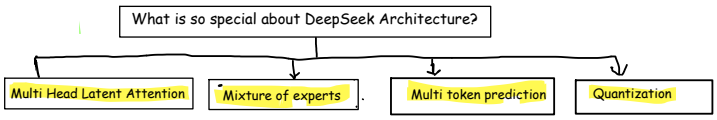
Deepseek Architecture

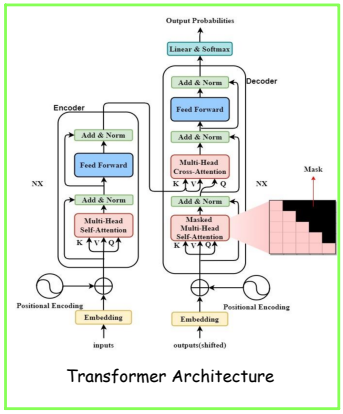
Wednesday, 2 July 2025 9:12 PM

DeepSeek Architecture



1. Multi Head Latent Attention

1. Multi Head Attention



Transformer Architecture

Encoder Block

Input Sentence: "Transformers are amazing"

1. Tokenization: Split the raw sentence into tokens using a tokenizer (e.g., BPE, Sentence piece, word-piece).
   Ex.

```
tokens = ["Transformers", "are", "amazing"]
```

2. Vocabulary Lookup (Token → Token ID): Use a predefined vocabulary to map each token to a unique integer ID.
   Ex.

Suppose the vocabulary has:

```
"Transformers" → 17
"are"          → 56
"amazing"      → 94
```

Result:

```
token_ids = [17, 56, 94]
```

3. Token Embedding (Learned)
   Use a learnable embedding matrix E ∈ ℝ^{|V| × d_model}
   Each token ID t maps to a vector E[t] ∈ ℝ^{d_model}

```
import torch.nn as nn

embedding_layer = nn.Embedding(num_embeddings=30000, embedding_dim=512)
token_embeddings = embedding_layer(torch.tensor([17, 56, 94]))  # shape: (3, 512)
```

Now:

```
TokenEmb = [
  E[17],  # "Transformers"
  E[56],  # "are"
  E[94]   # "amazing"
] → shape: (seq_len=3, d_model=512)
```

4. Positional Encoding: Transformers have no recurrence — so we **add position info**.

Compute fixed sinusoidal encoding:
For each position `pos` and dimension `i`:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Create a positional encoding matrix of shape `(seq_len, d_model)`.

5. Add Token and Positional Embeddings: The input to the transformer is the **element-wise sum** of token embedding and positional Embedding

```
input_to_attention = token_embeddings + positional_encoding
```

Shape = (3, 512) → sequence length × embedding dimension
This is the final matrix passed into the **transformer block**.

PyTorch Style Code Skeleton:

```
import torch
```

Decoder Block

Let's say we're decoding this: ""Les trans""

1. Tokenization → Embeddings + Positional Encoding

Just like the encoder:

- Tokens → IDs → token embeddings `x ∈ R^{tgt_seq_len × d_model}`
- Add positional encoding (fixed or learned)

Let:

```
tgt_seq_len = 3
d_model = 512
```

You get:

```
x ∈ R^{3 × 512}
```

2. Masked Multi-Head Self Attention:

Allow each position to attend **only to previous or same positions**, not to future tokens.
Why?
At training time, all tokens are known.
At inference, we generate tokens one by one — future tokens aren't available.

A triangular mask is applied:

```
Allowed:
Position 0 → [0]
Position 1 → [0, 1]
Position 2 → [0, 1, 2]
```

So attention scores at position `i` are masked such that:

```
mask[i][j] = -inf if j > i else 0
```

This is applied **before softmax**:

```
attn = softmax((QK^T / √d_k) + mask)
```

Let input `x ∈ R^{3 × 512}`
After Q, K, V projections (for all `h` heads):

