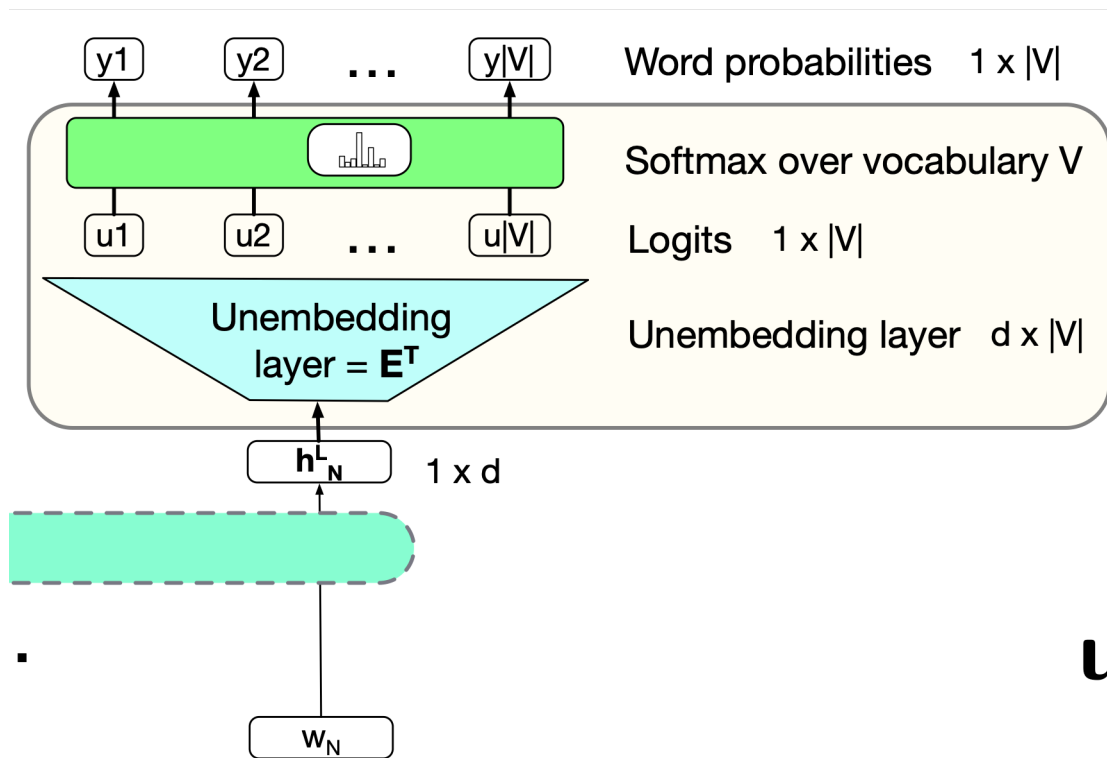


Why "unembedding"? **Tied** to E^T

Weight tying, we use the same weights for two different matrices

Unembedding layer maps from an embedding to a $1 \times |V|$ vector of logits

Language modeling head



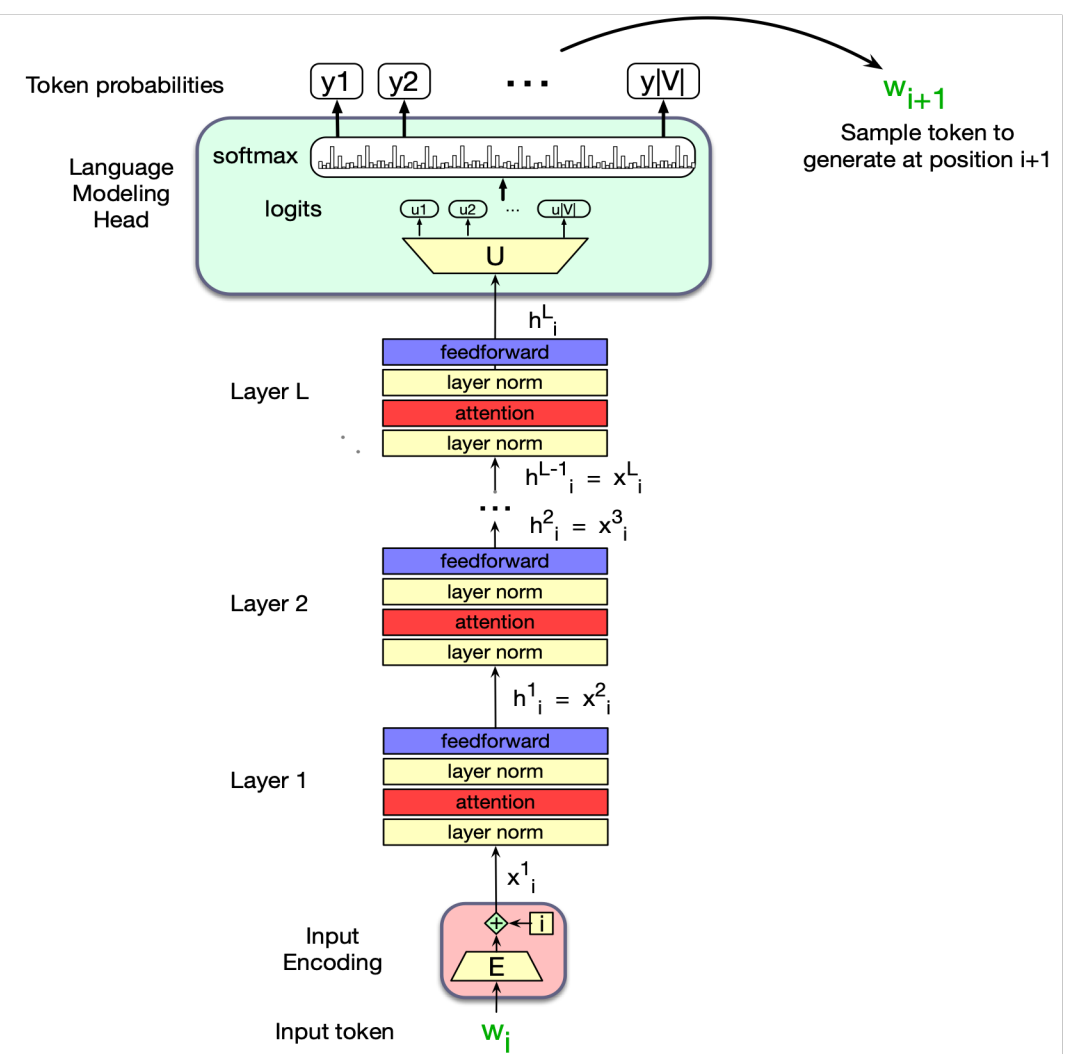
Logits, the score vector u

One score for each of the $|V|$ possible words in the vocabulary V .
Shape $1 \times |V|$.

Softmax turns the logits into probabilities over vocabulary.
Shape $1 \times |V|$.

$$\mathbf{u} = \mathbf{h}_N^L \mathbf{E}^T$$
$$\mathbf{y} = \text{softmax}(\mathbf{u})$$

The final transformer model



Large
Language
Models

Mixture of Experts

Mixture-of-Experts (MoE) architecture

The **Mixture-of-Experts (MoE)** architecture splits a model block into multiple experts, which are specialized sub-networks trained to handle different types of inputs.

- In Transformer-based MoE models, the standard Feed-Forward Network (FFN) inside the Transformer block is replaced with an **MoE layer**.
- A **routing mechanism** selects which experts (smaller FFNs) should process each input token.

Instead of all parameters being used for every input, only **a small subset of experts are activated**, which allows the model to maintain a very large overall capacity while keeping computation efficient.

- This design has shown that models can achieve state-of-the-art performance while keeping inference more practical and cost-effective.

