Transformers

Introduction to 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 lukaszkaiser@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

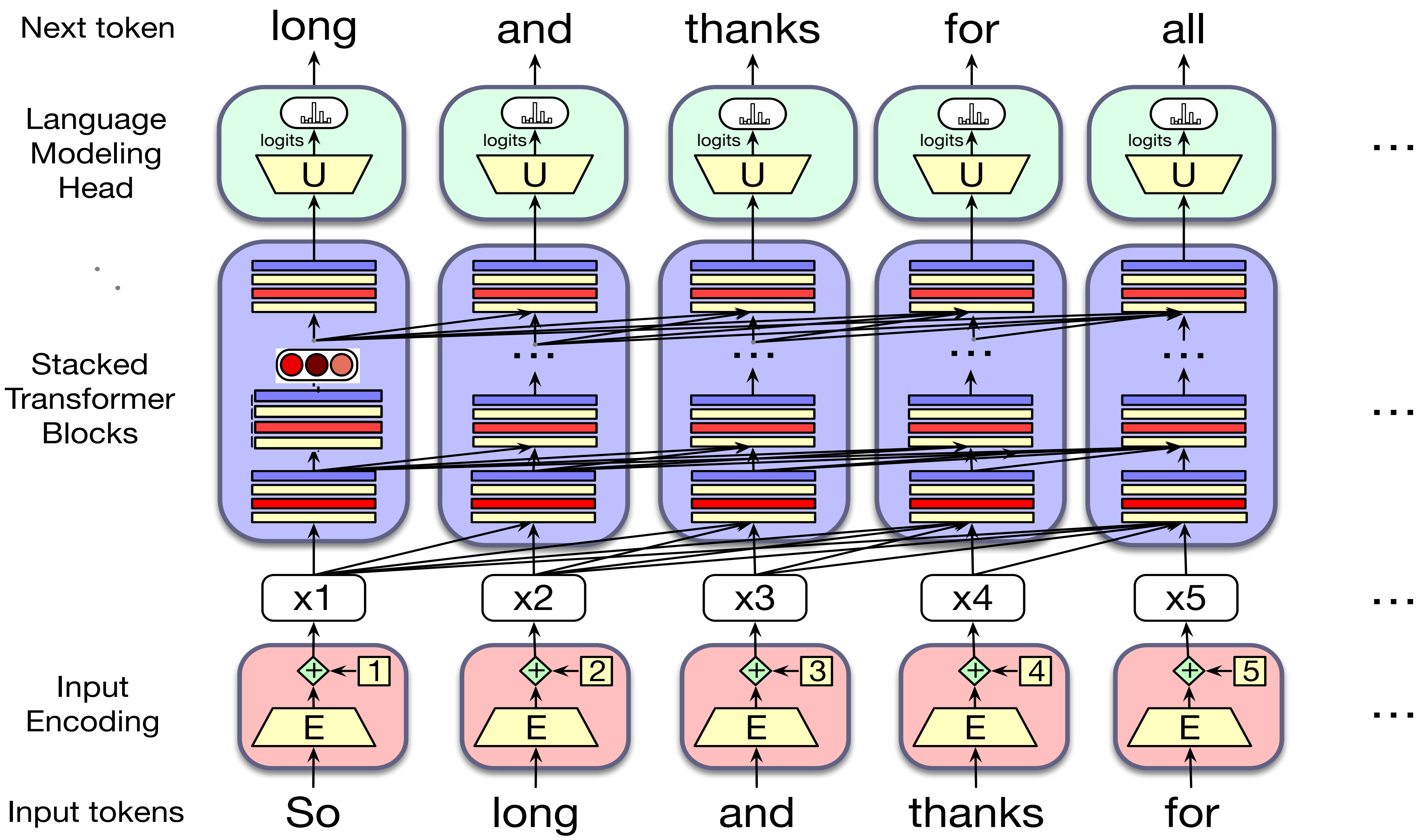
2019 Prompting

Transformers

Attention

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`**

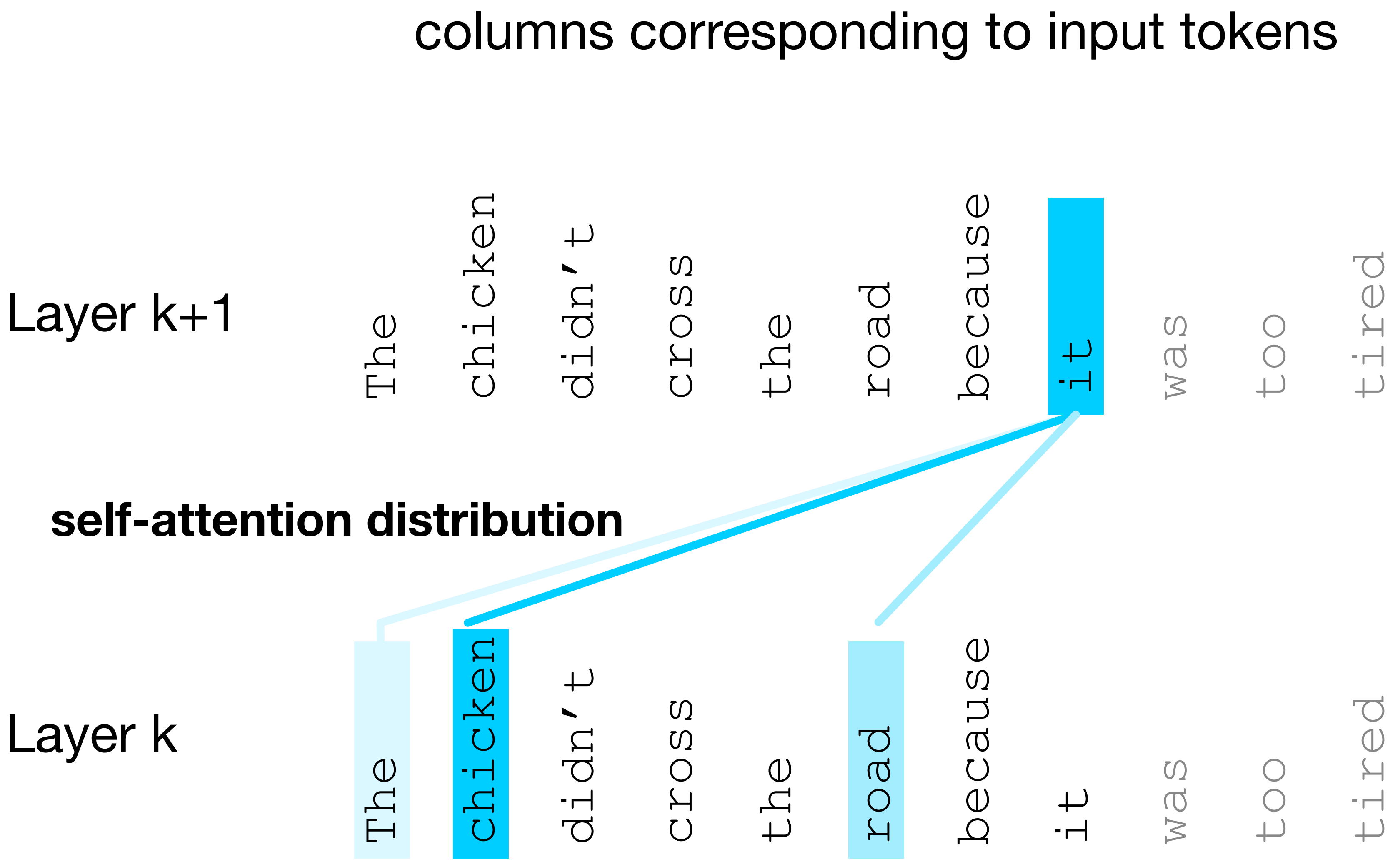
At this point in the sentence, it's probably referring to either the chicken or the street

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:

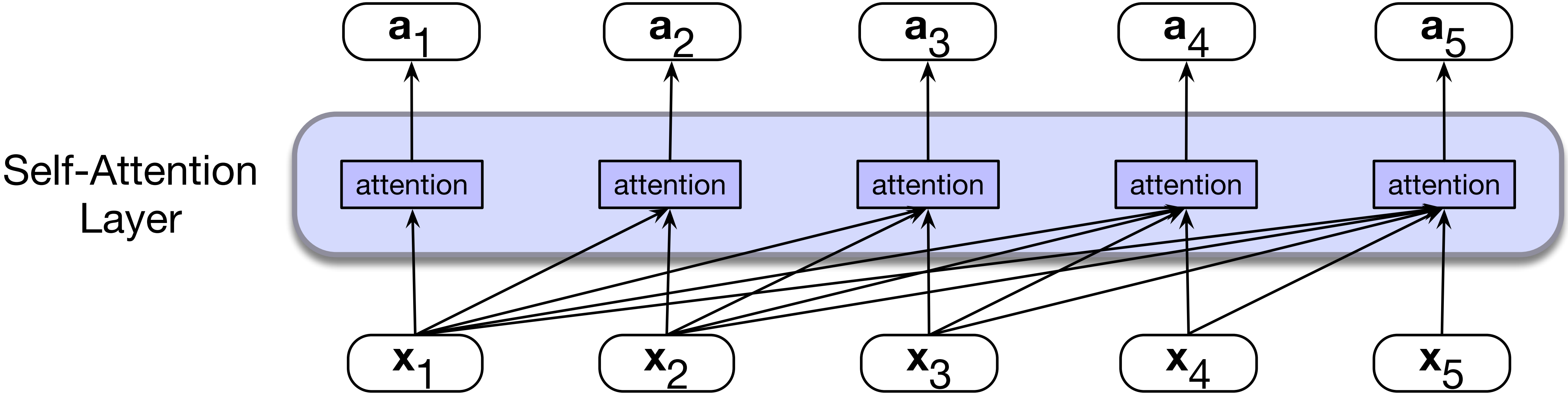


Attention definition

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 (in Decoder models)



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

Given a sequence of token embeddings:

$$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \mathbf{x}_4 \quad \mathbf{x}_5 \quad \mathbf{x}_6 \quad \mathbf{x}_7 \quad \mathbf{x}_\mathrm{i}$$

