Member-only story

# Multi-Head Latent Attention Is The Powerful Engine Behind DeepSeek

A deep dive Into DeepSeek's innovative Attention mechanism that makes its LLMs so good

Dr. Ashish Bamania · Following

Published in Level Up Coding · 18 min read · 13 hours ago

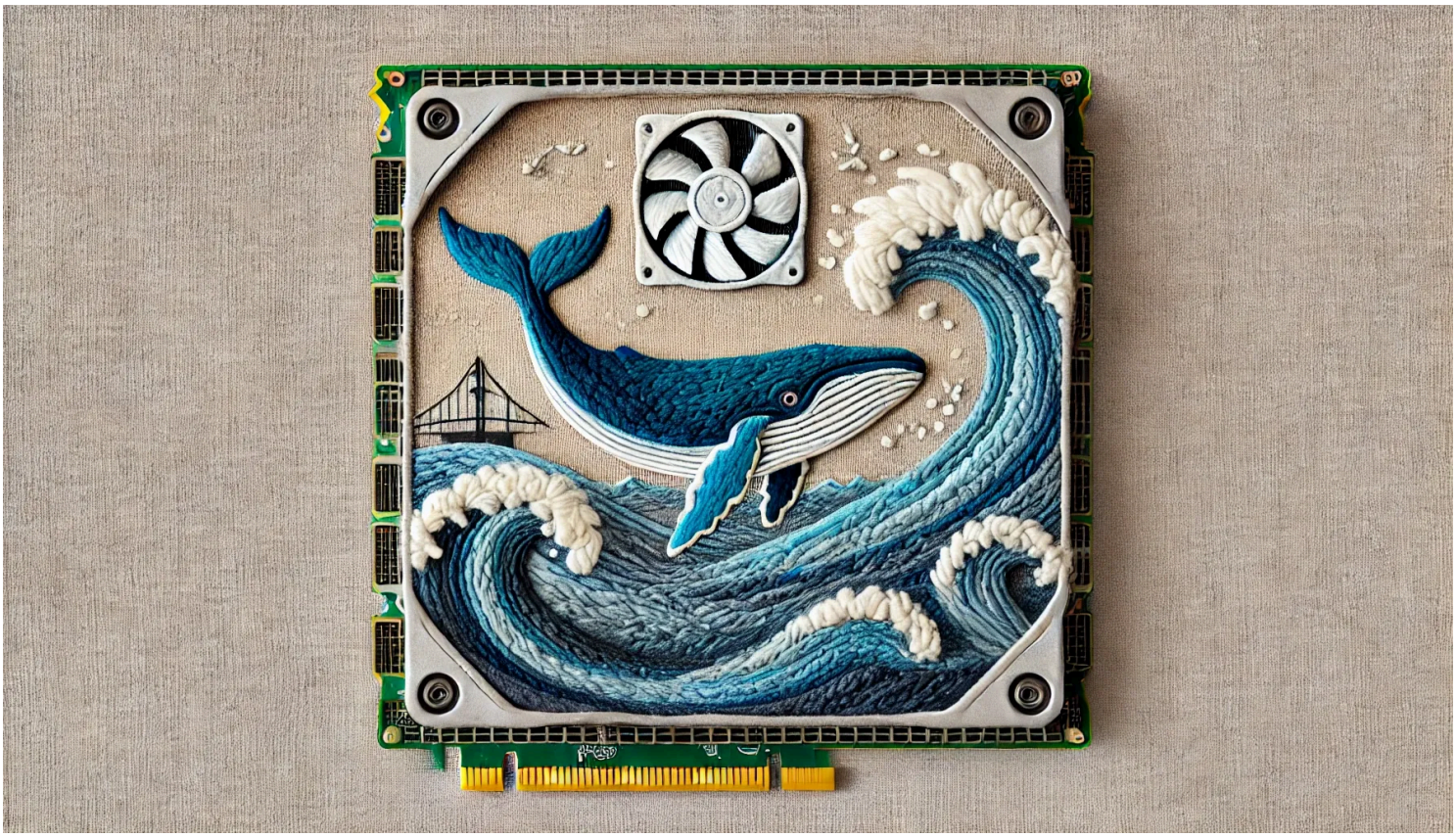


Image generated with DALL-E 3

DeepSeek is all over the news.

Its models have achieved state-of-the-art performance across multiple benchmarks (educational, factuality, math and reasoning, coding) and are competing head-to-head with OpenAI's o1.

Since its release, nearly $1 trillion of value has been lost in U.S. technology stocks in the S&P 500.

However, many popular beliefs about it are not correct.

A $6 million training cost for DeepSeek-R1 is being peddled all across the internet, but this is far from the truth (the official figures remain undisclosed).

This figure is actually for the official training of DeepSeek-V3 (the predecessor model for R1) and even then excludes the costs associated with prior research and ablation experiments on architectures, algorithms, or data associated with V3.

| Training Costs | Pre-Training | Context Extension | Post-Training | Total |
|---|---|---|---|---|
| in H800 GPU Hours | 2664K | 119K | 5K | 2788K |
| in USD | $5.328M | $0.238M | $0.01M | $5.576M |

Training costs for DeepSeek-V3 assuming that the rental price of NVIDIA's H800 GPU is $2/ GPU hour (Image from ArXiv research paper titled 'DeepSeek-V3 Technical Report')

Even though the costs are not accurately described, DeepSeek-V3 is still trained with significantly fewer resources than the models of the other prominent players in the LLM market.

*But how has this been made possible?*

Here's a story about a groundbreaking architectural change called **Multi-Head Latent Attention**, which is one reason DeepSeek's models perform so well at low training resources.

Let's begin!

## We Start Our Journey With 'Attention'

Attention is a mechanism that allows a model to focus on different parts of the input sequence when making predictions.

The mechanism weighs the importance of each token in the sequence and captures relationships between different tokens of the sequence regardless of their distance from each other.

This helps the model decide which tokens from the input sequence are most relevant to the token being processed.

Attention is not new.

It was popularly introduced for neural machine translation tasks using Recurrent Neural Networks (RNNs).

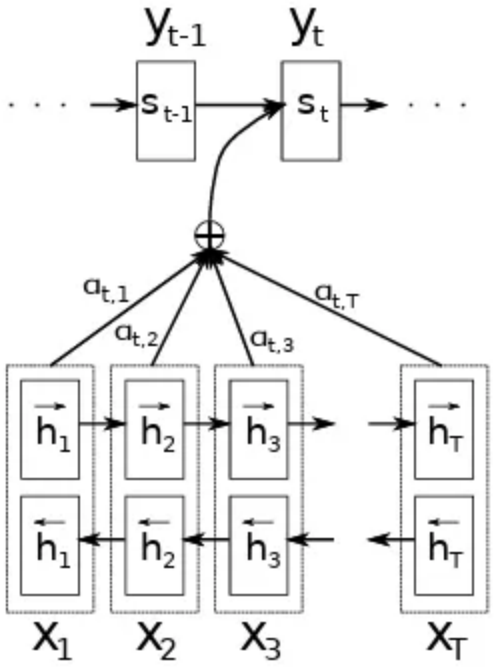
This version, called the **Bahdanau Attention mechanism**, uses a bi-directional RNN as an Encoder (*bottom blocks in the image*) to process input sequences `x(1) to x(T)` to generate hidden states `h(1) to h(T)`.

The Attention mechanism computes the attention scores for each encoder hidden state, telling how relevant these are to the current decoding step.

These scores are transformed into attention weights `a(t,1) to a(t,T)` (`T` is to the total number of input tokens) using the softmax function.

The encoder's hidden states are weighted according to these and summed to produce a **context vector** `c(t)`.

At each time step `t`, the decoder (*top blocks in the image*) then generates the next output `y(t)` based on its current hidden state `s(t)` combined with the context vector `c(t)`, previous hidden state `s(t-1)` and the previous output `y(t-1)`.




Bahdanau Attention Mechanism (Image from the research paper titled 'Neural Machine Translation by Jointly Learning to Align and Translate')

A particular type, called **Scaled Dot-Product Attention,** was later introduced in the Transformer architecture, which is calculated using the following three values obtained from the input token embeddings:

- **Query (Q):** a vector representing the current token that the model is processing.
- **Key (K):** a vector representing each token in the sequence.
- **Value (V):** a vector containing the information associated with each token.

These are used in the formula as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention where d(k) is the dimensionality of the key vector (Image obtained from the research paper titled 'Attention Is All You Need' )

Transformers use this in three ways:

1. **Self-Attention**

This is used in the Transformer's Encoder.

Here, the Queries, Keys, and Values come from the same input sequence — the previous layer's output in the Encoder.

