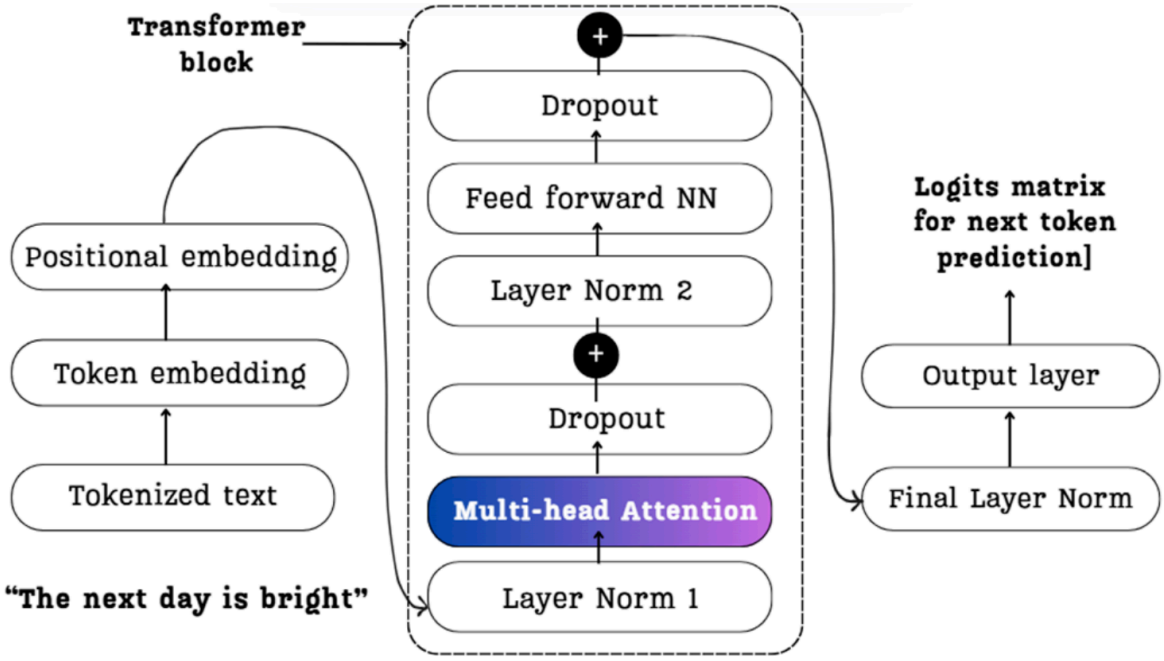
@April 28, 2026

🔗 References provided during lecture

Inference Engineering (Lecture 2)



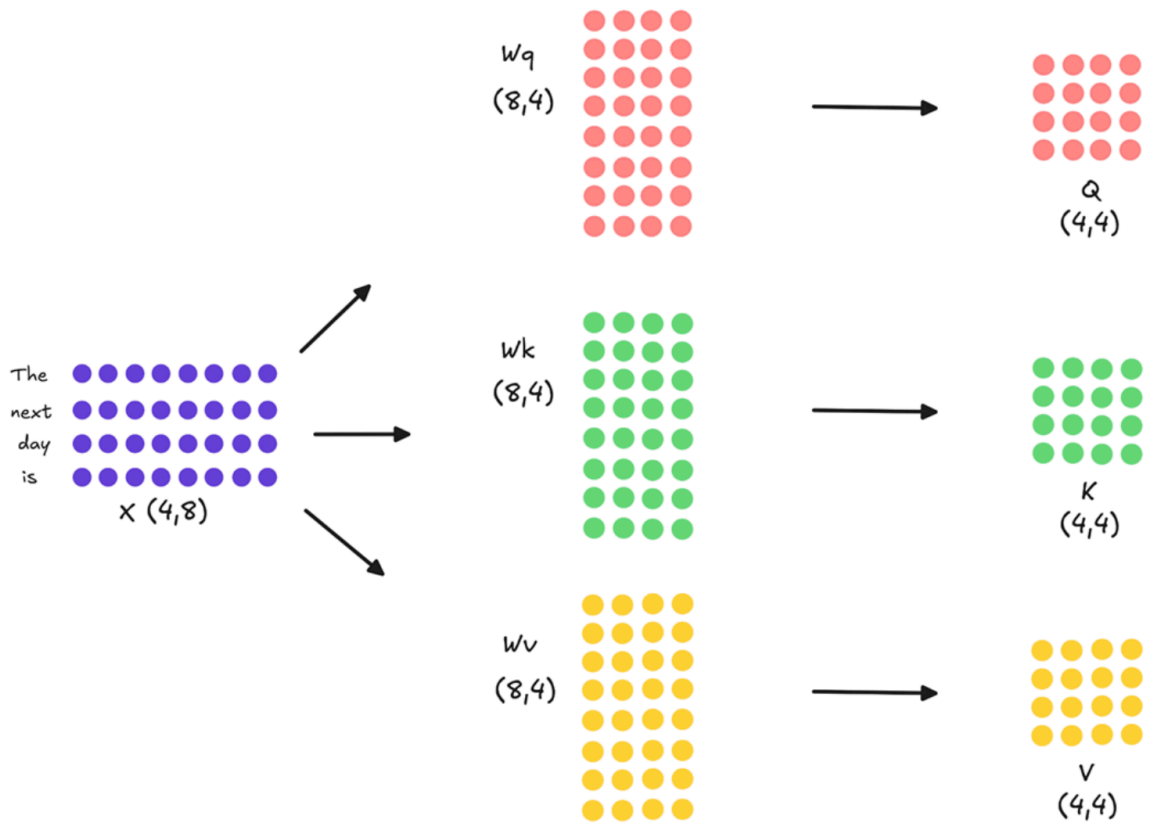
Remember this architecture diagram in your mind. We do tokenization. We get token embeddings. We have position embedding. Add the token embedding to position embedding. That gives us **input embedding**. And this input embedding goes inside **Transformer Block**.

We will work on the sentence "*The next day is bright*".

Some considerations done for simplicity are:

- Dimensions = 8

- Assuming already we have **input embedding**, entered **Transformer Block**, we have applied **Layer Norm 1** and we are behind the door of **Multi-Head Attention** Block.



What happens now?

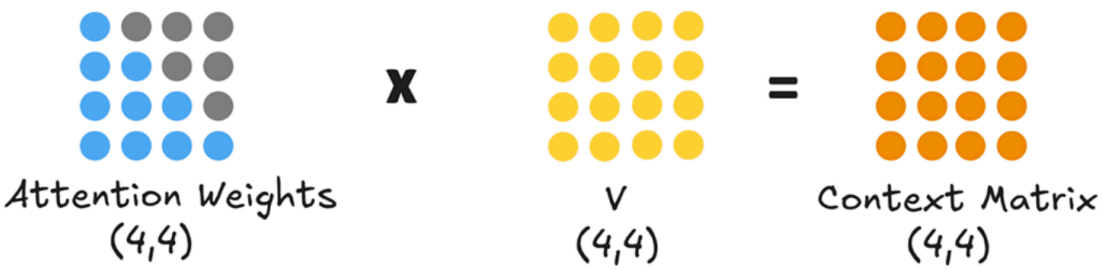
- Size of the input = $(4, 8)$
- Multiply this **input embedding** matrix with the **trainable query weight matrix** (W_q) and multiply with **trainable keys weight matrix** (W_k) and multiply with **trainable value weight matrix** (W_v) with dimensions $(8, 4)$.
- Output we get: **Query** matrix $(Q_{(4,4)})$, **Key** matrix $(K_{(4,4)})$ and **Value** matrix $(V_{(4,4)})$
- Multiply **Query** $(Q_{(4,4)})$ matrix with **Key** $(K_{(4,4)})$ matrix to get **Attention Scores** $(4, 4)$

- We apply **Scaling** + **Causal Attention** + **Softmax**. But why?
  - **Scaling**
    - Suppose attention scores are: $S = QK^T$
    - Each entry becomes: $S_{ij} = q_i \cdot k_j$, measures how much token i should attent to token j
    - Raw dot products aren't enough.
    - We compute: $\frac{QK^T}{\sqrt{d_k}}$ where $\sqrt{d_k}$ is key/query dimension
    - Problem here is dot products grow with dimension. If **Query** (Q) dimension = 64, entries roughly variance 1
    - Then $q.k$ can become large.
    - Large scores entering **Softmax** causes extremely peaked probabilities, softmax saturation and tiny gradients.
    - Without scaling: $[20, 21, 22]$, Softmax becomes: $[0.09, 0.24, 0.67]$. For even large: $[200, 210, 220]$, Softmax nearly becomes $[0, 0, 1]$. Training becomes unstable.
    - But why specifically $\sqrt{d_k}$? It is because variance of dot products grows proportional to dimension: $Var(q \cdot k) \propto d_k$. Hence dividing by $\sqrt{d_k}$ normalizes variance back near 1.
  - **Causal Attention**
    - In autoregressive LLM, token t must not see future tokens.
    - Sentence: "*The sky is bright*" when predicting "*bright*" the token "*is*" should not look ahead at "*bright*". Otherwise training cheats.
    - So we apply triangular mask.
    - Before masking: $[q_i \cdot k_1, q_i \cdot k_2, q_i \cdot k_3, q_i \cdot k_4]$

        - After masking: $[score1, score2, score3, -\infty]$
        - Softmax of $-\infty$ becomes zero. Thus future tokens receive zero attention.
        - Causal mask creates autoregressive behavior:
            - Without mask: model becomes bidirectional like BERT
            - With mask: model predicts left-to-right like GPT
    - **Softmax**
        - Raw scores are arbitrary numbers: $[2.1, 5.3, -1.7]$
        - We need normalized weights, non-negative values, interpretable importance
        - Softmax converts scores into probabilities: $[0.04, 0.94, 0.02]$
        - Now attention becomes weighted averaging: $output = \sum_j \alpha_{ij} \cdot v_j$ where $\alpha_{ij} = softmax(S_{ij})$
    - **One-Line Intuition**
        - Below table helps:

Component	Purpose
QK^T	similarity
scaling	stabilize gradients
causal mask	prevent cheating
softmax	probability distribution
multiply with V	retrieve information

- We the multiply **Attention Weights** $(4, 4)$ with **Value** $(V_{(4,4)})$ matrix to get **Context Matrix** $(4, 4)$



    - Why is it called **Context Vector** after multiplying **Attention Weights** $(4, 4)$ with $V_{(4,4)}$?
        - If you see whole LLM Architecture, there is only one place where one token pays attention to all the other tokens and that is **Attention** Block. Here one token learns to pay attention to other neighbor tokens and it learns to quantify how much attention should I pay to each neighbor.
        - So its own embedding becomes a weighted sum of those attentions. Now each token embedding gets context of its neighbors. Hence it is called **Context Vector**.
- When the **Token Embedding** (now with **Context**) comes out of Attention block, all operations that happens later is on individual embeddings. Only in the Attention block each **Token Embedding** interacts to with one other. All operations that happen before and after Attention block happens on individual **Token Embedding**.
- And this cycle happens with the number of **Transformer** blocks in the architecture. Apparently, life of a **Token** is not easy.

▼ **QUESTION: If one Token Embedding attends to all other Tokens in Attention Block, why do you need multiple Transformer Blocks?**

One attention layer lets a token **look at** all other tokens.

But a single layer is usually not enough to perform the kinds of multi-step reasoning and feature construction needed in language.

The key idea is:

> Attention lets tokens communicate once.
>
> Multiple transformer blocks let information be progressively refined and composed.

What happens in ONE block?

Suppose input is:

```
"The animal didn't cross the street because it was too tired."
```

To understand:

```
"it" → refers to animal
```

one layer may already help.

But for harder reasoning:

```
"The trophy doesn't fit in the suitcase because it is too big."
```

the model must determine:

- what objects exist
- relative sizes
- causal relation
- pronoun resolution

That often requires multiple sequential transformations.

Important Insight

Attention is not “thinking forever.”

Each layer performs roughly:

1. gather information
2. mix information
3. transform representations

Then the next layer works on the transformed representations.