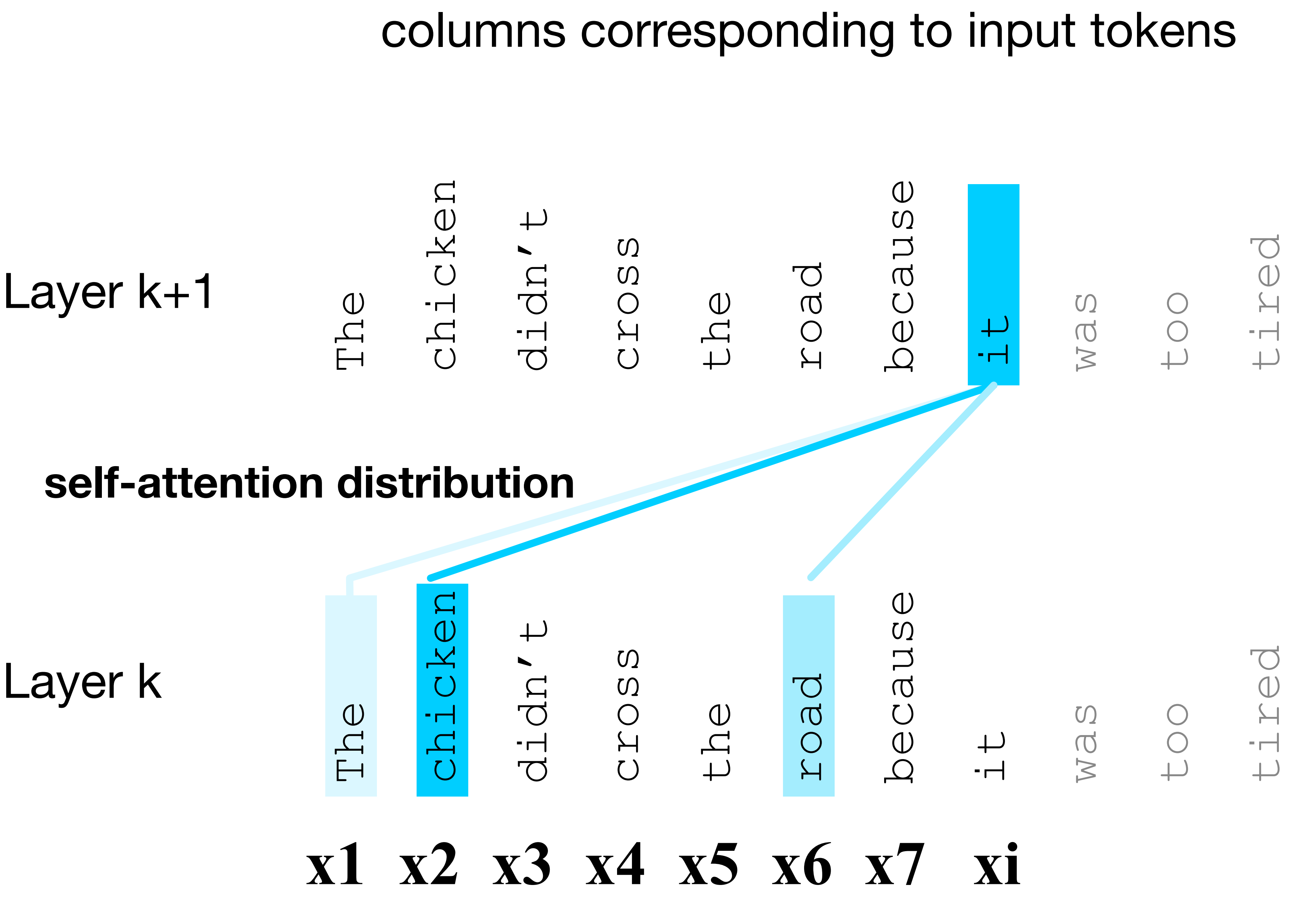
Produce: $\mathbf{a}_\mathrm{i}$ = a weighted sum of $\mathbf{x}_1$ through $\mathbf{x}_7$ (and $\mathbf{x}_\mathrm{i}$)
Weighted by their similarity to $\mathbf{x}_\mathrm{i}$

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

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

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

Intuition of attention:

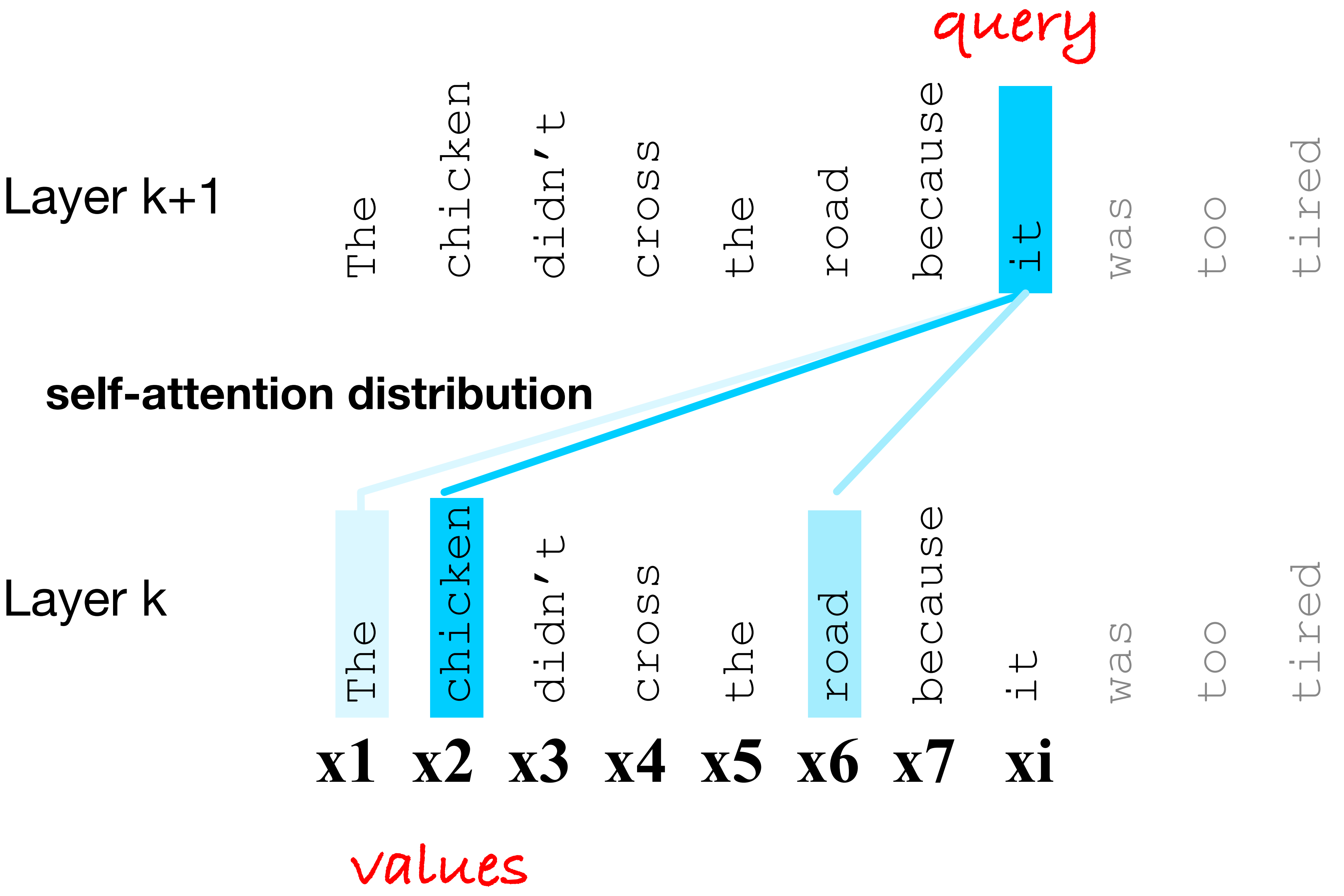


An Actual Attention Head: slightly more complicated

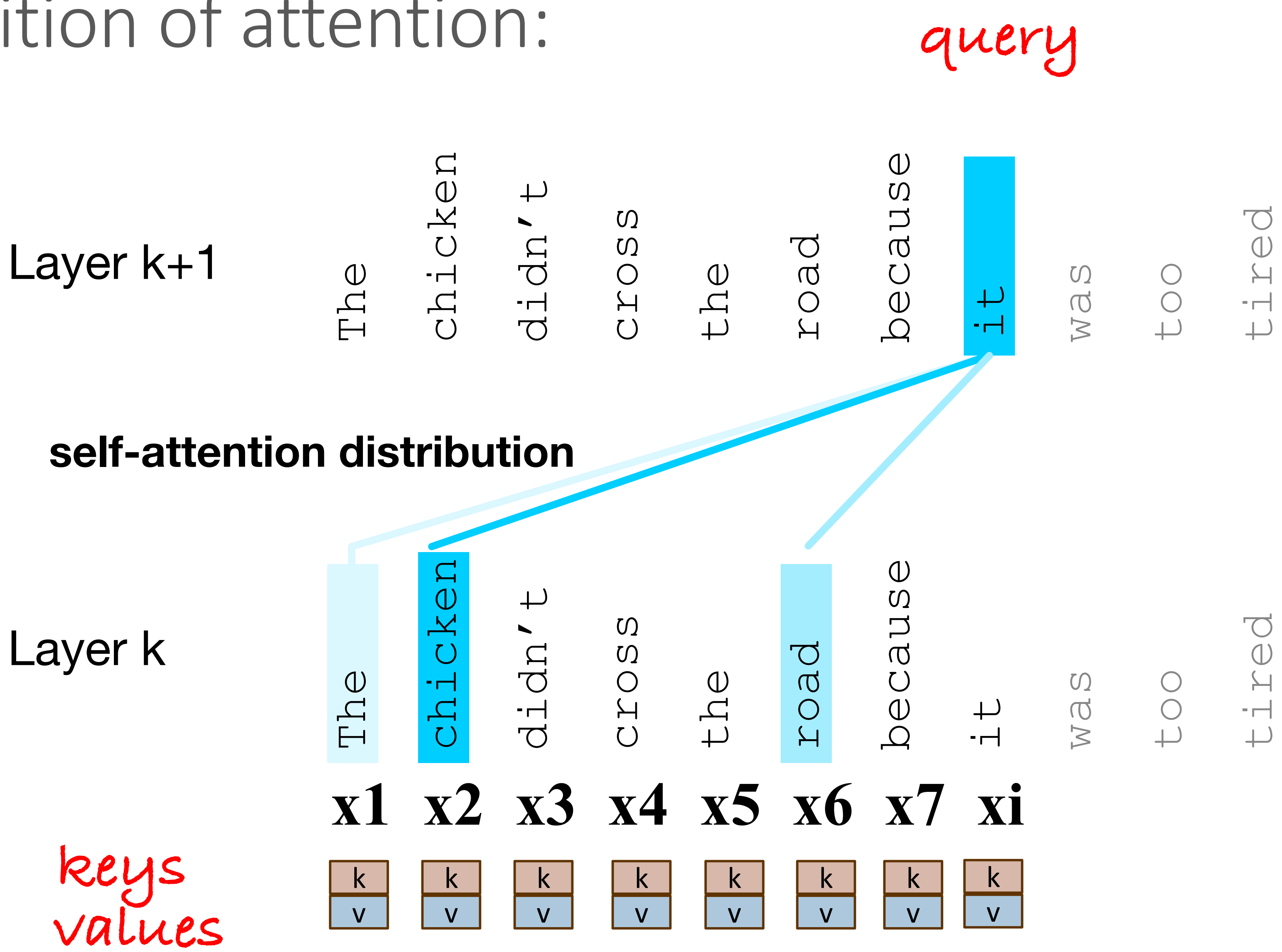
High-level idea: instead of using vectors (like x_i and x_4) directly, we'll represent 3 separate roles each vector $\mathbf{x}_i$ plays:

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

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

An Actual Attention Head: slightly more complicated

Given these 3 representation of $\mathbf{x}_i$

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

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 $\mathbf{k}_j$.

