Introduction to Large Language Models

Spring 2026

Transformers Architecture

(Some slides adapted from Ralph Grishman at NYU,
Yejin Choi at UWashington, N. Tomura at UDepaul, Jurafsky and
Martin, CS224N, CS224d at Stanford and other resourses on the web)

Attention

Attention is a *general* Deep Learning technique

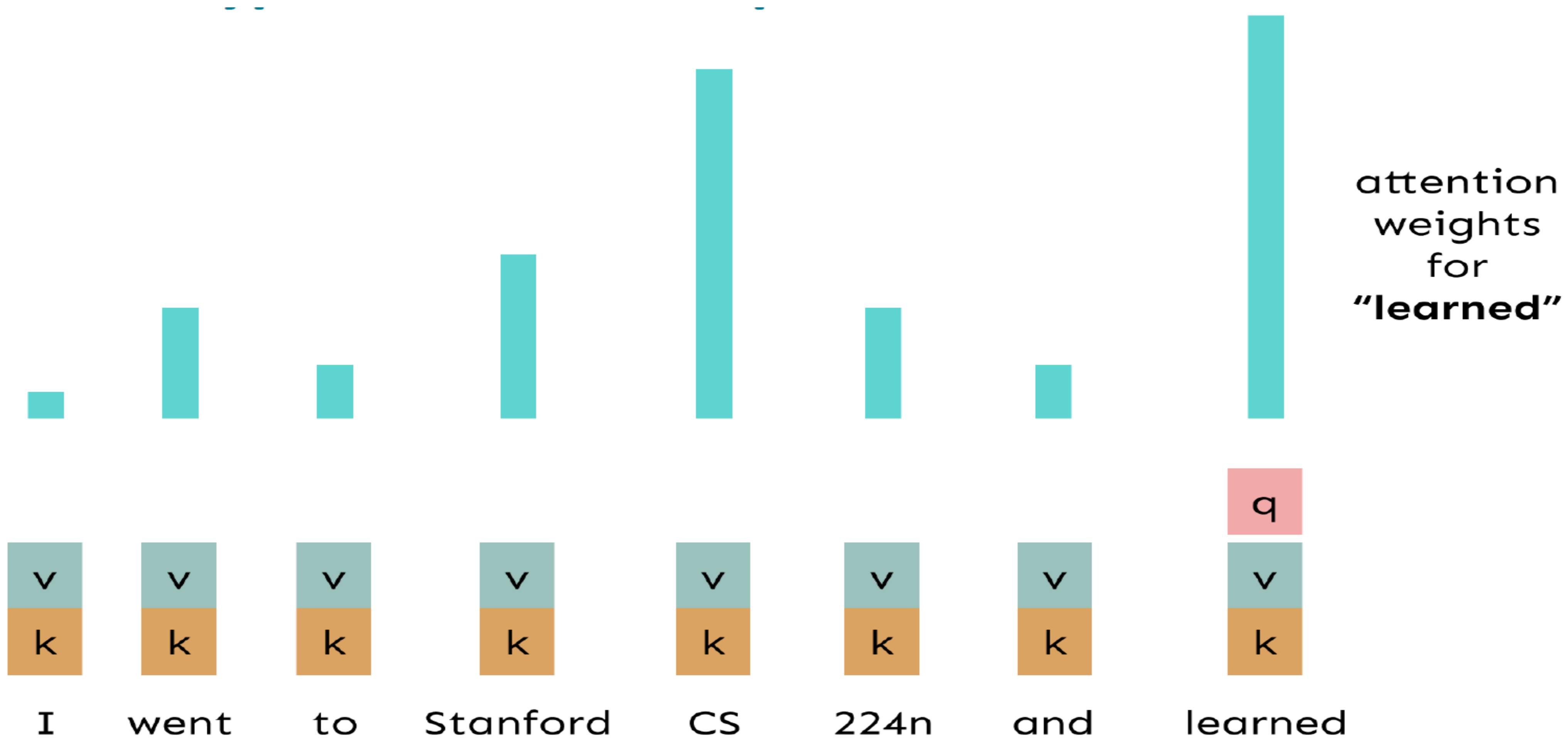
- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- However: You can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

- More general definition of attention:
  - Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the query *attends to* the values.
- For example, in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).

Self-Attention Hypothetical Example

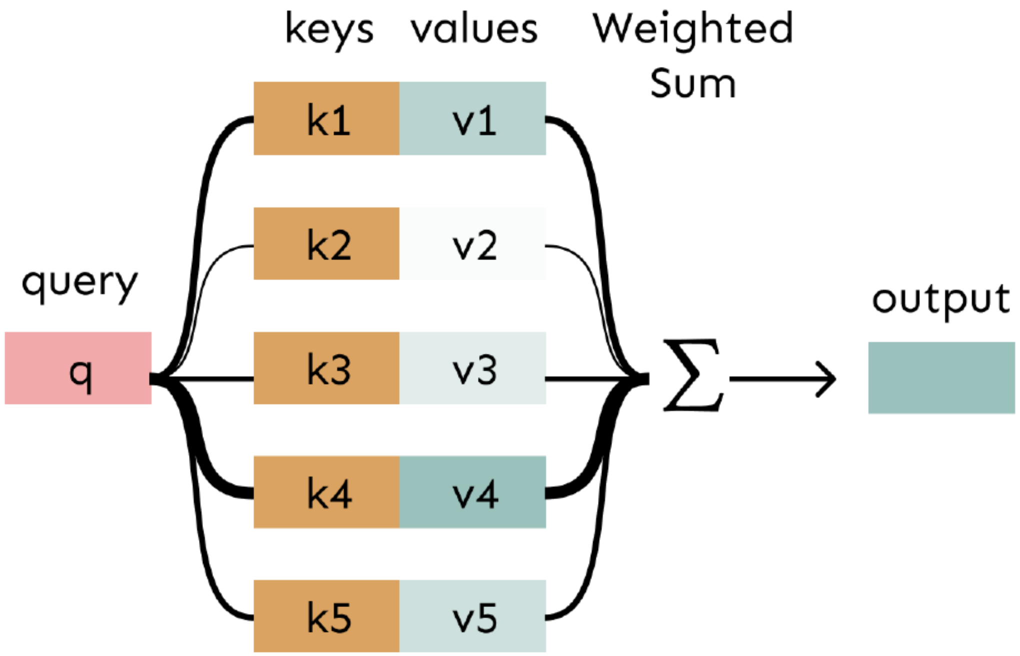
- **Self** attention: to generate y_t , we need to pay attention to $y_{<t}$



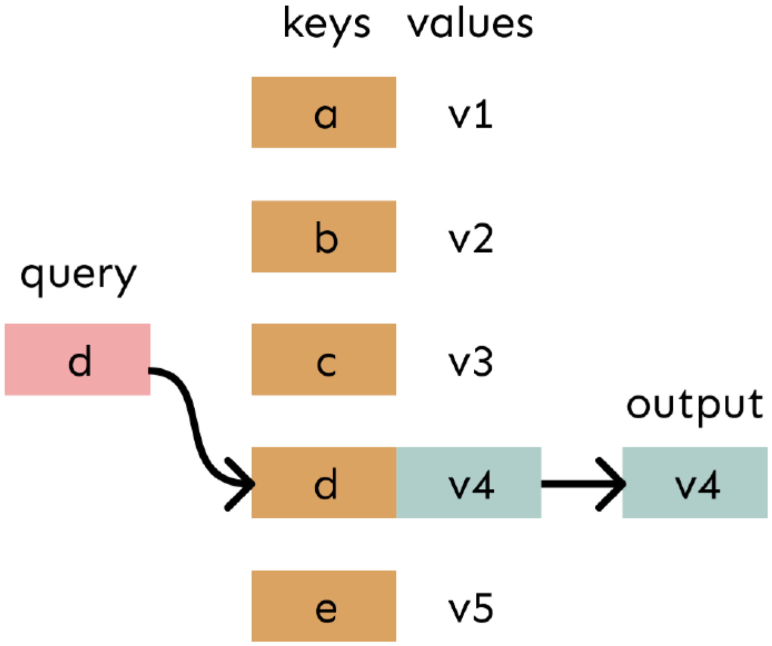
Attention is weighted averaging, which lets you do lookups!

Attention is just a **weighted** average – this is very powerful if the weights are learned!

In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



Self-Attention: keys, queries, values from the same sequence

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_{\boldsymbol{i}}$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices $\mathrm{Q}, \mathrm{K}, \mathrm{V}$, each in $\mathbb{R}^{d\times d}$

$$\boldsymbol{q}_i = Q\boldsymbol{x}_{\boldsymbol{i}} \text{ (queries)} \qquad \boldsymbol{k}_i = K\boldsymbol{x}_{\boldsymbol{i}} \text{ (keys)} \qquad \boldsymbol{v}_i = V\boldsymbol{x}_{\boldsymbol{i}} \text{ (values)}$$

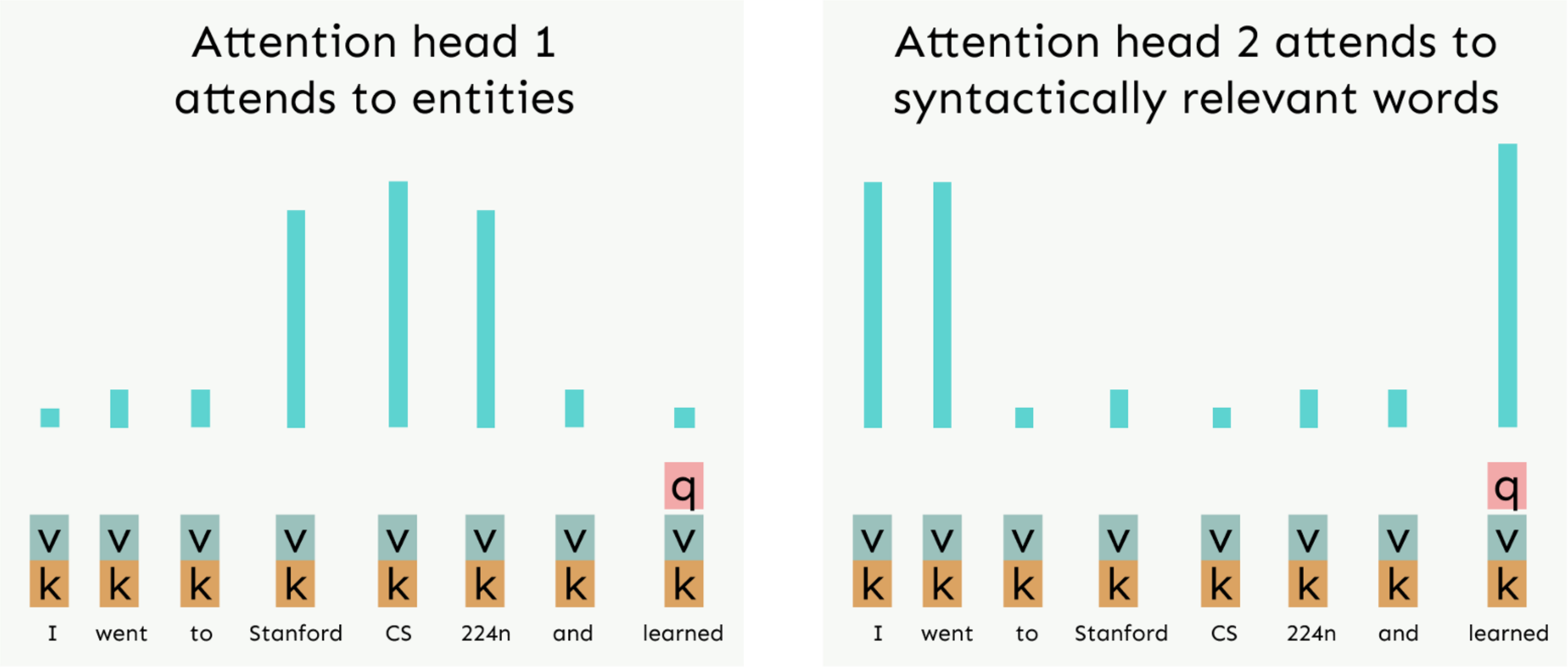
2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_{\boldsymbol{i}}^{\top}\boldsymbol{k}_{\boldsymbol{j}} \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'}\exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_{\boldsymbol{j}} \boldsymbol{\alpha}_{ij}\,\boldsymbol{v}_{\boldsymbol{j}}$$

Hypothetical Example of Multi-Head Attention


Attention head 1
attends to entities
q
v v v v v v v v
k k k k k k k k
I went to Stanford CS 224n and learned
Attention head 2 attends to
syntactically relevant words
q
v v v v v v v v
k k k k k k k k
I went to Stanford CS 224n and learned

I went to Stanford CS 224n and learned

Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x_i^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h.
- Each attention head performs attention independently:
  - $\text{output}_\ell = \text{softmax}\left(X Q_\ell K_\ell^\top X^\top\right) * X V_\ell$, where $\text{output}_\ell \in \mathbb{R}^{n \times d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1, \dots, \text{output}_h] Y$, where $Y \in \mathbb{R}^{d \times d}$