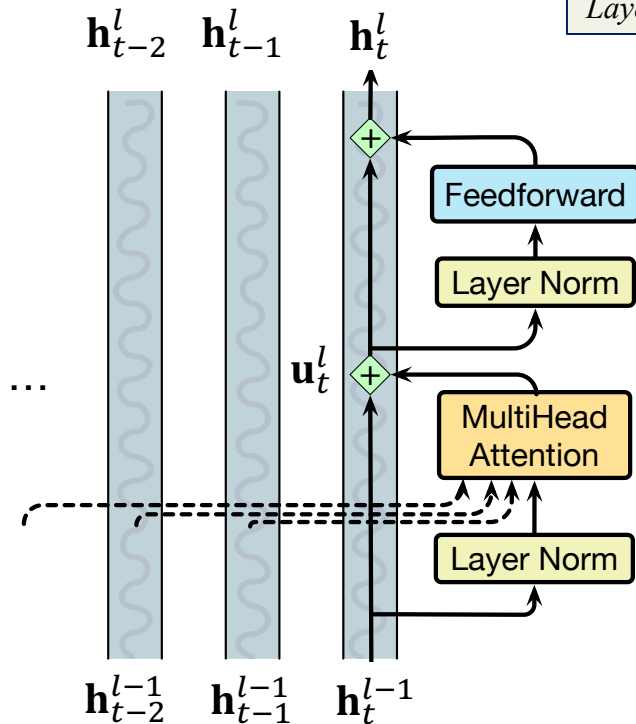
Mixture-of-Experts for Transformers

Standard Transformer block:

$$\mathbf{u}_{1:T}^l = \text{Self-Att}(\mathbf{h}_{1:T}^{l-1}) + \mathbf{h}_{1:T}^{l-1},$$

$$\mathbf{h}_t^l = \text{FFN}(\mathbf{u}_t^l) + \mathbf{u}_t^l,$$

LayerNorm not shown

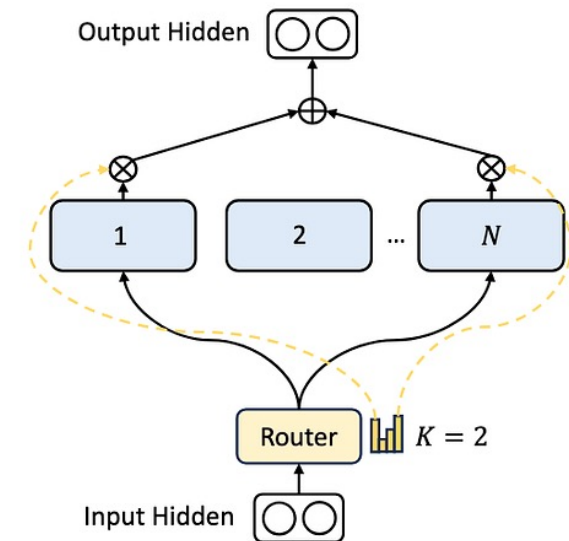


Conventional MoE layer with Routed experts:

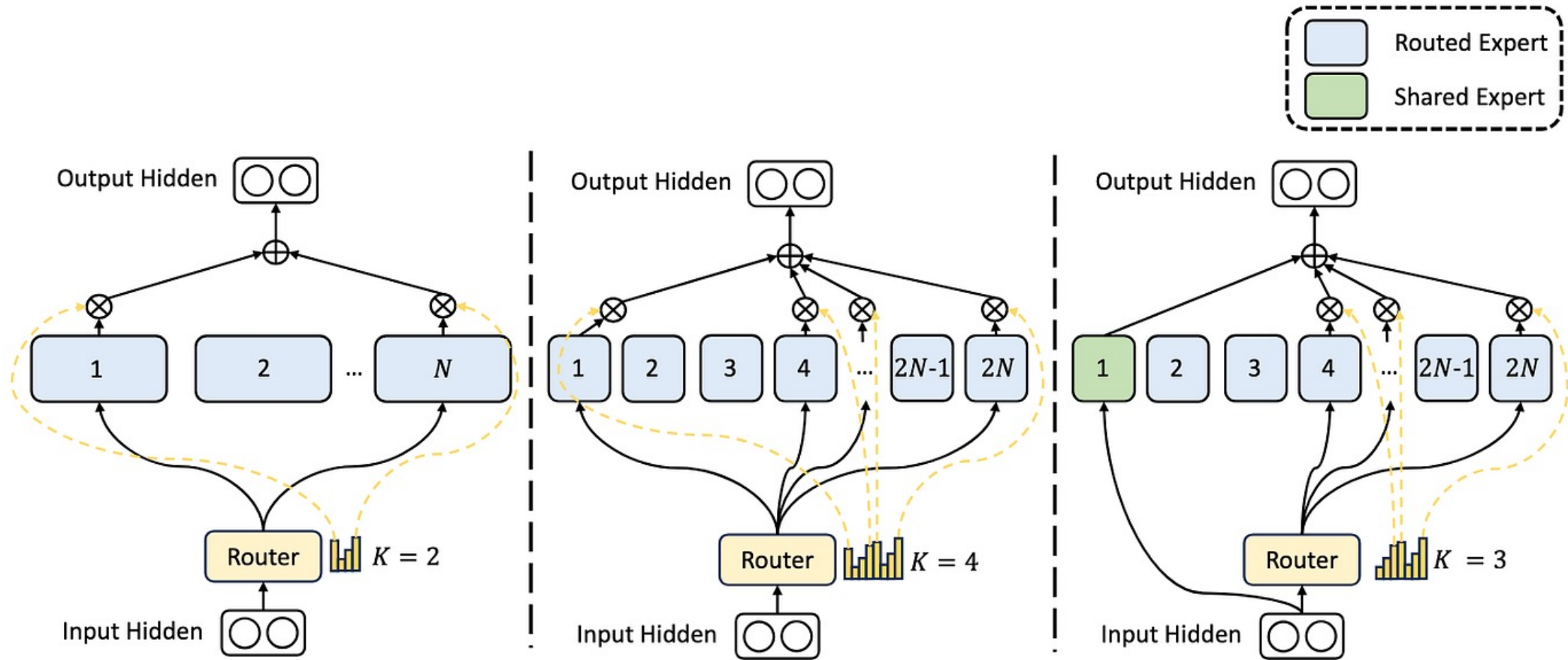
$$\mathbf{h}_t^l = \sum_{i=1}^N \left(g_{i,t} \text{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l,$$