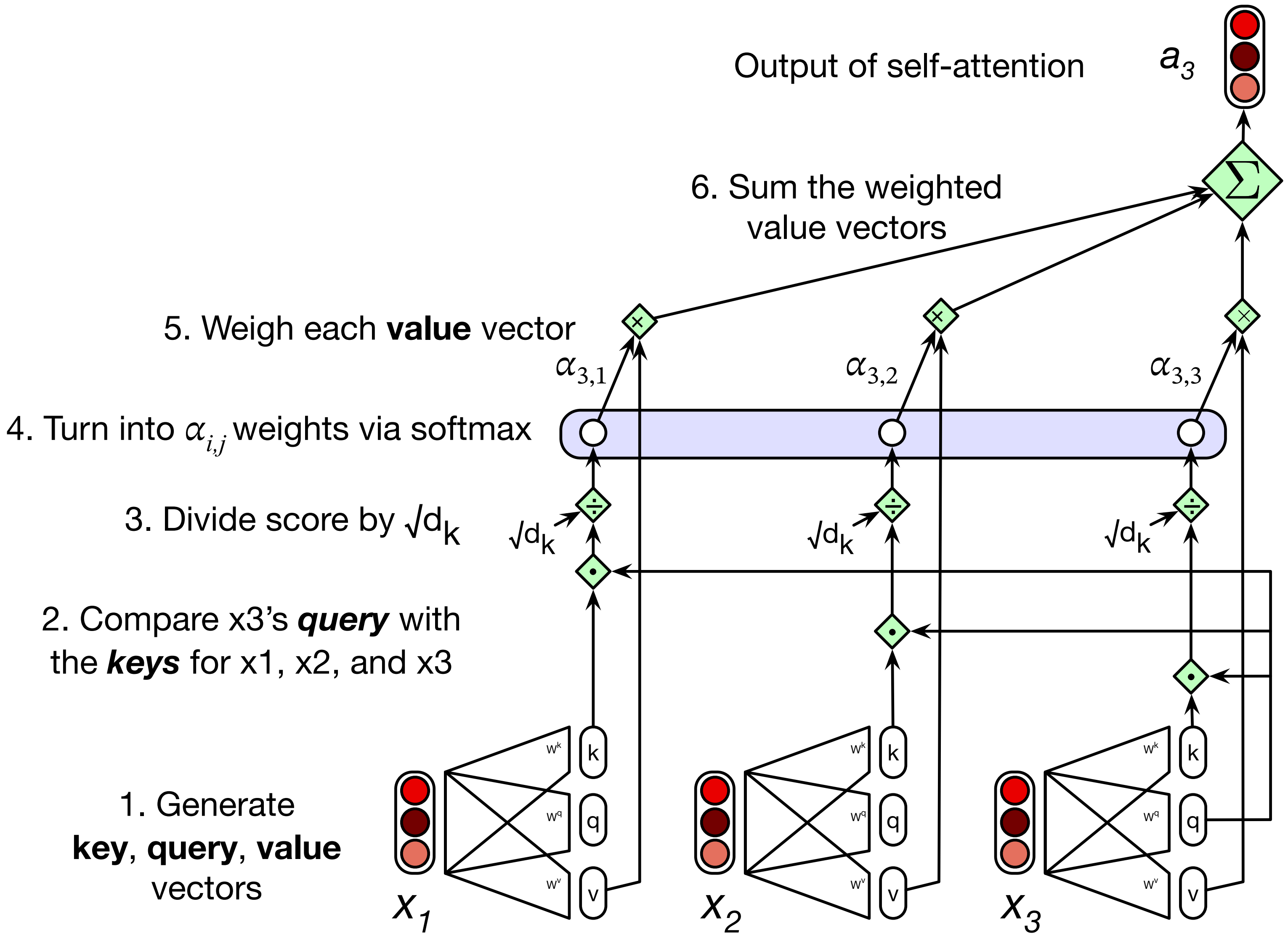
And instead of summing up $\mathbf{x}_j$, we'll sum up $\mathbf{v}_j$

Final equations for one attention head

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

$$\begin{aligned}
\text{score}(\mathbf{x}_i, \mathbf{x}_j) &= \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \\
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \\
\mathbf{a}_i &= \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j
\end{aligned}$$

Calculating the value of a3

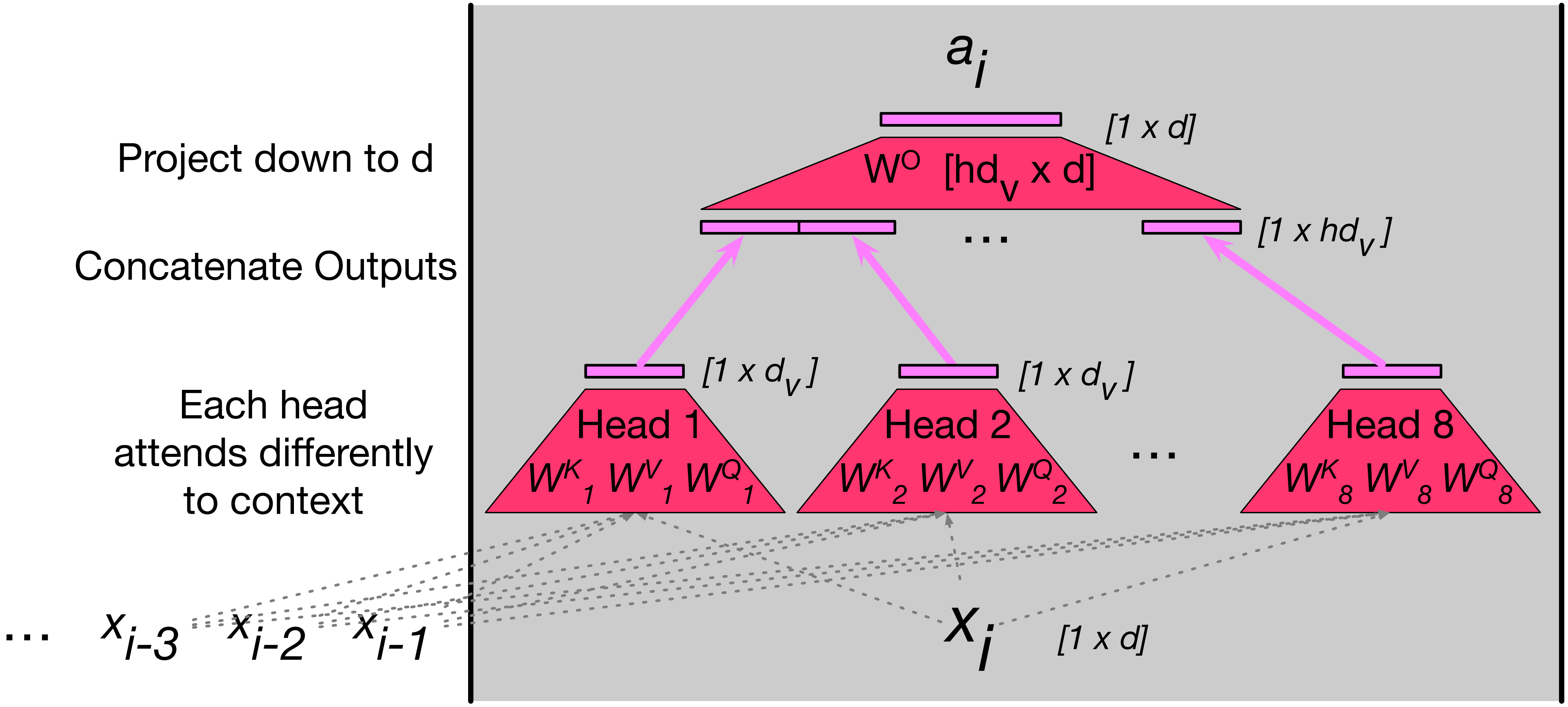


Actual Attention: slightly more complicated

- 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

$$
\begin{aligned}
\mathbf{q}_i^c = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}c}; \quad \mathbf{k}_j^c = \mathbf{x}_j \mathbf{W}^{\mathbf{K}c}; \quad \mathbf{v}_j^c &= \mathbf{x}_j \mathbf{W}^{\mathbf{V}c}; \quad \forall\, c \quad 1 \le c \le h \\
\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) &= \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}} \\
\alpha_{ij}^c &= \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j)) \ \forall j \le i \\
\mathbf{head}_i^c &= \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c \\
\mathbf{a}_i &= (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^h)\mathbf{W}^O \\
\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_N]) &= \mathbf{a}_i
\end{aligned}
$$

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.