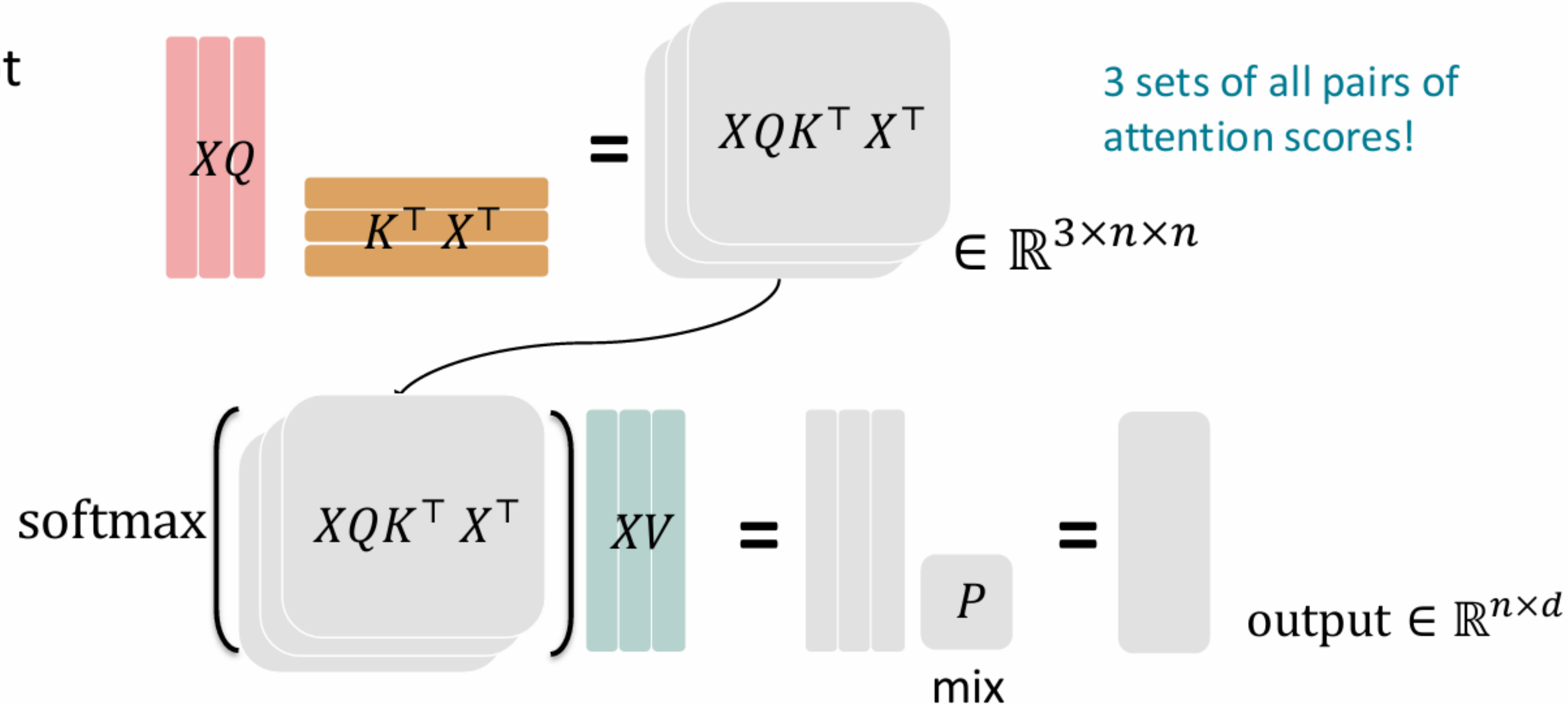
- Each head gets to "look" at different things, and construct value vectors differently.

Multi-head self-attention is computationally efficient

- Even though we compute h many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK, XV.)
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.
  - Almost everything else is identical, and the **matrices are the same sizes.**

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$

Next, softmax, and compute the weighted average with another matrix multiplication.


XQ
$K^\top X^\top$
=
$XQK^\top X^\top$
$\in \mathbb{R}^{3 \times n \times n}$
softmax
$XQK^\top X^\top$
XV
=
P
mix
=
output $\in \mathbb{R}^{n \times d}$

3 sets of all pairs of attention scores!

Scaled Dot Product [Vaswani et al., 2017]

- **"Scaled Dot Product"** attention aids in training.
- When dimensionality d becomes large, dot products between vectors tend to become large.
  - Because of this, inputs to the softmax function can be large, making the gradients small.
- Instead of the self-attention function we've seen:

$$\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$$

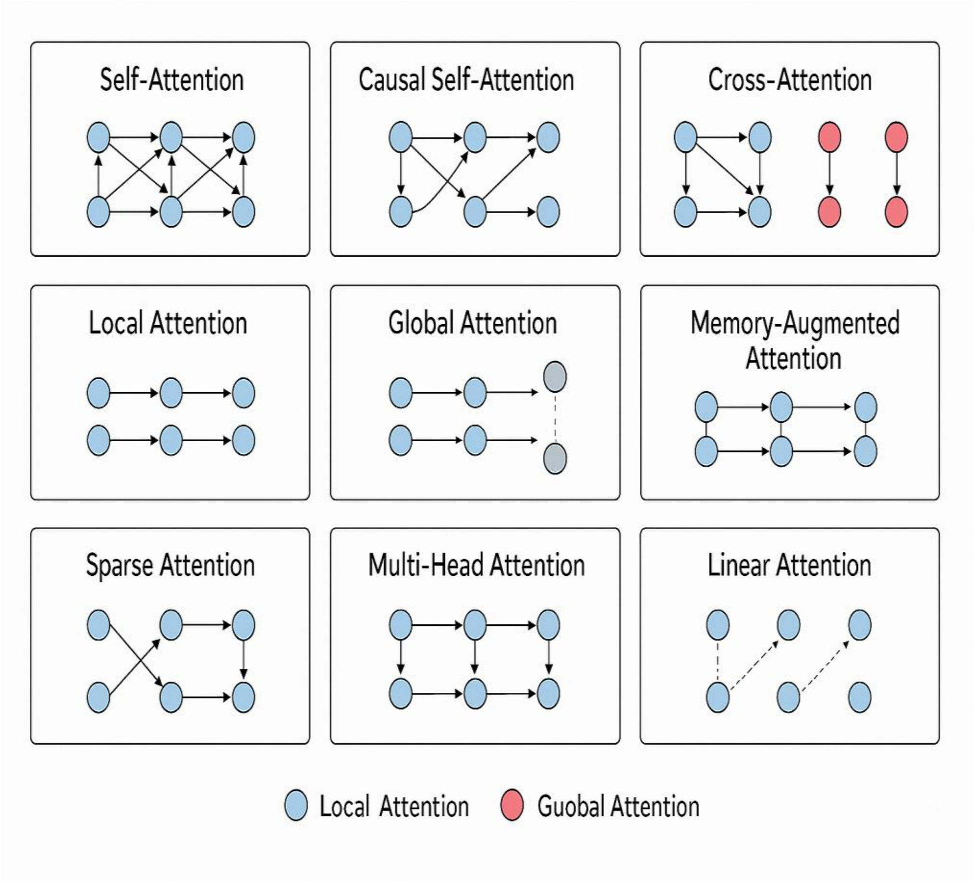
- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

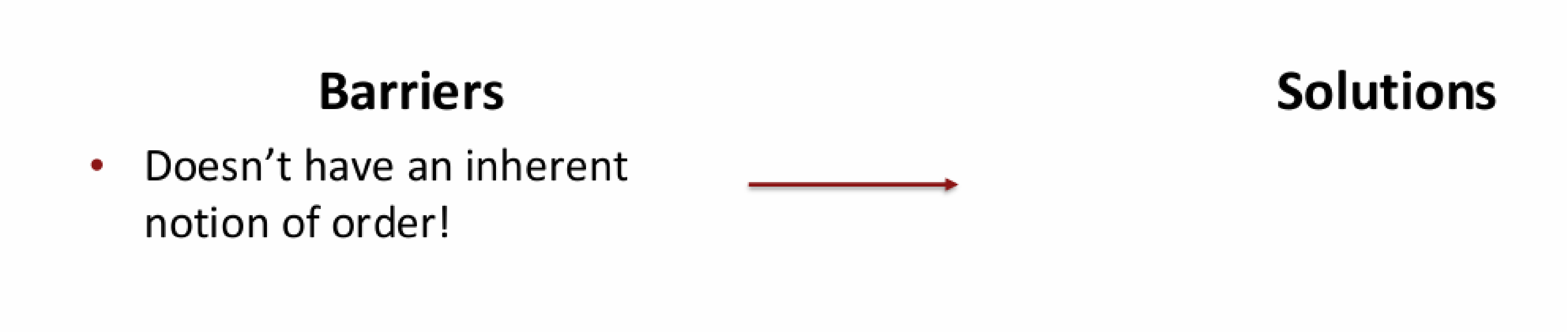
Types of Attention Mechanisms

- Attention mechanisms allow a model to *focus* on different parts of the input sequence when making predictions. Over time, many variants have been developed. Below is some of them.

Category	Examples	Use Case
Self-Attention	GPT, BERT	Learning dependencies in sequence
Cross-Attention	T5, multimodal models	Encoder→decoder, multimodal fusion
Sparse / Local / Global	BigBird, Longformer	Long-context efficiency
Memory-Based	TXL, RETRO, KV Cache	Very long sequences
Multi-Query / Grouped-Query	PaLM, LLaMA	Faster inference
Linear Attention	Performer	Linear time attention
Positional Attention	RoPE	Encoding position inside attention
Flash Attention	FlashAttention	High-speed GPU attention



Barriers and solutions for a transformer building block



Barriers

- Doesn’t have an inherent notion of order!

Solutions

Fixing the first self-attention problem: **sequence order**

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

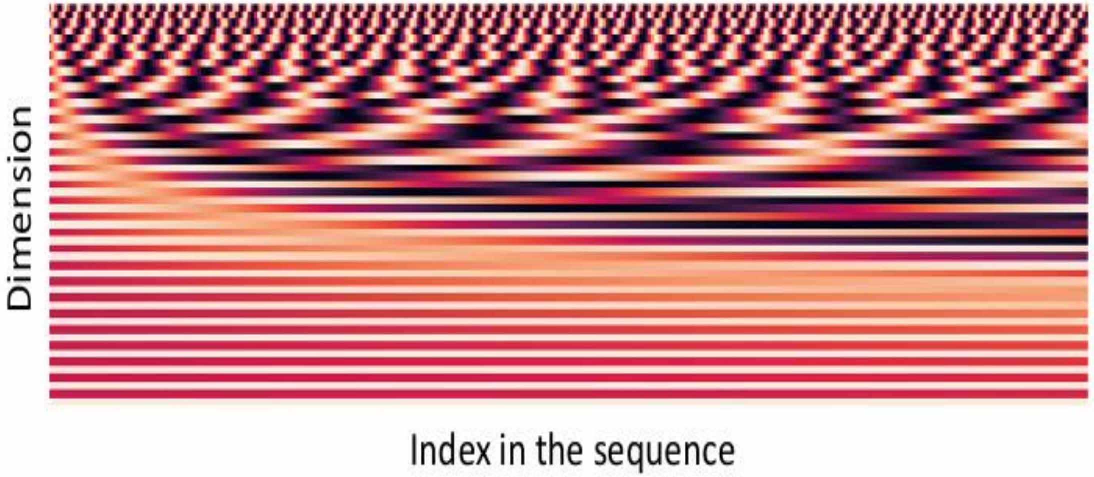
$$\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
  - Periodicity indicates that maybe “absolute position” isn’t as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation doesn’t really work!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Positional encoding = a stack of sin and cos waves with different wavelengths.

What each P_i means:

- For each position iii in the sequence (like token index 0, 1, 2, 3…),
- You generate a d-dimensional vector made of sin and cos waves.
- Each dimension uses a sinusoid with a different frequency.
- Lower dimensions have long-period smooth waves.
- Higher dimensions have short-period rapid oscillations.

This gives each position a unique, structured signature.

The heatmap shows visually:

- The lower rows (bottom of the heatmap) have slowly changing color bands → these are low-frequency sinusoids.
- The upper rows have highly oscillatory patterns → high-frequency sinusoids.

Together these frequencies combine to give the model a way to represent positions uniquely.

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!

  Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$*, and let each* $\boldsymbol{p}_i$ *be a column of that matrix!*

Instead of using a fixed mathematical function (like sinusoids), the model learns a vector for each position during training—similar to how it learns word embeddings.

- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \ldots, n$.
- Most systems use this!

- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Positional Encodings

- Transformers do not have any built-in notion of order because attention is **permutation-invariant**. Positional encodings inject sequence order information into token embeddings.
- There are **six major families** of positional encoding methods:

Method Type	Example Methods	Pros	Cons
Absolute	Sinusoidal, Learned	Simple, stable	Poor extrapolation (except sinusoidal)
Relative	Shaw, T5 bias, ALiBi	Great generalization, long context	More complex
Rotary	RoPE, NTK-scaled RoPE	Best for LLMs, long context	Needs careful scaling
Shift-based	Transformer-XL, XLNet	Handles long-range	Architectural complexity
Continuous/Fourier	FPE, RFF	Smooth, flexible	Harder to tune
Implicit / Architecture-Driven	Mamba, RWKV	No embeddings needed	Model-specific

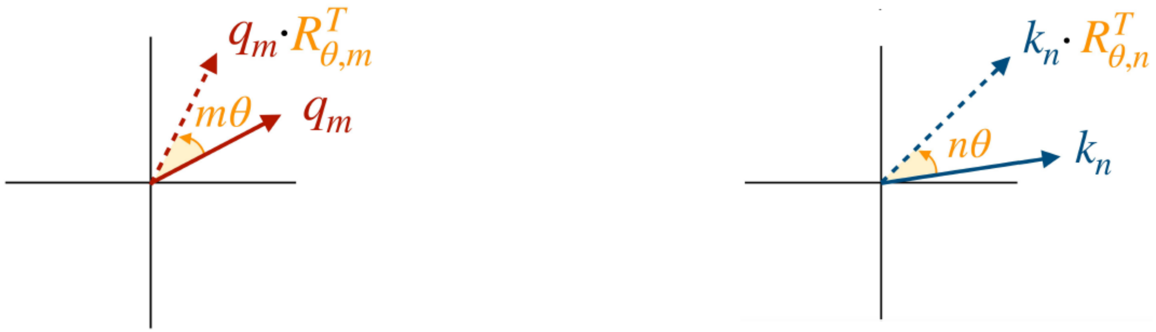
Which methods are used in modern LLMs?

- **Llama 1–3:** RoPE
- **GPT-J / GPT-NeoX:** RoPE
- **GPT-4 family (likely):** Relative + rotary variants
- **Mistral / Mixtral:** RoPE
- **Qwen / Yi / DeepSeek:** RoPE (with NTK scaling)
- **Anthropic Claude:** Relative position bias + proprietary adaptations

RoPE = Rotary Position Embeddings

- RoPE is a positional encoding method that injects information about token positions directly into the attention mechanism by rotating the query and key vectors in a way that depends on the position index.
- Default choice nowadays: allows rotations in attention layer

Idea. Rotate query and key vectors with rotation matrix



Formula. Use rotation matrix: $R_{\theta,m} = \begin{pmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{pmatrix}$

RoPE (Rotary Position Embedding) encodes token position by **rotating** each pair of query/key dimensions in a position-dependent way.

The rotation is controlled by:

- token position i
- frequency index m
- rotation angle $\theta_{i,m} = i \cdot \omega_m$

"RoFormer: Enhanced Transformer with Rotary Position Embeddings", Su et al., 2021.
Figure from "Super Study Guide: Transformers & Large Language Models", Amidi et al., 2024.

RoPE = Rotary Position Embeddings

- **How RoPE Is Applied Inside Attention**
- Let the model have embedding dimension d.
  - the **raw query** Q_i
  - the **raw key** K_i
  - the **rotation matrix** $R_{\theta,m}$
- RoPE works by *splitting* the vectors into **2-dimensional pairs**:
  $\left(Q_{i,0}, Q_{i,1}\right),\ \left(Q_{i,2}, Q_{i,3}\right),\ \ldots,\ \left(Q_{i,d-2}, Q_{i,d-1}\right)$

- **The Frequency $\boldsymbol{\omega_m}$** : Classically:
  - $\omega_m = 10000^{-\frac{2m}{d}}$
  - This gives higher frequencies to higher dimensions.

- **Use the Rotated Q′ and K′ in Attention**
- Standard attention score:

$$score(i,j) = Q_i' \cdot K_j'$$

- Because:

$$Q_i' = R_{\theta_i} Q_i, K_j' = R_{\theta_j} K_j$$

- We get:

$$Q_i' \cdot K_j' = Q_i^{\top} R_{\theta_i}^{\top} R_{\theta_j} K_j$$

Barriers and solutions for Self-Attention as a building block

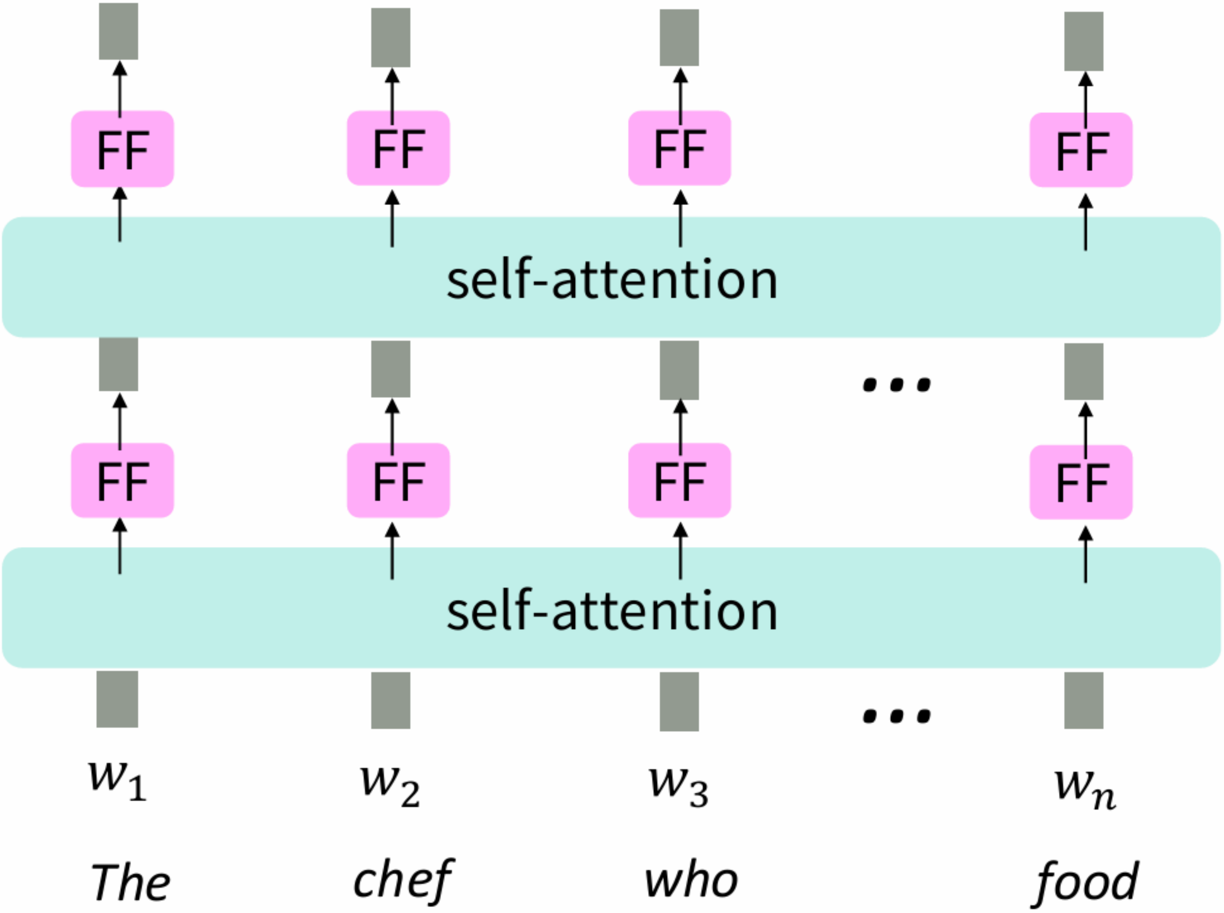
Barriers		Solutions
• Doesn’t have an inherent notion of order!	→	• Add position representations to the inputs
• No nonlinearities for deep learning! It’s all just weighted averages	→	

Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)

- Easy fix: add a **feed-forward network** to post-process each output vector.

$$
\begin{aligned}
m_i &= MLP(\text{output}_i) \\
&= W_2 * \text{ReLU}(W_1 \,\text{output}_i + b_1) + b_2
\end{aligned}
$$



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

Barriers	Solutions
• Doesn’t have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning magic! It’s all just weighted averages	• Easy fix: apply the same feedforward network to each self-attention output.
• Need to ensure we don’t “look at the future” when predicting a sequence • Like in machine translation • Or language modeling	

Masking the future in self-attention

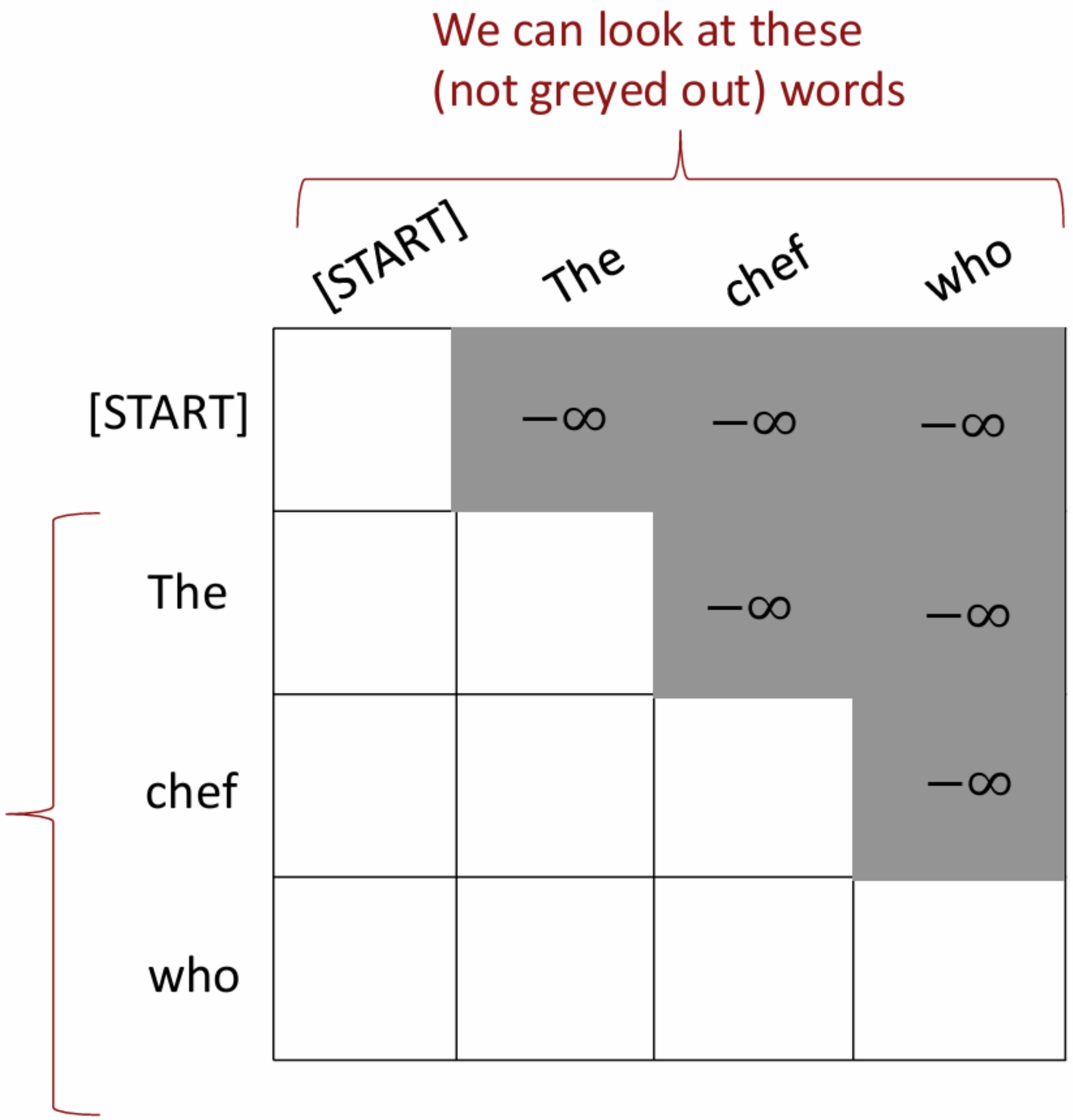
- To use self-attention in **decoders**, we need to ensure we can't peek at the future.

- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)

- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^{\top} k_j, j \le i \\ -\infty, j > i \end{cases}$$

For encoding these words



We can look at these (not greyed out) words

	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

Masking the future in self-attention

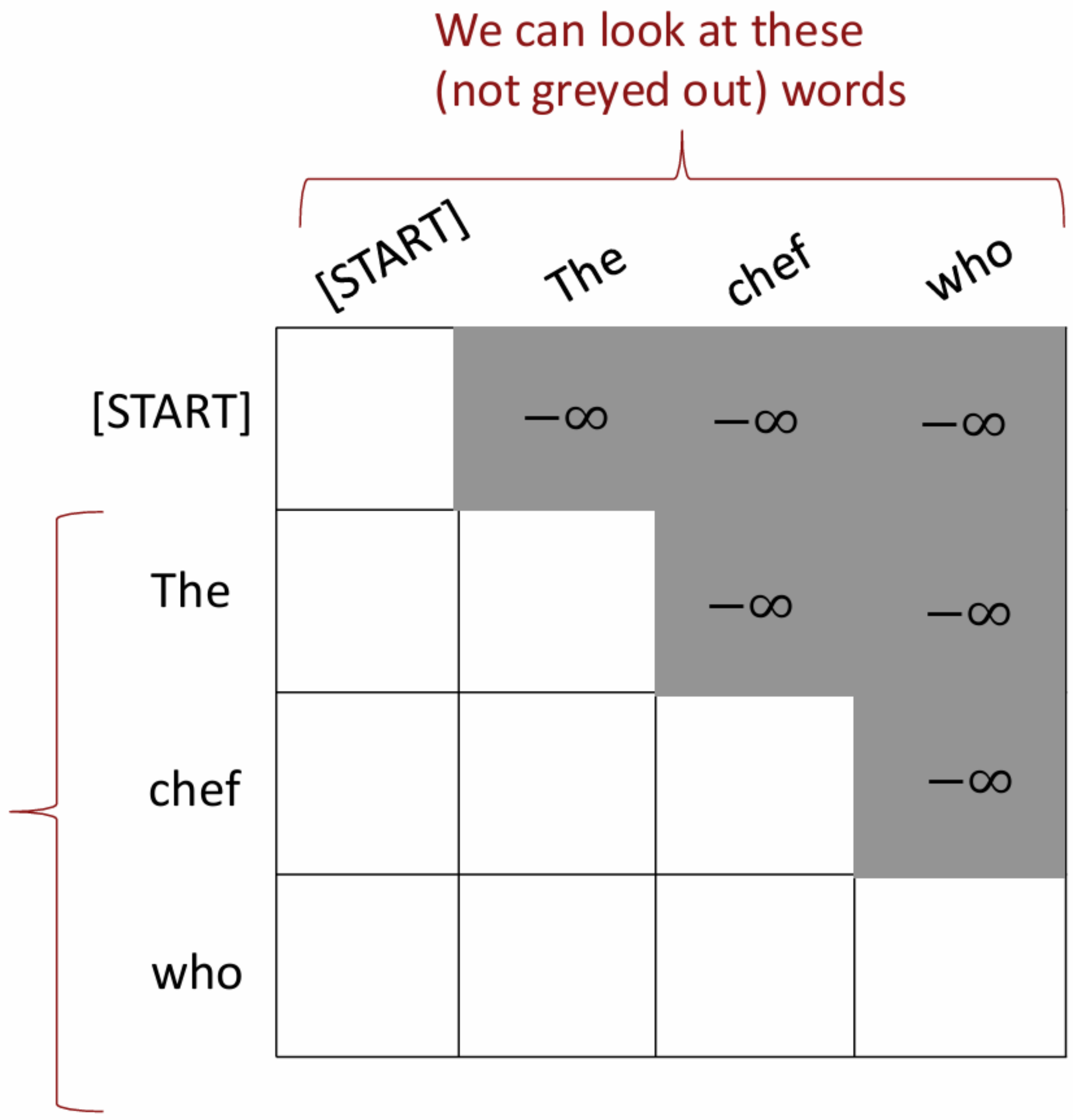
- To use self-attention in **decoders**, we need to ensure we can't peek at the future.

- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)

- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \le i \\ -\infty, j > i \end{cases}$$

For encoding these words



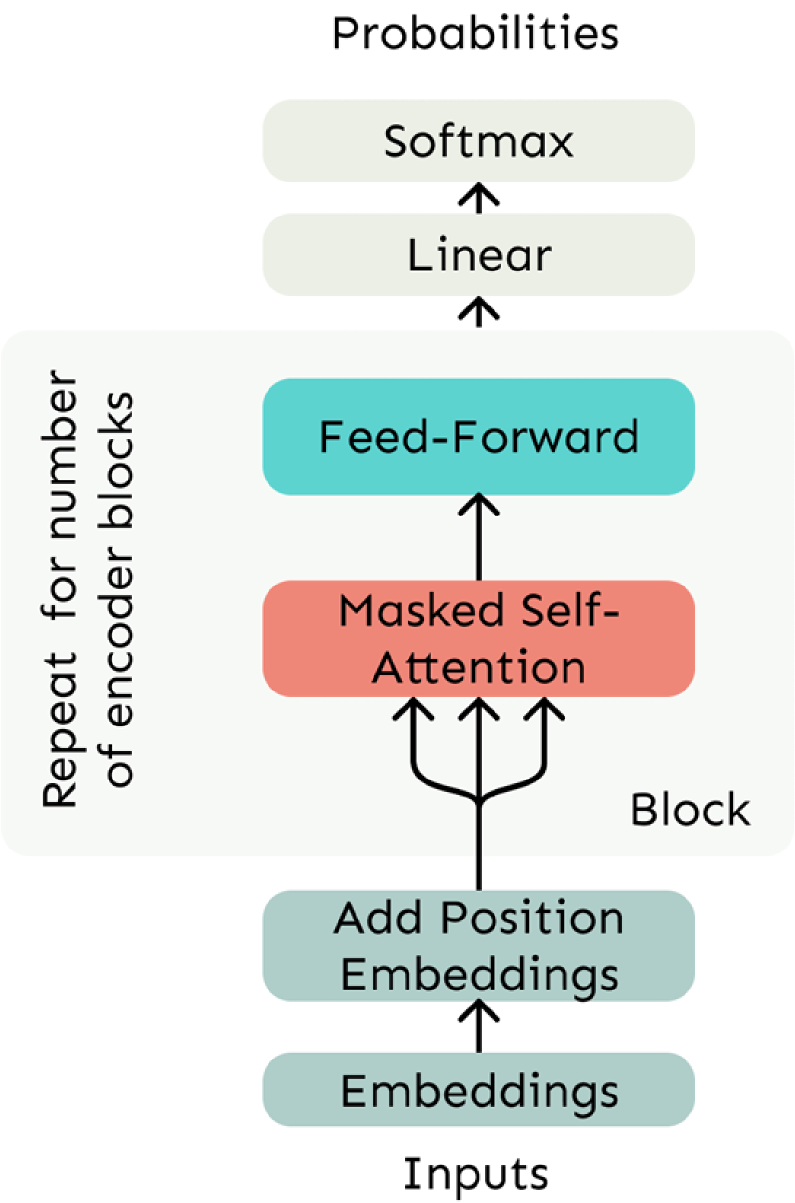
We can look at these (not greyed out) words

	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

Necessities for a self-attention building block:

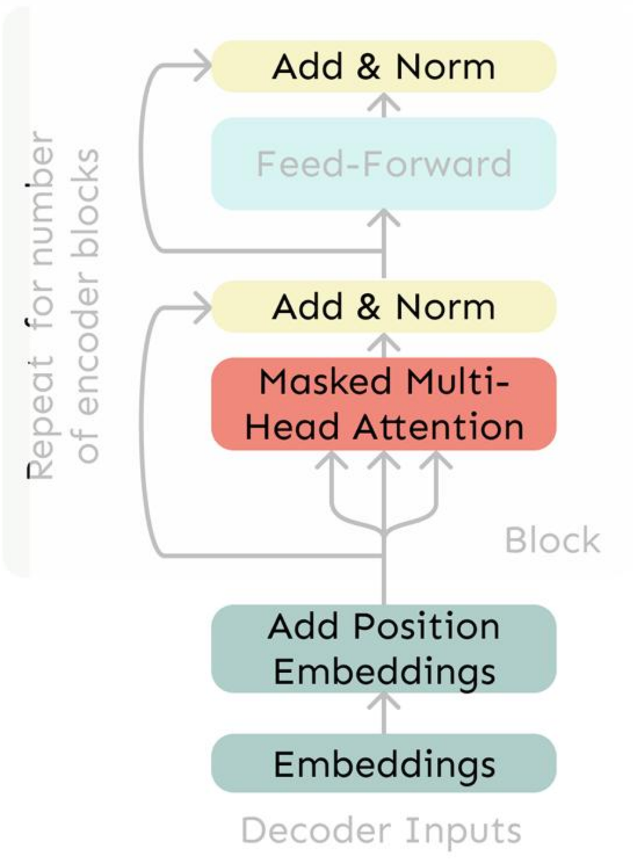
- **Self-attention**:
  - the basis of the method.
- **Position representations**:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities**:
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- **Masking**:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from “leaking” to the past.

Transformer **Decoder**



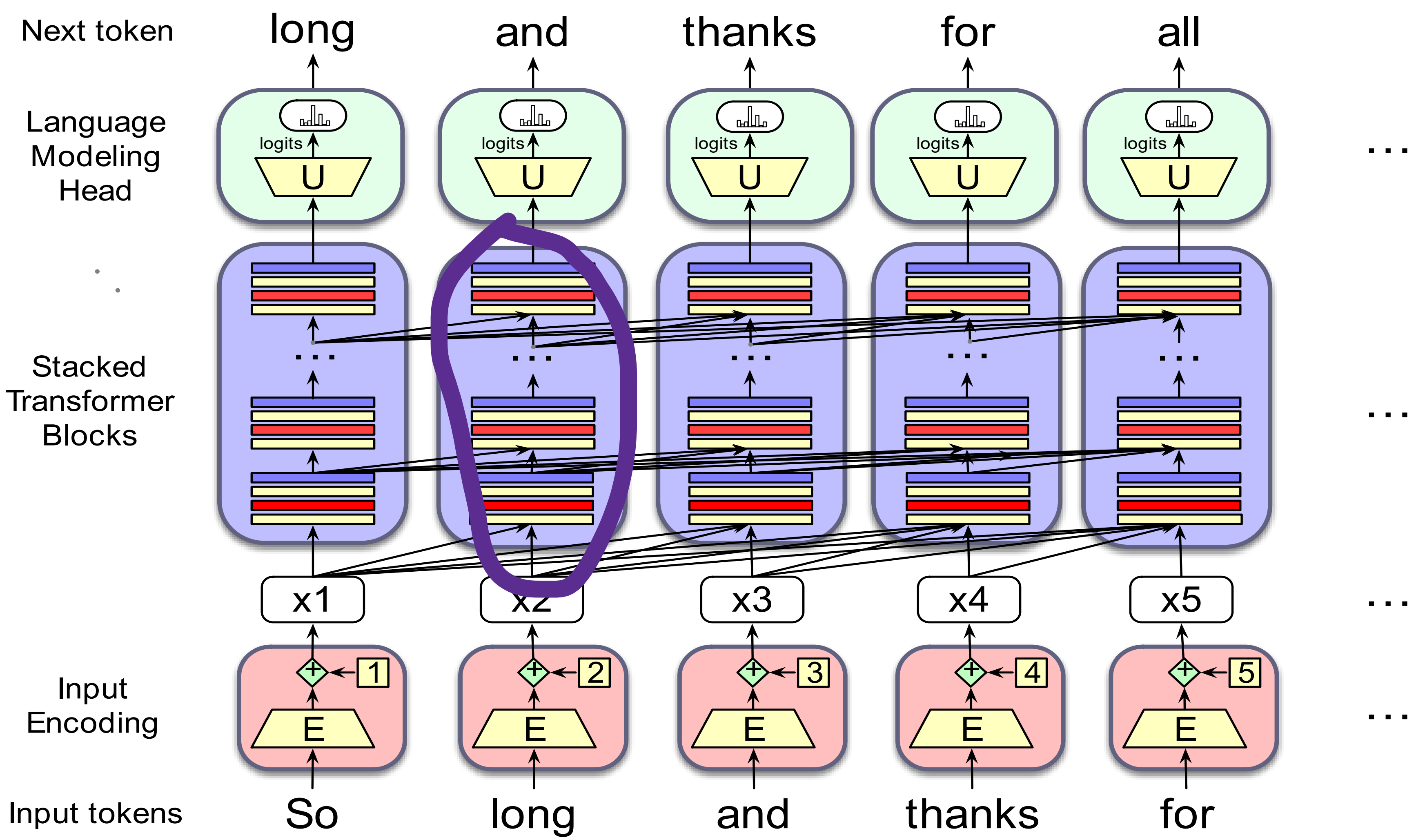
The Transformer Decoder

- Two optimization tricks that end up being :
  - Residual Connections
  - Layer Normalization
- In most Transformer diagrams, these are often written together as “Add & Norm”



Transformer Decoder

Reminder: transformer language model



The residual stream: each token gets passed up and modified

- Each token has a hidden state: h_i

For every token in the input sentence (e.g., words or subwords):

- The model keeps a vector representing that token at each layer.
- These vectors are shown as vertical bars labeled h_{i-2}, h_{i-1}, and h_i.

These vectors get passed upward from layer to layer.