- `Q, K, V ∈ R^{3 × d_k}`
- For all heads → output shape = `R^{3 × 512}`

3. Add & LayerNorm:

```
x1 = LayerNorm(x + MaskedSelfAttention(x))
```

4. Encoder-Decoder Cross-Attention

Goal:
Let the decoder **attend to the encoder's output** — the encoded source sequence.
At each decoder position (e.g., French word being generated), we ask:
"Which parts of the source sentence are relevant to me?"

```python
import torch.nn as nn

# Dummy inputs
token_ids = torch.tensor([17, 56, 94])  # shape: (seq_len,)

# Embedding
embedding = nn.Embedding(30000, 512)
token_emb = embedding(token_ids)  # (3, 512)

# Positional Encoding
def get_pos_enc(seq_len, d_model):
    pos = torch.arange(seq_len).unsqueeze(1)
    i = torch.arange(d_model).unsqueeze(0)
    angle_rates = 1 / torch.pow(10000, (2 * (i//2)) / d_model)
    angles = pos * angle_rates
    pe = torch.zeros(seq_len, d_model)
    pe[:, 0::2] = torch.sin(angles[:, 0::2])
    pe[:, 1::2] = torch.cos(angles[:, 1::2])
    return pe

pos_enc = get_pos_enc(seq_len=3, d_model=512)  # shape: (3, 512)

# Final Input to Attention
x = token_emb + pos_enc  # (3, 512)
```

6. Multi-Head Self-Attention

At each position in the sequence, we want the token to *understand the context it's in.*

For example, in the sentence:

```
"Bank of the river"
```

"bank" should attend more to "river", not "money." So, we need a way for each token to "look at" all others and decide *who to pay attention to*.

A) Linear Projections for Q, K, V

We project x into **queries (Q)**, **keys (K)**, **values (V)** using learnable matrices:

$$Q = xW^Q, \quad K = xW^K, \quad V = xW^V$$

- `W^Q, W^K, W^V ∈ R^{d_model × d_k}` where `d_k = d_model / num_heads = 64`

B) Scaled Dot-Product Attention (per head)

For each head:

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

🧠 **Intuition:**

- `Q · K^T` gives a **score**: how much each token wants to pay attention to another.
- Softmax turns this into a **distribution** over tokens.
- Multiplying by `V` tells the model: *"What should I take from those tokens I'm attending to?"*

Each token now becomes a **weighted sum of all other tokens** — but learned, context-aware, and dynamic for each head.

C) Concatenate Heads & Final Linear Layer

$$\text{MultiHead}(x) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Where `W^O ∈ R^{d_model × d_model}`

🔸 **Why multi-head?**

Each attention head can focus on **different relationships**:

- One head may focus on subject-verb
- Another may track time relationships
- Another may resolve ambiguity ("bank" context)

Instead of one single attention operation, we have:

- Split `d_model` into `h` smaller dimensions (e.g., 512 → 8 heads × 64)
- Run attention independently in each
- Concatenate outputs → project back to `d_model`

$$\text{MultiHead}(x) = \text{Concat}(head_1, ..., head_h)W^O$$

7. Add & LayerNorm

Apply residual connection and layer normalization:

$$x_1 = \text{LayerNorm}(x + \text{MultiHead}(x))$$

Why residual connection?

The network can always choose to copy the input - which stabilizes training and helps gradient flow. It also allows the model to learn refinements, not entirely new representations.

Why LayerNorm?

To normalize the input — this reduces internal covariate shift, stabilizes learning, and improves convergence. It ensures that all features have comparable scale and dynamics.

8. Feed-Forward Network (FFN)

Apply two linear layers with ReLU:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

Where:

- `W_1 ∈ R^{d_model × d_ff}`, `W_2 ∈ R^{d_ff × d_model}`

Typically, d_ff is 4 x d_model

Why FFN?

Attention mixes **information across positions**. FFN adds **non-linearity and transformation power at each token independently**.

Think of this as "refining" the representation for each token after it's looked at everyone else.

9. Add & Layer Norm (again)

Another residual connection:

$$x_2 = \text{LayerNorm}(x_1 + \text{FFN}(x_1))$$

10. Final Ecoder ouput

You **stack N such blocks**

$$\text{EncoderOutput} = \text{EncoderLayer}_N(\ldots(\text{EncoderLayer}_2(\text{EncoderLayer}_1(x))))$$

Shape remains the same: seq_len x d_model

But the content of each token's embedding now encodes **contextualized information** from all other tokens in the input.