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \text{Topk}(\{s_{j,t} | 1 \leq j \leq N\}, K), \\ 0, & \text{otherwise,} \end{cases}$$

$$s_{i,t} = \text{Softmax}_i(\mathbf{u}_t^{lT} \mathbf{e}_i^l),$$



Three Expert Structures for MoE Architectures



(a) Conventional Top-2 Routing → **(b) + Fine-grained Expert Segmentation** → **(c) + Shared Expert Isolation (DeepSeekMoE)**

DeepSeekMoE: Shared Experts

Shared Expert Isolation:

- Isolate certain experts to serve as **shared experts** that are always activated, aiming at capturing and consolidating common knowledge across varying contexts.
- Through compressing common knowledge into these shared experts, redundancy among other **routed experts** will be mitigated.
- This can enhance the parameter efficiency and ensure that each routed expert stays specialized by focusing on distinctive aspects.

Mixture-of-Experts for Transformers

Standard Transformer block:

$$\mathbf{u}_{1:T}^l = \text{Self-Att}(\mathbf{h}_{1:T}^{l-1}) + \mathbf{h}_{1:T}^{l-1},$$
$$\mathbf{h}_t^l = \text{FFN}(\mathbf{u}_t^l) + \mathbf{u}_t^l,$$

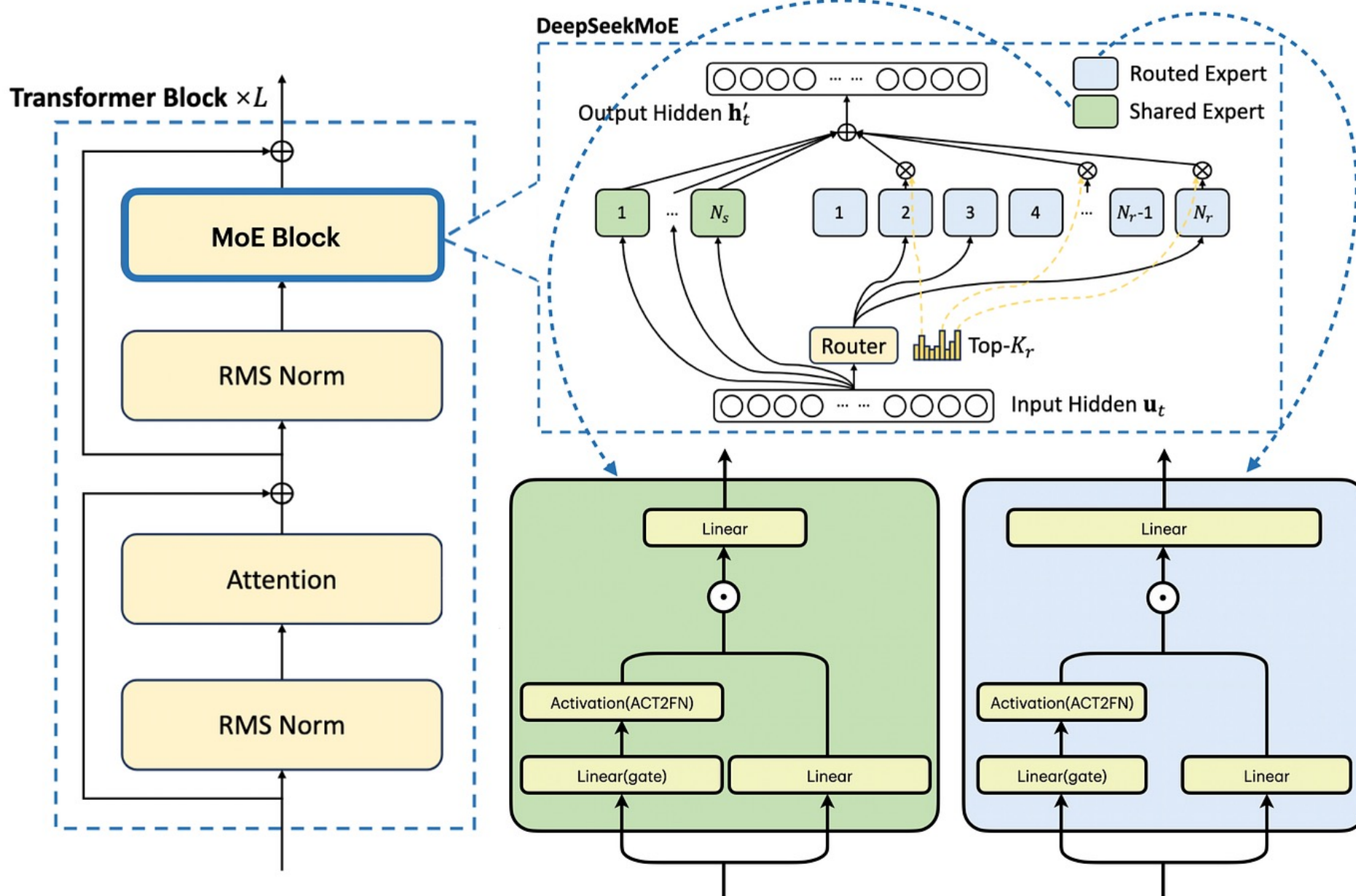
Conventional MoE layer with Routed experts:

$$\mathbf{h}_t^l = \sum_{i=1}^N \left(g_{i,t} \text{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l,$$
$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \text{Topk}(\{s_{j,t} | 1 \leq j \leq N\}, K), \\ 0, & \text{otherwise,} \end{cases}$$
$$s_{i,t} = \text{Softmax}_i(\mathbf{u}_t^{lT} \mathbf{e}_i^l),$$

MoE layer with Shared + Routed experts:

$$\mathbf{h}_t^l = \sum_{i=1}^{K_s} \text{FFN}_i(\mathbf{u}_t^l) + \sum_{i=K_s+1}^{mN} \left(g_{i,t} \text{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l,$$