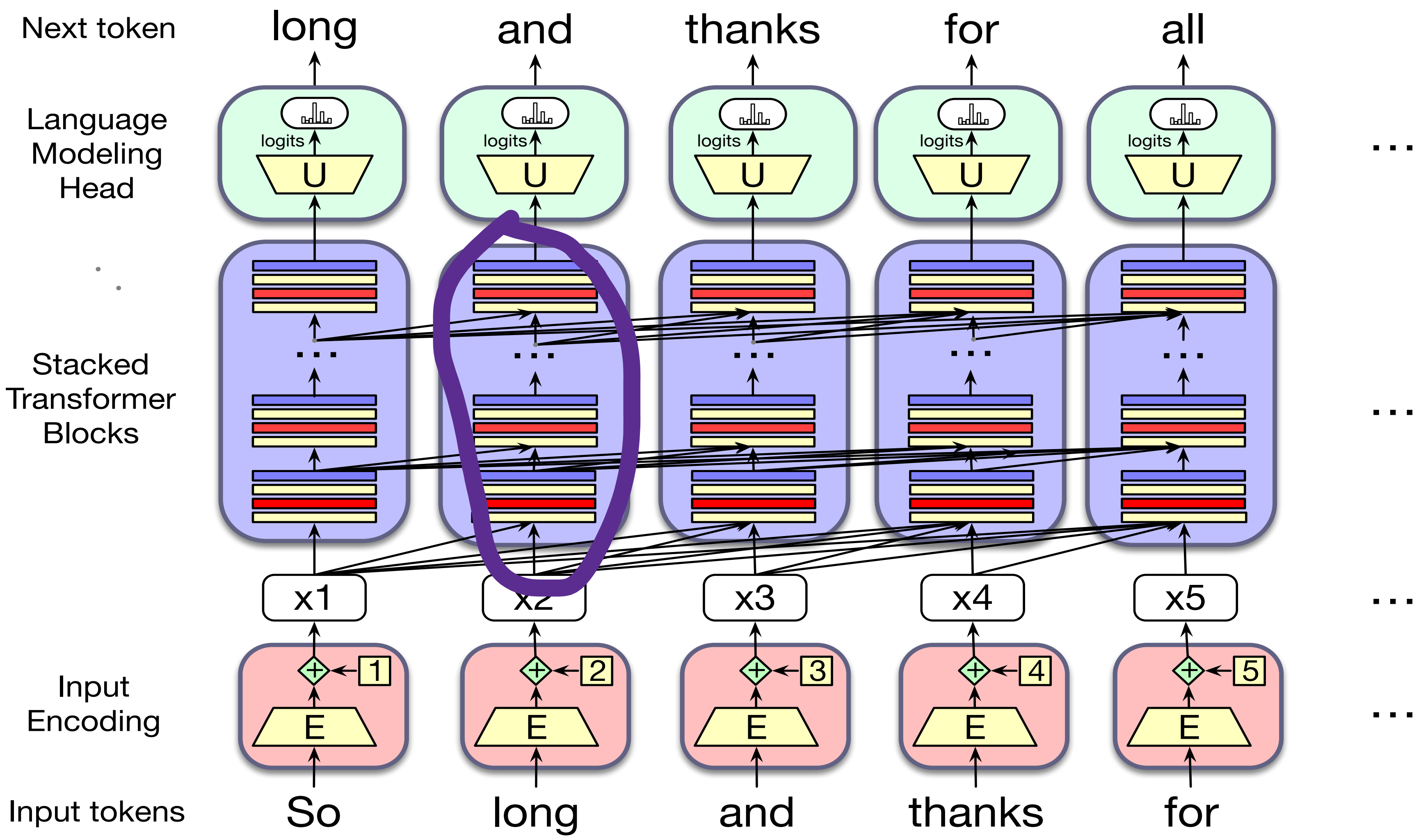
Transformers

Attention

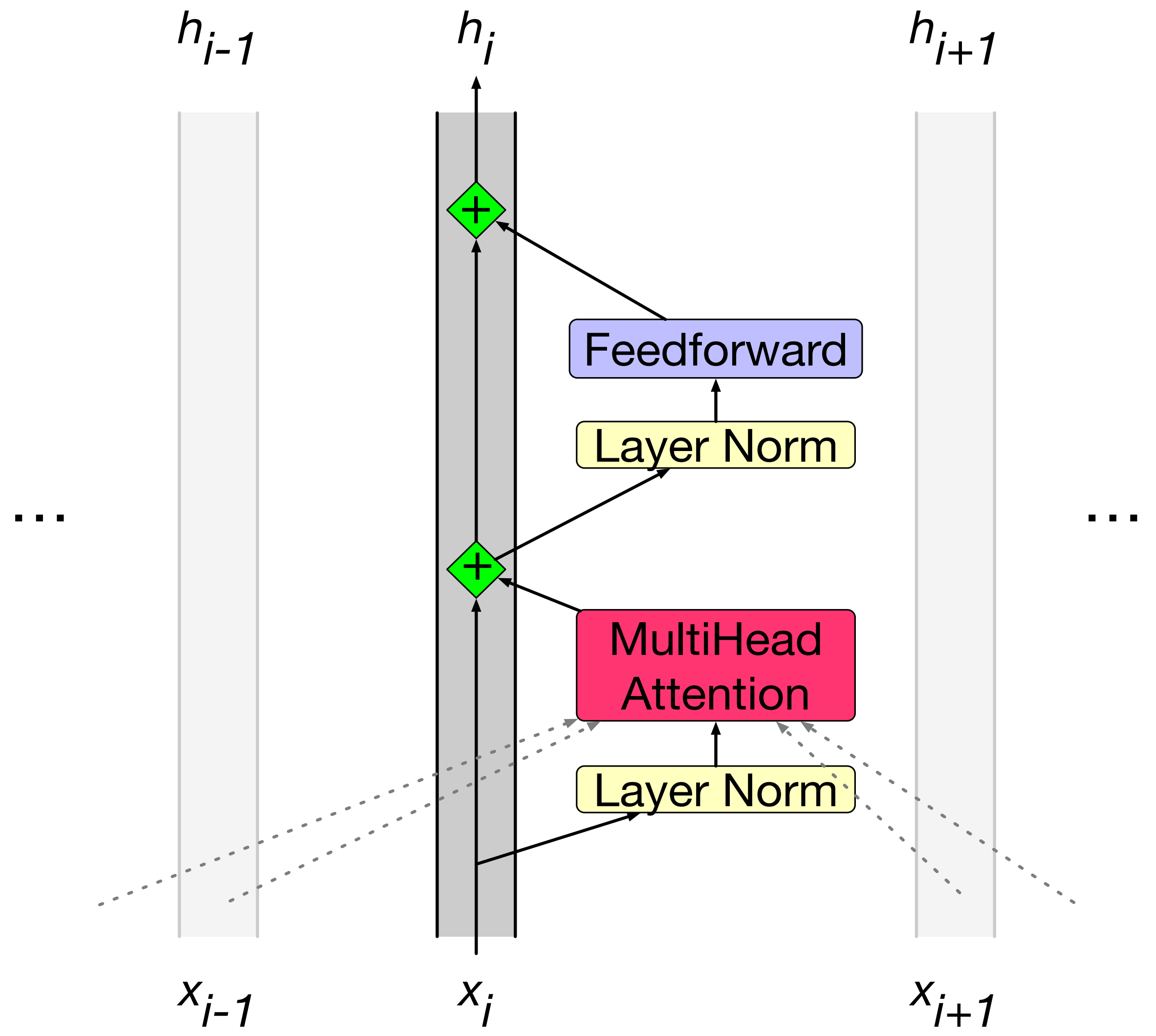
Transformers

The Transformer Block

Reminder: transformer language model

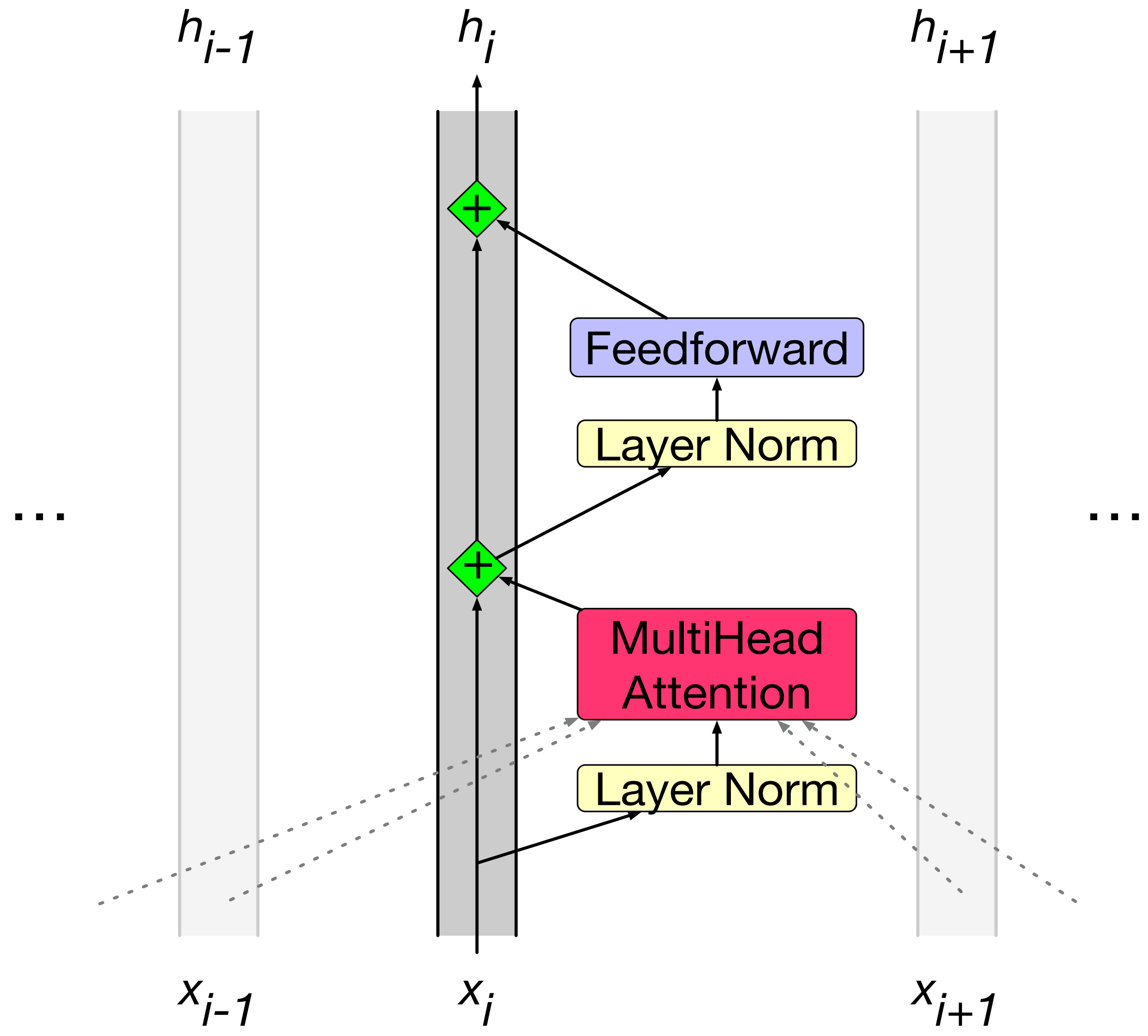


The residual stream: each token gets passed up and modified

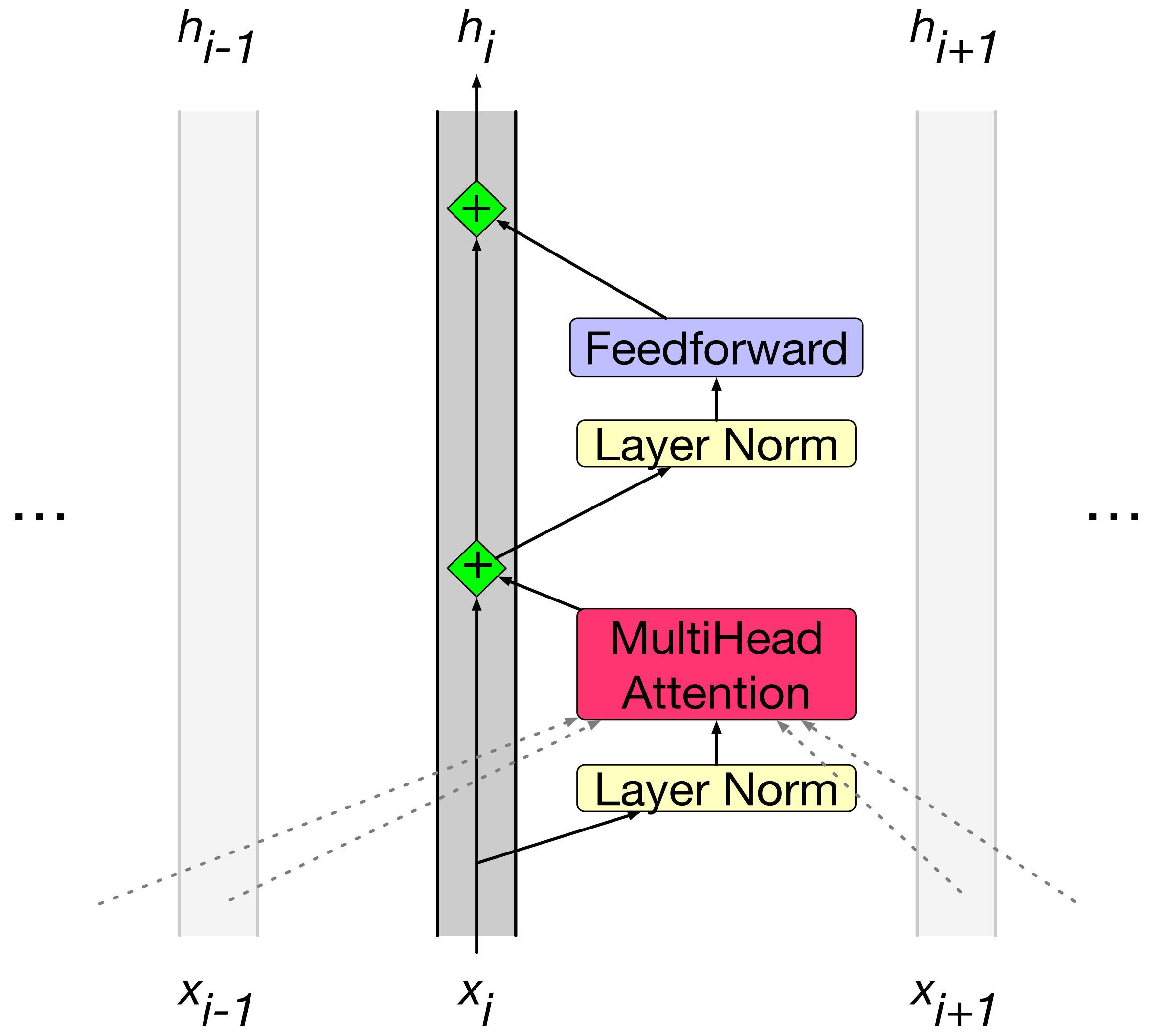


We'll need nonlinearities, so a feedforward layer

$$FFN(\mathbf{x}_i) = ramp(\mathbf{x}_i W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



Layer norm: the vector $\mathbf{x}_i$ 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

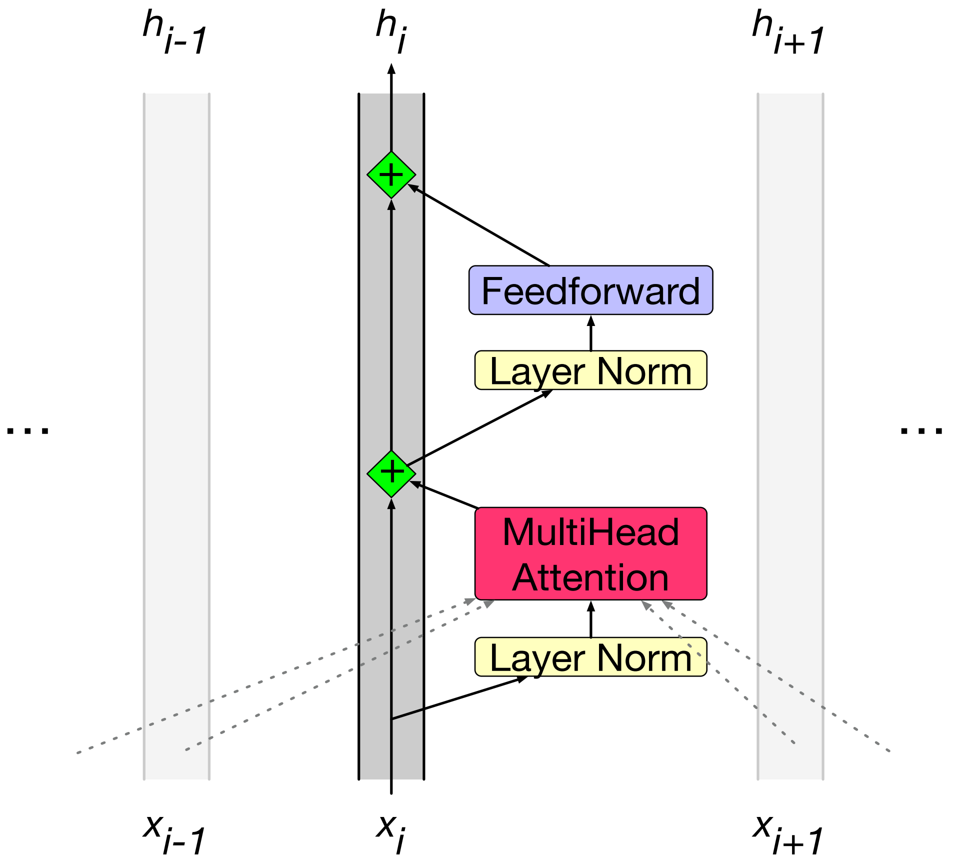
$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

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

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

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

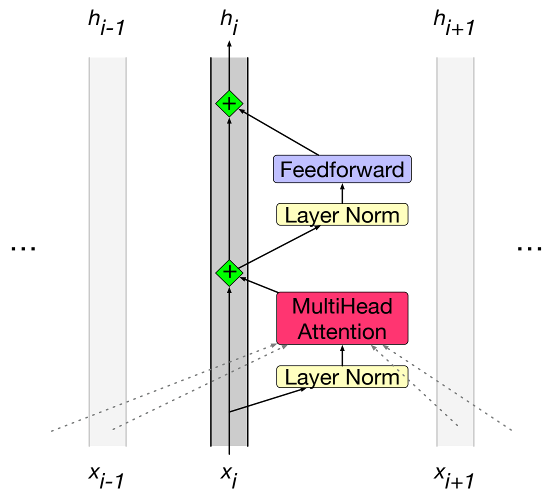
Putting together a single transformer block



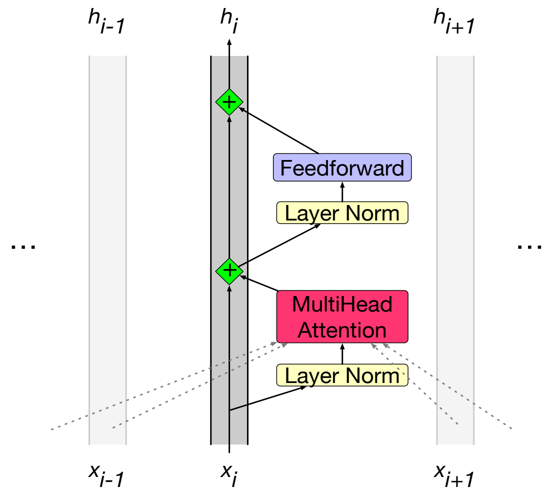
$$
\begin{aligned}
\mathbf{t}_i^1 &= \mathrm{LayerNorm}(\mathbf{x}_i) \\