Q, K, V in Cross-Attention

- Q comes from the decoder input `x1`
- K, V come from encoder output `enc_out`

```python
Q = x1 W^Q,    K = enc_out W^K,    V = enc_out W^V
```

- `x1 ∈ R^{tgt_len × d_model}`
- `enc_out ∈ R^{src_len × d_model}`

📐 **Dimensions:**

- `Q ∈ R^{tgt_len × d_k}`
- `K, V ∈ R^{src_len × d_k}`

Then compute:

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

This allows the decoder to **attend over the source sentence** at every position.

Why not mask here?

Because:

- **Source sentence is fully available**, even during inference.
- So we **do not** apply masking in cross-attention.

5. Add & LayerNorm

```python
x2 = LayerNorm(x1 + CrossAttention(x1, enc_out))
```

6. Feedforward Network

Same as in encoder:

```python
FFN(x) = ReLU(xW1 + b1)W2 + b2
```

- `W1 ∈ R^{512 × 2048}`
- `W2 ∈ R^{2048 × 512}`

7. Add & LayerNorm

Final residual connection:

```python
x3 = LayerNorm(x2 + FFN(x2))
```

This is the output of **one decoder block**.

8. Stack N Decoder Blocks

Just like the encoder, we repeat this **N times**.

Each decoder layer builds deeper representations, now jointly shaped by **past target context** + **source context**.

9. Final Output

Decoder output ∈ `R^{tgt_seq_len × d_model}`

Pass this through a final linear layer and softmax:

```python
logits = decoder_out @ W_vocab.T  # shape: (tgt_len × vocab_size)
```

Used to compute probabilities of the next tokens.

2. KV Caching

KV caching is a critical optimization used during **inference** (not training) in **decoder-only** or **auto-regressive** transformer models like GPT. It helps speed up inference by avoiding redundant computation across time steps

Problem Before KV Caching (Without Cache)

When generating a sequence (e.g., a sentence), say "The cat sat", the transformer processes the **entire prefix** at every new token step.

1. To predict the first token, say "The" → compute self_attention(The)
2. To predict "cat" → recompute attention over "The, cat"
3. To predict "sat" → recompute attention over "The, cat, sat"

This is redundant. Past key-value projections are recalculated again and again.

KV Caching: Motivation

Avoid recomputing attention on tokens you've already processed. Instead, **cache the key and value tensors** from previous time steps.

How KV Caching Works

1. Token Embedding

- Input: One token at a time (during inference), say token t_i
- Output: Embedded representation x_i ∈ ℝ^{d_model}

2. Linear Projections to Q, K, V

For each input token x_i, compute:

- Query: Q_i = x_i · W_Q
- Key: K_i = x_i · W_K
- Value: V_i = x_i · W_V

Shapes:

- Q_i: (num_heads, 1, d_k)
- K_i: (num_heads, 1, d_k)
- V_i: (num_heads, 1, d_v)

3. KV Caching Begins

- For the **first token**:
  - Compute K_1, V_1, and store them in the cache
  - cached_K = [K_1], cached_V = [V_1]
- For **subsequent tokens**:
  - Only compute K_i, V_i for the current token
  - **Append** to cache: cached_K = [K_1, ..., K_i], same for cached_V

4. Attention Computation

At each time step i:

- Use only the **new query** Q_i
- Use all **cached keys/values**: cached_K = [K_1..K_{i-1}], cached_V = [V_1..V_{i-1}]

5. No Re-computation of Past K/V

Since the cached K/V are reused, the model avoids redoing linear projections for all past tokens. The expensive attention matrix grows only by 1 step per token.