- The "residual stream" is the *running memory* of the token

Each vertical bar is the **residual stream** — a continuous vector that:

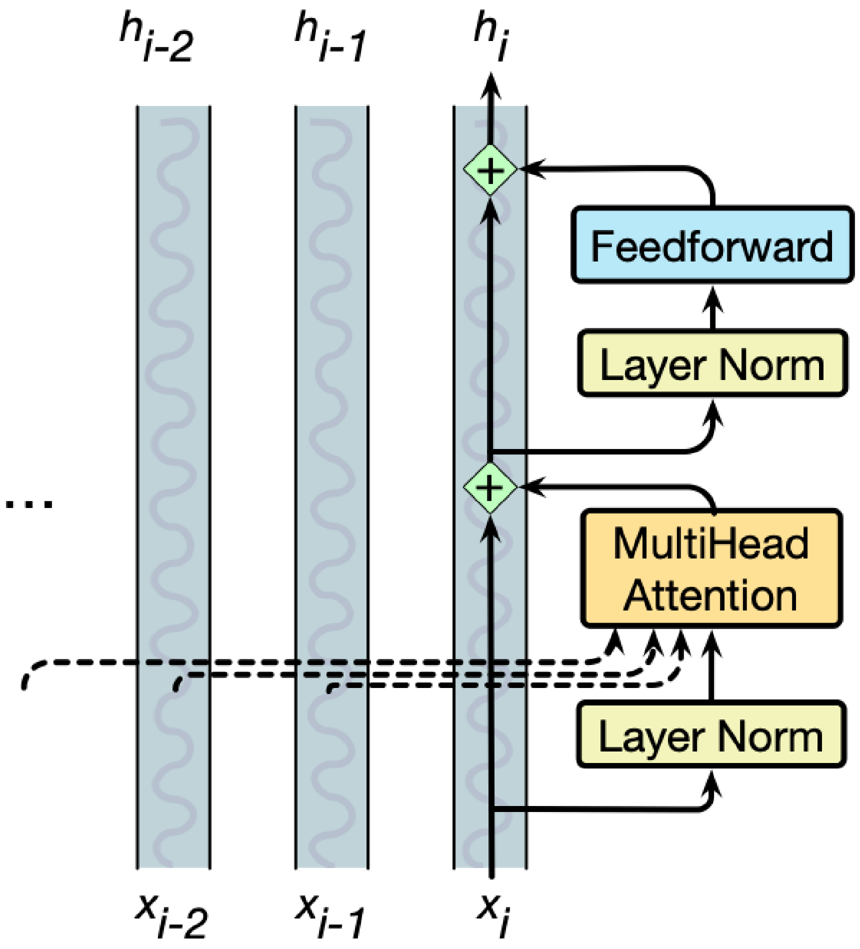
- **Carries the token's representation up through the layers**
- **Gets modified** by attention and feedforward blocks
- **Never gets overwritten**, only updated by adding new information

This is why it's called *residual*: Each module adds a "residual" (a change) to the existing state.

- Why is the residual stream important?

Residual connections were originally invented to:

- Help deep networks train more easily
- Preserve information across many layers
- Allow each layer to make *incremental improvements* rather than rewrite everything
- Holds the entire evolving representation of each token
- Accumulates all information from all layers
- Carries both attention outputs and feedforward transformations



We'll need nonlinearities, so a feedforward layer

$$\mathrm{FFN}(\mathbf{x}_i) = \mathrm{ReLU}(\mathbf{x}_i \mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$

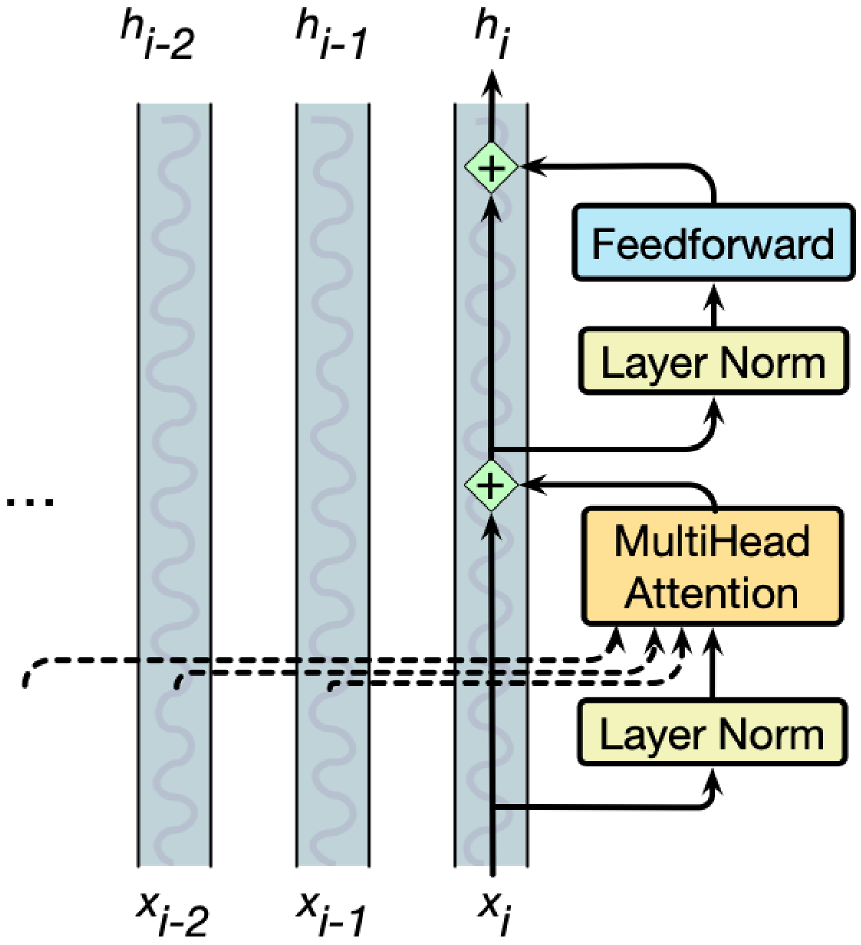
Why Transformers Need Feed-Forward Network (FFN)

Self-attention alone is mostly *linear*. Linear models have limited expressive power.

The FFN gives the model:

- **nonlinear transformations**
- **extra learning capacity**
- **richer, more flexible representations**

It's similar to adding a small neural network after attention.



Layer norm: the vector x_i is normalized twice

Each Transformer block has two major sublayers:

1. **Multi-Head Attention (MHA) sublayer**
2. **Feed-Forward Network (FFN) sublayer**

Each sublayer is wrapped with:

- a **residual (skip) connection**
- a **LayerNorm operation**

Why Is Layer Normalization Needed?

LayerNorm helps in three major ways:

✓ **1. Stabilizes training**

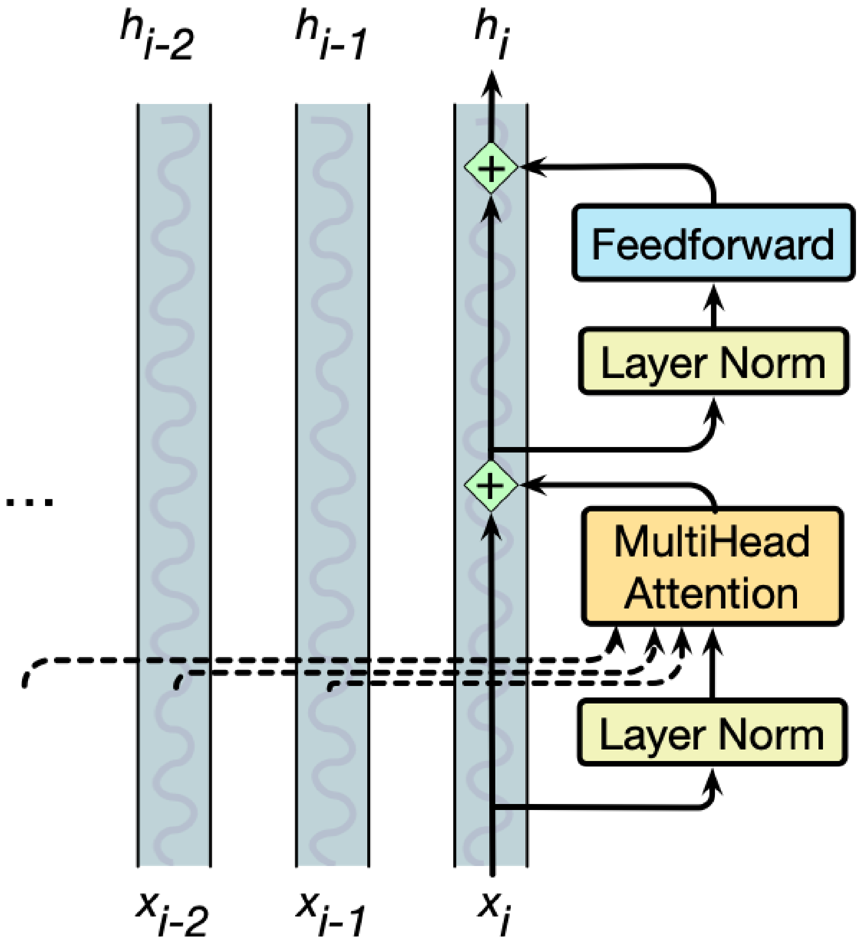
During training, activations can explode or vanish.
LayerNorm keeps values in a stable range.

✓ **2. Balances the residual connections**

Residual connections add the input directly to the output.
The scale of these two vectors must be compatible.
LayerNorm makes sure they "play nicely together."

✓ **3. Prevents attention from dominating**

Without normalization, attention scores or FFN activations could overwhelm early layers.



Layer Norm

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

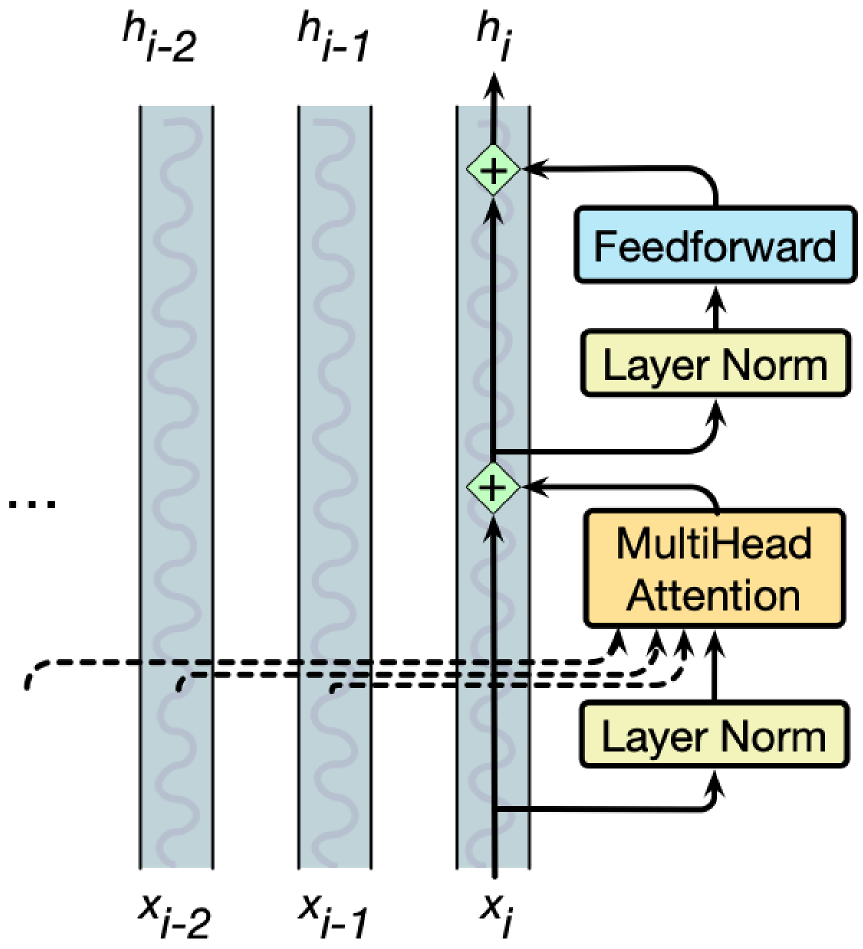
$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

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

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

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

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

Every Block Keeps the Same Dimensionality

A Transformer consists of many identical blocks stacked on top of each other.

Every block takes as input a sequence of vectors:

$$x_1, x_2, \dots, x_n \in \mathbb{R}^d$$

and produces output vectors of the **same dimension**:

$$h_1, h_2, \dots, h_n \in \mathbb{R}^d$$

Why stack multiple blocks?

Each block:

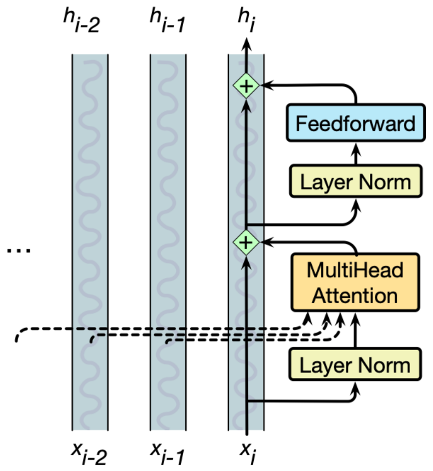
- attends to information
- normalizes it
- processes it nonlinearly
- passes it forward

- Stacking them allows the model to build **richer and more abstract representations**.