Analogy

Imagine a room full of people.

One attention layer

Everyone gets:

- ONE round of communication

After that, each person updates their notes once.

Multiple layers

Now:

- round 1: exchange raw facts
- round 2: combine facts
- round 3: infer relationships
- round 4: build abstractions
- round 5: reason at higher level

Deep reasoning emerges from repeated refinement.

Mathematically

One block computes something like:

$$H^{(1)} = \mathrm{TransformerBlock}(X)$$

Next block:

$$H^{(2)} = \mathrm{TransformerBlock}(H^{(1)})$$

and so on.

$$H^{(l+1)} = \mathrm{TransformerBlock}(H^{(l)})$$

Each layer sees richer representations than the previous one.

What deeper layers learn

Empirically:

Layer depth	Typical behavior
Early layers	syntax, local patterns
Middle layers	phrases, dependencies
Later layers	semantics, reasoning
Final layers	task-specific prediction

This hierarchical abstraction is similar to CNNs:

- early CNN layers detect edges

- deeper layers detect objects

Transformers do hierarchical language abstraction.

Why attention alone is insufficient

Attention itself is mostly: $\text{weighted averaging}$

A single weighted average cannot express highly compositional reasoning.

The power comes from:

- repeated attention
- nonlinear MLPs
- residual connections
- stacking many layers

Another crucial point:

Receptive field vs computation depth

One layer gives full visibility:

```
every token can see every other token
```

But visibility is NOT the same as computation depth.

Example:

A human can see an entire chessboard instantly.

But strong chess reasoning still requires many sequential reasoning steps.

Transformers are similar.

Role of MLP inside each block

A transformer block is not just attention.

It contains:

1. Attention

2. Feed-forward network (MLP)

The MLP performs nonlinear feature transformation.

Without stacking:

- expressivity becomes limited

Depth allows repeated nonlinear composition.

Key intuition

A single attention layer answers:

> "What information from other tokens is relevant?"

Multiple layers answer:

> "After integrating information repeatedly, what higher-level concepts emerge?"

That's why large LLMs use:

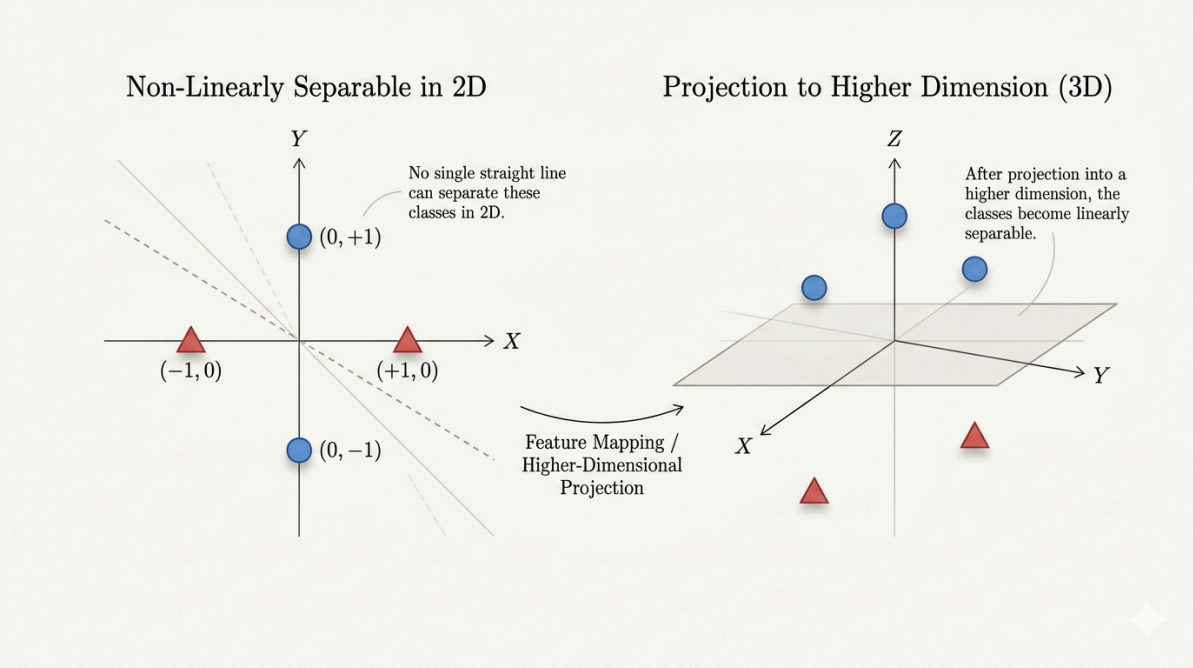
- 32 layers
- 80 layers
- sometimes 100+ layers

even though every layer already has global attention.

▼ **QUESTION: Why do we want more parameters to be thrown at the model?**

In low-dimensional space, some patterns are impossible to separate with a simple boundary. Deep learning models solve this by projecting data into higher-dimensional spaces, where hidden structures become easier to distinguish. More parameters allow models to learn richer transformations and capture more perspectives of the data. In essence, higher-dimensional representations make complex problems simpler to separate and understand.

In low-dimensional space, some patterns are impossible to separate linearly. By projecting the data into a higher-dimensional representation, neural networks uncover perspectives where simple decision boundaries become possible.

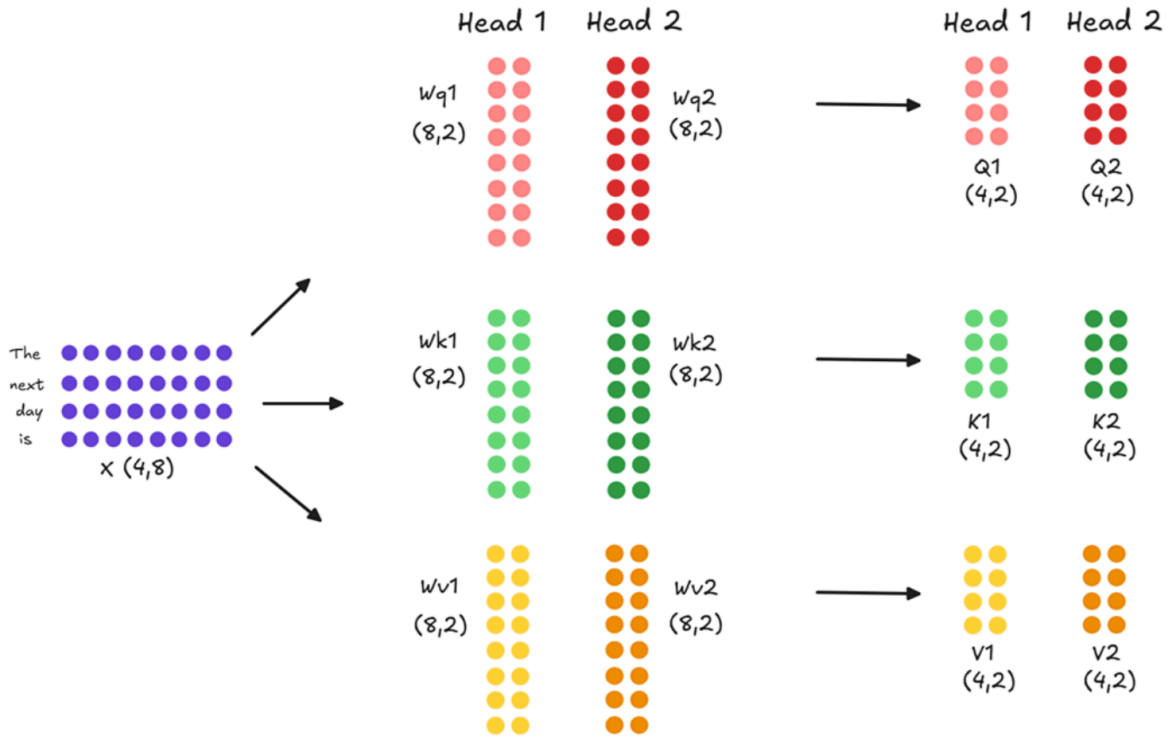


This illustrates the core intuition behind feature mappings and the kernel trick: transform the representation so that a nonlinear problem becomes linearly separable.

Multi-Head Attention

One assumption we did until we get **Context Vector** now is we had just one **Head**. Then where does this Multi-Head comes from?

With the same example that is seen above, we have the same **Input Embedding**. We still multiply this **input embedding** matrix with the **trainable query weight matrix** (W_q) and multiply with **trainable keys weight matrix** (W_k) and multiply with **trainable value weight matrix** (W_v) but it is just that **these trainable matrices are split into** 2. And we call it as **Head 1** and **Head 2**.



Instead of:

- $W_q(8,4)$ we have $W_{q_1}(8,2)$ and $W_{q_2}(8,2)$
- $W_k(8,4)$ we have$W_{k_1}(8,2)$ and $W_{k_2}(8,2)$
- $W_v(8,4)$ we have $W_{v_1}(8,2)$ and $W_{v_2}(8,2)$

After Multiplying Input Embedding with these trainable weight matrices we get:

- $Q_1(4,2)$ and $Q_2(4,2)$
- $K_1(4,2)$ and $K_1(4,2)$
- $V_1(4,2)$ and $V_2(4,2)$

This is just splitting the Q, K and V into 2 parts.

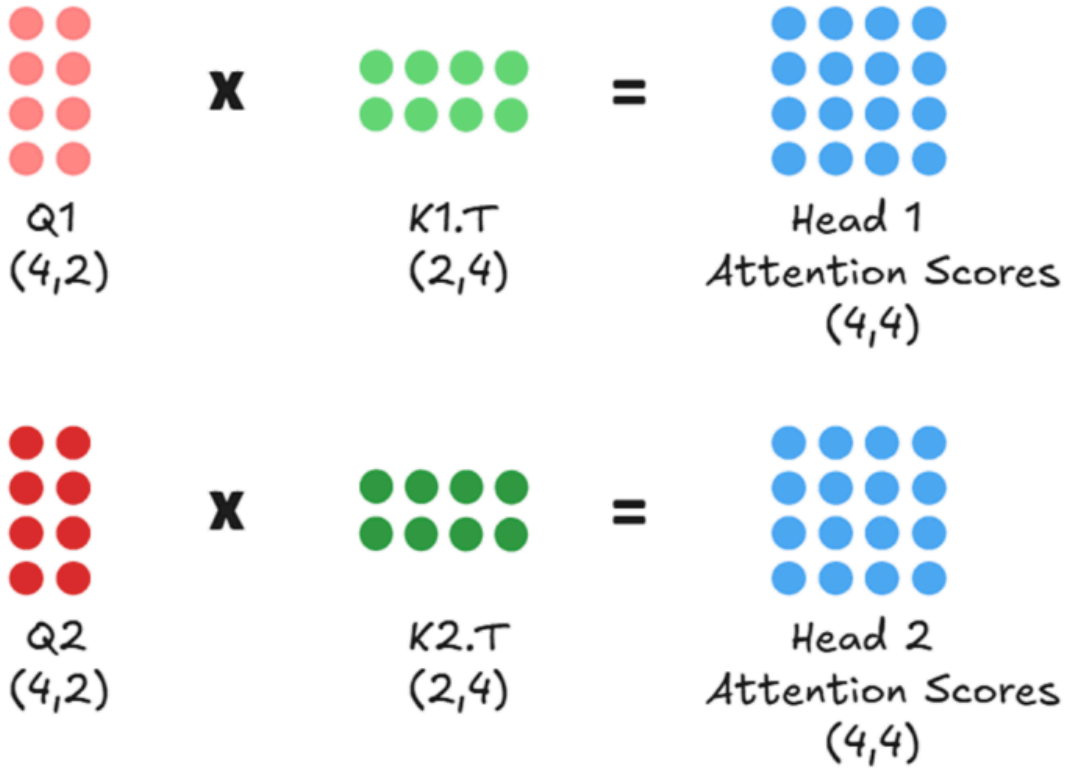
▼ **QUESTION: How many attention scores matrices will we have and what will be the dimensions of the attention score matrices?**

Size of the attention score matrices remains exactly the same

$$\text{Number of attention score matrices} = \text{Number of attention heads}$$

So, we get 2 attention score matrices and the size of the attention score matrix remains the same.