6. Caching Across All Layers

Each transformer block maintains **its own** KV cache. So if you have 24 layers:

- Each layer keeps cached_K, cached_V per layer
- These are updated layer-wise per token during autoregressive decoding

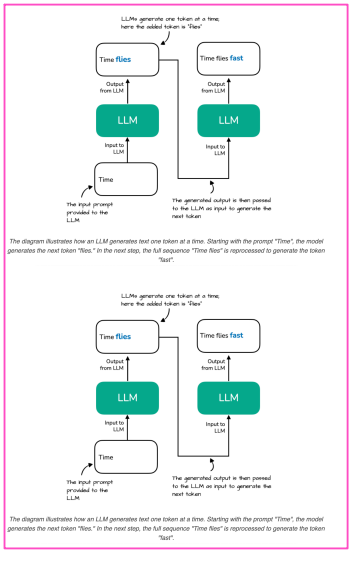
7. Memory vs Speed Tradeoff

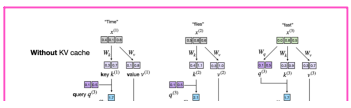
- **Pros**: Huge **speedup** during inference (especially for long sequences)
- **Cons**: Requires additional **memory** for storing cached K/V for every layer and head

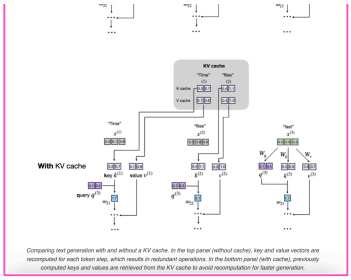
8. Why Not Cache Queries?

- Queries are computed only for the current token (during inference), and are **not reused**.
- Keys/Values are reused across all future time steps — only they are cached.

9. Diagramatic Representation:







Reference: https://sebastianraschka.com/blog/2025/coding-the-kv-cache-in-llms.html

10. Size of KV Cache

$$L \times B \times N \times H \times S \times 2 \times 2$$

L: Number of Transformer blocks
B: Batch Size
N: Number of Attention Heads
H: Attention Head Size
S: Context Length
2: No. of Caches per transformer block (K,V)
2: No. of Bytes per floating point Number (Assuming each parameter is 16 bit)

GPT-2: 36MB
GPT-3: 4.5GB

3. Multi Query Attention

Multi-Query Attention (MQA) is a variant of multi-head attention in transformers where:

- **Each attention head has its own Query projection**.
- **All heads share the same Key and Value projections**.

This design differs from standard multi-head attention (MHA), where each head has its own **Q, K, and V** projections.

1. Input Embeddings
- Suppose you have an input sequence of shape (batch_size, seq_len, d_model).
- Let's assume d_model = 768 and num_heads = 12.

2. Linear Projections
- In **standard MHA**, you create separate projection matrices for Q, K, and V per head.
  - So you get Q = W_Q * X, K = W_K * X, V = W_V * X, where all are shaped (batch_size, seq_len, d_k) for each head.
  - Resulting in **12 Q, 12 K, and 12 V** for 12 heads.
- In **MQA**:
  - You have **12 different W_Q**, one per head → 12 separate **Q** matrices.
  - But **only 1 shared W_K and 1 shared W_V**, so all heads **share the same K and V**.
  - So: Q_1, ... Q_12 and a **shared** K and V.

3. KV Caching in Decoding
- In autoregressive decoding (e.g., GPT), to generate one token at a time efficiently, we **cache past K and V values** for reuse.
- In **standard MHA**, each head has its own K/V → you must store (num_heads, seq_len, d_k) for both K and V.
- In **MQA**, all heads share the same K and V → you only store one set of (seq_len, d_k) for K and V.

4. Memory Savings
- **Significant reduction in KV cache size** during inference.
  - From **12x** to **1x** for K and V.
  - Especially important for **long-context generation** and **edge devices**.

5. Faster Generation
- Less memory → better **cache locality**.
- Smaller key/value caches → **faster memory reads**.
- Improves **latency** in autoregressive decoding (e.g., LLaMA-2 Chat uses MQA for this reason).

6. Loss of Head Diversity
- In standard MHA, each head can learn **different semantic attention patterns** via separate K/V projections.
- MQA limits this by forcing all heads to use **the same K and V** → reduces model expressiveness.
- Heads may **collapse into similar attention maps**, since only Q differs.