\mathbf{t}_i^2 &= \mathrm{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{x}_1^1, \cdots, \mathbf{x}_N^1]) \\
\mathbf{t}_i^3 &= \mathbf{t}_i^2 + \mathbf{x}_i \\
\mathbf{t}_i^4 &= \mathrm{LayerNorm}(\mathbf{t}_i^3) \\
\mathbf{t}_i^5 &= \mathrm{FFN}(\mathbf{t}_i^4) \\
\mathbf{h}_i &= \mathbf{t}_i^5 + \mathbf{t}_i^3
\end{aligned}
$$

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

Block 2



Block 1

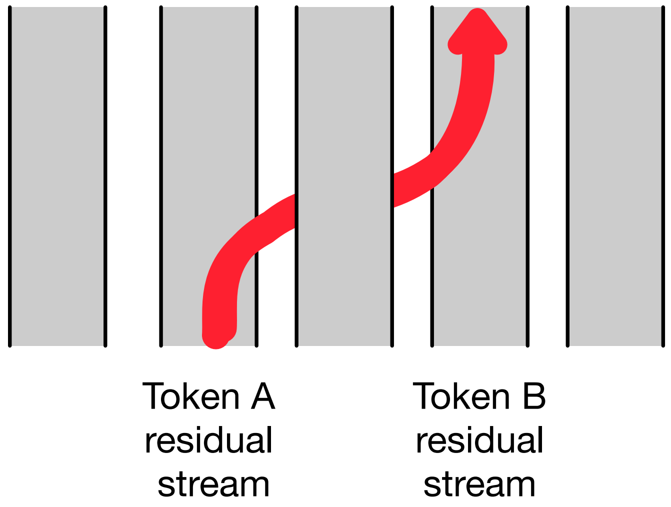


Residual streams and attention

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

The Transformer Block

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**)

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}};\ \ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}};\ \ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

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

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

Masking out the future

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

- What is this mask function?
  QK^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