- **Head 1** attention score matrix is $(4, 4)$
- **Head 2** attention score matrix is $(4, 4)$



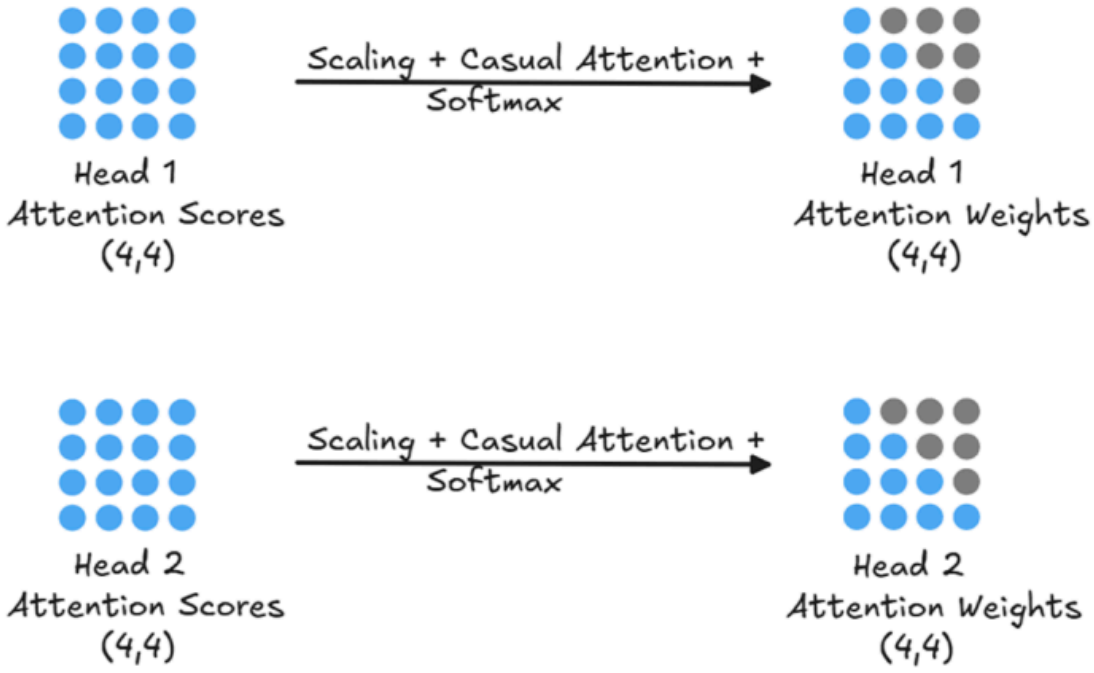
But wondering why do we need two attention score matrices? This is because, a given sentence might have two perspectives which can be captured.

Consider the sentence, "*The artist painted the portrait of a woman with a brush*". It can mean the artist actually painted the portrait of a woman who had a brush in her hand and it can also mean the artist painted the portrait of a woman using a painting brush. The brush can be with painter or woman here in this sentence.

If you have one **Head** and get one **Attention Score** matrix, that matrix will probably have a high attention between artist and brush. You clearly need another attention score matrix which has high attention score between woman and brush. This will never happen with just one head. That is why you have 2 **Heads**. Each head captures different perspective.

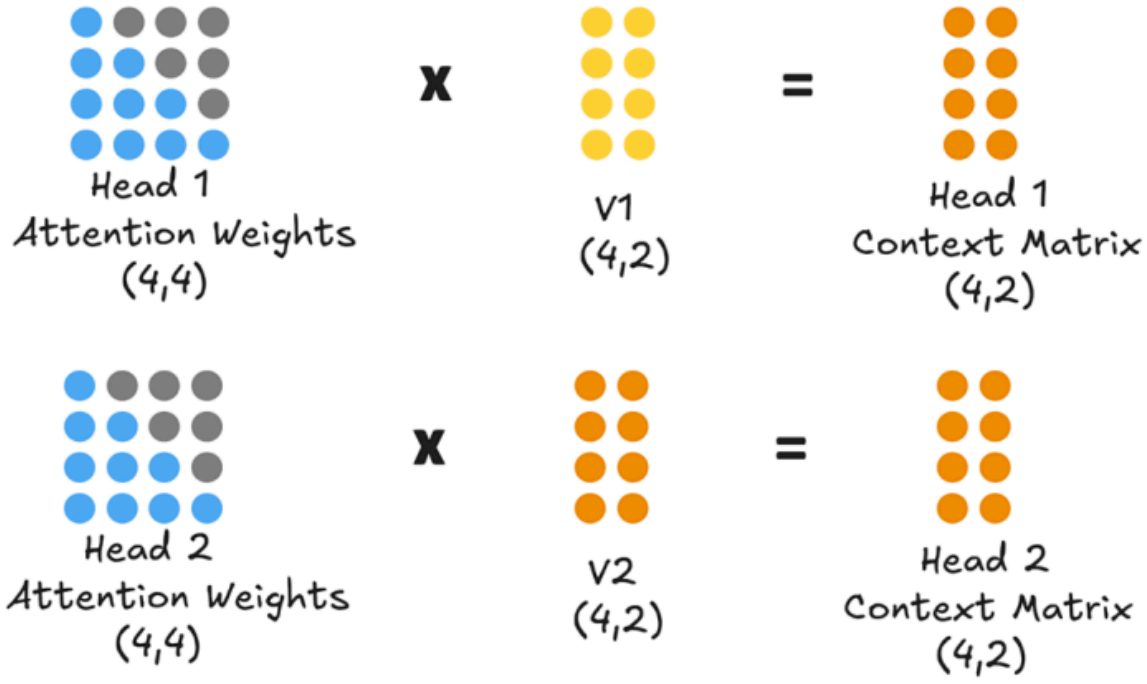
This is just example with 2 **Heads**; **DeepSeek** models, they have 48 Heads, 96 Heads etc. And it is not just limited to perspectives, it learns many things like one **Head** can focus on grammar, tone and other **Head** focus on subject, object and other **Head** can focus on verb, proverb and other **Head** can focus on voice etc.

What happens next is same for general **Attention Scores** like **Scaling**, **Causal Attention** and **Softmax**. **Scaling** can be a bit different as keys dimension is different as keys dimension would have been reduced due to number of **Heads** but the gist remains the same.



So, we now have **Head 1 Attention Weight** and **Head 2 Attention Weight** and it is $(4, 4)$. Even in the case of single head attention it was $(4, 4)$.

Now what we do is multiply it with $Value$ $V_1(4, 2)$ and $V_2(4, 2)$ to get **Head 1 Context Matrix** $(4, 2)$ and **Head 2 Context Matrix** $(4, 2)$



We only need one **Context Matrix** $(4, 2)$. What do we do? We just concatenate them together and that gives **Final Context Matrix** $(4, 4)$. This Final Context Matrix consists of two perspectives.

Research Topic Area

We do not know which context matrix to prioritize. So best option right now is concatenate. What we can do is add a penalty parameter like α for each Heads and we can learn the α_1 and α_2 on the go to make sure which perspective matters the most.

Common Terminology

When you see **Head** dimension, it is the embedding dimension divided by number of heads.

$$\text{Head Dimension} = \frac{\text{Embedding Dimension}}{\text{Total Number of Heads}}$$

In our example, Embedding Dimension = 4, Total Number of Heads = 2

Hence, $\text{Head Dimension} = 4/2 = 2$

So, in $(4, 2)$ the 2 comes from this formula.

▼ **QUESTION: Doesn't the dot product between Input Embedding and different Heads always going to be relatively the same?**

It is not going to be the same because the weight values itself are fundamentally different. So the dot product will not be the same.

▼ **Why projection after obtaining Context Matrix?**

Step 1 — Single-head attention

You said:

Input embedding: $X \in \mathbb{R}^{4\times 8}$

One head: $W_Q, W_K, W_V \in \mathbb{R}^{8\times 4}$

So finally you get: $H \in \mathbb{R}^{4\times 4}$

This single head learns ONE way of relating tokens.

Step 2 — Two-head attention

Now suppose:

Each head uses:

$W_Q^{(1)}, W_K^{(1)}, W_V^{(1)} \in \mathbb{R}^{8\times 2}$

$W_Q^{(2)}, W_K^{(2)}, W_V^{(2)} \in \mathbb{R}^{8\times 2}$

Each head produces:

$H_1 \in \mathbb{R}^{4\times 2}$

$H_2 \in \mathbb{R}^{4\times 2}$

Then concatenate: $[H_1; H_2] \in \mathbb{R}^{4\times 4}$

So far so good.

Important point: Concatenation does NOT combine the heads intelligently

After concatenation, the representation is basically:

```
[ perspective_1 | perspective_2 ]
```

The heads are merely sitting side-by-side.

Nothing has yet decided:

- which perspective matters more
- how to combine them
- whether some dimensions from head 1 should interact with head 2

That is exactly the role of: W_O

the output projection matrix.

What your fellow student meant

People say:

> "The output projection does not change at all."

Meaning:

Whether you use:

- 1 head
- 2 heads
- 32 heads

eventually transformers always project back into the SAME model dimension.

Example:

Concatenated heads: $\mathbb{R}^{4\times4}$

Then: $W_O \in \mathbb{R}^{4\times8}$

projects back to: $\mathbb{R}^{4\times8}$

So the transformer block interface stays consistent.

This is the deeper idea.

Suppose:

Head 1 learns:

```
"brush" relates to painter using brush
```

Head 2 learns:

```
"brush" relates to woman holding brush
```

Now after concatenation:

```
[perspective_from_head1 | perspective_from_head2]
```

The output matrix: W_O

learns how much importance to give to each perspective.

Very important intuition

W_O is NOT just reshaping dimensions. It is a learned mixing operation.

It can:

- amplify one head
- suppress another head
- combine pieces of multiple heads
- create interactions across heads

The key mathematical intuition

Suppose concatenated output is:

$H = [h_1, h_2]$

Then: HW_O is a linear combination of ALL head outputs.

So each output dimension can become:

$0.8h_1 + 0.2h_2$ or $-0.3h_1 + 1.7h_2$ etc.

Thus:

```
W_O learns how to merge multiple perspectives
```

Analogy

Imagine:

- Head 1 = one detective's interpretation
- Head 2 = another detective's interpretation

Concatenation is just placing both reports on a table.

But W_O acts like the chief investigator who decides:

- which report matters more
- how to combine evidence
- what final conclusion to form

Crucial deeper insight

Without W_O heads remain mostly isolated.

With W_O information across heads can interact.

This is why multi-head attention becomes far more expressive than merely “many independent attentions.”

One subtle but VERY important thing

People often think:

```
multi-head attention = independent perspectives
```

That is only half true.

The REAL power comes from:

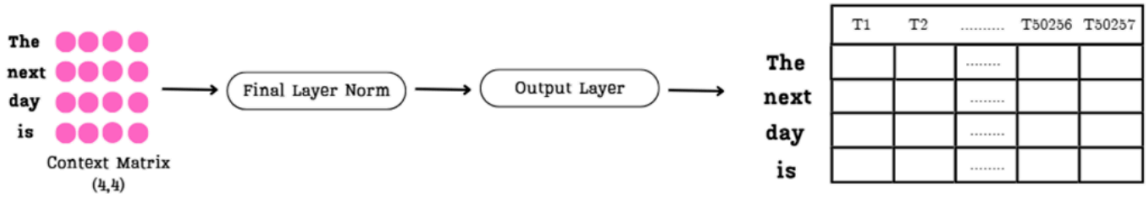
```
independent perspectives + learned recombination via W_O
```

That recombination is essential.