**2. Masked Self-Attention**

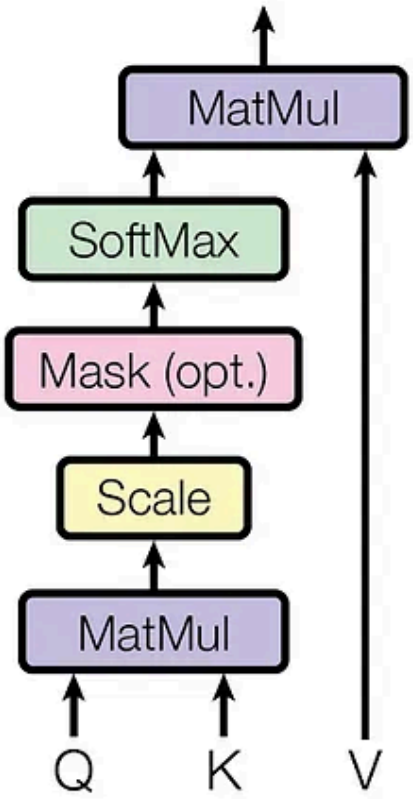
This is used in the Transformer's Decoder.

Here, the Queries, Keys, and Values come from the same sequence — the output sequence generated so far with the future tokens masked.

**3. Cross-Attention or Encoder-Decoder Attention**

This is used in the Transformer's Decoder.

Here, the Query comes from the Decoder's previous layer, while the keys and values come from the Encoder's output.




Scaled Dot-Product Attention where the ‘Scale’ operation represents multiplication with 1/ √d(k) and an optional ‘Mask’ used in the Decoder (Image obtained from the research paper titled ‘Attention Is All You Need’ )

## Moving Towards Multi-Head Attention

Instead of calculating the Attention score just once, the Transformer runs multiple Attention mechanisms in parallel using multiple heads.

Each head focuses on different aspects of the input sequence (short vs. long-range dependencies, grammatical rules, etc.).

This helps the architecture capture better semantic relationships between different tokens.

Let’s learn how this works.

Let's say that the Transformer architecture has an overall model dimension represented with `d(model)` . This represents the dimensionality of the input/ hidden layer representations `X` .

Instead of working with full-dimensional vectors, each head works with lower-dimensional projections of the Queries, Keys and Values.

These are obtained using learned projection matrices as follows:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V$$

The matrices `W(Q)(i)` , `W(K)(i)` , and `W(V)(i)` projects `X` into lower-dimensional Query, Key, and Value vectors for each attention head.

`d(model)` is reduced into smaller dimensions `d(k)` for Queries and Keys, and `d(v)` for Values as follows —

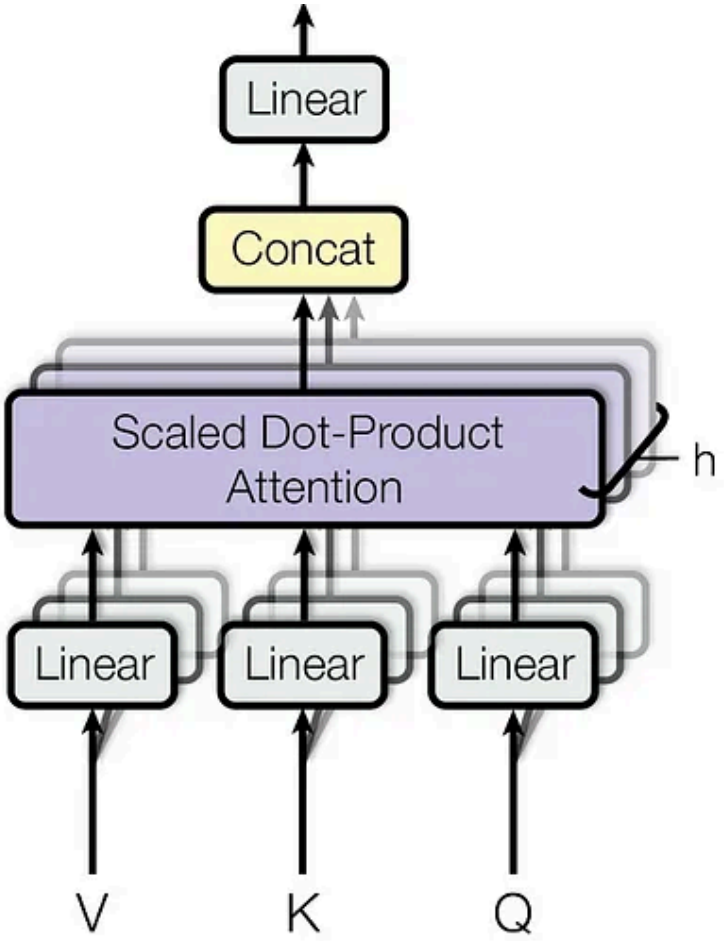
> `d(k) = d(v) = d(model) / h` where `h` is the number of heads

Next, Scaled dot-product Attention is calculated for each head using its own set of projected Query, Key, and Value matrices.

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

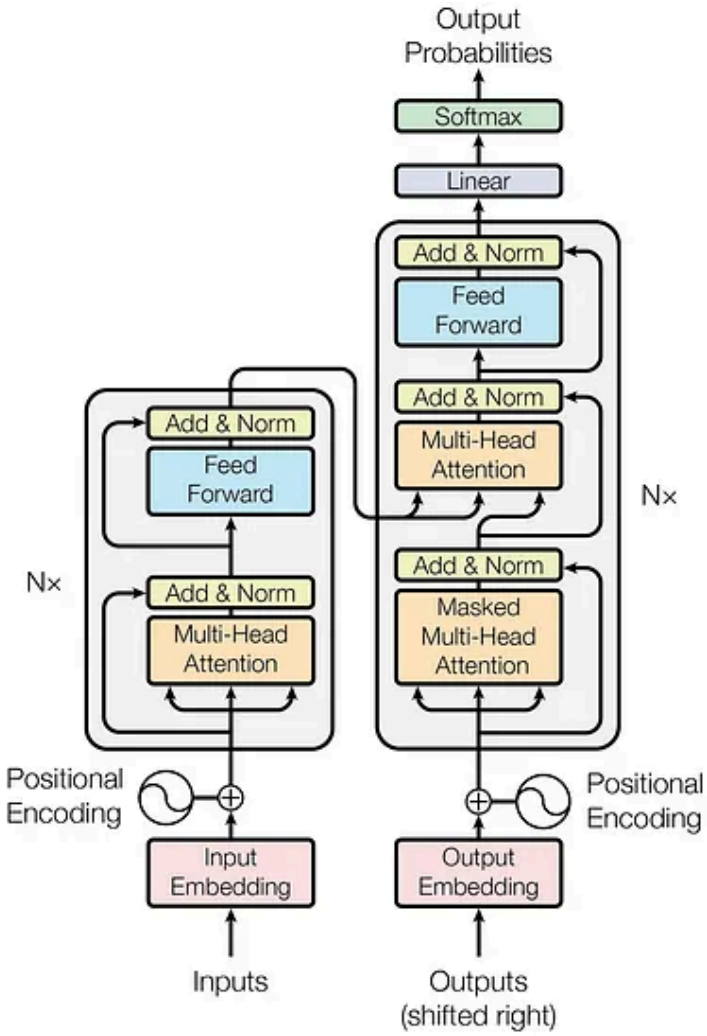
These individual Attention scores are concatenated and linearly transformed using a learned matrix `W(O)` as shown below.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$




Multi-head Attention (Image obtained from the research paper titled 'Attention Is All You Need' )

Going back to the Transformer architecture, it is Multi-head Attention (MHA) that is actually used instead of the basic Attention mechanism (*described earlier*) for both Self-Attention & Cross-Attention.




The Transfromer Architecture (Image obtained from the research paper titled 'Attention Is All You Need' )

## But MHA Is Memory Expensive At Inference

Multi-head Attention (MHA) is a powerful mechanism used by most LLMs today to capture dependencies between tokens, but it causes an issue during inference/ token prediction.

When the LLM generates a token, it must compute the attention scores with all the previous tokens.

Instead of recomputing all keys and values for previous tokens at every time step, they are stored in a **Key-Value (KV) Cache**.

(*Queries are not cached since they are dynamically calculated for each new token, and only Keys and Values need to be reused for future tokens.*)

This is a fantastic optimisation step to speed up inference, but as the sequence length increases, the number of stored key-value pairs grows linearly with it.

**For a transformer with `L` layers, `n(h)` heads per layer, and the per-head dimension of `d(h)`, `2 x n(h) x d(h) x L` elements need to be cached for each token.**

This is because, for each token, each head independently stores a separate key and value vector of size `d(h)`, and there are `n(h)` heads per layer and `L` layers in total.

This cache can become massive over time, especially in long-context models, leading to massive GPU memory usage during cache retrieval and slower inference.

Let's learn how this memory bandwidth bottleneck is overcome in modern LLMs.