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

Add $-\infty$ to cells in upper triangle

The softmax will turn it to 0

N

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

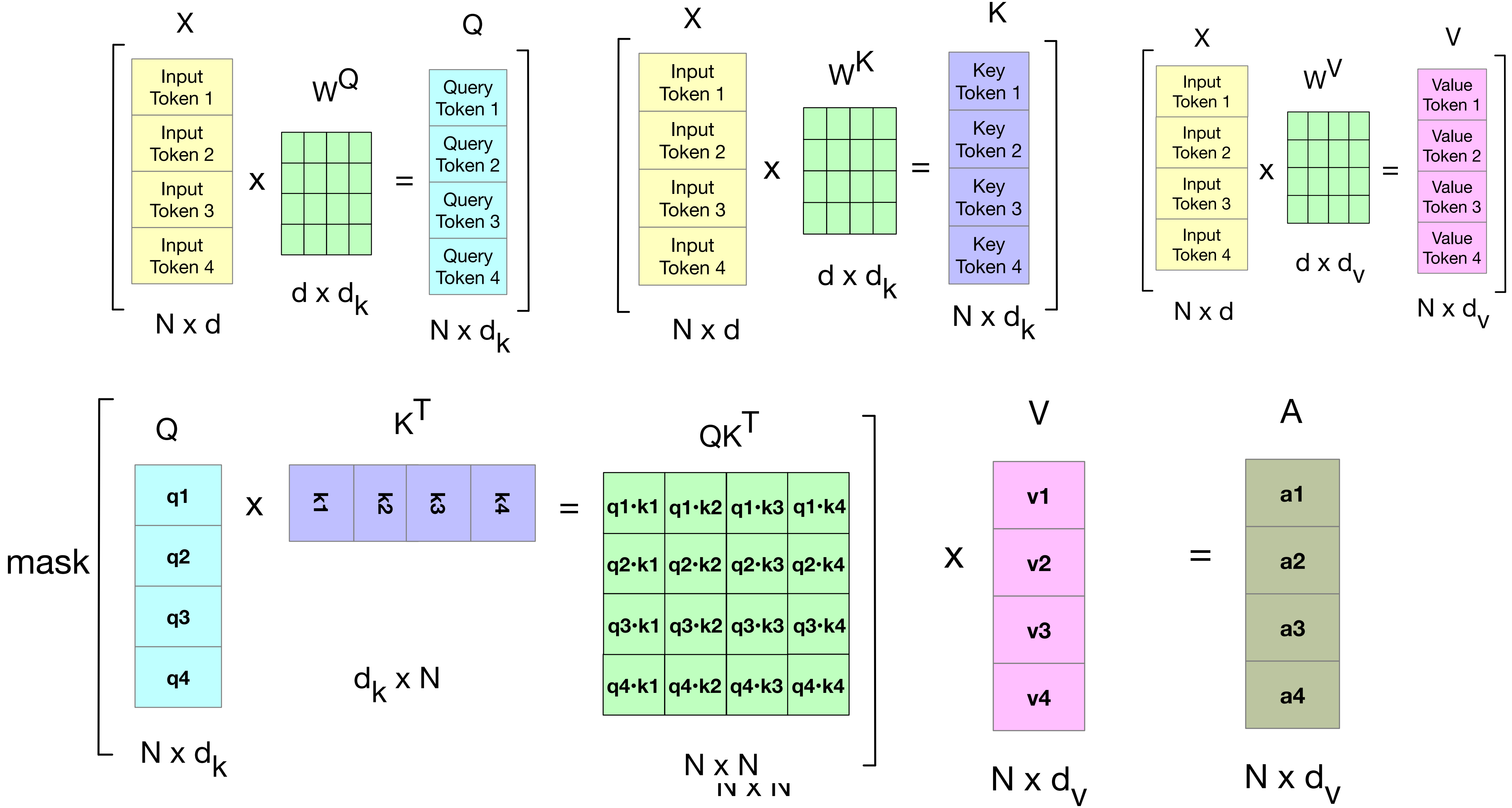
N

Another point: Attention is quadratic in length

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

$$\begin{array}{cc} & \\ N & \begin{array}{|c|c|c|c|} \hline q1{\cdot}k1 & -\infty & -\infty & -\infty \\ \hline q2{\cdot}k1 & q2{\cdot}k2 & -\infty & -\infty \\ \hline q3{\cdot}k1 & q3{\cdot}k2 & q3{\cdot}k3 & -\infty \\ \hline q4{\cdot}k1 & q4{\cdot}k2 & q4{\cdot}k3 & q4{\cdot}k4 \\ \hline \end{array} \\ & N \end{array}$$

Attention again



Parallelizing Multi-head Attention

$$\mathbf{Q^i} = \mathbf{XW^{Qi}}\,; \quad \mathbf{K^i} = \mathbf{XW^{Ki}}\,; \quad \mathbf{V^i} = \mathbf{XW^{Vi}}$$

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

$$\mathrm{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W^O}$$

Parallelizing Multi-head Attention

$$\begin{aligned} \mathbf{O} &= \mathrm{LayerNorm}(\mathbf{X} + \mathrm{MultiHeadAttention}(\mathbf{X})) \\ \mathbf{H} &= \mathrm{LayerNorm}(\mathbf{O} + \mathrm{FFN}(\mathbf{O})) \end{aligned}$$

or

$$\begin{aligned} \mathbf{T}^1 &= \mathrm{MultiHeadAttention}(\mathbf{X}) \\ \mathbf{T}^2 &= \mathbf{X} + \mathbf{T}^1 \\ \mathbf{T}^3 &= \mathrm{LayerNorm}(\mathbf{T}^2) \\ \mathbf{T}^4 &= \mathrm{FFN}(\mathbf{T}^3) \\ \mathbf{T}^5 &= \mathbf{T}^4 + \mathbf{T}^3 \\ \mathbf{H} &= \mathrm{LayerNorm}(\mathbf{T}^5) \end{aligned}$$

Transformers

Parallelizing Attention Computation

Transformers

Input and output: Position embeddings and the Language Model Head

Token and Position Embeddings

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 embedding for each input

- token embedding
- positional embedding

Token Embeddings

Embedding matrix 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

w = [5,4000,10532,2224]

2. Select the corresponding rows from E, each row an embedding

- (row 5, row 4000, row 10532, row 2224).

Position Embeddings

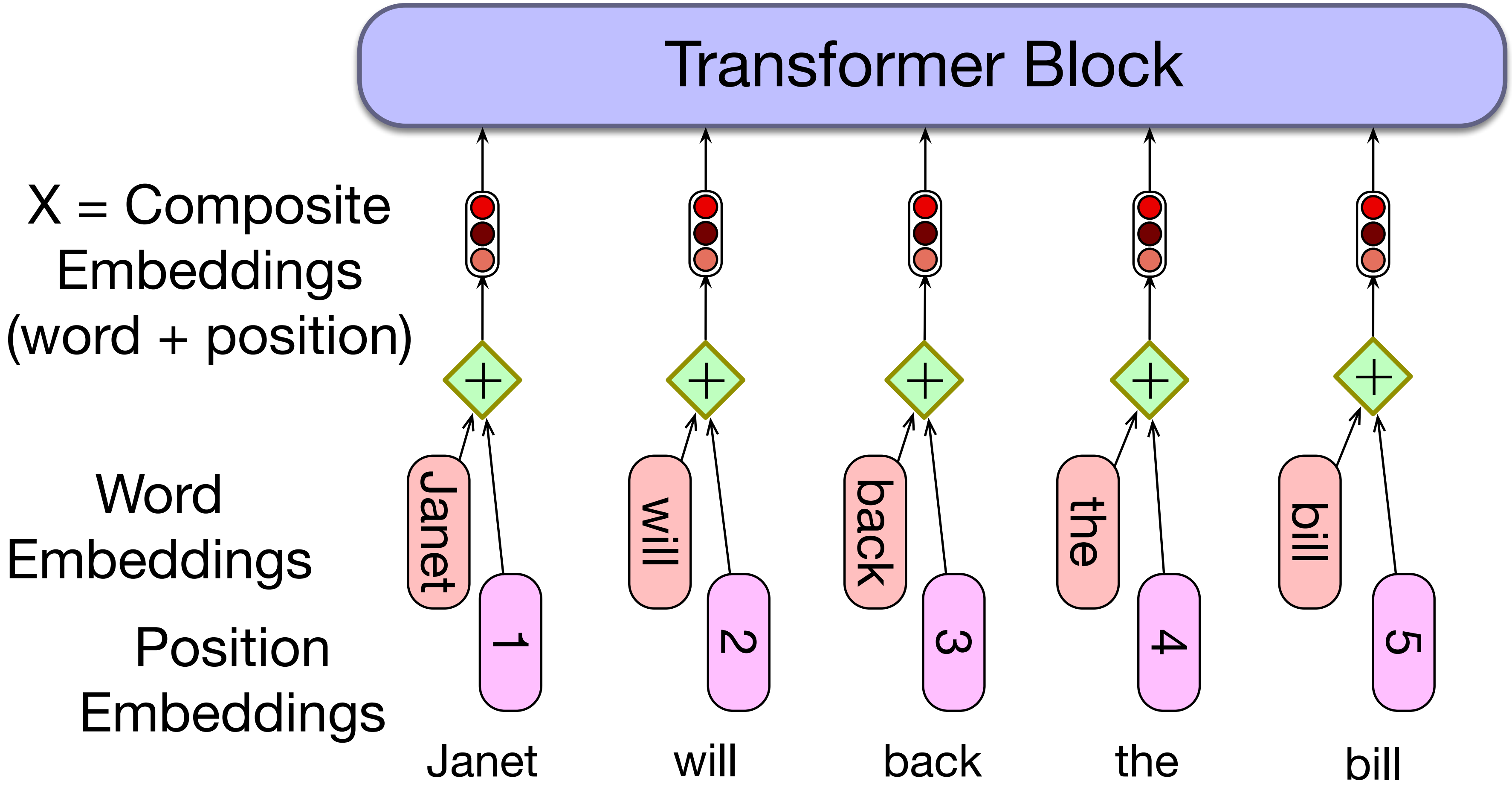
There are many methods, but we'll just describe the simplest: absolute position.

Goal: learn a position embedding matrix *E*pos of shape [1 × *N*].