Final intuition in one sentence

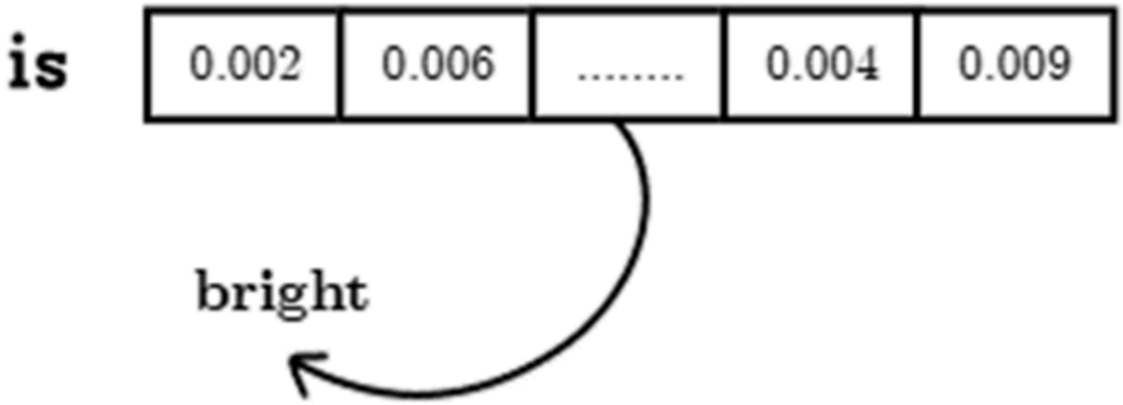
Multi-head attention lets different heads capture different relationships, and the output projection matrix (W_O) learns how to intelligently combine those relationships into a unified contextual representation.

Once we come out of **Multi-Head Attention** block, we have the **Context Matrix** which has to go through **Feed Forward Neural Network**, it has to go through **Dropout**, it has to go through **Residual Block**. This is the path from **Context Vector** to **Logits**.



In the Output Layer, the **Context Vector** is converted from $(4, 4)$ to $(4, 50257)$. Here 50257 is the size of vocab. This is where prediction of the next token actually happens.

We look at the last token that is "*is*", we apply **softmax** and then look at the values and then we choose the value with the maximum probability. So the next token is now "*bright*".



▼ **Are probabilities different here? Softmax applied twice?**

Confusion comes from mixing up:

1. attention probabilities
2. output token probabilities

These are completely different things.

Softmax to get **Logits** is about the SECOND one.

Step-by-step

Suppose input is:

```
"The sky is"
```

After all transformer blocks, the final hidden/context vectors may be:

$H \in \mathbb{R}^{4\times4}$

(Using your toy example dimensions.)

Each row corresponds to one token position.

So:

Position	Token	Final context vector
1	The	(h_1)
2	sky	(h_2)
3	is	(h_3)
4	current position	maybe masked

To predict next token, we only use: h_3 the final context vector of “is”.

Output projection layer

Now this vector is projected into vocabulary space:

$z = h_3 W_{vocab}$

where:

$W_{vocab} \in \mathbb{R}^{4 \times 50257}$

Result:

$z \in \mathbb{R}^{50257}$

$z = hW_{vocab}$

These 50,257 numbers are called:

```
logits
```

Critical point

These are NOT embedding values.

They are:

```
scores for every vocabulary token
```

Example:

Token	Logit
bright	8.2
blue	7.9
cloudy	2.1
pizza	-4.0

Then softmax

Softmax converts logits into probabilities:

Token	Probability
bright	0.42
blue	0.31
cloudy	0.02
pizza	0.00001

Now the model has:

$$P(\text{next token} \mid \text{"The sky is"})$$

Then token selection

Now inference chooses next token.

Common methods:

Method	Behavior
Greedy	choose highest probability
Sampling	sample probabilistically
Top-k	sample from top k
Top-p	sample from cumulative mass

Your mentor described:

```
greedy decoding
```

which simply picks maximum probability token.

So:

```
bright
```

gets selected.

Your important confusion

You asked:

> "Is it because taking that value determines which token had highest attention?"

NO.

This is the key misunderstanding.

Attention probabilities are INTERNAL

Inside attention:

$\text{softmax}(QK^T)$

These probabilities determine:

```
which previous tokens to focus on
```

Example:

Previous token	Attention weight
The	0.05
sky	0.70
is	0.25

This happens INSIDE the transformer.

Output probabilities are DIFFERENT

Final softmax:

$$\text{softmax}(z)$$

determines:

```
which vocabulary token to generate next
```

Example:

Vocabulary token	Probability
bright	0.42
blue	0.31

Totally different meaning.

Important intuition

Attention softmax:

```
"Where should I look?"
```

Output softmax:

```
"What word should I say next?"
```

These are two entirely separate probability distributions.

Why highest probability token?

Because the model was trained to maximize:

$$P(\text{correct next token})$$

So during inference, choosing highest probability is:

```
choosing the token the model believes is most likely next
```

Not the token with highest attention.

Final mental model

Attention mechanism Produces: context vectors by mixing information from previous tokens.

Output layer

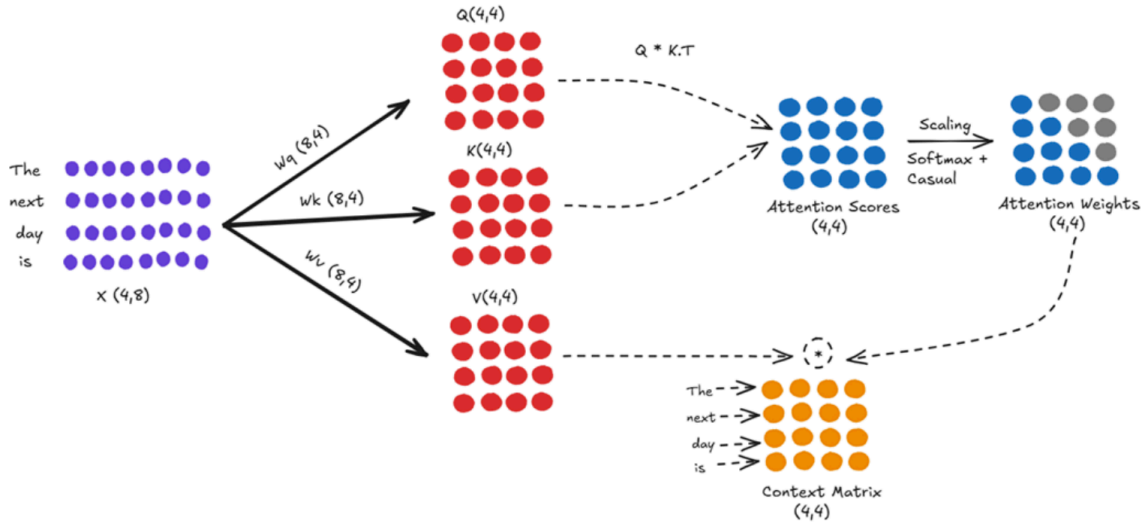
Transforms context vector into: probabilities over vocabulary for next-token prediction.

These are two separate stages of the transformer.

Pay attention here..

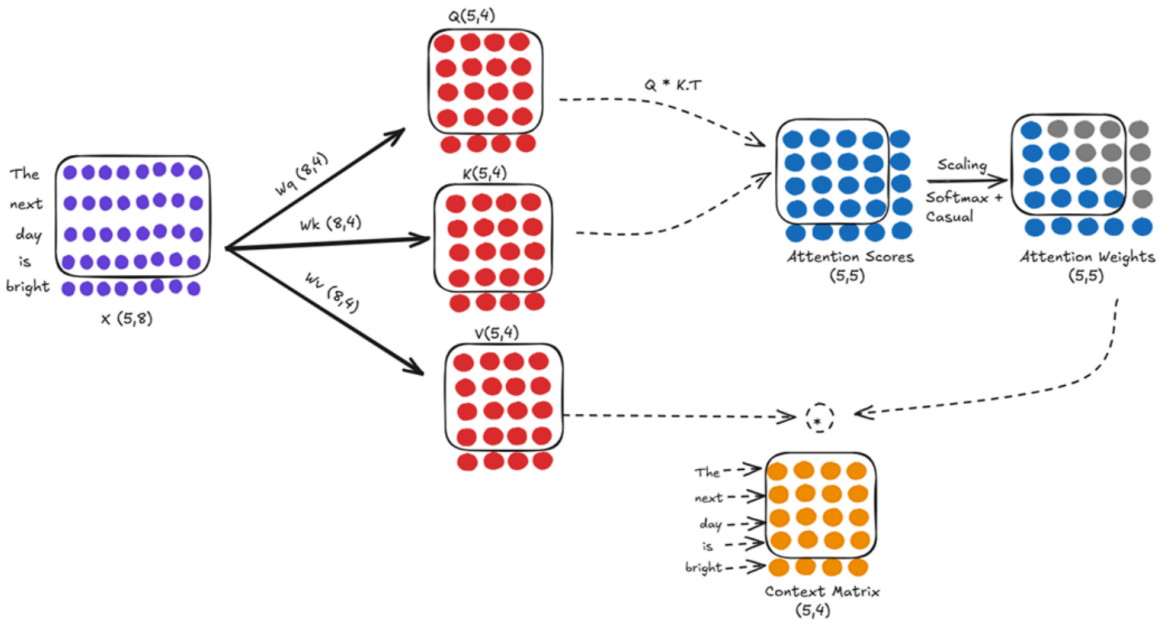
To predict the next token, only this last row ("*is*" logits) is important, all other 3 rows in **Logits Matrix** are not needed. In fact the **softmax** is not necessarily applied to other rows. It is applied only on the last row to get the **Logits**. We will now see the problem of redundant computations.

Our intuition is that, as new tokens are predicted **are we calculating the same thing over and over?**



Let assume we have the query "*The next day is*" from above image. We have seen how we arrive at **Context Matrix**. Now, if we get the next token as "bright", this is appended to the query and let us assume we start the same process again to get the **Context**

Matrix. One observation we can find is we are computing the same thing again and again. Every thing we see in black box in below image, we already know and computed before.



We are doing a huge number of repeated computations if we are doing **Naive Inference**.

At prefill, for an input of 4 tokens → we are doing 4×4 (16 scores)

At decode, we have one new token, now the input of 5 we calculate 5×5 (25 scores) again.

At decode, if we have 1000 tokens, we would be doing $1004 \times 1004 \approx (1\text{M scores})$ computations. (Note: originally 4-token prompt).

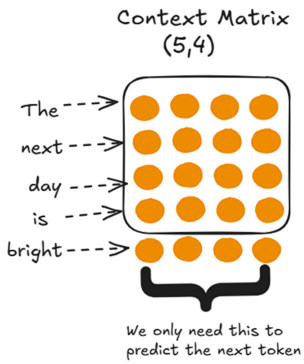
If Prefill has N tokens, Decode also had N tokens, **what is the order of magnitude of computations we are doing in Naive Inference?** It is N^3 because for every new token, we need to do N^2, for N new tokens added, it is $N \times N^2 = N^3$ number of operations again.

We do not want this. Something needs to be caches. Something needs to be stored.

KV Cache

Let's start with our end goal and work backward

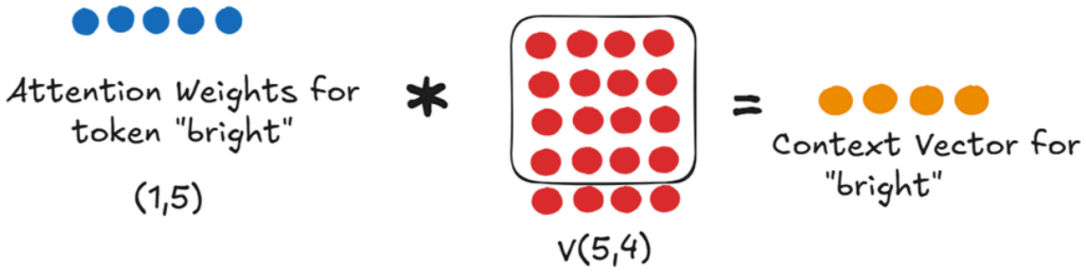
Goal: At every inference step, we need to produce a single context vector for the most recent token!