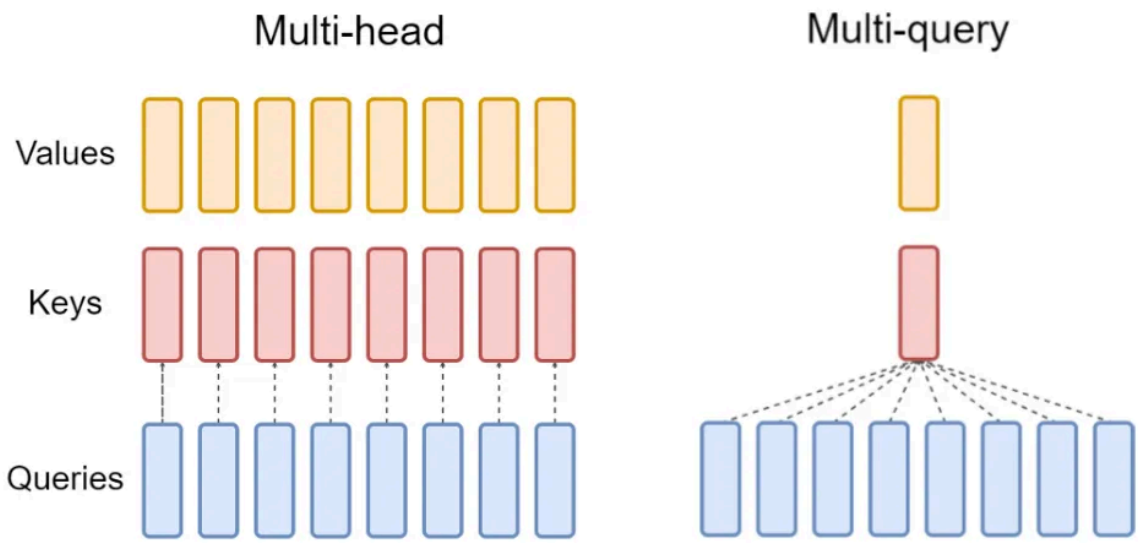
## Levelling Up To Multi-Query Attention (MQA)

Unlike traditional Multi-head Attention (MHA), which caches separate key-value pairs per head, Multi-Query Attention (MQA) shares a single set of keys and values across all heads.

$$Q_i = XW_i^Q, \quad K_{\text{shared}} = XW^K, \quad V_{\text{shared}} = XW^V$$

**This reduces the KV cache size from `2 x n(h) x d(h) x L` (as in MHA) to `2 x d(h) x L`.**

Since only one set of keys and values needs to be fetched, this reduces GPU memory usage, allowing large batch sizes to be processed at inference time.




Multi-head Attention vs Multi-query Attention (Image obtained from the ArXiv research paper titled 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints')

LLMs like PaLM and Falcon use MQA instead of MHA.

But MQA isn't perfect.

Since all heads share the same Keys and Values, this reduces effective learned representations, making the LLM less expressive and struggling to track long-range dependencies.

It is also seen that using MQA leads to instability during LLM fine-tuning, particularly with long input tasks.

*What's the fix?*

## Grouped-Query Attention (GQA) To The Rescue

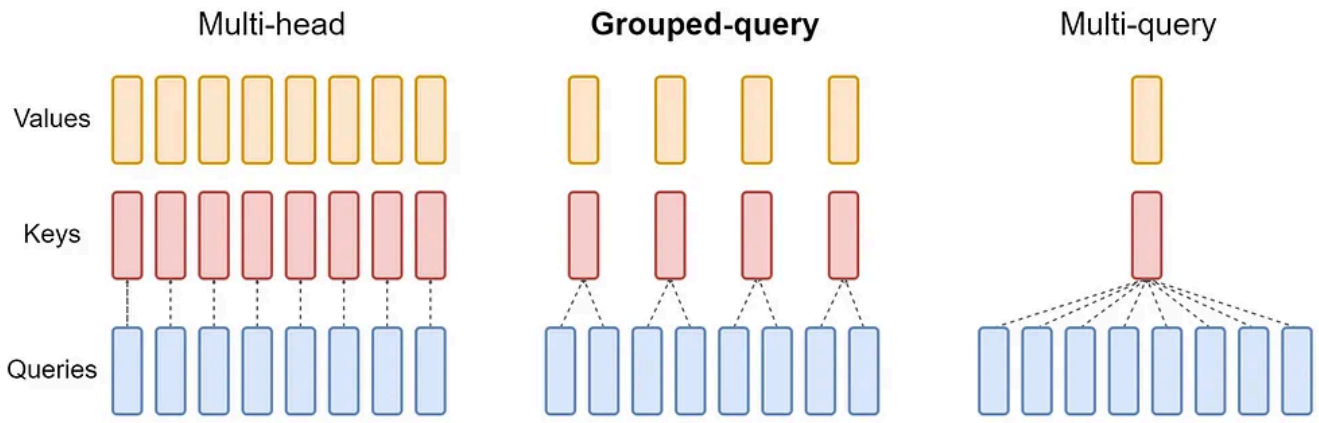
Published by a team of Google researchers, Grouped-Query Attention (GQA) is a tradeoff between MHA and MQA.

Instead of one KV pair per head (like in MHA) or one KV pair for all heads (like in MQA), GQA groups multiple heads together that share a single KV pair.

Each group processes its own set of queries but shares the same keys and values.

**This reduces the KV cache size from `2 x n(h) x d(h) x L` (as in MHA) to `2 x d(h) x G`, where `G` is the number of groups.**

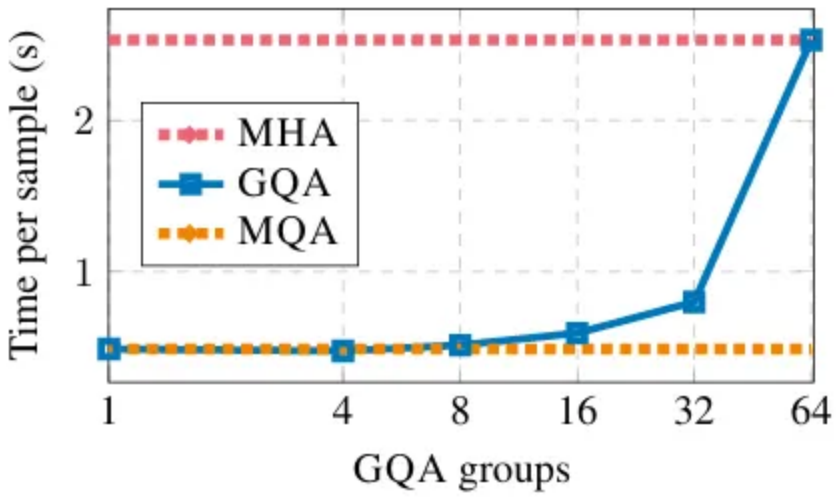
This makes the inference much faster and, at the same time, allows the LLM to learn its representations effectively.




Visual representation of the different Attention mechanisms: MHA, GQA, MQA (Image obtained from the ArXiv research paper titled 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints')

To summarise —

> *MHA's generation quality is the best, but its inference speed is the lowest.*
>
> *MQA leads to the fastest inference speed but the lowest generation quality.*
>
> *GQA balances and interpolates between these two.*




For G = 1, GQA works similarly to MQA, while for G = n(h), it works similarly to MHA. The best performance is observed at G = 4 to 8, where G is the number of groups and n(h) is the number of heads, respectively. (Image

obtained from the ArXiv research paper titled 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints')

Popular LLMs that use GQA in their architecture include Llama 3 series of models, Mistral 7B and DeepSeek LLM (the first version of DeepSeek models).

DeepSeek's further versions push this performance even more.

Let's learn how.

## Here Comes Multi-Head Latent Attention (MLA)

MLA, or Multi-Head Latent Attention, is built to further reduce the size of the KV cache while still achieving the best performance compared to the previously discussed attention mechanisms (including MHA).

**It does this by compressing the KV cache into a lower-dimensional latent space, reducing its size by 93.3%!**

Let's break it down step by step.

### Low-rank Key-Value Joint Compression

Firstly, instead of computing and storing the Keys and Values for each token, these are compressed into a latent vector `C(KV)` using a down-projection matrix `W(DKV)`.

$$C_{KV} = XW_{D_{KV}}$$

The KV pairs can be reconstructed from this latent vector during inference using per-head up-projection matrices `W(UK)` (for Keys) and `W(UV)` (for Values).

$$K_i = C_{KV} W^i_{U_K}, \quad V_i = C_{KV} W^i_{U_V}$$

To reduce the computational cost:

- Instead of explicitly calculating the Keys, the matrix `W(UK)` is merged into `W(Q)`

This is how we were calculating Query and Keys from the latent vector before —

$$Q_i = X W^Q_i, \quad K_i = C_{KV} W^i_{U_K} = X W_{D_{KV}} W^i_{U_K}$$

But instead, we redefine the query projection as:

$$W'_Q = W_Q W_{U_K}$$

And then compute the Query as:

$$Q'_i = X W'_Q$$

This eliminates the need to explicitly compute `K(i)` since `Q'(i)` already includes this information.