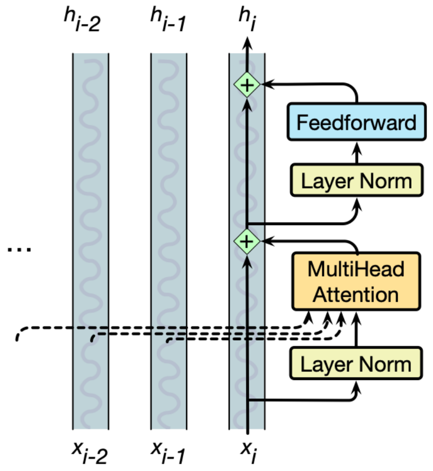
 - Lower blocks → focus on **local patterns**

 - Higher blocks → understand **deeper meaning**, long-range relationships, global structure

Block 2

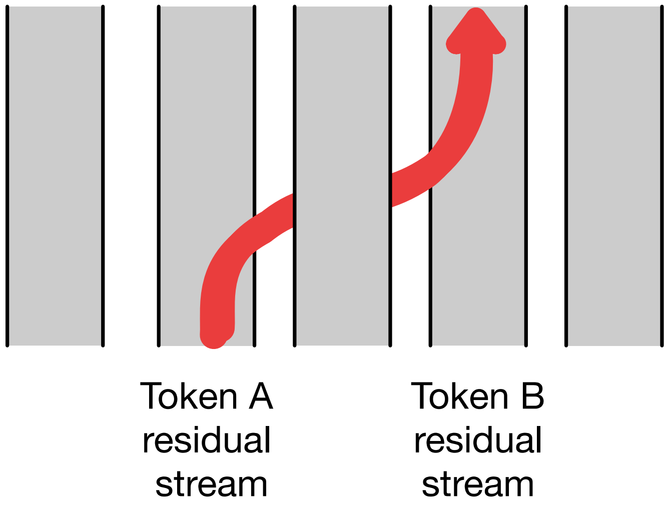


Block 1



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 .



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* × *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

N

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

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

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

Masking out the future

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

- 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

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

- Mask out attention to future words by setting attention scores to –∞ in cells in upper triangle
- The softmax will turn it to 0

N

q1•k1	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞
q4•k1	q4•k2	q4•k3	q4•k4

N

Another point: Attention is quadratic in length

Matrix $QK^\top$

- Q= matrix of *queries*
- K= matrix of *keys*

If the sequence length is **N**, then:

- $Q$is $N \times d_k$
- $K$is $N \times d_k$

So:

$$QK^\top \text{is an } N \times N \text{ matrix}$$

Each cell is:

$$q_i \cdot k_j \text{(a dot-product)}$$

Attention is quadratic

Because the matrix $QK^\top$requires N × N operations.

So the computational cost grows like:

$O(N^2)$(quadratic in sequence length)

Quadratic attention becomes very expensive when:

- N = 4 → OK
- N = 512 → already heavy
- N = 8192 → extremely expensive
- N = 100k → impossible with standard attention

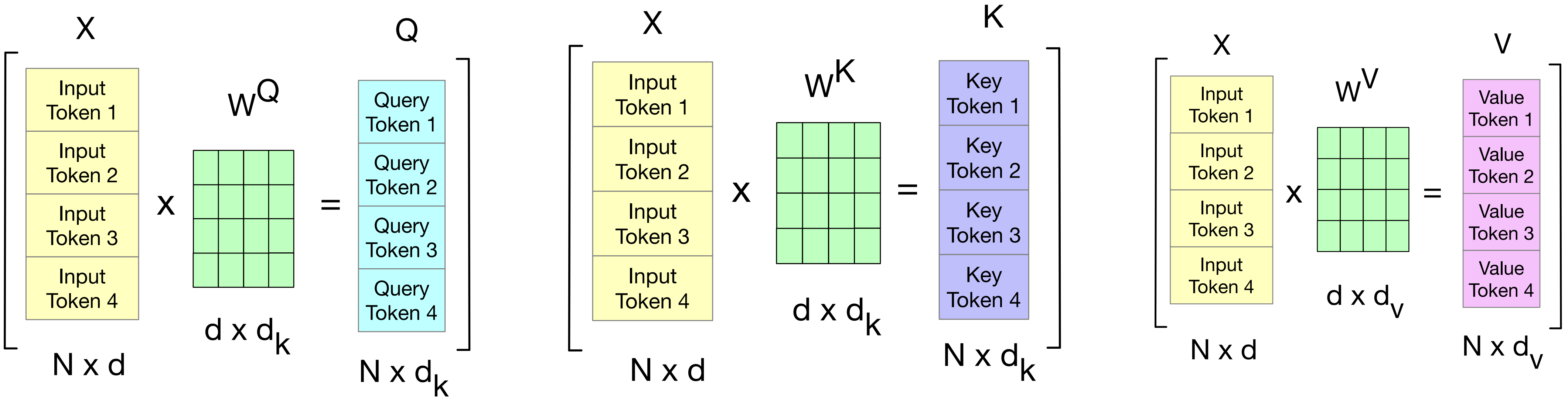
$$\mathbf{A} = \mathrm{softmax}\left(\mathrm{mask}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\right)\mathbf{V}$$

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

Attention again



X × W^Q = Q

Input Token 1
Input Token 2
Input Token 3
Input Token 4 — $N \times d$

W^Q — $d \times d_k$

Query Token 1
Query Token 2
Query Token 3
Query Token 4 — $N \times d_k$

X × W^K = K

Input Token 1
Input Token 2
Input Token 3
Input Token 4 — $N \times d$

W^K — $d \times d_k$

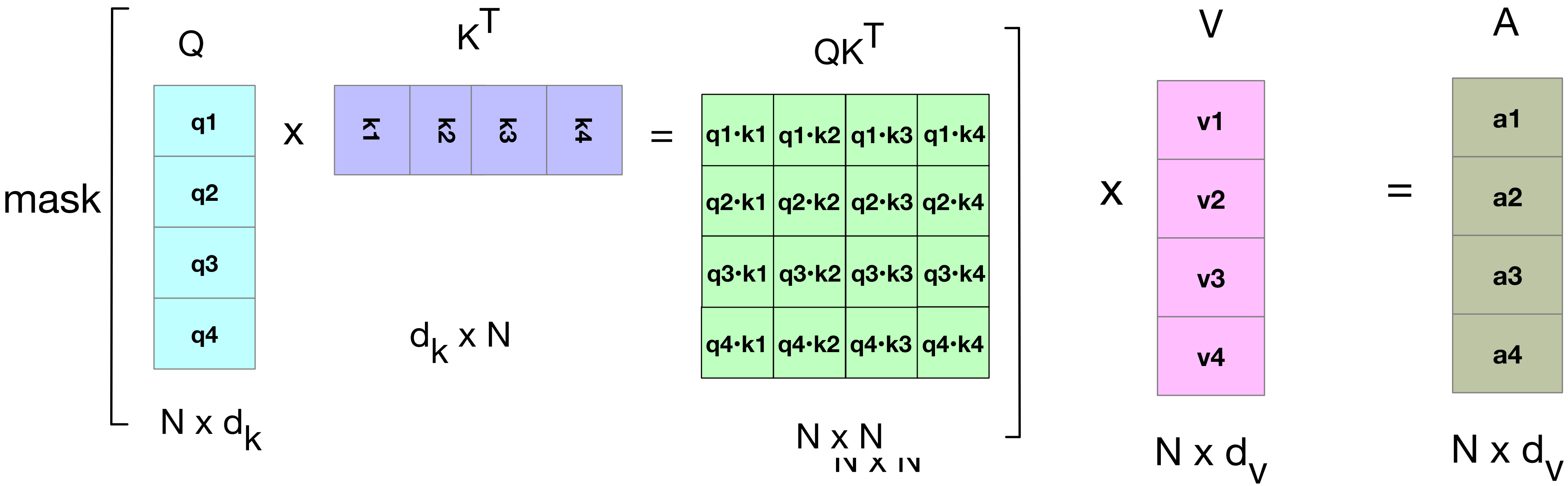
Key Token 1
Key Token 2
Key Token 3
Key Token 4 — $N \times d_k$

X × W^V = V

Input Token 1
Input Token 2
Input Token 3
Input Token 4 — $N \times d$

W^V — $d \times d_v$

Value Token 1
Value Token 2
Value Token 3
Value Token 4 — $N \times d_v$



mask [Q × K^T = QK^T] × V = A

Q: q1, q2, q3, q4 — $N \times d_k$

K^T: k1, k2, k3, k4 — $d_k \times N$

QK^T — $N \times N$

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

V: v1, v2, v3, v4 — $N \times d_v$

A: a1, a2, a3, a4 — $N \times d_v$

Parallelizing Multi-head Attention

$$\mathbf{Q}^{\mathrm{i}} = \mathbf{X}\mathbf{W}^{\mathrm{Qi}};\;\; \mathbf{K}^{\mathrm{i}} = \mathbf{X}\mathbf{W}^{\mathrm{Ki}};\;\; \mathbf{V}^{\mathrm{i}} = \mathbf{X}\mathbf{W}^{\mathrm{Vi}}$$

Because all heads use the same X, these matrix multiplications are **parallelizable**.

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

This is standard scaled dot-product attention, applied independently in each head.

Every head can compute this operation **simultaneously**.

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

Concatenate the heads and multiply with W$^{\mathrm{o}}$

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

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

Multi-head attention is parallelizable because each head operates independently.

- All heads compute Q, K, V in parallel
- All heads compute attention in parallel
- Finally they are concatenated

This is a major reason Transformers run efficiently on GPUs/TPUs.

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

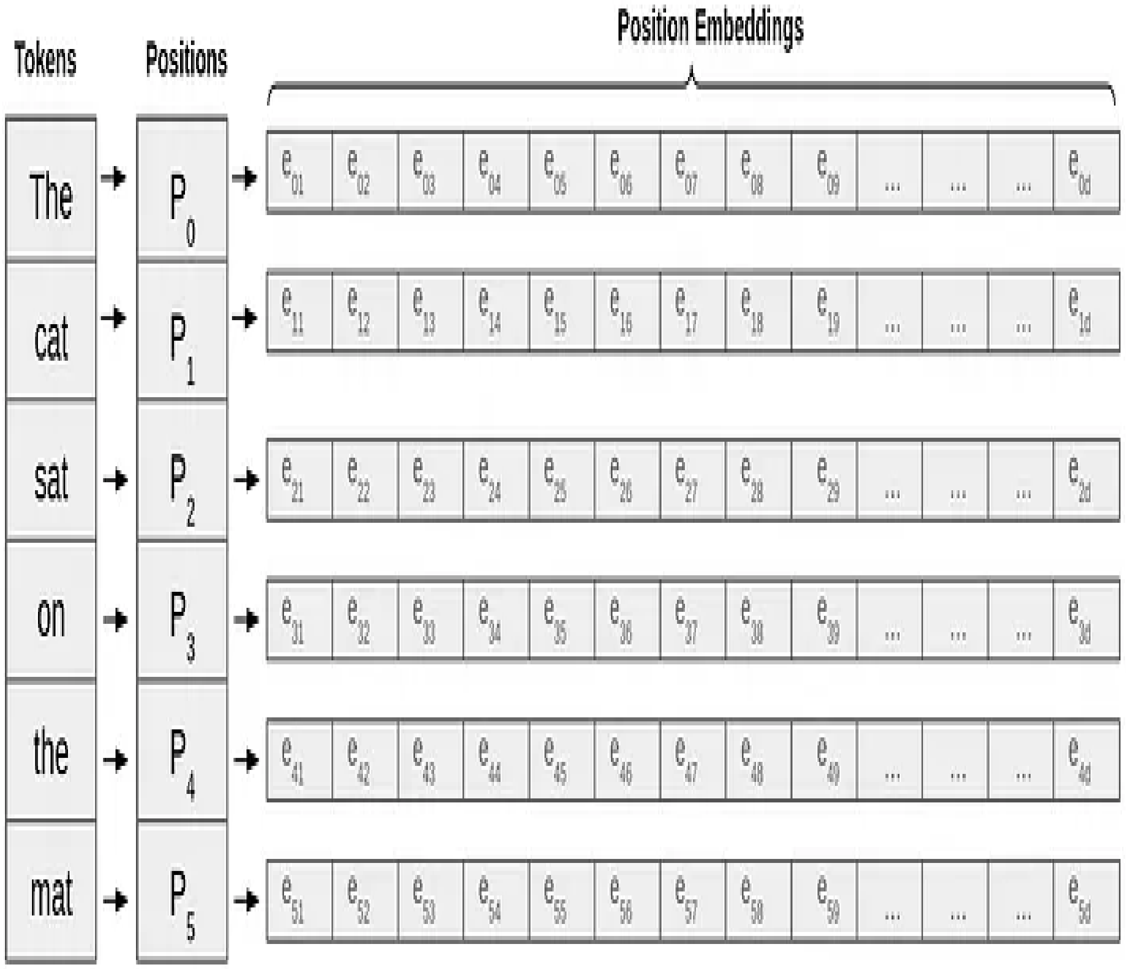
- Given: string "*Thanks for all the*"
  - *1.* Tokenize with BPE and convert into vocab indices

    w = [5,4000,10532,2224]
  - 2. Select the corresponding rows from E, each row an embedding