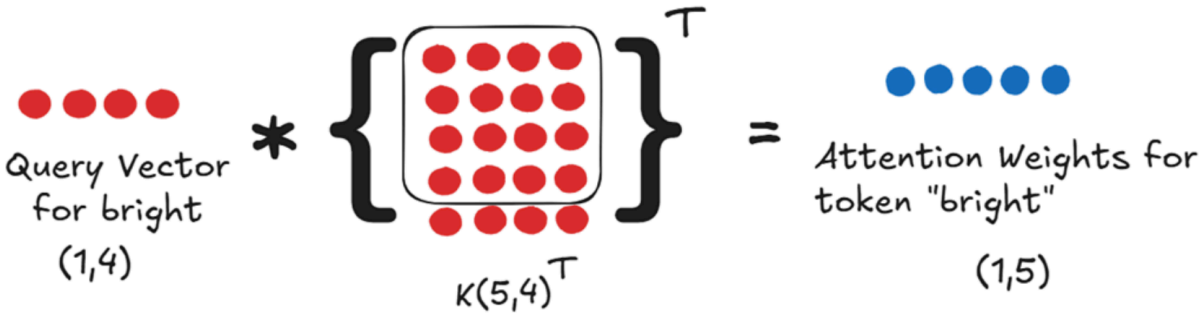
We do not need previous **context vectors** at all. For the current inference step, we only need a single context vector from the most recent token.

"*The next day is bright*" is the input token now and we want to predict the next token after "*bright*", so we need the **context vector** of "*bright*". We take that **context vector**, convert it to **logits** and get the next token.

Let us take a step back, if we need the **context vector** for "*bright*". How do we get it? We multiply the **Attention Weights** $(1, 5)$ for "*bright*" token with entire $Values(5, 4)$ matrix. In this $(5, 4)$, the black box is already **Cached** (below image). The black box can already be calculated in previous iteration.



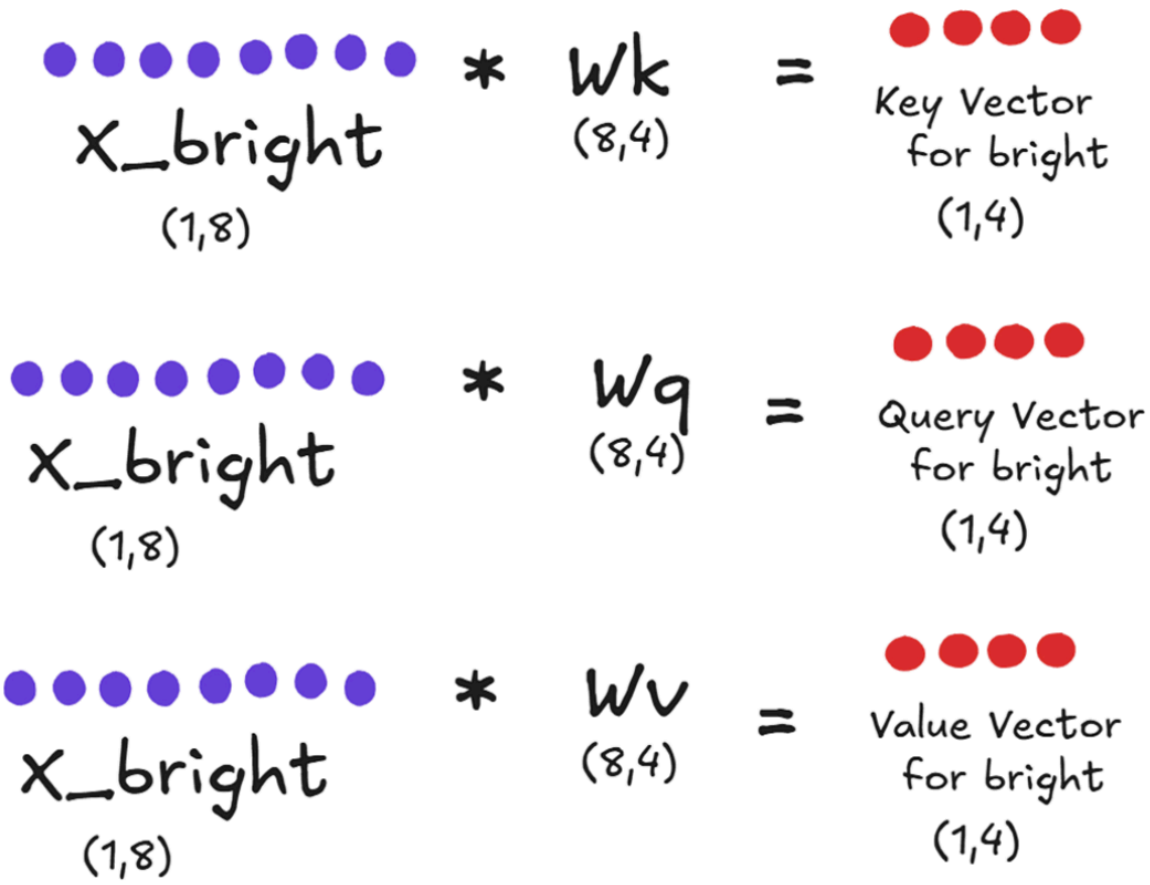
But then how did we get the Attention Weights for the token "*bright*"? We get that by multiplying Query $Q(1, 4)$ vector for "*bright*" with Keys transpose (K^T). Even the black box in Keys matrix can be **Cached** (see below image)



If we carefully observe, we only need:

- Query (Q) vector for "*bright*"
- Key (K) vector for "*bright*"
- Value (V) vector for "*bright*"

The black box in the above images are **Cached** and these above 3 vectors are appended to Q, K, V respectively (see below image).



These three vectors are appended to respective cache

Now to predict the next token after "*bright*", the current existing $K(5,4)$ and $V(5,4)$ will be **Cached**.

Inference Loop with KV Cache

1. Receive new token: The model looks up the embedding for new token
2. Compute Query vector, Key vector and Value vector
3. Retrieve from cache: K and V Cache of earlier tokens
4. Append to Cache: Append K and Value vector of new token to existing cache
5. Multiply Query vector with Keys → Attention Scores → Attention Weights
6. Multiply Attention Weight with Values → Content vector for new token
7. Context vector * W_o → get out of Attention block
8. Feed Forward Network
9. Multiple transformer blocks
10. Logits vector for new token
11. Select next token

This is repeated until:

- `<EOS>` token is reached
- Max New Tokens is reached
- Session terminates or crashes

Intuition why only KV needs to be Cached

- Keys (K) need to be cached because Attention Scores cannot be calculated without previous Keys
- Values (V) need to be cached because Context Vectors cannot be calculated without previous Values.

Storing the KV Cache is one of the biggest bottlenecks of inference engineering.

▼ Is the KV Cache unique for each transformer block?

Yes, each transformer block has its OWN separate KV cache. This is extremely important.

Why?

Because every transformer layer produces different representations.

Remember: $X^{(1)}, X^{(2)}, X^{(3)}, \ldots$

Each layer transforms the token representations into more abstract forms.

So:

- Layer 1 keys/values represent low-level features
- Layer 10 keys/values represent richer semantics
- Layer 30 keys/values represent highly contextual abstractions

Therefore:

```
K/V tensors from different layers are NOT interchangeable
```

Example

Suppose model has:

```
32 transformer blocks
```

Then you have:

Layer	KV Cache
Block 1	$(K^{(1)}, V^{(1)})$
Block 2	$(K^{(2)}, V^{(2)})$
Block 3	$(K^{(3)}, V^{(3)})$
...	...
Block 32	$(K^{(32)}, V^{(32)})$

So there are:

```
32 separate KV caches
```

(one per layer).

During inference

Suppose new token arrives.

For EACH layer:

1. compute new Q/K/V
2. append K/V to that layer's cache
3. query against THAT layer's cached K/V

So layer-by-layer:

Layer 1

$Q^{(1)}new(K^{(1)}cache)^T$

Layer 2

Uses completely different:
$Q^{(2)}new(K^{(2)}cache)^T$
because hidden representations changed after layer 1.

Very important intuition

The model is not storing:

```
one universal memory
```

Instead:

```
each layer maintains its own contextual memory
```

Why memory usage becomes huge

KV cache memory scales with: $\text{layers} \times \text{sequence length} \times \text{heads} \times \text{head dimension}$

$$\text{KV Cache Memory} \propto L \times T \times H \times d_{head}$$

where:

- L = layers
- T = sequence length
- H = number of heads

This is why long-context inference becomes memory-heavy.

Another subtle point

KV cache is also unique per:

- user session
- sequence
- request

Two users chatting with same model do NOT share KV cache.

They only share model weights.

Final intuition

Think of each transformer block as building its own evolving interpretation of the sequence.

Its KV cache stores:

```
what this specific layer currently remembers about all previous tokens
```

So every layer needs its own separate cache.

▼ Can I assume maximum size of KV Cache is the context window?

Yes, for standard autoregressive transformers, you can approximately think of:

$$\text{Maximum KV Cache Length} \approx \text{Context Window}$$

That intuition is correct.

Why?

KV cache stores:

- one K vector per token
- one V vector per token
- for every layer
- for every head

So as more tokens arrive:

```
KV cache grows token-by-token
```

until reaching the model's maximum supported context length.

Example

Suppose model has:

- context window = 8192

Then during inference:

Tokens processed	KV cache size
100	stores 100 tokens
1000	stores 1000 tokens
8192	full cache
>8192	old tokens must be dropped/slid/compressed

Important nuance

The cache size is not:

```
one giant 8192×8192 matrix
```

Instead, per layer it is more like:

$K \in \mathbb{R}^{T \times H \times d_{head}}$

$V \in \mathbb{R}^{T \times H \times d_{head}}$

where:

- T = current sequence length
- H = number of heads

Total KV cache memory

Across all layers:

$$\text{KV Cache} \propto L \times T \times H \times d_{head}$$

which is why long contexts consume enormous VRAM.

What happens at context limit?

Depends on architecture.

Possible strategies:

Strategy	What happens
Hard cutoff	reject more tokens
Sliding window	discard oldest tokens
Memory compression	summarize old tokens
Retrieval augmentation	move old info externally

Most classic LLMs historically used:

```
fixed-size sliding attention window
```

Important subtle point

Even if context window is 128K:

model does NOT necessarily “remember” all 128K equally well.

Attention quality often degrades with distance.

This is called:

- lost-in-the-middle problem
- long-context degradation

Another subtle distinction

Context window refers to:

```
maximum tokens model can attend to
```

KV cache refers to:

```
stored runtime representations enabling that attention
```

Closely related, but not identical concepts.