7. Reduced Representational Power
- With shared K/V, model has **less flexibility** to attend to **diverse contextual signals**.
- May hurt **fine-grained attention**, especially in tasks needing nuanced alignment (e.g., translation, QA).

8. Accuracy Drop in Some Tasks
- Empirically, MQA can perform worse than MHA on **complex reasoning or multilingual** tasks.
- Works well for **text generation**, but **not always** for other modalities (vision, multimodal) or for **short context** tasks.

9. Trade-off Between Speed and Accuracy
- MQA trades **speed and memory efficiency** for a **small drop in quality**.
- Acceptable in large models like GPT-3.5, LLaMA-2–7B+, but may not suit **small models** or **low-resource training**.

10. Size of KV Cache

$$L \times B \times H \times S \times 2 \times 2$$

L: Number of Transformer blocks
B: Batch Size
H: Attention Head Size
S: Context Length
2: No. of Caches per transformer block (K,V)
2: No. of Bytes per floating point Number (Assuming each parameter is 16 bit)

GPT-3 with MHA : 4.5GB
GPT-3 with MQA: 48MB

4. Grouped Query Attention

Grouped Query Attention is a hybrid between **Multi-Head Attention (MHA)** and **Multi-Query Attention (MQA)** where:

- Each **group of attention heads** shares a **single Key and Value** projection.
- But **each head still gets its own Query projection**, like in MHA.

This provides a **balance** between MQA's memory efficiency and MHA's representational power.

1. **Input Embeddings**
   - Suppose you have an input sequence with shape (batch_size, seq_len, d_model).
   - Let's assume d_model = 768 and num_heads = 12.
   - Let's say you **group the 12 heads into 4 groups**, so each group has 3 heads.

2. **Linear Projections**
   - In **standard MHA**:
     - Each head has its **own** Q, K, and V projection.
     - You get 12 Qs, 12 Ks, and 12 Vs.
   - In **GQA** (for 4 groups and 12 heads):
     - You have:
       - 12 different W_Q matrices → one **Query** projection per head.
       - Only 4 different W_K and 4 different W_V matrices → one shared **Key** and **Value** per **group**.
     - So:
       - Q_1, ..., Q_{12} (1 per head)
       - K_1, ..., K_4 (1 per group)
       - V_1, ..., V_4 (1 per group)
   - Each group of 3 heads shares the same K and V but uses their own Q.

3. **KV Caching in Decoding**
   - During **autoregressive decoding**:
     - In MHA: You cache 12 separate K/V → expensive.
     - In MQA: You cache only 1 K/V → very efficient.
     - In GQA (with 4 groups): You cache **4** sets of K/V → a middle ground.
   - KV cache shape becomes (num_groups, seq_len, d_k) instead of (num_heads, seq_len, d_k).

4. **Memory Savings**
   - **Moderate reduction** in KV cache size compared to full MHA.
     - If num_heads = 12 and num_groups = 4, cache size drops by **3x**.
   - This is useful for **long-sequence generation** and **deployment on limited hardware**.

5. **Faster Generation**
   - Fewer KV tensors → better memory locality.
   - Smaller cache → faster memory access during token generation.
   - Improves **inference speed**, though less so than MQA.

6. **Preserves Partial Head Diversity**
   - Unlike MQA, **not all heads share the same K/V**.
   - Each group can attend to different contexts via their own K/V.
   - Retains **some of the expressiveness** of standard MHA.
   - Heads within a group may correlate more, but **inter-group diversity remains**.

7. **Better Representational Power than MQA**
   - Because K/V are shared **per group**, not globally:
     - More **fine-grained attention** than MQA.
     - Still less expressive than full MHA, but better suited for **complex reasoning** tasks than MQA.

8. **Improved Accuracy over MQA**
   - Empirical studies show GQA performs **better than MQA** on tasks like:
     - Multilingual understanding
     - Question answering
     - Token classification
   - Widely used in models like **LLaMA-3**, **Mistral**, and **Gemma**.