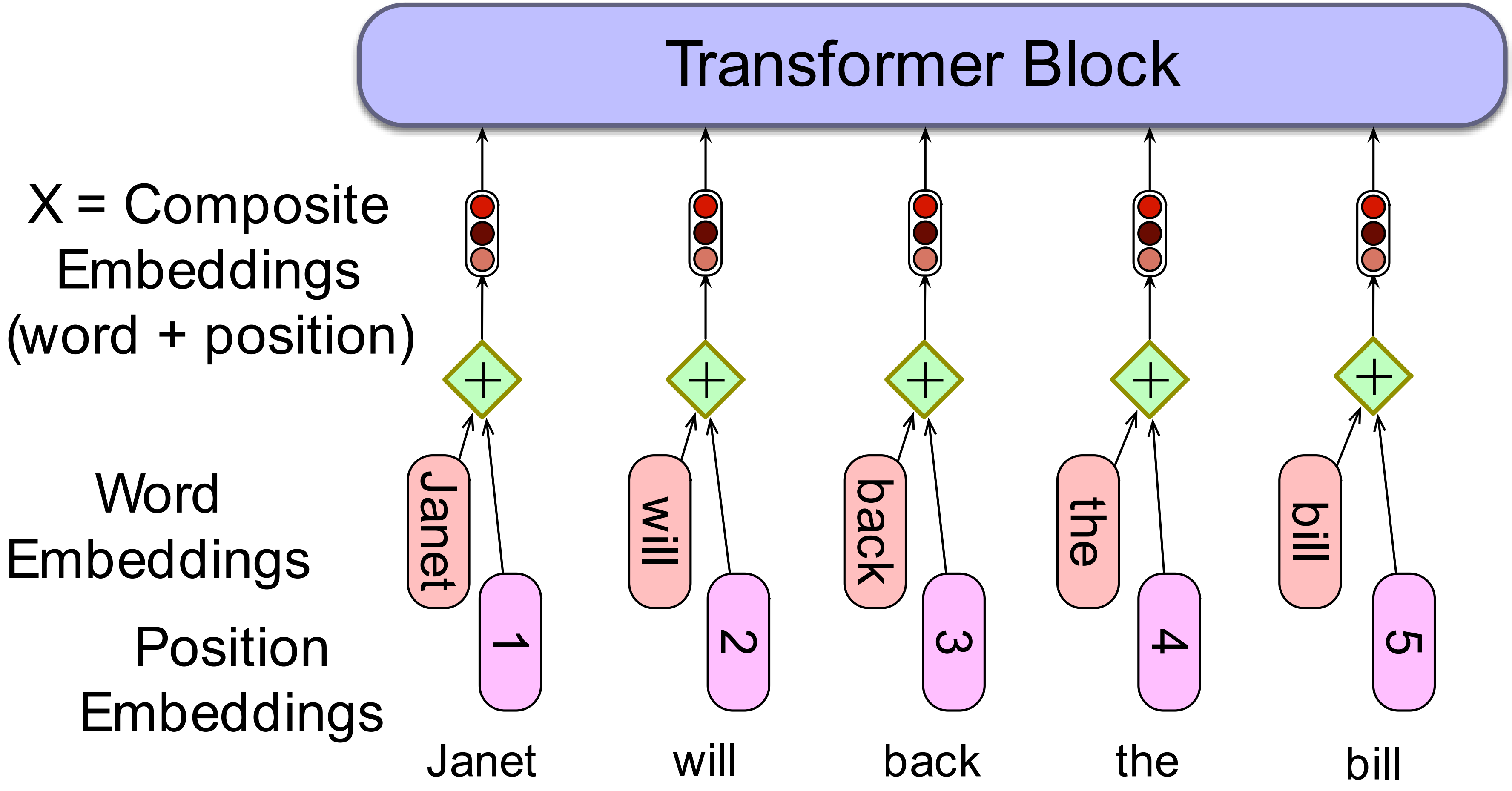
    (row 5, row 4000, row 10532, row 2224).

Position Embeddings

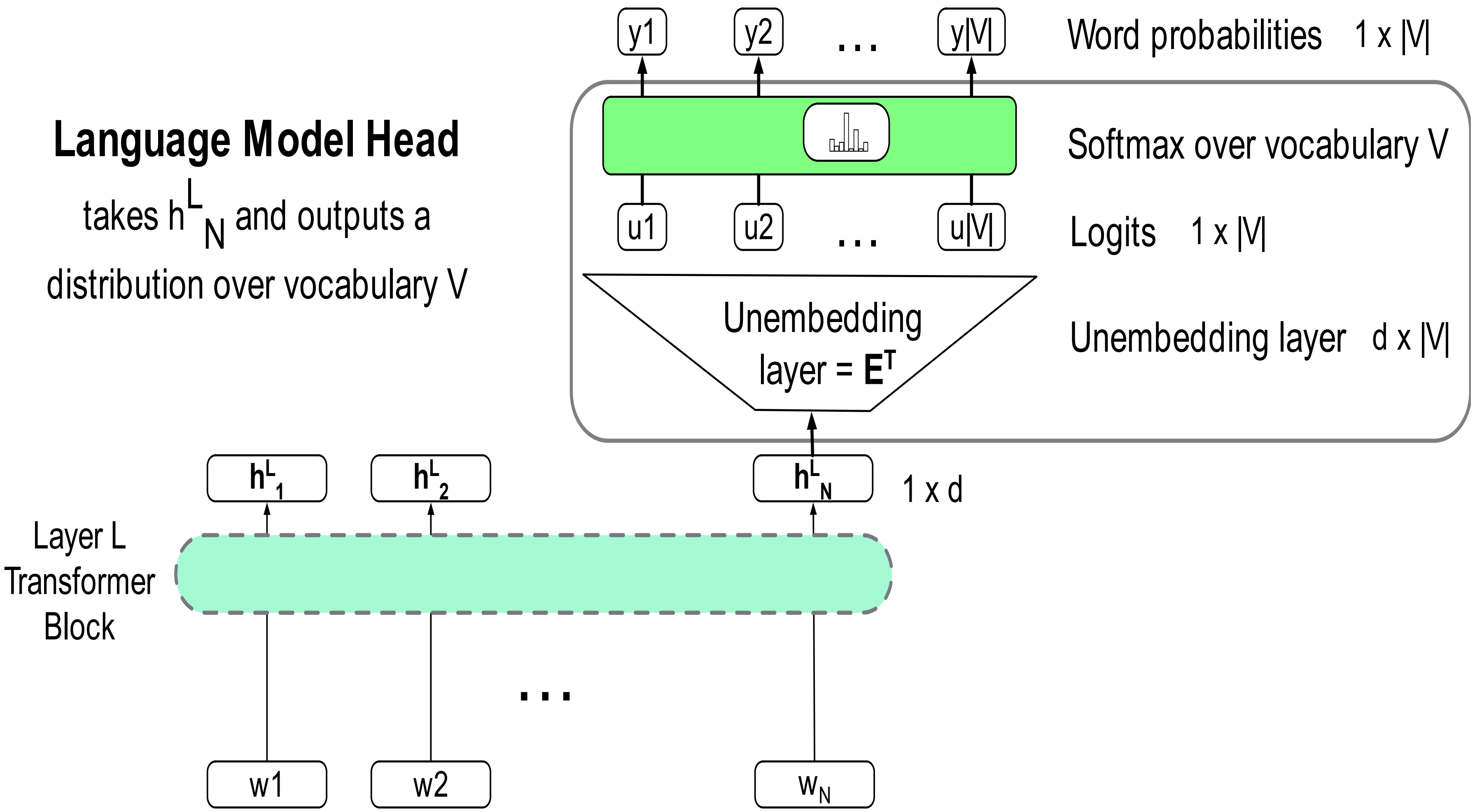
- There are many methods as stated earlier, but we'll just describe the simplest: absolute position.
  - Goal: learn a position embedding matrix E_{pos} of shape [1 × *N*].
  - Start with randomly initialized embeddings
    - one for each integer up to some maximum length.
    - i.e., just as we have 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.


Tokens
Positions
Position Embeddings
The
cat
sat
on
the
mat
P0
P1
P2
P3
P4
P5
e01 e02 e03 e04 e05 e06 e07 e08 e09 ... ... e0d
e11 e12 e13 e14 e15 e16 e17 e18 e19 ... ... e1d
e21 e22 e23 e24 e25 e26 e27 e28 e29 ... ... e2d
e31 e32 e33 e34 e35 e36 e37 e38 e39 ... ... e3d
e41 e42 e43 e44 e45 e46 e47 e48 e49 ... ... e4d
e51 e52 e53 e54 e55 e56 e57 e58 e59 ... ... e5d

Each x is just the sum of word and position embeddings



Language modeling head



y1
y2
…
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
…
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
…
w1
w2
w_N

Language Model Head

takes h^L_N and outputs a distribution over vocabulary V

- For autoregressive LM (GPT-style), we usually focus on **the last position** h_N, because that state encodes the meaning of the **entire prefix**.

- The final step of a Transformer language model:

→ The model takes the hidden state of the last layer

$$h_N \in \mathbb{R}^d$$

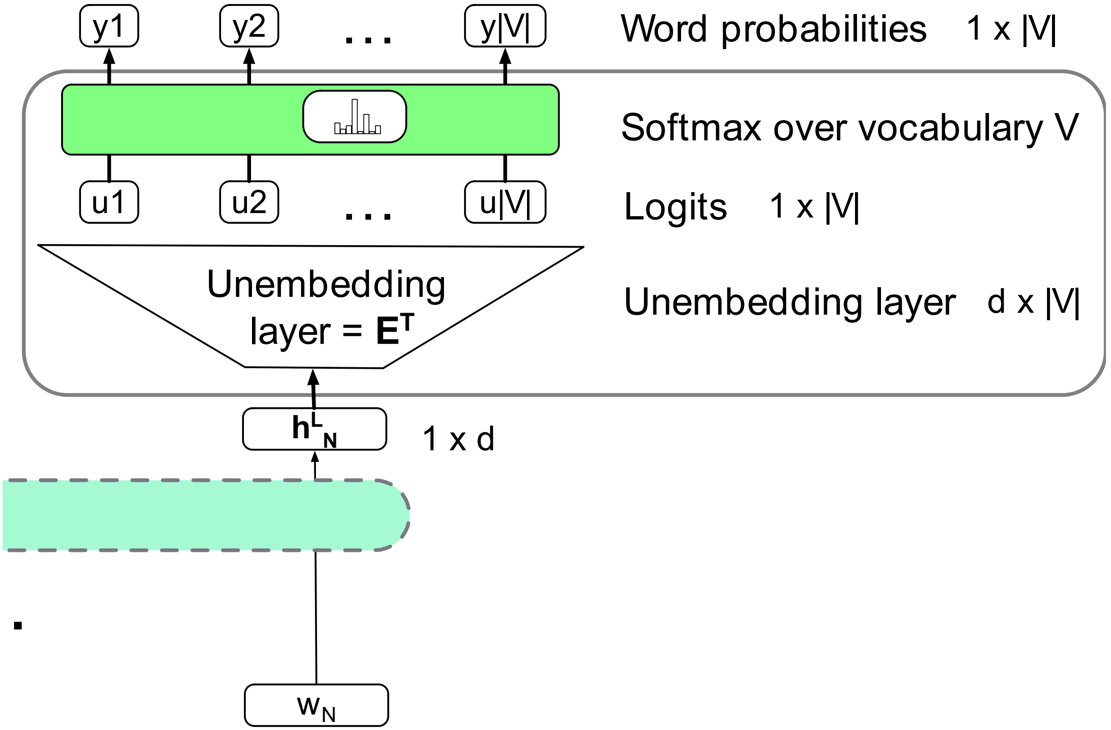
→ and converts it into a probability distribution over the vocabulary

$$P(|)$$

This "conversion" is done by a module called the: **Language Modeling Head**

Language modeling head

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


y1
y2
. . .
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
. . .
u|V|
Logits 1 x |V|
Unembedding layer = $\mathbf{E}^T$
Unembedding layer d x |V|
$\mathbf{h}^L{}_N$
1 x d
w_N

Unembedding layer: multiply by E^T

- E= embedding matrix (shape $V \times d$)
- V= vocabulary size

logits are computed by:

$$u = h_N E^T$$

- $h_N$is 1 × d
- $E^T$is d × V
- output u is 1 × V

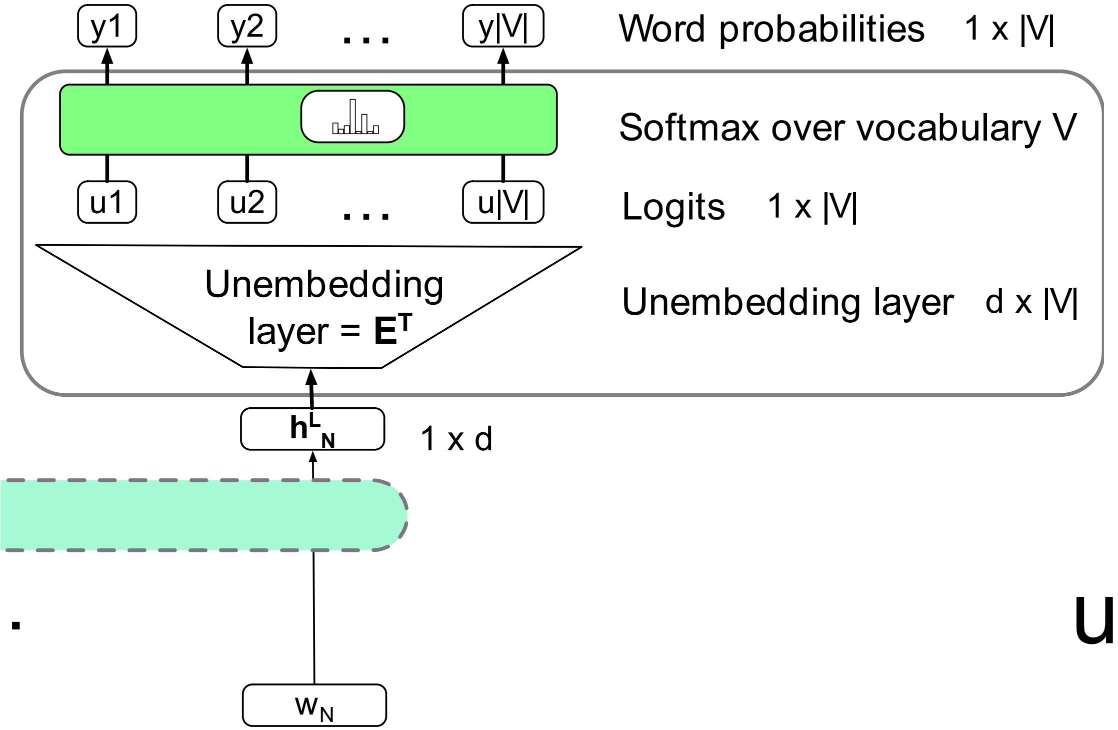
This vector u contains **logits** for all vocabulary tokens.