Final intuition

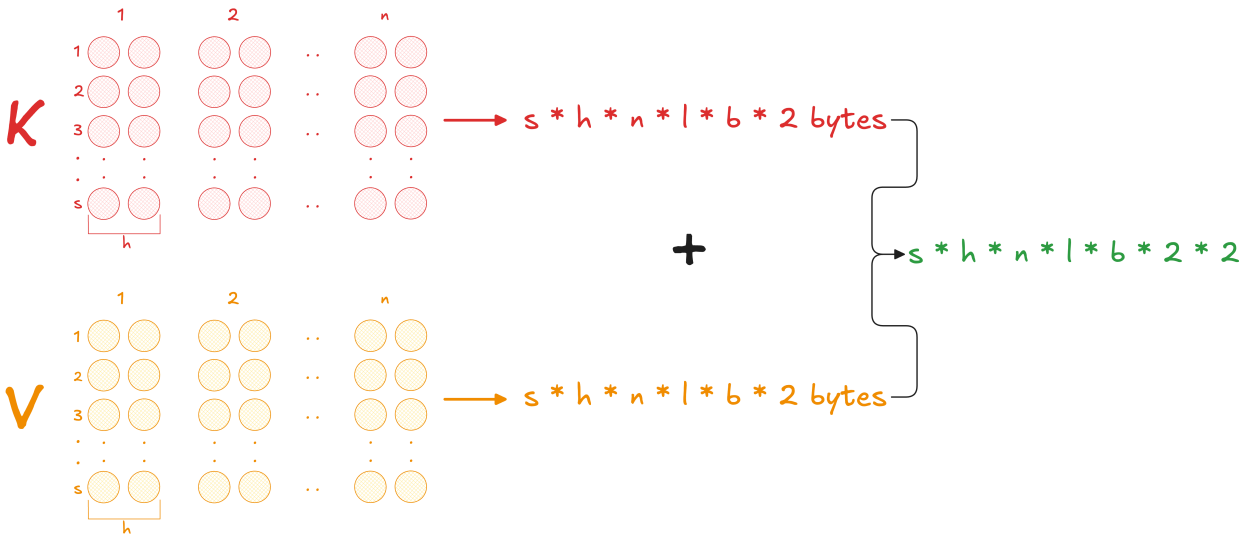
You can think of KV cache as:

```
the runtime memory implementation of the context window
```

So in standard transformers:

```
max KV cache size is effectively bounded by the context window
```

Size of the KV Cache Formula



Given:

- l : number of layers
- b : batch size
- n : number of attention heads
- h : head dimension
- s : sequence length

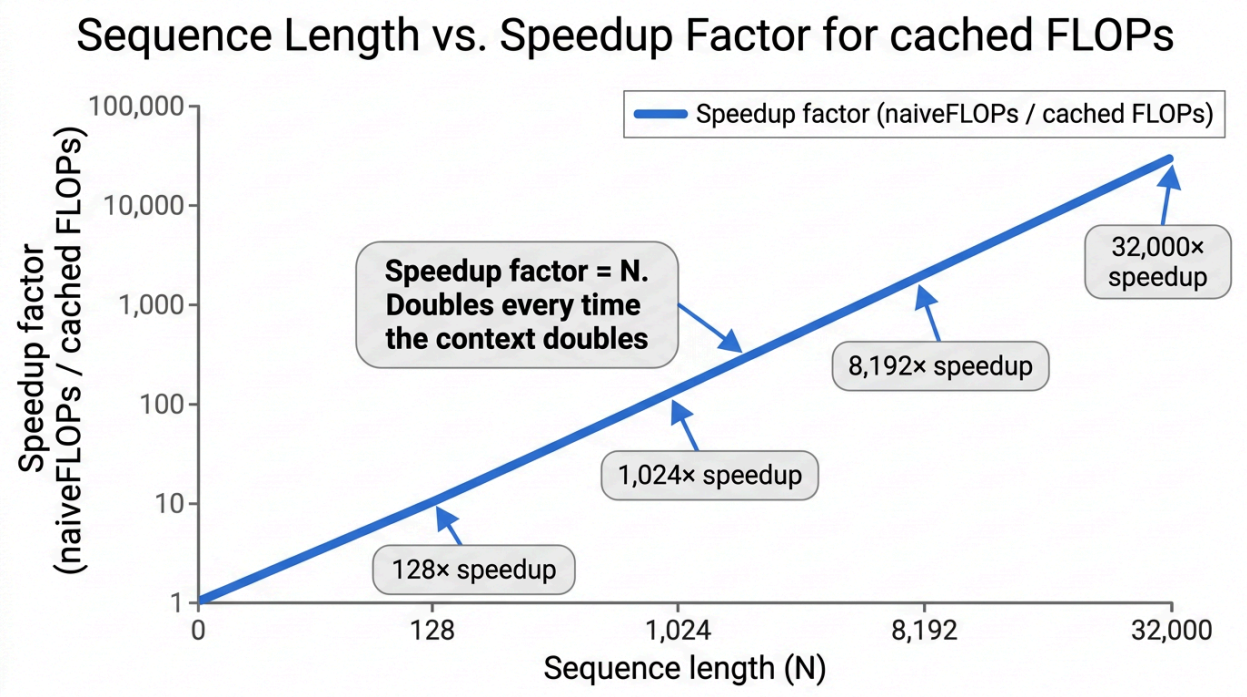
> For $FP16$ (2 bytes) :
> $\text{KV Cache Size} = l \times b \times n \times h \times s \times 2 \times 2 \quad \text{bytes taken up by KV cache}$

The Bright Side of KV Cache

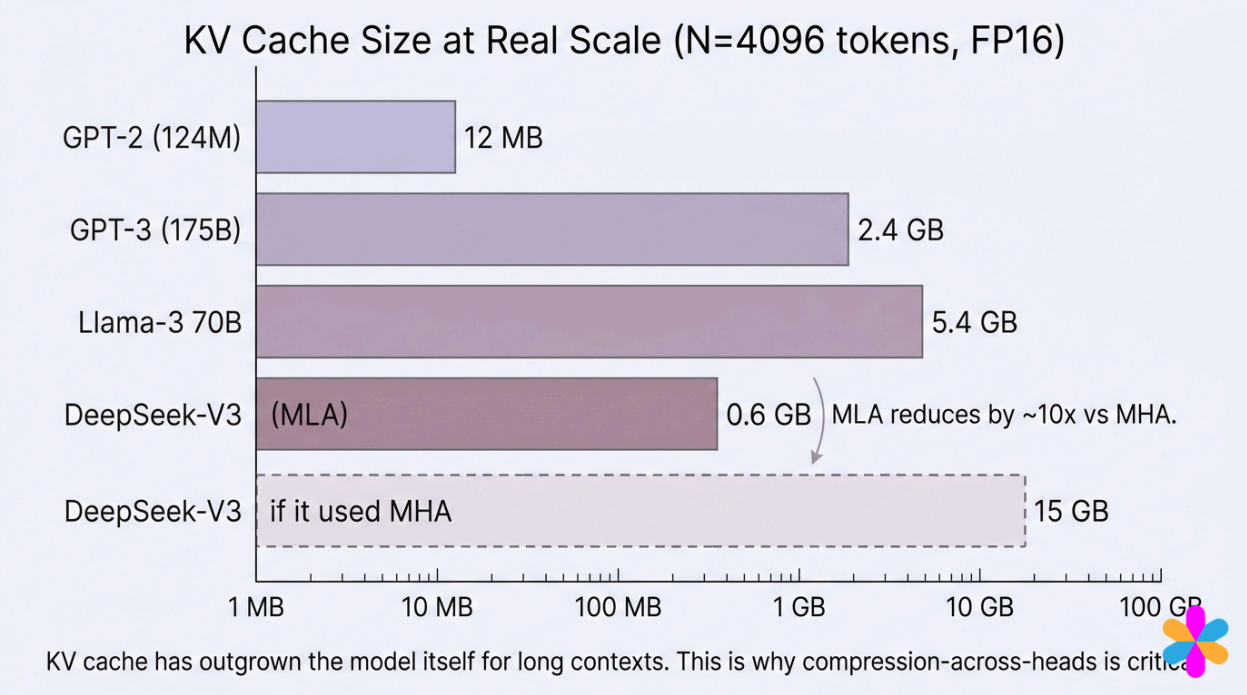
Tying it back to the metrics of inference engineering, it really helps **ITL** because it reduces the redundant computations. **TTFT** does not get benefited as it depends on prefill and prefill does not use KV Cache.

Because KV cache saves redundant computations so of course it saves the number of FLOPs for us and the number of FLOPs that saves for us goes on increasing with **Sequence Length** (N) and in fact it is directly proportional in the fraction of N. It directly gives the speed up during the inference.

$$\text{KV Cache} \propto \text{Sequence Length}$$



The Dark Side of KV Cache



Observe the above image, if you have GPT-2 (124M), you'd have 12MB of KV cache size. Even the powerful GPU now like H100 that has L1 and L2 cache where L1 is shared memory, is about 256KB per SM and unified 50MB of L2 cache. And no one uses GPT-2

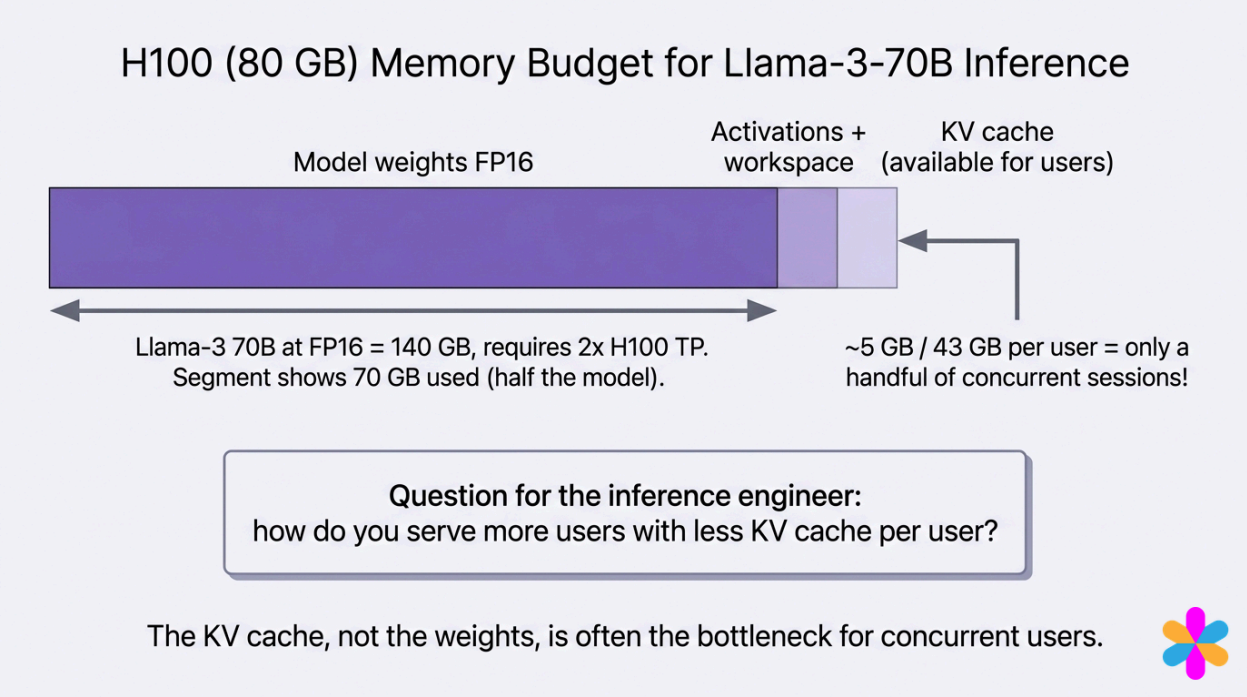
anymore. As size of model grows, the size of KV cache increases, with the number of layers, attention heads etc.

DeepSeek-V3 has about 15GB of KV cache. The only place where you can store this is GPU's SRAM (shared memory per SM).

Why is this a problem?

Let's say you are using H100 for inference, there are three things that take up space for inference: the model weights, the activations and the KV cache.

If less size is provided for KV cache to persist, then it is bad as only few number of users can do inference per second. Every user will need their own KV cache.



A horizontal stacked bar representing the $80GB$ of an H100's HBM, with segments labeled Model weights ($70GB$, with a footnote that Llama-3-70B at FP16 is actually $140GB$ and requires 2× H100 TP), Activations / workspace ($5GB$), KV cache available (~$5GB$).