- Instead of explicitly calculating the Values, the matrix `W(UV)` is merged into `W(O)`

This is how we calculated the attention for each head and concatenated them:

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) C_{KV} W_{U_V}^i$$

$$U = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O$$

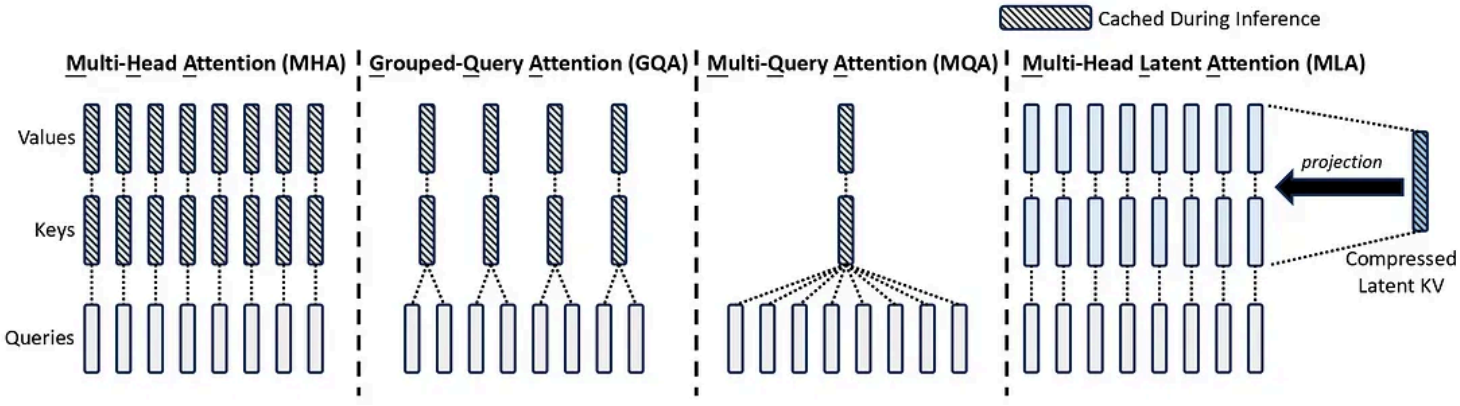
Instead, the output transformation matrix `W(O)` is merged with `W(UV)` as follows:

$$W_O' = W_O W_{U_V}$$

And the final attention output is calculated as:

$$U' = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W_O'$$

This eliminates the need to explicitly compute `V(i)`.




Comparison between different Attention mechanisms: MHA vs. GQA vs. MQA vs. MLA

### Low-rank Compression For Queries

Similar compression is done for Queries as well.

These are compressed into a latent (compressed) representation `C(Q)` using a down-projection matrix `W(DQ)`.

$$C_Q = XW_{D_Q}$$

These are reconstructed when needed using an up-projection matrix `W(UQ)`.

$$Q_i = C_Q W^i_{U_Q}$$

This does not reduce the KV cache size but decreases activation memory usage **during training.**

(Activation memory is the memory used to store intermediate activations during forward propagation in training. These activations are needed for

backpropagation to compute gradients.)

During training with MHA, each layer computes and stores Queries explicitly in memory, and this number grows linearly with the number of layers.

Instead, only the compressed representation of the Queries is stored in MLA. This reduces the total activations stored for backpropagation.

To clarify, activations for backpropagation are not stored during inference when the Queries are computed once per token and discarded.

Therefore, compression for Queries only improves training efficiency but keeps the inference performance unchanged.

Researchers try using RoPE to include token position information when using MLA.

However, this is not possible because of how the previous calculations are performed.

To understand why, we must first learn how position encoding works in LLMs.

## Positional Encoding — What's That?

Transformers process tokens in parallel rather than sequentially. This is what gives them the computational advantage over RNNs.

However, this also makes Transformers position-agnostic, meaning they do not have a sense of the order of the tokens they process.

Consider two sentences:

> *"The cat sits on the mat."*
>
> *"The mat sites on the cat."*

To a Transfromer, both of them are the same.

This isn't good for language processing; therefore, positional information in the form of positional embeddings (vector) is added with token embeddings before Transformers process them.

Positional Embeddings are of two main types:

1. **Absolute Positional Embeddings** — where each token is assigned a unique encoding according to its position
2. **Relative Positional Embeddings** — where information about how far apart tokens are from each other is encoded rather than their absolute positions

Each of them can be either:

- Fixed (calculated using a mathematical function)

- Learned (has trainable parameters that are updated with backpropagation during model training)

While relative positional embeddings are popularly used by the T5 Transformer, in the original Transformers paper, the authors use fixed absolute positional embeddings, as shown below.

(Learned positional embeddings are not used in this paper because the fixed embeddings produced nearly identical results.)

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Calculations for Positional Embeddings where 'pos' is the token index, 'i' is the index of the token embedding dimension, and 'd' is the total token embedding dimension

For each token position, Positional embeddings (PE) are calculated using alternating sine and cosine functions on even and odd dimensions, respectively.

The denominator `10000 ^ (2i/ d(model))` controls the wavelength of these functions.

This means that for lower dimensions (smaller `i`), the frequency is high, and the wavelengths are short.

Similarly, for the higher dimensions (larger `i`), the frequency is low, and the wavelengths are long.

These wavelengths form a Geometric progression, with the smallest and the largest values being $2\pi$ and $10000 \times 2\pi$, respectively.

This makes the model capture the short-term and long-term dependencies in the input sequence using the high and low frequencies, respectively.

Since the dimensions of these Positional Embeddings are the same as the token embeddings, they can be directly summed as follows before being passed to the Transformer to process.

$$X' = X + PE$$

X' is final input embedding passed into the Transformer, X is the original input embedding, and PE is the set of positional embeddings for each token in the input sequence

These embeddings can also capture the relative relationships between tokens, as their positional embeddings are related linearly.

### Improving Learning Further With RoPE

Previous embedding approaches can struggle to capture dependencies in sequences longer than those seen during training.

These approaches also *add* positional embeddings to the token embeddings, which increases the total parameters and complexity, leading to increased computational training costs and slower inference.

A 2023 research introduced a new way of directly encoding both absolute and relative positions in the attention mechanism.

Their method is called **Rotary Position Embedding** or **RoPE**.

Instead of adding position embeddings, as in the previous approaches, RoPE rotates the token embeddings according to their positions.

For a token embedding `x(m)` of dimension `d` at position `m`, it is transformed into Query (`q(m)`) and Key (`k(n)`) vectors as follows using weight matrices `W(q)` and `W(k)`, respectively.

$$q_m = W_q x_m, \quad k_n = W_k x_n$$

RoPE rotates these vectors before they are used in the self-attention calculation. This is done using a position-dependent rotation matrix `R(m)`.

$$q'_m = R_m q_m, \quad k'_m = R_m k_m$$

`R(m)` acts independently on each **pair** of dimensions in `q` and `k`.

For a case where these vectors are two-dimensional (`d = 2`), the rotation matrix `R(m)` is defined as:

$$R_m = \begin{bmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{bmatrix}$$

This 2D rotation matrix rotates these vectors counter-clockwise, proportional to their position `m` by an angle of `mθ` where `θ` is a constant (`1` in the case of `d = 2`).

For a two-dimension query vector `q(m) = (q1, q2)`, this transformation will result in:

$$\begin{bmatrix} q_1' \\ q_2' \end{bmatrix} = \begin{bmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

$$q_1' = q_1 \cos(m\theta) - q_2 \sin(m\theta)$$
$$q_2' = q_1 \sin(m\theta) + q_2 \cos(m\theta)$$

But query and key vectors are usually higher-dimensional (assumed an even number of dimensions), and to work with them, they are paired and separate 2D rotations are applied to each adjacent pair of dimensions.

For example, for a 4-dimension query vector `q(m) = (q1, q2, q3, q4)`, two independent rotations are applied for `(q1, q2)` and `(q3, q4)` as shown below.

$$(q_1', q_2') = R_m^{\theta_1}(q_1, q_2), \quad (q_3', q_4') = R_m^{\theta_2}(q_3, q_4)$$

These can be expanded to:

$$\begin{bmatrix} q_1' \\ q_2' \end{bmatrix} = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) \\ \sin(m\theta_1) & \cos(m\theta_1) \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

$$\begin{bmatrix} q_3' \\ q_4' \end{bmatrix} = \begin{bmatrix} \cos(m\theta_2) & -\sin(m\theta_2) \\ \sin(m\theta_2) & \cos(m\theta_2) \end{bmatrix} \begin{bmatrix} q_3 \\ q_4 \end{bmatrix}$$

$$\begin{aligned} q_1' &= q_1 \cos(m\theta_1) - q_2 \sin(m\theta_1) \\ q_2' &= q_1 \sin(m\theta_1) + q_2 \cos(m\theta_1) \\ q_3' &= q_3 \cos(m\theta_2) - q_4 \sin(m\theta_2) \\ q_4' &= q_3 \sin(m\theta_2) + q_4 \cos(m\theta_2) \end{aligned}$$

The angle of rotation `θ(i)` for each pair of dimensions is different and is calculated as follows:

$$\theta_i = 10000^{-2(i-1)/d}, \quad i = 1, 2, \ldots, d/2$$

This ensures that the low-frequency rotations for higher dimensions capture long-range dependencies and the high-frequency rotations for lower dimensions capture short-term dependencies.