9. **Trade-off: Accuracy vs. Efficiency (Sweet Spot)**
   - GQA offers a **balance**:
     - Much **faster and more memory-efficient** than MHA.
     - Retains **much more accuracy** than MQA.
   - Ideal for models where:
     - **Scalability** is key (e.g., LLMs with long contexts)
     - **Edge deployments** or **limited compute environments** matter
     - **High-quality outputs** are still necessary.

10. **Size of KV Cache**

$$L \times B \times G \times H \times S \times 2 \times 2$$

L: Number of Transformer blocks
B: Batch Size
G: No. of Groups
H: Attention Head Size
S: Context Length
2: No. of Caches per transformer block (K,V)
2: No. of Bytes per floating point Number (Assuming each parameter is 16 bit)

Bottom line:
***GQA* ≈ "Goldilocks zone"** — not as expensive as MHA, not as lossy as MQA. A solid compromise used in modern LLMs.

5. Multi Head Latent Attention

Diagram from paper - DeepSeek-V2:

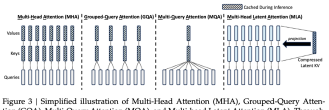


Figure 3 | Simplified illustration of Multi-Head Attention (MHA), Grouped-Query Attention (GQA), Multi-Query Attention (MQA), and Multi-head Latent Attention (MLA). Through jointly compressing the keys and values into a latent vector, MLA significantly reduces the KV cache during inference.

Attention Mechanism Type	Number of unique key-value pair	KV cache size	Performance (Context Understanding)
Multi-Head Attention (MHA)	Each head has its own key and value	Largest	Best
Multi-Query Attention (MQA)	All heads share same key and value	Smallest	Worst
Grouped Query Attention (GQA)	Heads are divided in G groups (each group share key and value)	Medium	Medium

Can we get best of both worlds? (KV cache size and Performance)

Size of KV cache
$L \times B \times N \times H \times S \times 2 \times 2$

Good Language Performance Model
For this, All heads should have different values in K, V matrix.

This needs to be reduced.

How do we achieve this?

What if we don't have to cache the Key and Values Separately?
What if we cache only one matrix?
What If this matrix has lesser dimensions than NxH?

To achieve this let's start by projecting this matrix into a latent space

X = (4x8)

Today
it
is
very

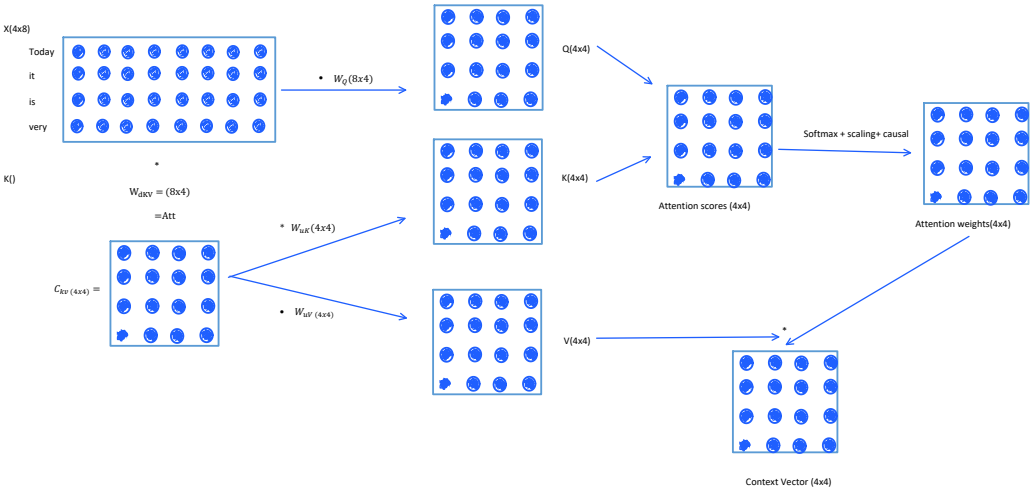
*

$W_{dkv} = (8x4)$

$C_{kv} = 4x4$

This is our latent matrix which we will cache.

What do we do next?



How does adding this latent matrix help?