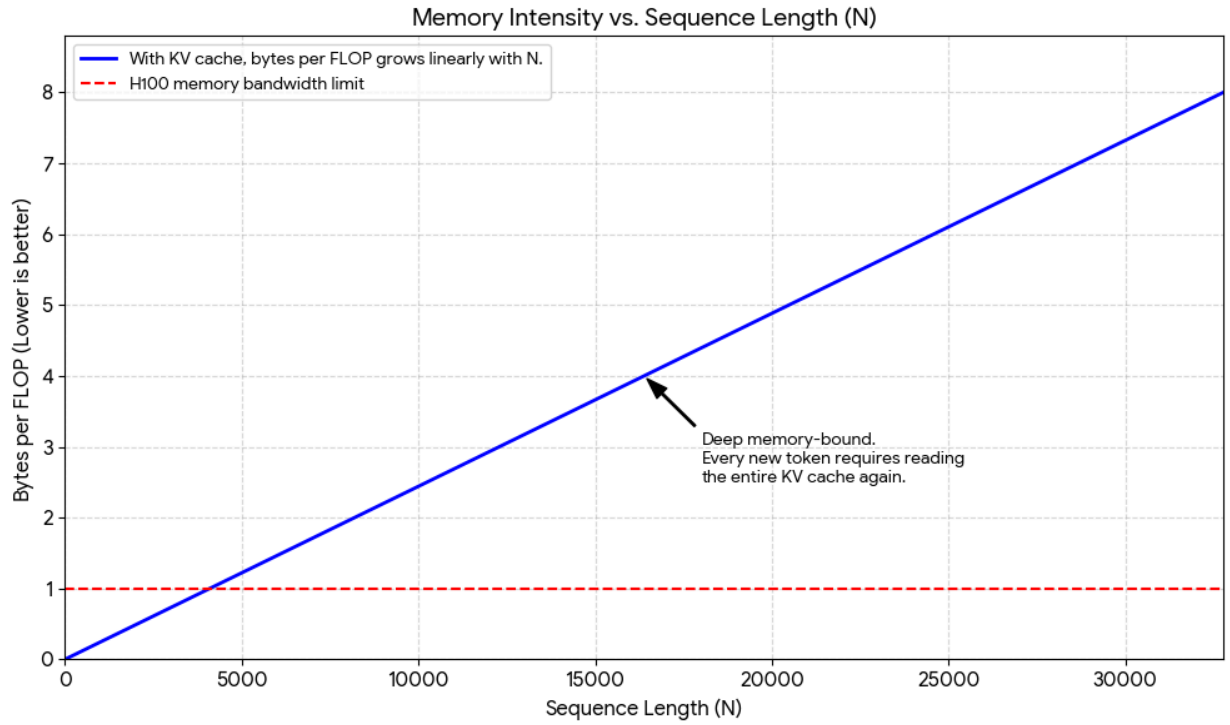
An annotation arrow points to the KV cache segment: ~$5GB/43GB$ per user = a handful of concurrent sessions!

Then **how do you even serve more users with less KV cache per user?**

Another issue w.r.t KV cache (also a dark side) is, since KV cache is stored in HBM, it needs to be brought to the compute area. And higher the **Sequence Length** (N), the

amount of bytes which needs to be brought back to the compute area just go on increasing.

Below image illustrates about this:



Understand the above with actual fact!

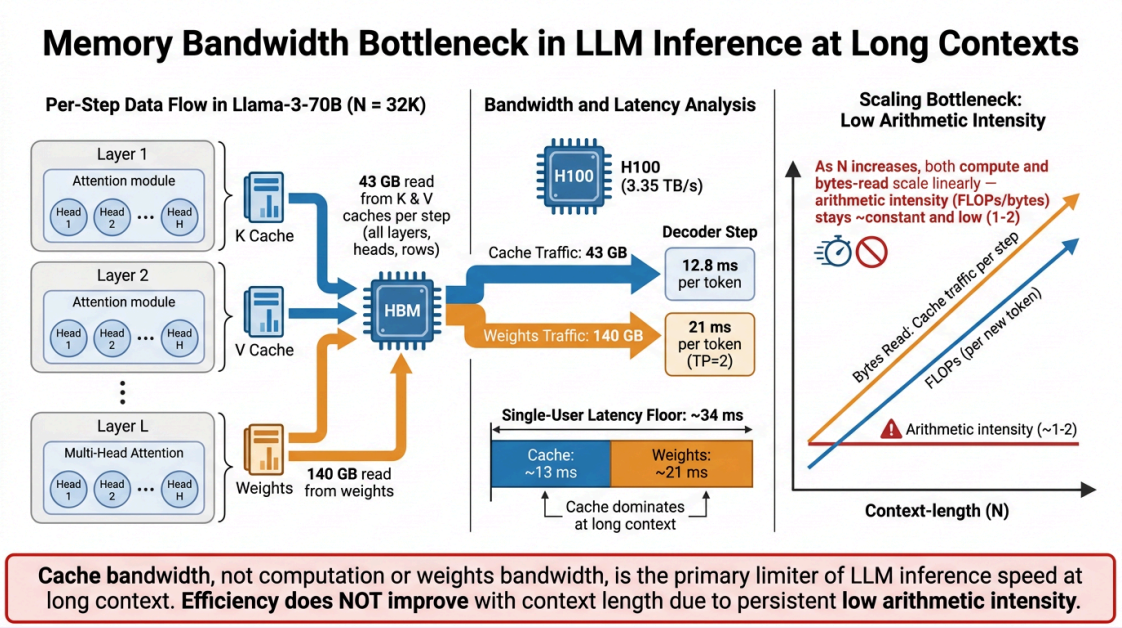
Per decode step, we read the entire K cache and the entire V cache from HBM, every row, every head, every layer. For Llama-3-70B at $N = 32K$, that is $43GB$ of cache traffic per decode step, on top of the $140GB$ of weights traffic.

At H100's $3.35TB/sec$, that is $43/3350 = 12.8ms$ of pure cache-reading time per token, plus $140/3350/2(TP) = 21ms$ of weight-reading time per token. The single-user ITL floor is $\sim 34ms$ before any optimization, and most of it is cache traffic, not weight traffic, at long context.

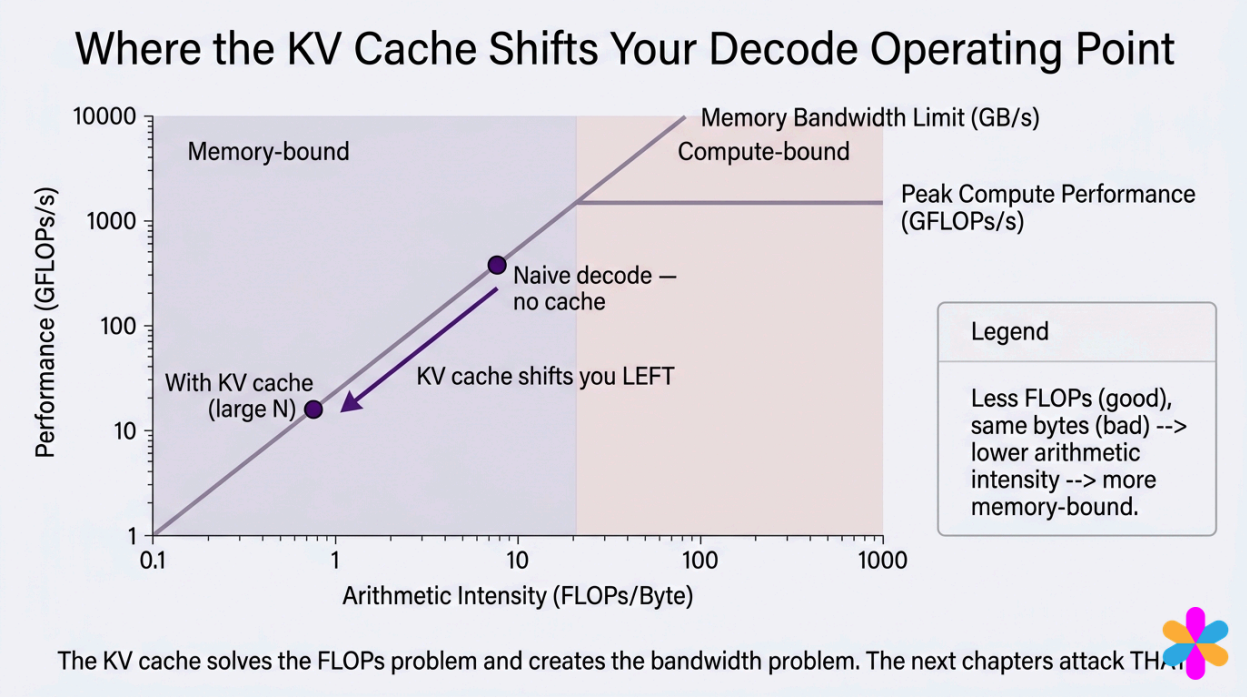
Worse: as N grows, bytes grow, but FLOPs per byte do not grow proportionally. The attention kernel's FLOPs scale as N (one new query dot product against N keys, times d), and the bytes-read scale as N (reading the K and V caches of length N).

Ratio: $FLOPs/bytes$ is roughly constant and small, arithmetic intensity stays around $1 - 2$, regardless of how long the context is.

(Visualize it here with a diagram)



Tying it back to the roofline plot, it actually shifts us to bottom side (left side) because our denominator in the Arithmetic Intensity, which is $bytes$ increases, thus decreasing Arithmetic Intensity and making it memory bound. This makes us extract less performance out of GPU.



Brainstorming the possible solutions

We now know the KV Cache formula: $\text{KV Cache Size} = l \times b \times n \times h \times s \times 2 \times 2bytes$

We can think of:

- **Quantization** (we are doing FP16 that takes 2 bytes so may be reducing the FP)
- Projection into latent space or compress → **Multi-Head Latent Attention (MLA)**
- Idea of temporal compression (compress past KV but keep recent ones)
- Reduce the number of heads → MQA or **GQA**
- Limit sequence length → **Sliding Window Attention**
- State Space Models → (**MAMBA**, **JAMBA**)

Live Demo Code Execution

Demo of: `L02_KV_Cache_Demo.ipynb`

TurboQuant

> **Fun fact:** *Paper published 1 month before this lecture*

It is an amazing technique and it directly relates to what we have learnt in this lecture. We will see how TurboQuant is implemented and see in practice.

To understand TurboQuant we need to understand two things: **Rotation** and **Quantization**.

We have Key and Value vectors. Simple way to think of reducing the size of KV cache is by **Quantization**. It will be explored in detail later but for now just think that it reduces your representation from `FP32` to `INT8` or `FP32` to `FP16` etc. It reduces the number of bytes each parameter takes in memory so that we get the reduction of the last "2" in the KV Cache Size formula.

But there is a catch, although we have **Layer Norm** before **Attention** Block and **Softmax** and **Layer Norm** after **Attention** Block, inside the **Attention** Block the KV vector's dimensions, let's assume in this case 768 dimensions, it is observed some value among those 768 dimensions will be a lot higher than the rest. And when we **Quantize** it, all the small ones becomes 0 and this one large value survives and thus appear spiked.

This is called the **Outlier Problem**.

Transformers are known to develop:

- heavy-tailed activations
- rare very large dimensions
- activation outliers

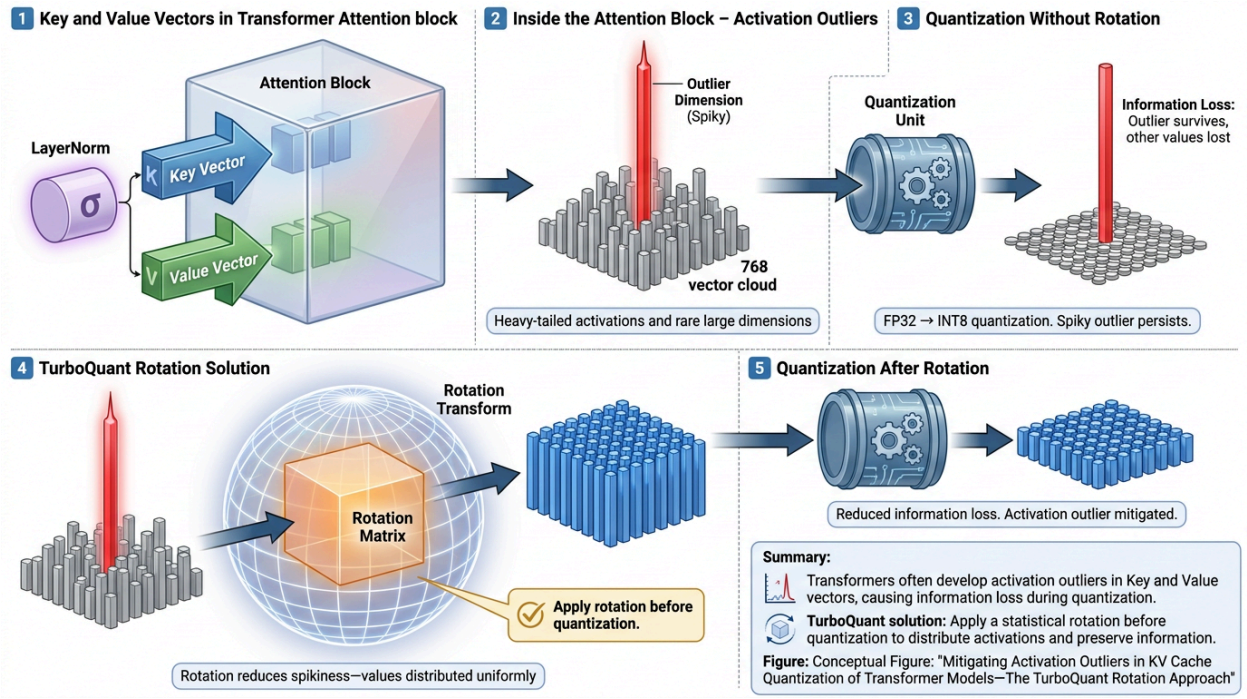
especially in:

- attention outputs
- residual streams
- KV activations

Research has repeatedly observed this. And when you **Quantize** this, information is lost. But we want to prevent this.