This `θ` is similar to what we learned when describing the sinusoidal embeddings from the original Transformers paper.

The overall rotation matrix for higher dimensions looks as follows.

$$R_m = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \dots \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \dots \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \dots \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

One can avoid explicit matrix multiplication to make the process of computing the rotated queries and keys computationally efficient.

This can be done by precomputing `cos(mθ(i))` and `sin(mθ(i))` for each position `m`, and then performing simple element-wise multiplications and additions between the terms.

For example, this is how we calculated the rotated query `q'(m)` previously:

$$q'_m = R_m q_m$$

$$\begin{bmatrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ \vdots \\ q'_{d-1} \\ q'_d \end{bmatrix} = \begin{bmatrix} \cos(m\theta_1)q_1 - \sin(m\theta_1)q_2 \\ \sin(m\theta_1)q_1 + \cos(m\theta_1)q_2 \\ \cos(m\theta_2)q_3 - \sin(m\theta_2)q_4 \\ \sin(m\theta_2)q_3 + \cos(m\theta_2)q_4 \\ \vdots \\ \cos(m\theta_{d/2})q_{d-1} - \sin(m\theta_{d/2})q_d \\ \sin(m\theta_{d/2})q_{d-1} + \cos(m\theta_{d/2})q_d \end{bmatrix}$$

This can be changed to:

$$R_m q_m = q_m \circ \cos(m\theta) + S(q_m) \circ \sin(m\theta)$$

Efficient calculation for R(m)q(m) where ◦ represents element-wise multiplication and S(q(m)) operation swaps adjacent elements in q(m), negating the even-indexed one

The complete calculation is shown below:

$$R_m q_m = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ \vdots \\ q_{d-1} \\ q_d \end{pmatrix} \circ \begin{pmatrix} \cos(m\theta_1) \\ \cos(m\theta_1) \\ \cos(m\theta_2) \\ \cos(m\theta_2) \\ \vdots \\ \cos(m\theta_{d/2}) \\ \cos(m\theta_{d/2}) \end{pmatrix} + \begin{pmatrix} -q_2 \\ q_1 \\ -q_4 \\ q_3 \\ \vdots \\ -q_d \\ q_{d-1} \end{pmatrix} \circ \begin{pmatrix} \sin(m\theta_1) \\ \sin(m\theta_1) \\ \sin(m\theta_2) \\ \sin(m\theta_2) \\ \vdots \\ \sin(m\theta_{d/2}) \\ \sin(m\theta_{d/2}) \end{pmatrix}$$

Next, the self-attention score is calculated as the dot product between the rotated queries and keys as follows:

$$q_m'^{\top} k_n'$$

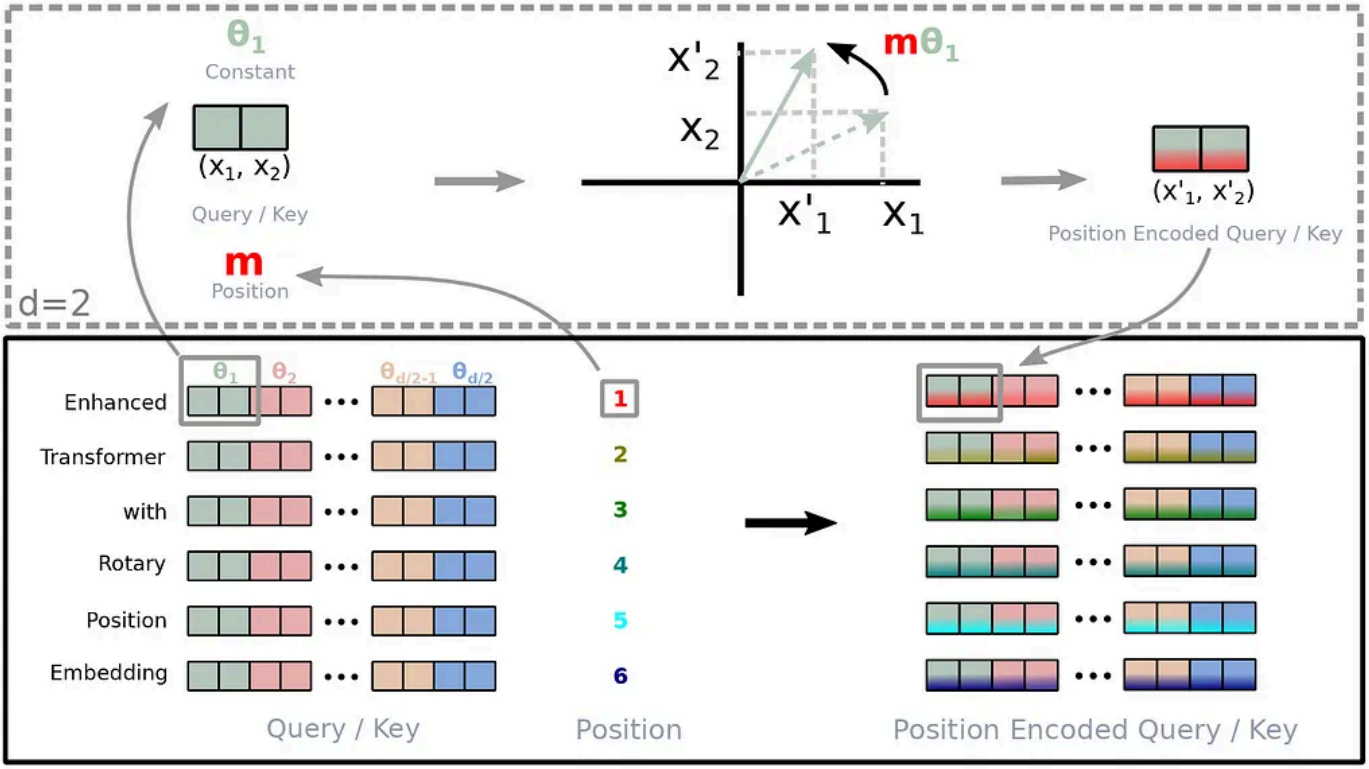
Since the rotation matrices can be written as follows,

$$R_m^{\top} R_n = R_{n-m}$$

The equation becomes:

$$q_m'^{\top} k_n' = q_m^{\top} R_{n-m} k_n$$

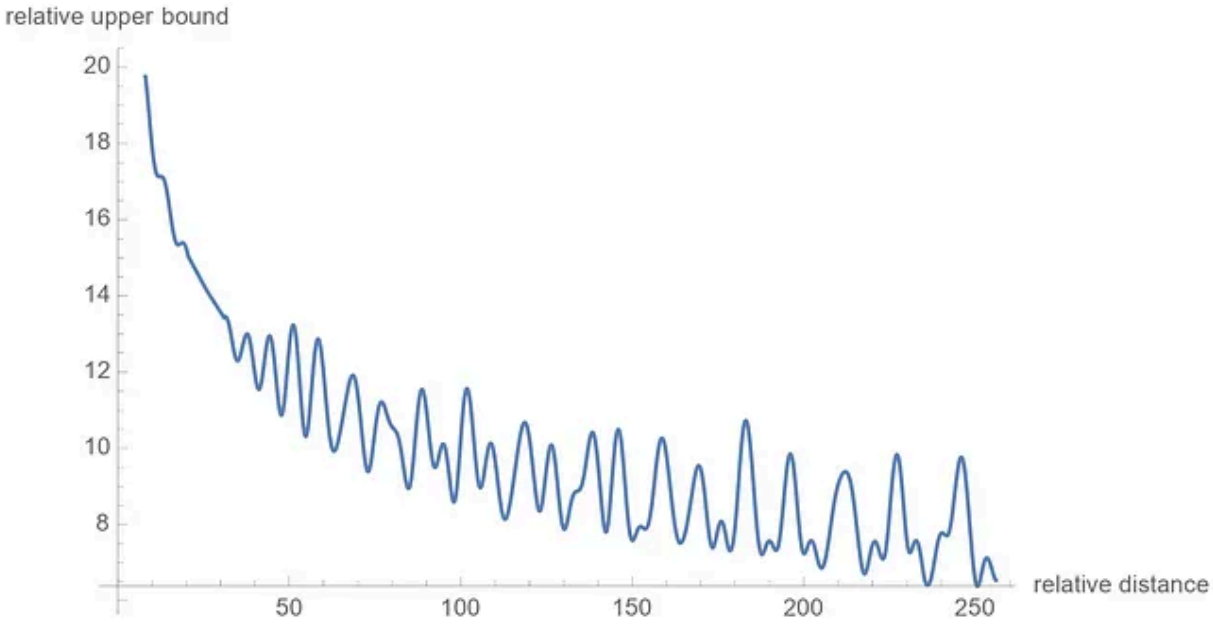
Thanks to RoPE, the attention score now encodes the relative position of tokens `(n - m)` without needing additional relative embeddings.