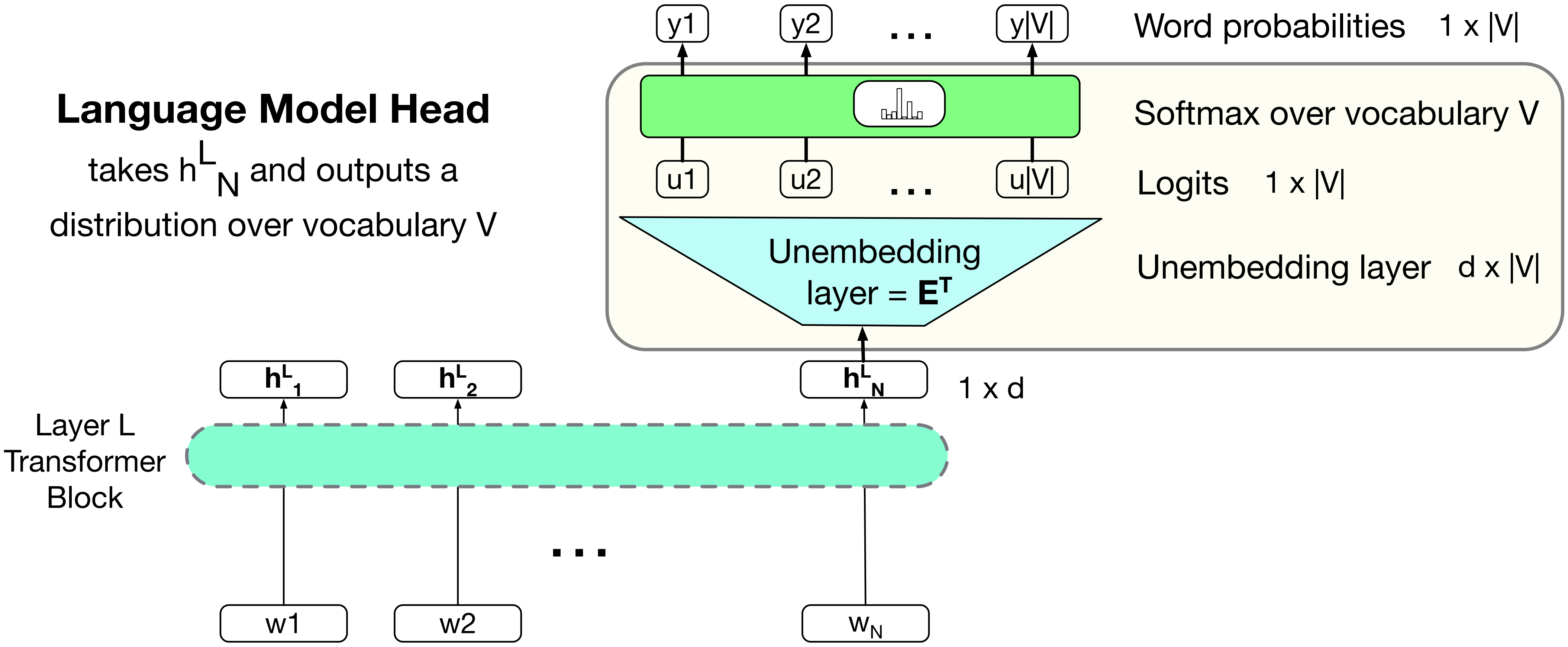
Start with randomly initialized embeddings

- one for each integer up to some maximum length.
- i.e., just as we have an embedding for token *fish*, we'll have an embedding for position 3 and position 17.
- As with word embeddings, these position embeddings are learned along with other parameters during training.

Each **x** is just the sum of word and position embeddings

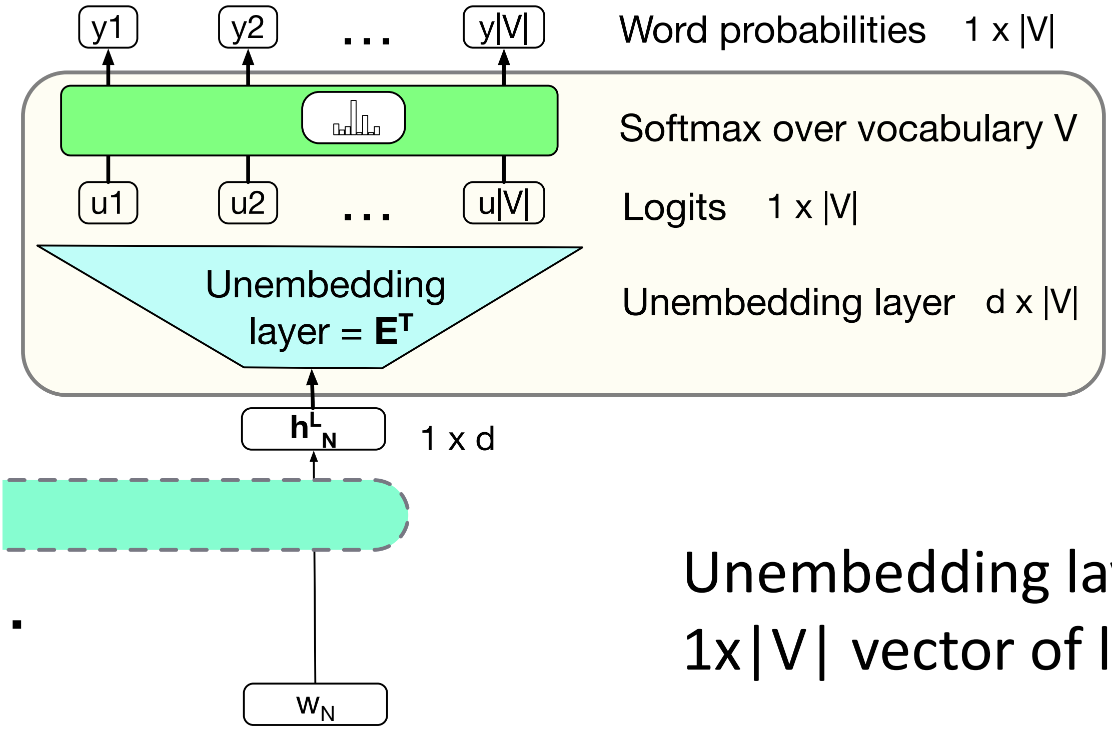


Language modeling head



Language modeling head

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

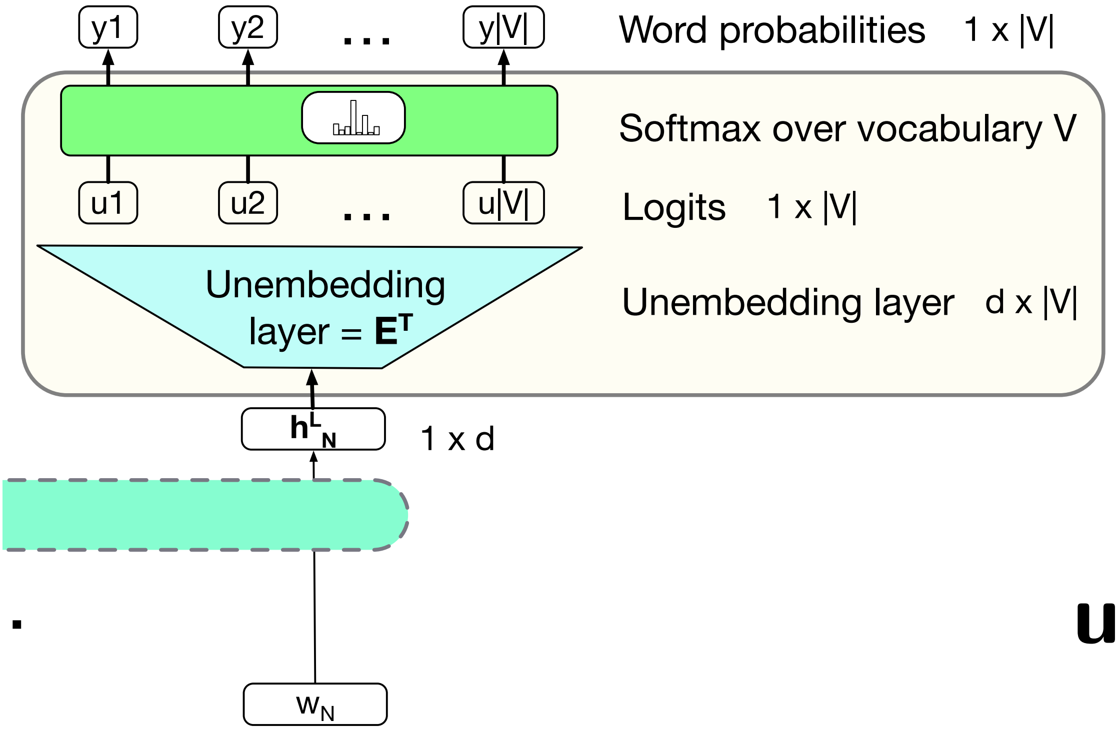


Why "unembedding"? **Tied** to $\mathbf{E}^T$

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

Unembedding layer maps from an embedding to a 1x|V| vector of logits

Language modeling head



Logits, the score vector u

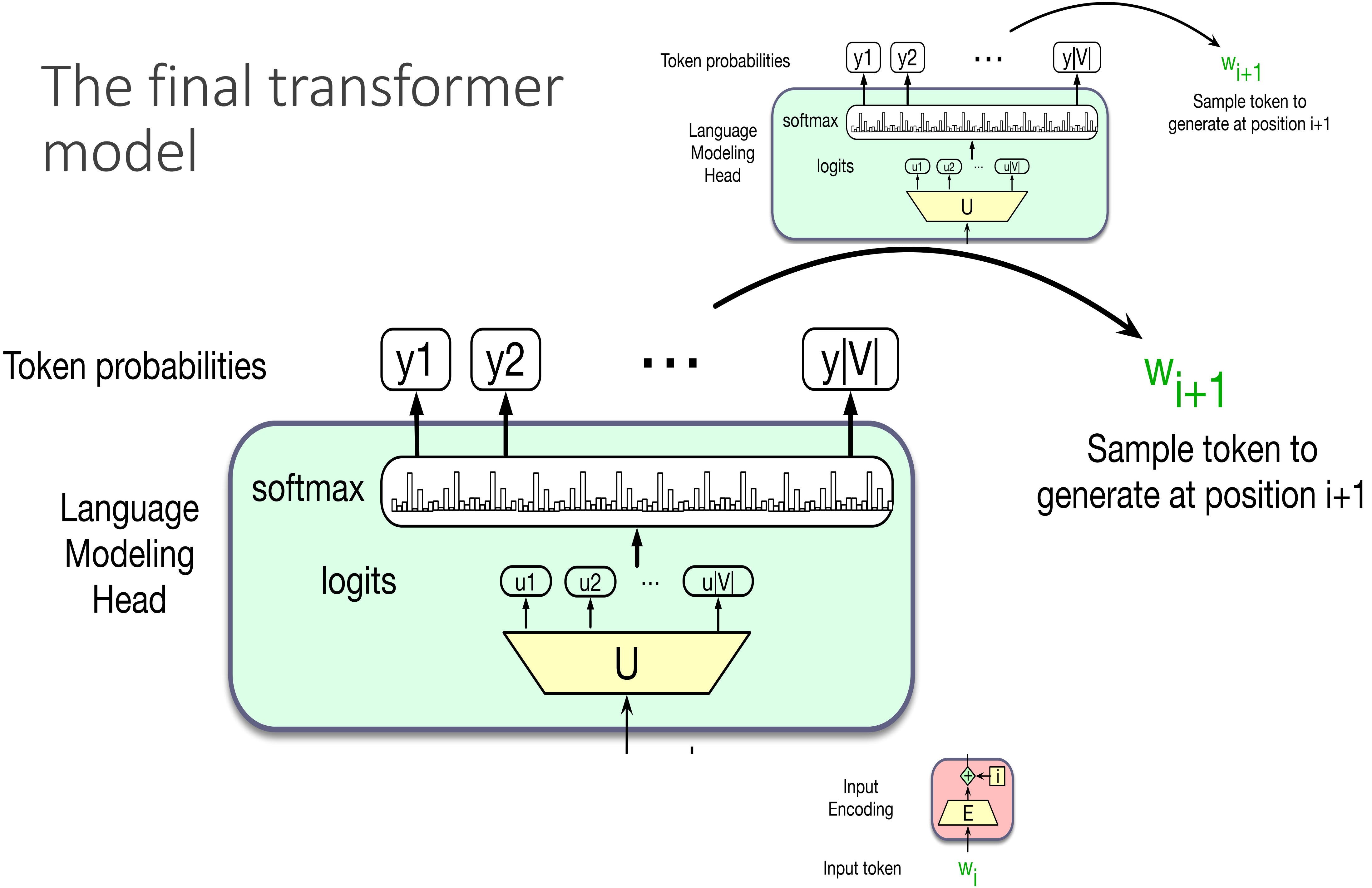
One score for each of the $|V|$ possible words in the vocabulary V. Shape $1 \times |V|$.