$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \text{Topk}(\{s_{j,t} | K_s + 1 \leq j \leq mN\}, mK - K_s), \\ 0, & \text{otherwise,} \end{cases}$$
$$s_{i,t} = \text{Softmax}_i(\mathbf{u}_t^{lT} \mathbf{e}_i^l).$$

DeepSeek V3 Transformer Block



DeepSeek V3

[DeepSeek-V3 Technical Report](#), DeepSeek-AI, 2024

Model Hyper-Parameters. We set the number of Transformer layers to 61 and the hidden dimension to 7168. All learnable parameters are randomly initialized with a standard deviation of 0.006. In MLA, we set the number of attention heads n_h to 128 and the per-head dimension d_h to 128. The KV compression dimension d_c is set to 512, and the query compression dimension d'_c is set to 1536. For the decoupled queries and key, we set the per-head dimension d_h^R to 64. We substitute all FFNs except for the first three layers with MoE layers. Each MoE layer consists of 1 shared expert and 256 routed experts, where the intermediate hidden dimension of each expert is 2048. Among the routed experts, 8 experts will be activated for each token, and each token will be ensured to be sent to at most 4 nodes. The multi-token prediction depth D is set to 1, i.e., besides the exact next token, each token will predict one additional token. As DeepSeek-V2, DeepSeek-V3 also employs additional RMSNorm layers after the compressed latent vectors, and multiplies additional scaling factors at the width bottlenecks. Under this configuration, DeepSeek-V3 comprises 671B total parameters, of which 37B are activated for each token.