Visual representation of RoPE embeddings (Image from the ArXiv research paper titled 'RoFormer: Enhanced Transformer with Rotary Position Embedding')

There's another cool property of these embeddings: the relative importance of the connection between far-apart tokens is lower than closer ones.

*The mathematical explanation of this can be found in the research paper, and curious readers are encouraged to check it out.*




Graph showing the decay of attention scores in RoPE as the relative distance between tokens increases. The further two tokens are apart, the weaker their maximum possible attention score becomes. (Image from the ArXiv research paper titled 'RoFormer: Enhanced Transformer with Rotary Position Embedding')

Now that we know how RoPE works, we need to figure out a way to apply it to MLA.

## Why Is RoPE Incompatible With MLA?

Returning to MLA, we first created a latent compressed representation of the keys and values `C(KV)` to reduce memory usage and improve inference efficiency.

$$C_{KV} = XW_{D_{KV}}$$

We know that RoPE involves rotating queries and keys according to positional information using the rotation matrix `R(m)` before calculating the

attention scores:

$$k'_m = R_m k_m$$

However, since MLA stores the compressed key-value cache rather than full keys, applying RoPE will mean that all previous keys must be recomputed every time a new token is generated.

This breaks the efficiency achieved by using the compressed KV representation in the first place.

Then there's another issue.

Remember how, instead of recalculating the keys during inference, the key up-projection matrix `W(UK)` was merged into `W(Q)` for optimization?

If we apply RoPE after key reconstruction as follows:

$$k'_m = R_m(C_{KV} W^i_{U_K}) = R_m(X W_{D_{KV}} W^i_{U_K})$$

This prevents the merging because matrix multiplication is **not commutative**.

$$R_m(X W_{D_{KV}} W^i_{U_K}) \neq (R_m X W_{D_{KV}}) W^i_{U_K}$$

Thus, `W(UK)` cannot be decoupled and merged with `W(Q)` as done originally.

## How Is RoPE Used In MLA Then?

In MLA, a new approach to applying RoPE called **Decoupled Rotary Position Embedding** is introduced.

Firstly, two types of keys are calculated:

1. Latent keys `K(C)` that are compressed keys calculated like we previously discussed.

$$K_C = C_{KV} W^i_{U_K} = X W_{D_{KV}} W^i_{U_K}$$

2. Position-sensitive or Decoupled keys `K(R)` that are non-compressed keys that store the position information required to apply RoPE as follows:

$$K_R = R_m(X W^i_{K_R})$$

Both of these are concatenated and shown below.

$$K = [K_C; K_R] = [X W_{D_{KV}} W^i_{U_K}; R_m(X W^i_{K_R})]$$

The same operations are performed for queries as well.

This results:

1. Latent queries `Q(C)`

$$Q_C = XW_{D_Q}W^i_{U_Q}$$

2. Position-sensitive or Decoupled queries `Q(R)` for RoPE

$$Q_R = R_m(XW^i_{Q_R})$$

Both of these are concatenated and are shown below.

$$Q = [Q_C; Q_R]$$

These are **calculated at inference and not stored.**

The approach gives us the best of both worlds, where we still benefit from low-rank KV compression while `K(R)` can be stored separately and allows the application of position-sensitive transformations without breaking the attention calculations with RoPE.

**In this approach, we cache `K(R)` and `C(KV)` and a total of `[d(c) + d(h)(R)] x L` elements per token need to be cached where `d(c)` is the latent key dimension, `d(h)(R)` is the per-head dimension of decoupled keys and `L` is the number of layers in MLA.**

This is much more efficient than for a transformer with `L` layers, `n(h)` heads per layer, and the per-head dimension of `d(h)`, where `2 x n(h) x d(h) x`

`L` elements need to be cached for each token.

Finally, the attention score is calculated using both compressed and position-sensitive queries and keys:

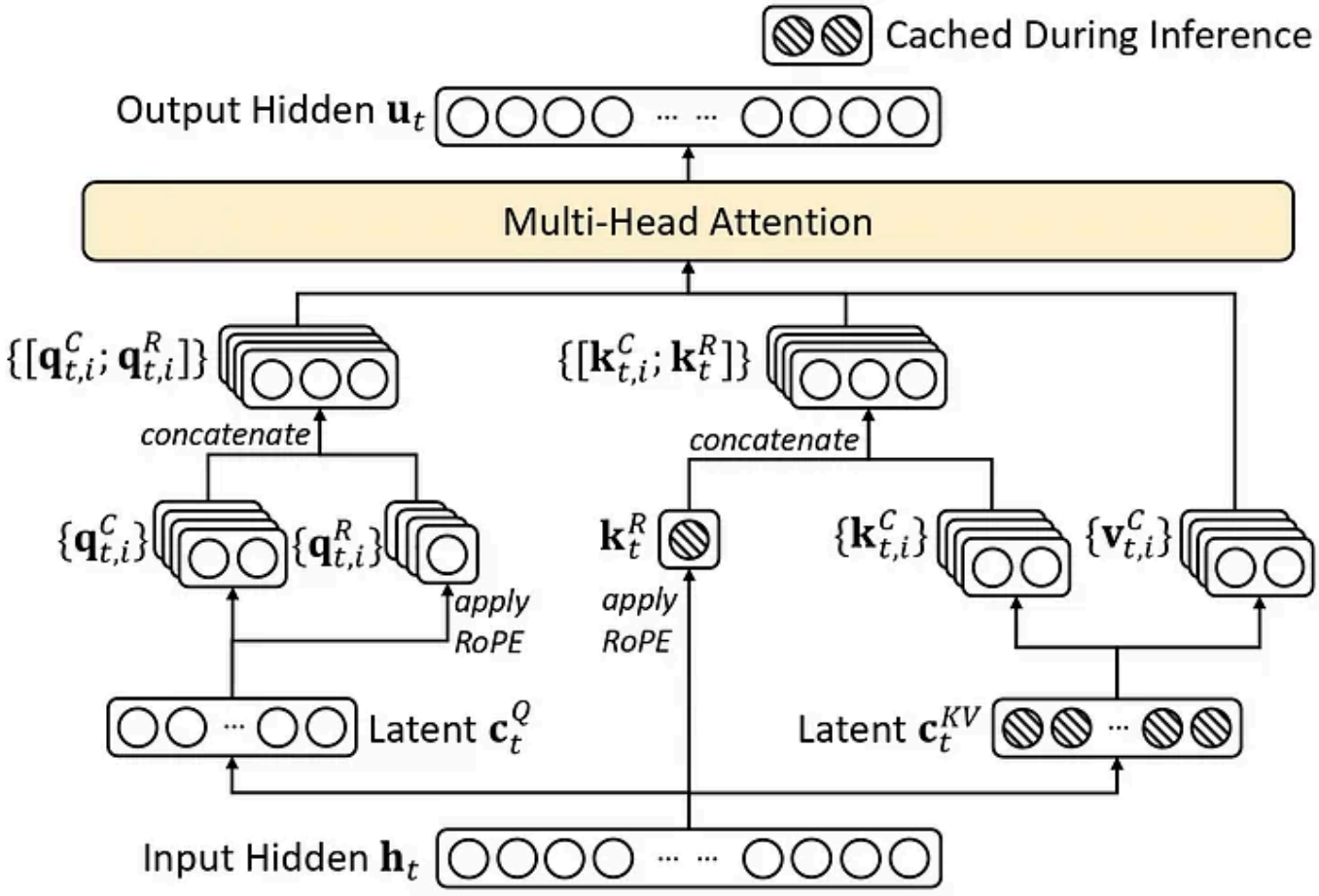
$$A_{m,n} = \frac{[Q_C; Q_R]^T [K_C; K_R]}{\sqrt{d_h + d_h^R}}$$

The final output is calculated as follows (where `V(C)` is retrieved dynamically from the key-value cache `C(KV)`):

$$O_t = \sum_{j=1}^{t} \text{Softmax}(A_{m,j}) V_C$$

***You must note that although these explicit calculations are shown to clarify the process, keys and values are never explicitly calculated, as we previously discussed.***

The complete MLA architecture is shown below.




Multi-head Latent Attention architecture (Terminology in the image might differ from what is used in the text)

## How Good Is MLA?

MLA stores a small number of elements in its KV cache, which is equal to GQA with only 2.25 groups but can achieve stronger performance than MHA.

| Attention Mechanism | KV Cache per Token (# Element) | Capability |
|---|---|---|
| Multi-Head Attention (MHA) | $2n_h d_h l$ | Strong |
| Grouped-Query Attention (GQA) | $2n_g d_h l$ | Moderate |
| Multi-Query Attention (MQA) | $2d_h l$ | Weak |
| MLA (Ours) | $(d_c + d^R_h)l \approx \frac{9}{2} d_h l$ | Stronger |

Comparison of elements in KV cache stored per token amongst different Attention mechanisms

When a dense model with 7B total parameters is trained with different attention mechanisms, **MHA performs significantly better** than both **GQA and MQA** on the tested benchmarks.

| Benchmark (Metric) | # Shots | Dense 7B w/ MQA | Dense 7B w/ GQA (8 Groups) | Dense 7B w/ MHA |
|---|---|---|---|---|
| # Params | - | 7.1B | 6.9B | 6.9B |
| BBH (EM) | 3-shot | 33.2 | 35.6 | **37.0** |
| MMLU (Acc.) | 5-shot | 37.9 | 41.2 | **45.2** |
| C-Eval (Acc.) | 5-shot | 30.0 | 37.7 | **42.9** |
| CMMLU (Acc.) | 5-shot | 34.6 | 38.4 | **43.5** |

Comparison between different dense models trained with MQA, GQA and MHA on different benchmarks

This shows that **GQA and MQA trade off some performance** to reduce the Key-Value (KV) cache size.