The diagram labels them:

- $u_1, u_2, \dots, u_{|V|}$

These are **unnormalized scores**.

Language modeling head


y1
y2
...
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_N
1 x d
.
w_N

Logits, the score vector u

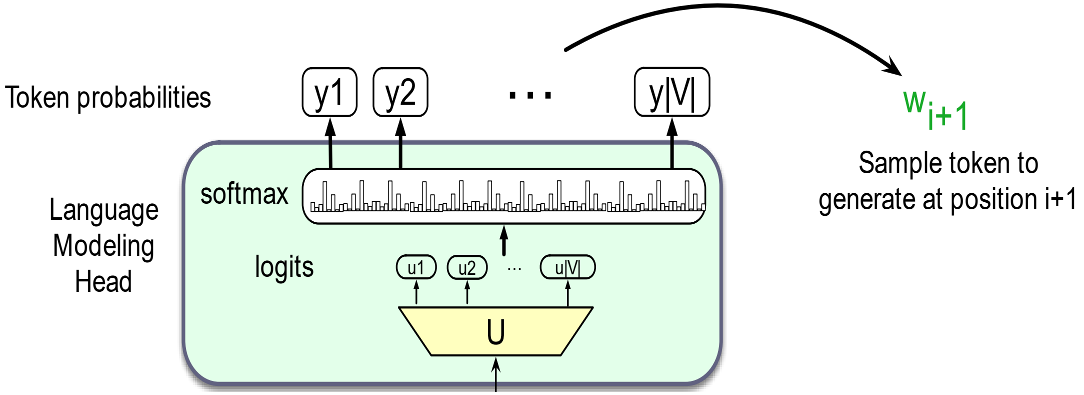
One score for each of the $|V|$ possible words in the vocabulary V. Shape $1 \times |V|$.

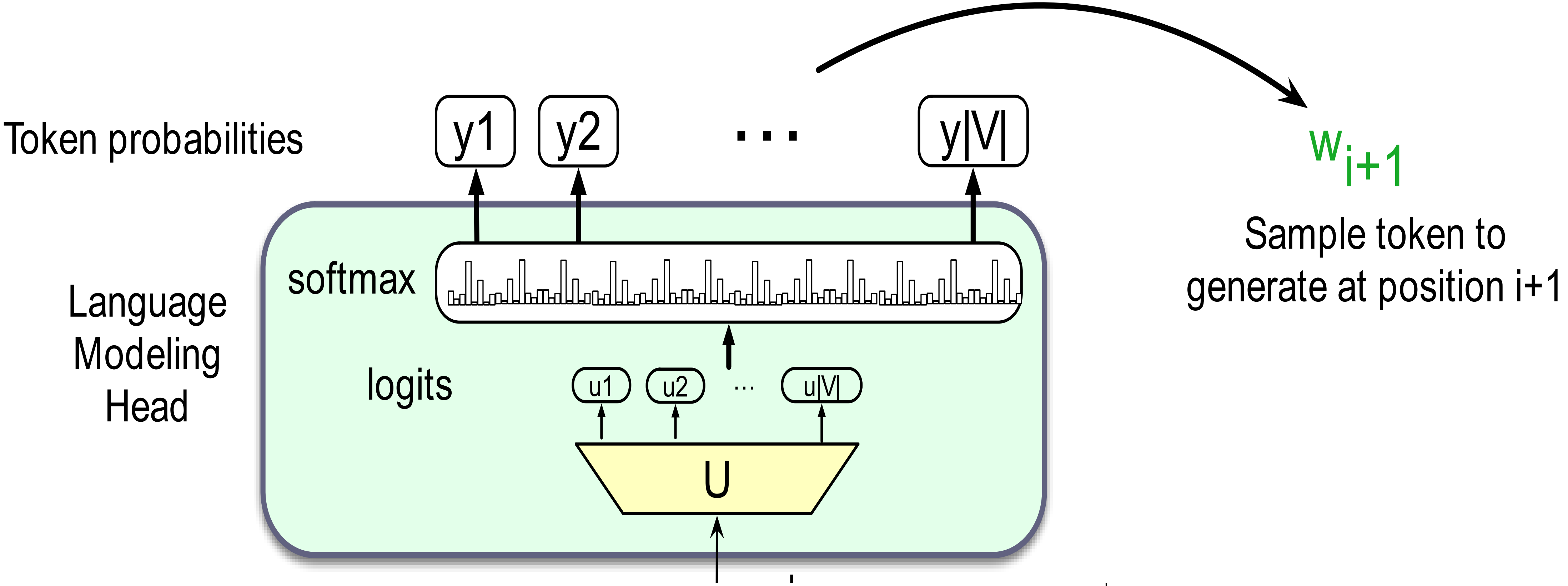
Softmax turns the logits into probabilities over vocabulary. Shape $1 \times |V|$.

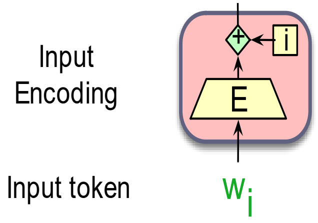
$$u = h^L_N \; E^T$$

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

The final n


Token probabilities
y1
y2
...
y|V|
w_{i+1}
Sample token to generate at position i+1
Language Modeling Head
softmax
logits
u1
u2
...
u|V|
U


Token probabilities
y1
y2
...
y|V|
w_{i+1}
Sample token to generate at position i+1
Language Modeling Head
softmax
logits
u1
u2
...
u|V|
U


Input Encoding
i
E
Input token
w_i

Summary

- The Transformer gives you a hidden vector representing all previous words.
- The LM head converts that vector into **vocabulary scores**.
- Softmax turns those scores into **probabilities**.
- The highest-probability token is the model’s prediction.

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)
- We 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 # of parameters *N*, dataset size *D*, compute budget *C* *(*if other two are held constant)

L(N, D, C) ≈ L(N) + L(D) + L(C)

$$
\begin{aligned}
L(N) &= \left(\frac{N_c}{N}\right)^{\alpha_N} \\
L(D) &= \left(\frac{D_c}{D}\right)^{\alpha_D} \\
L(C) &= \left(\frac{C_c}{C}\right)^{\alpha_C}
\end{aligned}
$$

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.

Scaling Laws

Loss as a function of model size

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

Meaning:

- N = the number of parameters you use
- N_c = a constant reference model size (the size where loss is measured)
- α_N = scaling exponent for model size

Interpretation:

As you make the model **larger**, i.e. N increases,

$$\frac{N_c}{N} \text{ gets smaller}$$

and raising a small number to a positive power makes $L(N)$ **smaller**.

➞ Bigger model ⇒ lower loss (smooth, predictable decrease).

Scaling Laws

Loss as a function of dataset size

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

Meaning:

- D = amount of training data
- D_c = reference dataset size
- α_D = scaling exponent for data

Interpretation:

If you increase the dataset size D:

$$\frac{D_c}{D} \text{ becomes smaller}$$

So the loss decreases.

➞ More data ⇒ lower loss.

Scaling Laws

Loss as a function of compute

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

Meaning:

- C = your compute budget
- C_c = reference compute
- α_C = scaling exponent for compute

Interpretation:

If you increase compute (longer training, bigger batches, etc.):

$$\frac{C_c}{C} \rightarrow \text{smaller value}$$

So the loss becomes smaller.

➞ **More compute ⇒ lower loss.**

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:

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

- 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

During inference: we generate *one token at a time*

When you chat with an LLM, the model **predicts the next token**, then the next, then the next…

Not all at once.

So at token step t, the model must compute:

- the query Q_t
- the key K_t
- the value V_t

for the **new token only**.

But to compute attention, we also need the keys and values of **all previous tokens**:

$$\left(Q_t K_1^T, Q_t K_2^T, \dots, Q_t K_{t-1}^T\right)$$

The problem: recomputing old keys and values is too expensive

If we recomputed **all previous** keys/values at every step:

- Step 1: compute K_1, V_1
- Step 2: compute K_1,V_1 again + compute K_2,V_2
- Step 3: compute K_1,V_1 again + K_2,V_2 again + compute K_3,V_3
- …

This would cost **quadratic time**, extremely slow.

For sequences of 8k, 32k, 100k tokens, this would be *impossible*.

The solution: KV Cache

➤ **Instead of recomputing previous keys and values every time,**

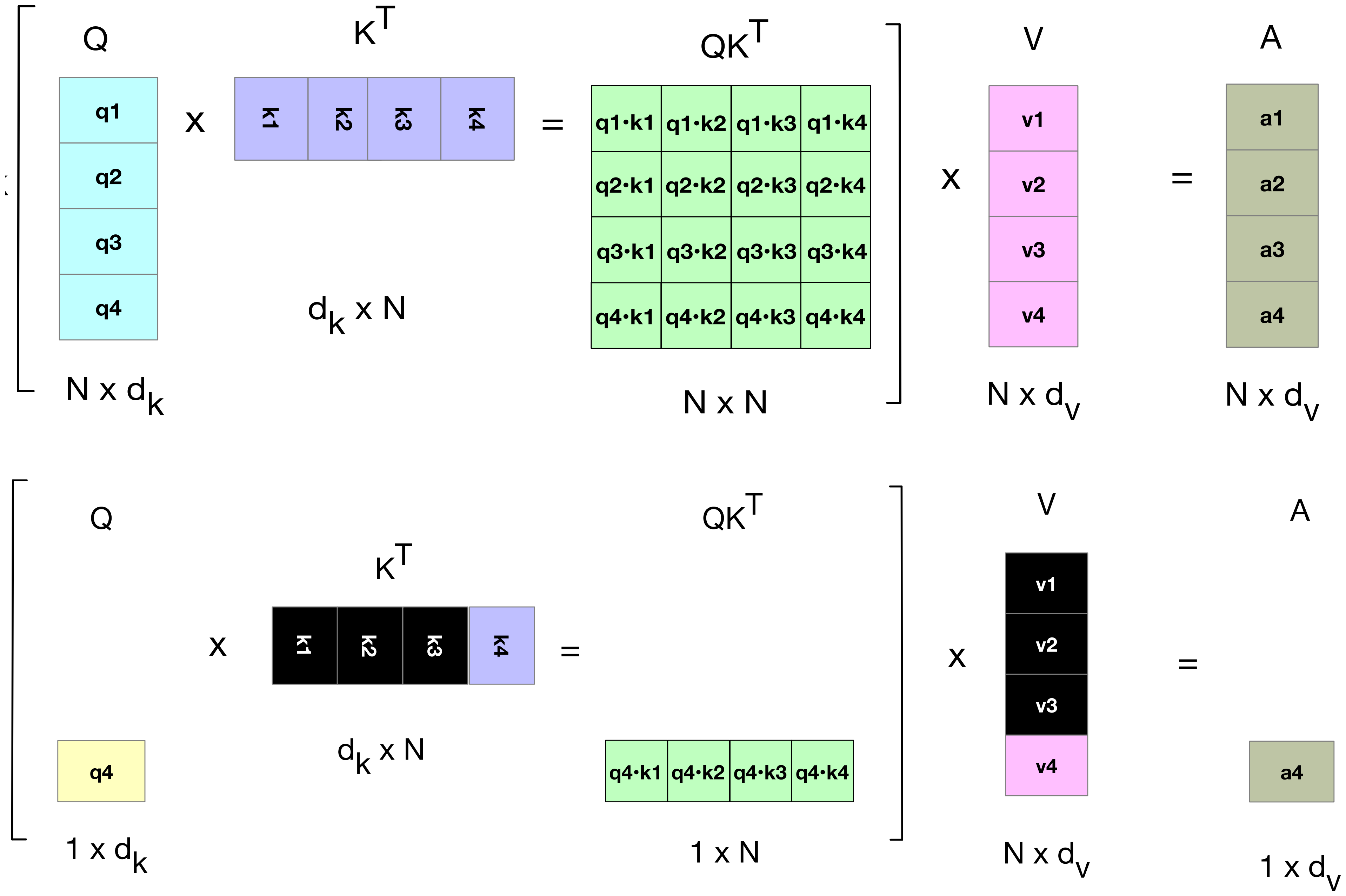
the model **stores** them in memory.

At each new token:

1. Compute Q, K, V for the new token only
2. Append the new K and V to the cache
3. To compute attention:
    - Use Q for the new token
    - Use **cached** Keys/Values for all previous tokens

This prevents any re-computation.

KV Cache


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

End

Thank you for your attention.