Large
Language
Models

Dealing with Scale

Scaling Laws

LLM performance depends on

- **Model** size: number of parameters (not counting embeddings).
- **Dataset** size: amount of training data.
- **Compute**: amount of compute, e.g., in FLOPS

Can improve a model by:

- adding **parameters** (more layers, wider contexts),
- using more **data**,
- training for more **iterations**.

The performance of a large language model (the loss) scales as a power-law with each of these three.

Scaling Laws

[Scaling Laws for Neural Language Models](#), Kaplan et al., 2020.

Loss L as a function of # parameters N , dataset size D , compute budget C (if other two are held constant)

$$L(N) = \left(\frac{N_c}{N} \right)^{\alpha_N}$$

$$L(D) = \left(\frac{D_c}{D} \right)^{\alpha_D}$$

$$L(C) = \left(\frac{C_c}{C} \right)^{\alpha_C}$$

Scaling laws can be used early in training to predict what the loss would be if we were to add more data or increase model size.

Number of non-embedding parameters N

$$\begin{aligned} N &\approx 2 d n_{\text{layer}} (2 d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12 n_{\text{layer}} d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned}$$

Thus GPT-3, with $n = 96$ layers and dimensionality $d = 12288$, has $12 \times 96 \times 12288^2 \approx 175$ billion parameters.

KV Cache

In **training**, we can compute attention very efficiently in parallel:

$$\mathbf{A} = \text{softmax} \left(\frac{\mathbf{QK}^T}{\sqrt{d_k}} \right) \mathbf{V}$$