What Google Researchers who wrote TurboQuant said is that if you have a vector like that which is spiky, why don't you **Rotate** it before you **Quantize** it? What does the

rotation do? Typically when you apply any Rotation to a spiky vector, it almost always leads to uniformity in the values. The same spiky vector, when got multiplied with Rotation vector, gets rotation another point in the sphere, the spikiness suddenly gets reduced.



What they do is that first they convert that vector into a unit dimension so it lives on a sphere. That is the first thing that TurboQuant does. After this, what they do is rotate the vector on that sphere so that the spikiness of that vector reduces.

> If you want to think about it geometrically, the reason why this works is, in d dimensions, the area of the sphere is overwhelmingly concentrated in a thin equatorial band, not at the poles. The fraction of the sphere within "spiky distance" of any axis shrinks exponentially with d. After rotation, every component is roughly the same size: $\pm 1/\sqrt{d}$. For a 128-dim attention head, that's about ± 0.088 per component, remarkably uniform.
> Note: The Rotation matrix (R) is a learned parameter

That is all the TurboQuant does. It helps you to Quantize the Keys and Values without leading to a degradation in the performance. What they proved in the paper is that non-spiky vectors remain non-spiky.

Actually what they showed was:

They took **spiky** vector (V_1) → **Quantized** it (V_2) → **De-Quantized** it (V_3) → Calculated Loss, $L_1 = Loss(V_1, V_3)$ → **Saving KV cache memory**

They took **spiky** vector (V_1) → **Rotate** → **Quantized** it (V_2) → **De-Quantized** it (V_3') → Calculated Loss, $L_2 = Loss(V_1, V_3')$ → **Saving KV cache memory + retains accuracy**

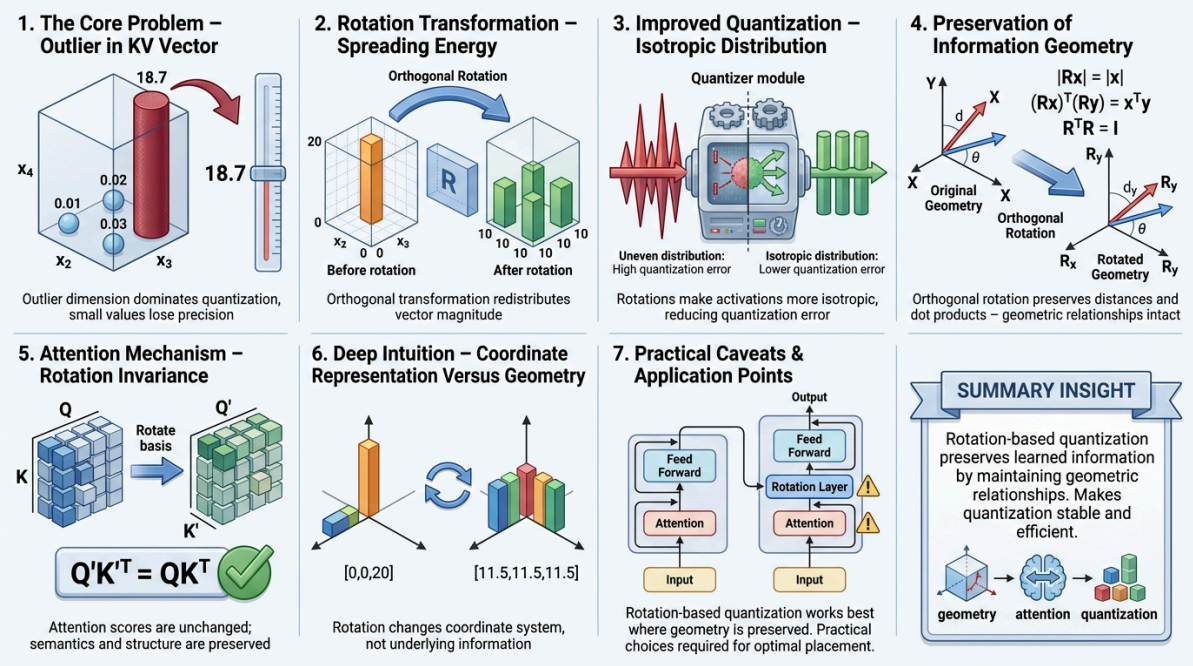
They found out that $L_2 <<< L_1$. Meaning, we did not lose the information we lost in the first case itself.

It seems like every time you need to solve something unknown, we either rotate it or throw a trainable weight matrix. Actually **Rotary Positional Encoding (RoPE)** did the same thing. Absolute **Positional Encodings** contaminate the token embeddings. So, how do we make sure the token embeddings are not contaminated? We rotate the Keys (K) and the Values (V). Here also they used **Rotation** trick.

Live Demo TurboQuant Code Execution

Demo of: `L02_KV_Cache_Demo_TurboQuant.ipynb`

▼ **QUESTION: In TurboQuant they introduced Rotation matrix. Doesn't Rotation also change the learned information?**



To summarize question:

> “If we rotate the representation, doesn’t that destroy the learned information?”

Fun fact: Rotation-based quantization are QuaRot/TurboQuant.

The beautiful part is:

```
certain rotations preserve information geometry
```

while making quantization MUCH easier.

This is the key insight.

The core problem again

Suppose a KV vector has:

```
[0.01, 0.02, 0.03, 18.7]
```

Huge outlier in one dimension.

Quantization scale becomes dominated by:

```
18.7
```

so small values lose precision.

What rotation tries to do

Instead of keeping energy concentrated in one dimension:

```
spread it across many dimensions
```

Example:

Before rotation:

```
[0, 0, 0, 20]
```

After rotation:

```
[10, 10, 10, 10]
```

(not exact mathematically, but intuition)

Now quantization range is much healthier.

No giant outlier dimension.

Why this helps quantization

Quantizers hate:

- highly uneven distributions
- large dynamic range
- sparse spikes

Rotations make activations:

```
more isotropic
```

meaning energy distributed more evenly.

This dramatically reduces quantization error.

Your important question:

> *"Doesn't rotation destroy learned information?"*

This is the beautiful linear algebra insight:

Certain rotations preserve distances and dot products.

These are called:

```
orthogonal transformations
```

If:

$R^T R = I$

then:

$|Rx| = |x|$

and:

$(Rx)^T(Ry) = x^T y$

$(Rx)^T(Ry) = x^T y$

This is the key.

Why attention still works

Attention fundamentally depends on: QK^T

If we rotate both consistently:

$$Q' = QR$$
$$K' = KR$$

then:

$$\begin{aligned} Q'K'^T &= (QR)(KR)^T \\ &= (QR)(R^T K^T) \\ &= Q(RR^T)K^T \end{aligned}$$

Since: $RR^T = I$, we get: QK^T unchanged.

This is the magic

The representation LOOKS different coordinate-wise, but relational structure remains the same.

So:

- semantics preserved
- attention preserved
- geometry preserved

while quantization becomes easier.

Deep intuition

A vector representation does NOT have inherent meaning tied to specific coordinates.

Example:

These can encode same information:

```
[0,0,20]
```

and

```
[11.5,11.5,11.5]
```

depending on basis choice.

Rotation changes:

```
coordinate system
```

not underlying information.

Why this is powerful

Transformers care more about:

- relative geometry
- similarities
- dot products

than exact coordinate values.

So if rotation preserves geometry:

the model still functions.

Why researchers discovered this

They realized:

```
outliers are often basis-dependent
```

Meaning:

a bad coordinate system can create spiky dimensions.

Rotate basis → smoother distribution.

This is a profound insight.

One subtle caveat

In practice:

- not every operation is perfectly rotation invariant
- nonlinearities complicate things
- residual paths complicate things

So implementations carefully choose where/how to rotate.

But overall the idea works surprisingly well.

Final intuition

Rotation does NOT destroy learned information because:

```
the information lives in geometric relationships, not individual coordinates
```

Orthogonal rotations preserve those relationships while redistributing magnitude across dimensions, making quantization much more stable.

That is the core idea behind rotation-based quantization methods.

▼ **QUESTION: If we quantize, rotate and de-quantize, doesn't this add latency during inference?**

At first glance, it seems contradictory:

```
quantize → rotate → dequantize
```

sounds like:

- extra computation
- extra memory movement
- extra latency

So why would inference become faster?

The answer is one of the most important principles in inference engineering:

> Modern LLM inference is usually memory-bandwidth bound, not compute bound.

The key bottleneck

For large LLMs, GPUs often spend more time:

```
moving tensors from memory
```

than actually doing arithmetic.

Especially for KV cache.

Example intuition

Suppose:

Without quantization:

- KV cache = 100 GB/s traffic

With quantization:

- KV cache = 25 GB/s traffic
- plus some extra rotation/dequant compute

Even if rotation adds compute, GPUs are extremely good at compute.

But memory bandwidth is precious.

So total latency can STILL improve.

Crucial modern GPU reality

GPUs today have:

- enormous FLOPS
- comparatively limited memory bandwidth

Example:

Resource	Modern GPU
Compute	insanely high
Memory bandwidth	bottleneck

So inference engineering often trades:

```
more math for less memory movement
```

This is counterintuitive but extremely common.

Why rotation overhead is acceptable

Rotation matrices are usually:

- structured
- lightweight
- fused into kernels
- highly parallelizable

Sometimes rotations use:

- Hadamard transforms
- orthogonal fast transforms
- blockwise operations

These are much cheaper than massive HBM traffic.

Important idea:

Bytes are expensive on GPUs:

```
moving data can cost more than computing on it
```

This is one of the deepest ideas in modern AI systems.

Why TurboQuant-like methods help

They improve:

- quantization quality
- cache compression ratio

without large accuracy loss.

So:

Before	After
large KV cache	much smaller KV cache
high bandwidth pressure	lower bandwidth pressure
worse GPU utilization	better utilization

Net result can still be faster.

Another subtle thing

De-quantization is often fused directly into attention kernels.

Instead of:

```
load → dequantize → store → use
```

they do:

```
load compressed → dequantize on-the-fly inside computation
```

So overhead becomes tiny.

Kernel fusion is critical.

Important systems insight

Inference optimization is often about:

```
minimizing memory traffic
```

NOT minimizing FLOPs.

This surprises many people initially.

Why this matters MORE for long contexts

As context grows:

KV cache dominates memory traffic.

Then quantization gains become enormous.

For long-context serving:

- bandwidth savings outweigh extra arithmetic

very quickly.

Deep roofline connection

This links back to your ridge-point question.

Attention/KV operations are often:

```
memory-bound
```

not compute-bound.

So adding a little compute is cheap if it reduces memory movement substantially.

Final intuition

TurboQuant-style methods work because:

```
compressed memory traffic savings are larger than added compute overhead
```

Modern GPUs have abundant compute capacity but limited memory bandwidth, so cleverly adding computation to reduce data movement can significantly improve inference speed.