But here's the surprising part!

When two MoE models (with 16B and 250B total parameters) trained with different attention mechanisms are evaluated, it is seen that **MLA outperforms MHA** across most benchmarks.

This performance is achieved **combined with a significantly smaller KV cache**, with a **14% KV cache usage** for the small MoE model and a **4% KV cache usage** for the large MoE model trained with MLA than MHA.

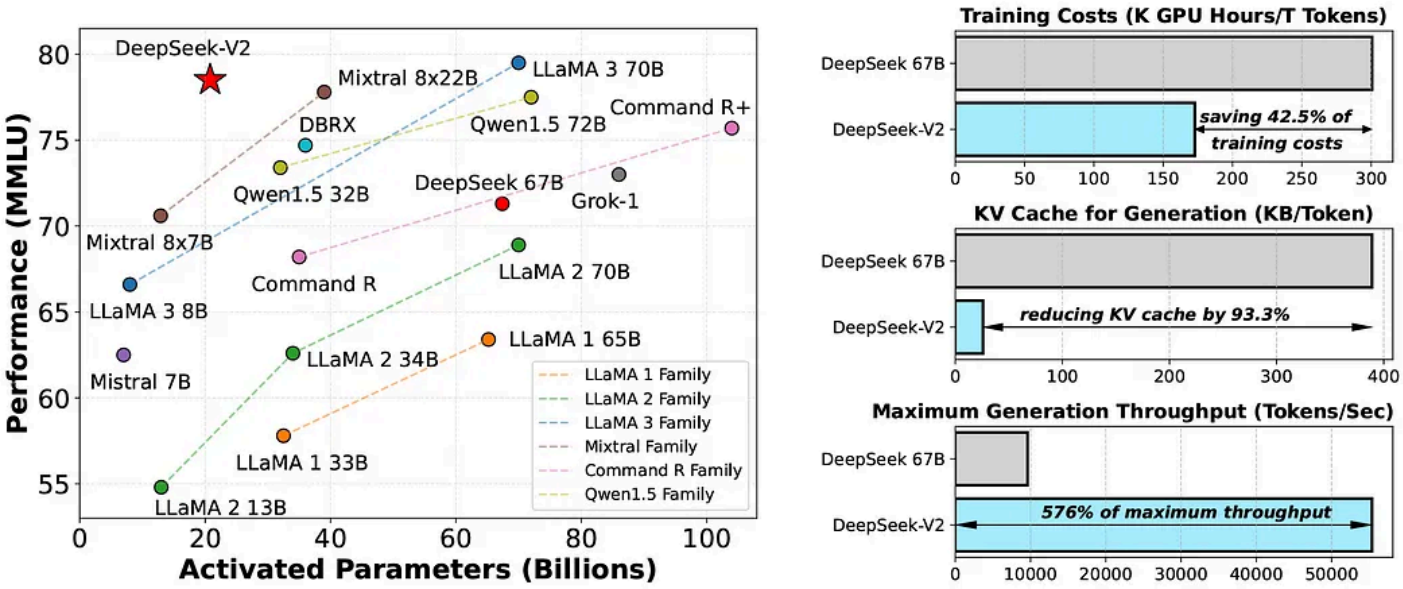
| Benchmark (Metric) | # Shots | Small MoE w/ MHA | Small MoE w/ MLA | Large MoE w/ MHA | Large MoE w/ MLA |
|---|---|---|---|---|---|
| # Activated Params | - | 2.5B | 2.4B | 25.0B | 21.5B |
| # Total Params | - | 15.8B | 15.7B | 250.8B | 247.4B |
| KV Cache per Token (# Element) | - | 110.6K | 15.6K | 860.2K | 34.6K |
| BBH (EM) | 3-shot | 37.9 | **39.0** | 46.6 | **50.7** |
| MMLU (Acc.) | 5-shot | 48.7 | **50.0** | 57.5 | **59.0** |
| C-Eval (Acc.) | 5-shot | **51.6** | 50.9 | 57.9 | **59.2** |
| CMMLU (Acc.) | 5-shot | 52.3 | **53.4** | 60.7 | **62.5** |

Comparison between MoE models trained with MLA and MHA on different benchmarks

**MLA does not trade off performance for the reduction in KV cache; instead, it improves performance and efficiency at the same time.**

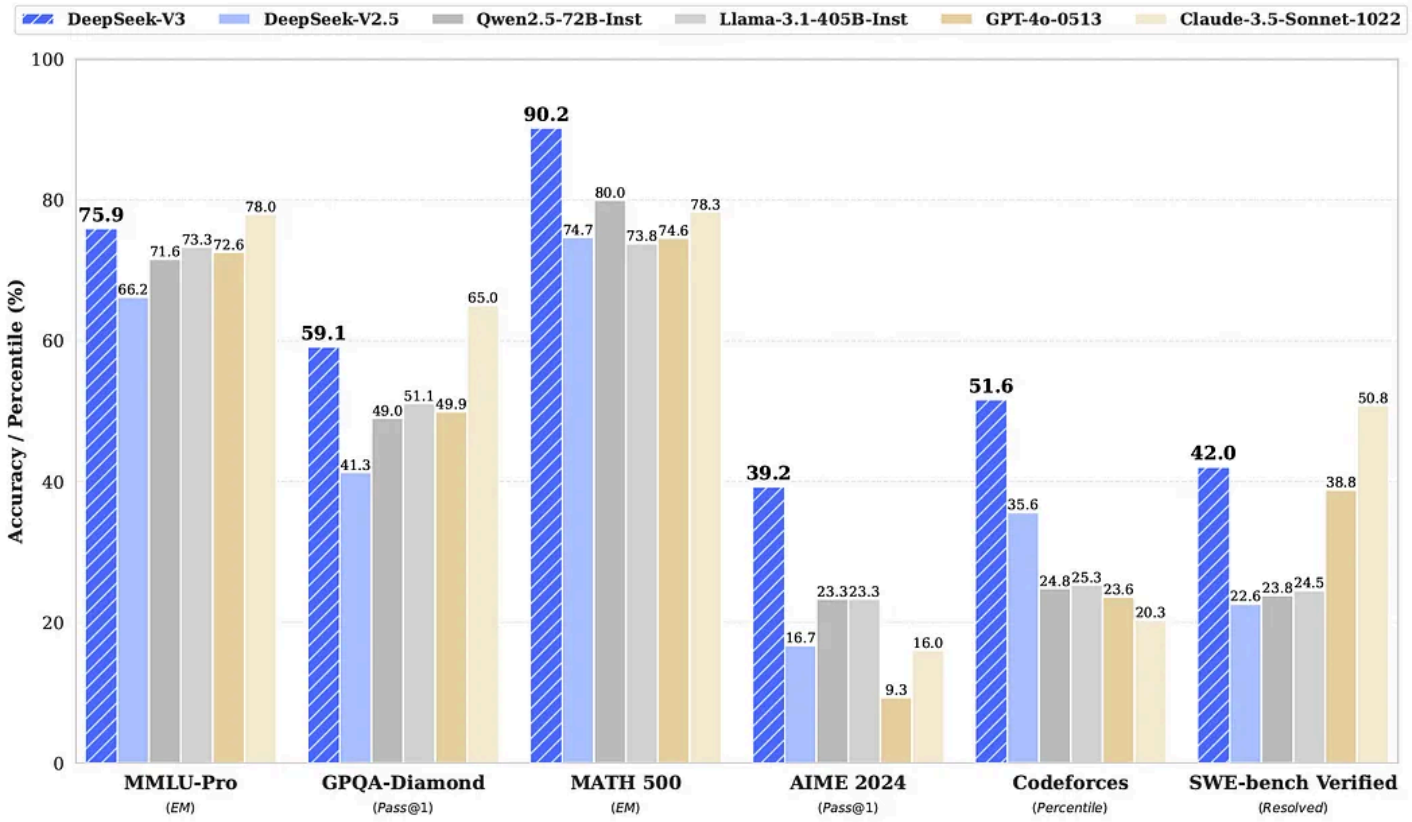
The following image shows how MLA is one of the reasons for the improvement in the MMLU performance, reduced training costs and size of the KV cache and improved generation throughput of DeepSeek-V2 compared to DeepSeek 67B.