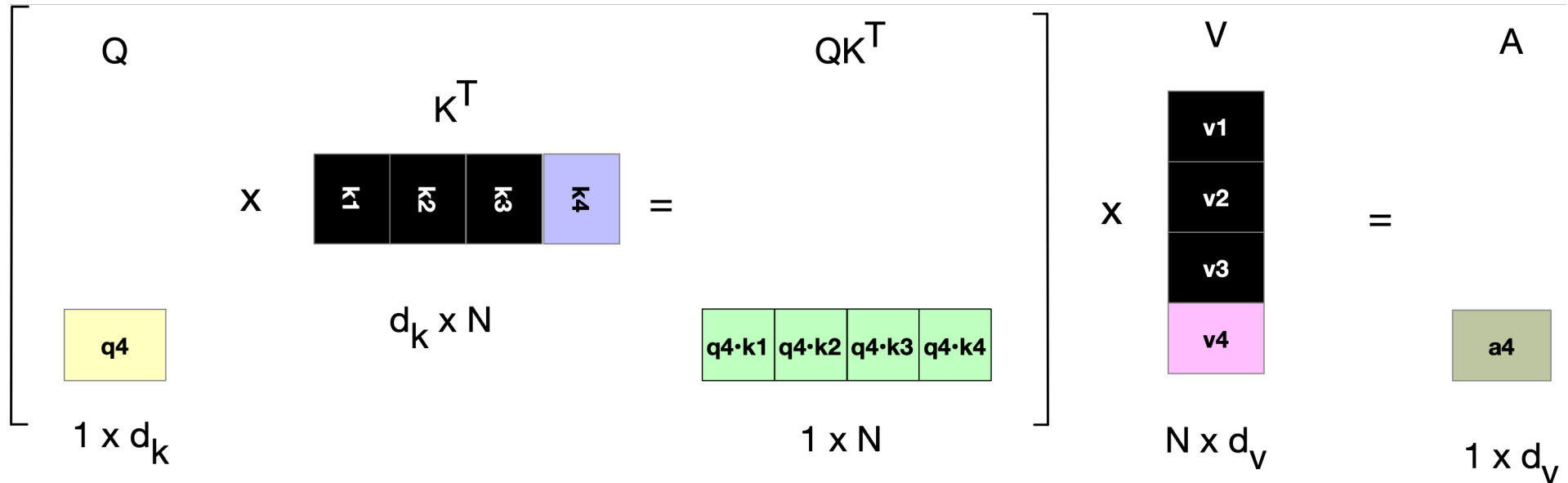
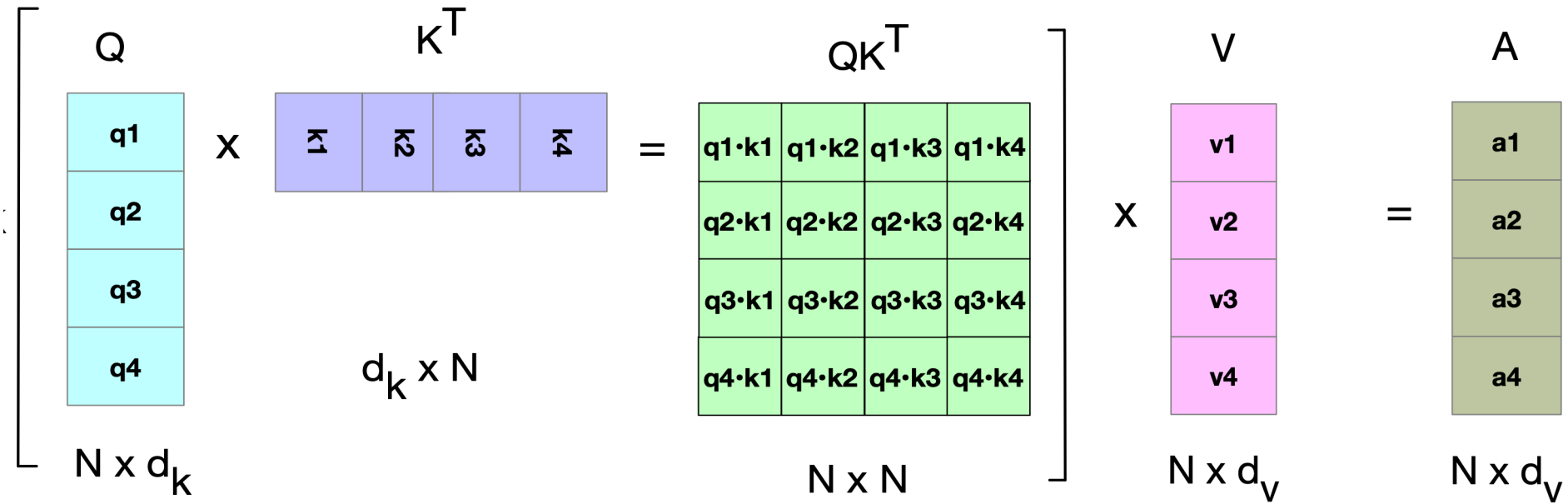
But not at **inference**! We generate the next tokens **one at a time**!

For a new token x , need to multiply by W^Q , W^K , and W^V to get query, key, values

But don't want to **recompute** the **key** and **value** vectors for all the prior tokens $x_{<i}$

Instead, store **key** and **value** vectors in memory in the **KV cache**, and then we can just grab them from the cache.

KV Cache



Parameter-Efficient Finetuning

Adapting to a new domain by continued pretraining (finetuning) is a problem with huge LLMs.

- Enormous numbers of parameters to train
- Each pass of batch gradient descent has to backpropagate through many many huge layers.
- Expensive in processing power, in memory, and in time.

Instead, **parameter-efficient fine tuning (PEFT)**

- Efficiently select a subset of parameters to update when finetuning.
- E.g., freeze some of the parameters (don't change them),
- And only update some a few parameters.

LoRA (Low-Rank Adaptation)

- Transformers have many dense matrix multiply layers
 - Like \mathbf{W}^Q , \mathbf{W}^K , \mathbf{W}^V , \mathbf{W}^O layers in attention
- Instead of updating these layers during finetuning,
 - Freeze these layers
 - Update a low-rank approximation with fewer parameters.

LoRA

- Consider a **matrix W** (shape $[N \times d]$) that needs to be updated during finetuning via gradient descent.
 - Normally updates are ΔW (shape $[N \times d]$)
- In LoRA, we freeze W and update instead a low-rank decomposition of W :
 - A of shape $[N \times r]$,
 - B of shape $[r \times d]$, r is very small (like 1 or 2)
 - That is, during finetuning we update A and B instead of W .
 - Replace $W + \Delta W$ with $W + BA$.

Forward pass: instead of

$$\mathbf{h} = \mathbf{xW}$$

We do

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB}$$

LoRA