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

$$\mathbf{u} = \mathbf{h}^L_N \, \mathbf{E}^T$$

$$\mathbf{y} = \mathrm{softmax}(\mathbf{u})$$

The final transformer model



Transformers

Input and output: Position embeddings and the Language Model Head

Large Language Models

Dealing with Scale

Scaling Laws

LLM performance depends on

- Model size: the number of parameters not counting embeddings
- Dataset size: the amount of training data
- Compute: Amount of compute (in FLOPS or etc

Can improve a model by adding parameters (more layers, wider contexts), more data, or 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

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 × 96 × 122882 ≈ 175 billion parameters.

KV Cache

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

$$\mathbf{A} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\intercal}}{\sqrt{d_k}}\right)\mathbf{V}$$

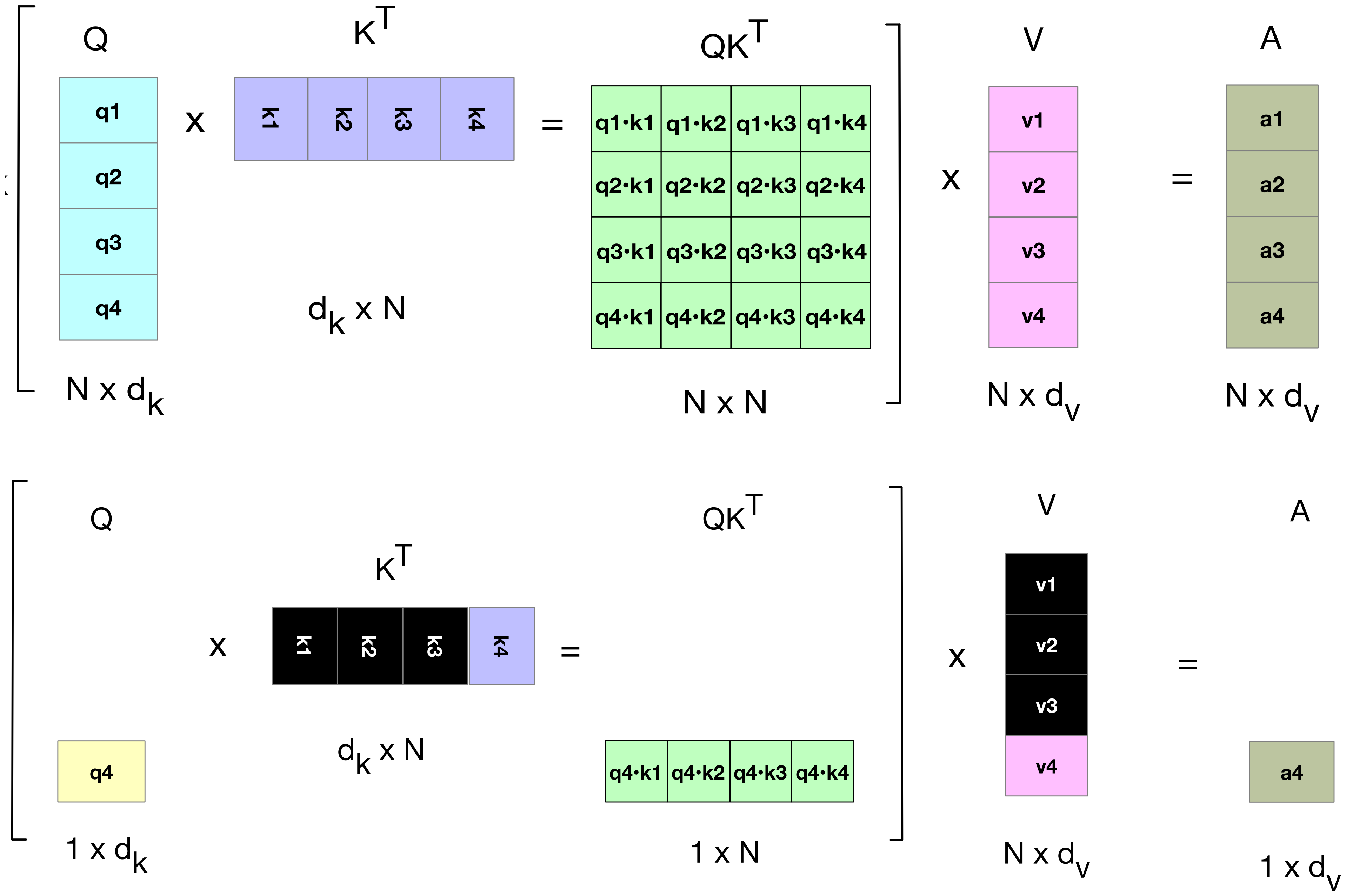
But not at inference! We generate the next tokens **one at a time!**

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


Q
q1
q2
q3
q4
N x d_k
x
K^T
k1
k2
k3
k4
d_k x N
=
QK^T
q1•k1 q1•k2 q1•k3 q1•k4
q2•k1 q2•k2 q2•k3 q2•k4
q3•k1 q3•k2 q3•k3 q3•k4
q4•k1 q4•k2 q4•k3 q4•k4
N x N
x
V
v1
v2
v3
v4
N x d_v
=
A
a1
a2
a3
a4
N x d_v
Q
q4
1 x d_k
x
K^T
k1
k2
k3
k4
d_k x N
=
QK^T
q4•k1 q4•k2 q4•k3 q4•k4
1 x N
x
V
v1
v2
v3
v4
N x d_v
=
A
a4
1 x d_v

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)

- Trransformers 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 + ΔW with W + BA.

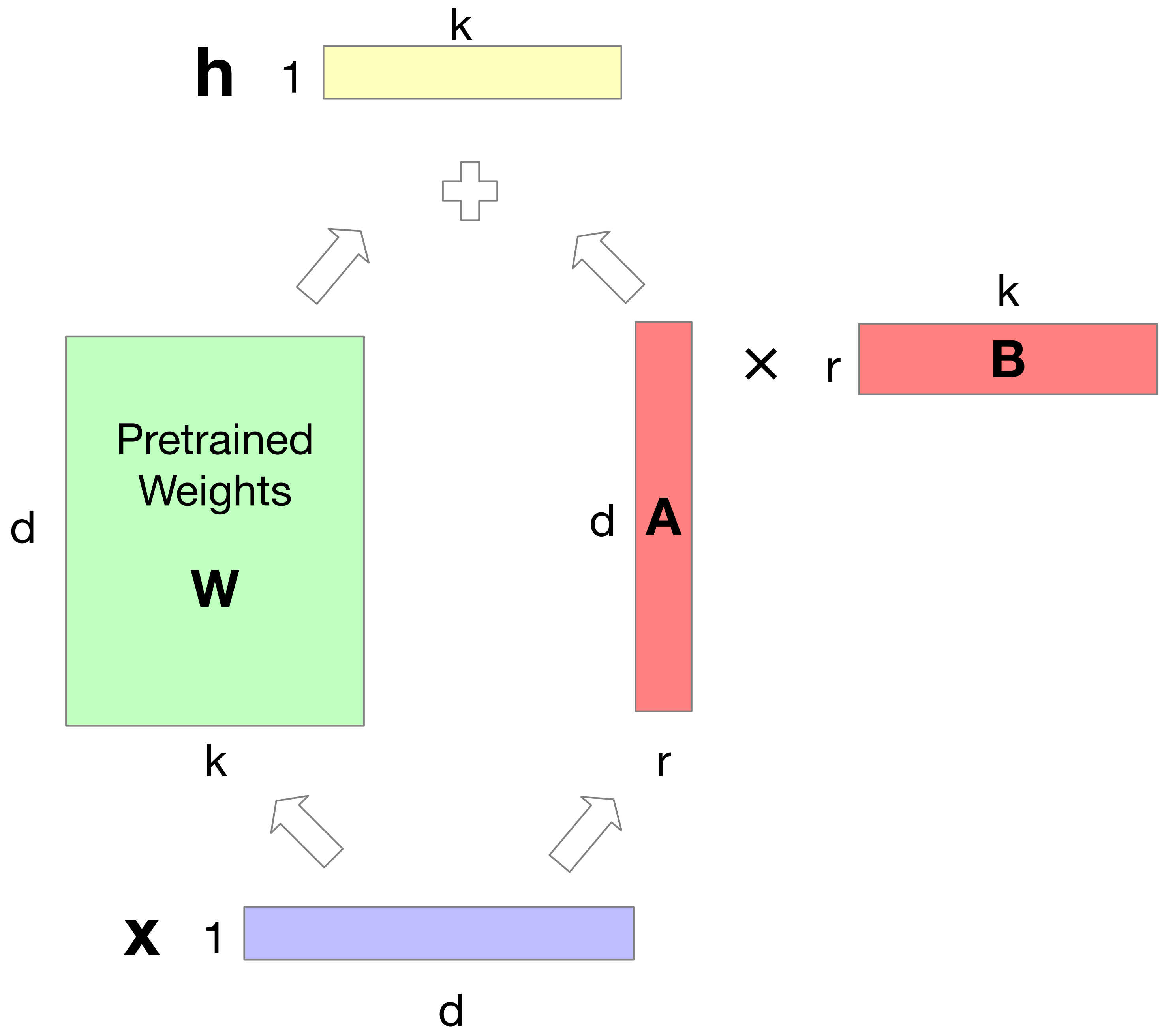
Forward pass: instead of

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

We do

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

LoRA



Large Language Models

Dealing with Scale