(Note that DeepSeek 67B is the first LLM from DeepSeek, which is a dense model (as opposed to MoE) that uses Grouped-Query Attention (GQA) with RoPE embeddings (as opposed to MLA) trained on lesser tokens and less sophisticated training methodologies.)




Performance comparison: (a) DeepSeek V2 vs. different LLMs (b) DeepSeek-V2 vs. DeepSeek 67B (Dense)

MLA is also one of the reasons why DeepSeek-V3 performs so well on diverse language, coding and mathematics benchmarks.




Performance of DeekSeek-V3 equiped with MLA on different benchmarks (Image from ArXiv research paper titled 'DeepSeek-V3 Technical Report')



Stay tuned for upcoming articles in this series, where we dive deep into the other architectural and training modifications that make DeepSeek models perform well.

Thanks for reading!

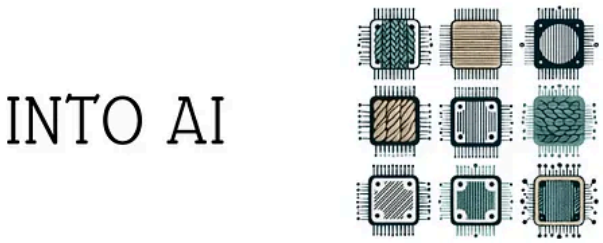
## Source Of Images

All images are obtained from the DeepSeek-V2 research paper unless stated otherwise.

## Further Reading

- *Research paper titled 'DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model' published in ArXiv*
- *Research paper titled 'DeepSeek-V3 Technical Report' published in ArXiv*
- *Research paper titled 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints' published in ArXiv*
- *Research paper titled 'Fast Transformer Decoding: One Write-Head is All You Need' published in ArXiv*
- *Research paper titled 'RoFormer: Enhanced Transformer with Rotary Position Embedding' published in ArXiv*

***Subscribe to 'Into AI' — my weekly newsletter where I help you explore Artificial Intelligence from the ground up by dissecting the original research papers.***




Artificial Intelligence | Programming | Data Science | Technology | Machine Learning

## Published in Level Up Coding

203K Followers · Last published 13 hours ago

Follow

Coding tutorials and news. The developer homepage gitconnected.com && skilled.dev && levelup.dev

## Written by Dr. Ashish Bamania

32K Followers · 412 Following

Following

🍰 I simplify the latest advances in AI, Quantum Computing & Software Engineering for you | 📰 Subscribe to my newsletter here: https://intoai.pub

## Responses (6)

What are your thoughts?

Respond

See all responses

## More from Dr. Ashish Bamania and Level Up Coding

In Level Up Coding by Dr. Ashish Bamania

### DeepSeek-R1 Beats OpenAI's o1, Revealing All Its Training Secrets...

A deep dive into how DeepSeek-R1 was trained from scratch and how this open-...

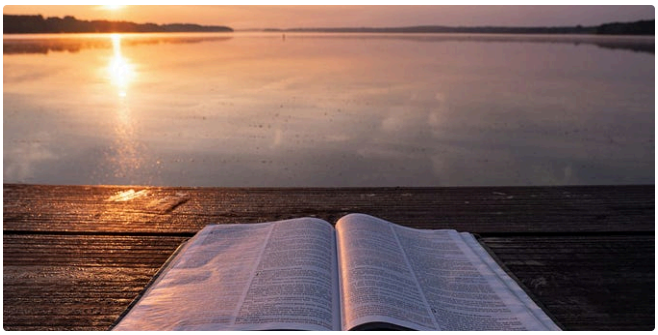
Jan 26 1.3K 32

In Level Up Coding by Salvatore Raieli

### The LLMs' Dilemma: Thinking Too Much OR Too Little?

Exploring the fine line between deep reasoning and computational overkill in larg...

13h ago 215 6

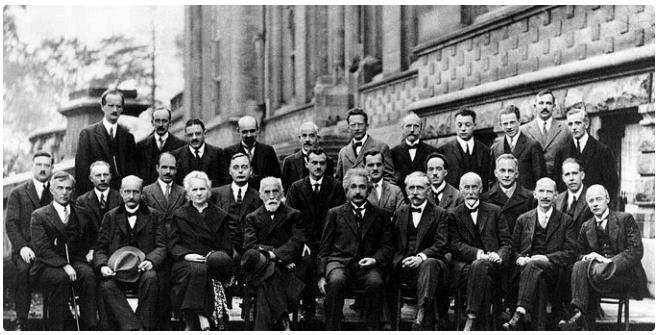


In Level Up Coding by Lucas Fernandes 👨‍💻

## The Ultimate Guideline For a Good Code Review

This Guideline Must Be Your Developer Teams Bible

✦ 2d ago 88 1



In Into Quantum by Dr. Ashish Bamania

## An Introduction To Bra-Ket (Dirac) Notation

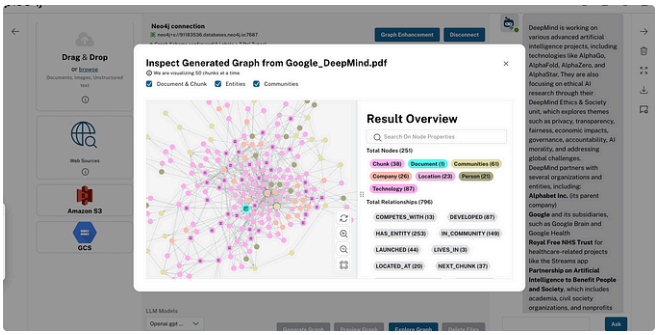
Learn about Bra-Ket or Dirac notation, a powerful mathematical tool used to describe...

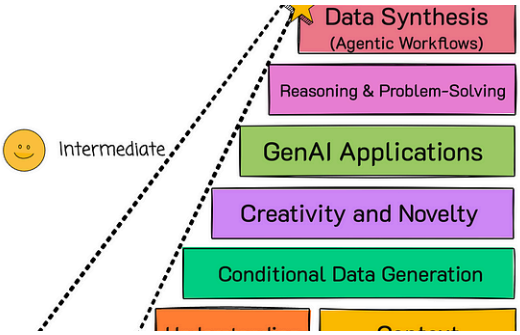
✦ Dec 28, 2024 234 1

See all from Dr. Ashish Bamania

See all from Level Up Coding

# Recommended from Medium



In Neo4j Developer Blog by Michael Hunger

## LLM Knowledge Graph Builder — First Release of 2025

New features include community summaries, parallel retrievers, and expanded model...

Feb 5 403 4

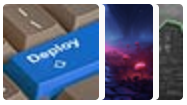
Cobus Greyling

## Why The Focus Has Shifted from AI Agents to Agentic Workflows

We find ourselves on a stairway from where Large Language Models were introduced to...

Feb 5 570 11

### Lists



**Predictive Modeling w/ Python**

20 stories · 1826 saves

**ChatGPT prompts**

51 stories · 2562 saves

**AI Regulation**

6 stories · 688 saves

**ChatGPT**

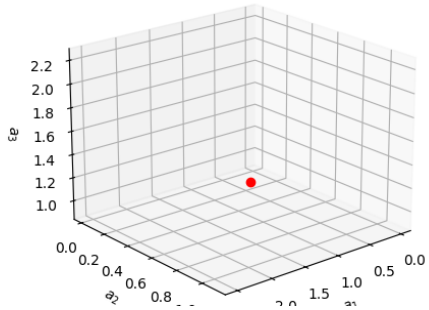
21 stories · 963 saves

In Artificial Intelligence in Plain Engl... by Mayur Ja...

## Mixture of Experts Explained — The Next Evolution in AI Architecture

Traditional transformer vs. MoE based Transformer

6d ago 1

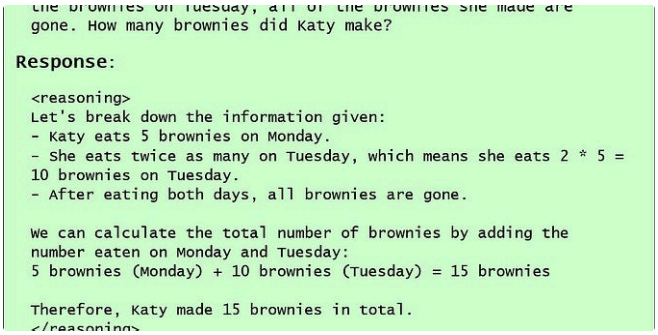



In Data Science Collective by Hylke C. Donker

## Exploiting the Structured State-Space Duality To Build Bayesian...

Building Bayesian Attention From Scratch

2d ago 174




Ertuğrul Demir

## Reasoning Models at Home

How to Train Your Own Specialized Reasoning Model at Home

6d ago 39

In Python in Plain English by Bradley Stephen Shaw

## A Cautionary Tale: Why You Might Want to Calibrate Your GBM Models

A real-world example of some of the dangers lurking below the surface of a “good” GBM...

4d ago 221 1

See more recommendations