1. $Q = X * W_Q$
2. $C_{KV} = X * W_{dkv}$
3. $K = Ckv * Wuk = X * Wdkv * Wuk$
4. $V = Ckv * Wuv = X * Wdkv * Wuv$

"The absorption trick "

Attention scores $= Q * K^T$

$= (X * W_Q) * (W_{uk}^T * W_{dkv}^T * X^T)$

$= (X * W_Q * W_{uk}^T) * (X * Wdkv)^T$

$= (X * (W_Q * W_{uk}^T) * (C_{kv})^T$

This is fixed during training. This needs to be cached

Context vector = Attention weights * V

$= (Q * K^T) * (X * Wdkv * Wuv) * W_O$

= Attention scores $* C_{kv} * (Wuv * W_O)$

Already Computed | Cached | fixed at training

What happens when the new token comes in?

Today
is
very
bright

1. Calculate Query (Q)
   $Q = X_{1x8} * WQ_{8x4} * Wuk_{4x4}$
2. Compute KV Vector
   $= X_{1x8} * Wdkv_{8x4}$ → Append this to KV cache
3. Now we multiply updated KV cache with absorbed query vector
   Attention scores = $Q_{1x4} * Updated\ KV_{5x4}^T$
   Attention score will have dimention (1x5)
4. Compute attention weights for the same
   Attention weights will also have dimention 1x5
5. Calculate Value matrix
   Value = $Updated\ KV_{5x4}^T * Wuv_{4x4}$

Value matric will have dimention (5x4)

6. Calculate context vector

$$Context\ Vector_{1x5} = Attention\ Weights_{1x5} * V_{5x4}$$

Did we actually solved the two problems we started with?

Size of KV cache
L x B x N x H x S x 2 x 2

Good Language Performance Model
For this, All heads should have different values in K, V matrix.

This Needs to be reduced.

Size in KV cache after MLA

L x B x dl x S x 2 ⟶ Reduction in size: $\frac{2 \times N \times H}{dl}$

For Deepseek:

$= \frac{2 * 128 * 128}{576}$ ~= 57 times

= 400GB to 6 GB

6. Positional Embedding

   A) Integer Positional Embedding

   B) Sinusoidal Positional Embedding

   C) Rotary Positional Embedding

7. Multi Head Latent Attention With Rotary Positional Embedding

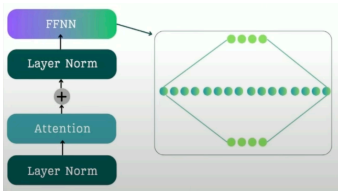
.

2. Mixture of Experts

Another Innovative thing deepseek did is creative usage of Mixture of Experts Technique
In traditional neural Networks standard feedforward neural network is used.

1. Traditional Mixture of Experts

   It's not a new idea, Mixture of expert was introduced in 1991 in neural computation.



As shown above. Feedforward network has one hidden layer with 4 times number of units as the input layer.

The output has same number of units as in input layer.

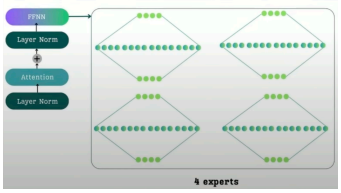
Parameters -

Assume input is 768 dimention so hidden layer will be 4 x 768.
= 768 x 4 x 768 Parameters

Same number of parameters will be at output side
= 4 x 768 x 768

Assume we have 12 number of heads
= 12 * (768*4*768) * 2
= 50Million

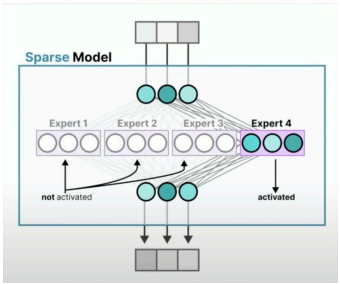
As parameters are huge this increases training and inference time. Mixture of experts come to rescue here.



In figure we have shown 4 experts, these are per transformer block.
In mixture of experts model we have multiple of such networks called **experts** in transformer block. So how does it help than FFN (dense)?

In a dense model, input token passes through all the parameters, (all layers and neurons)

In mixture of input we have multiple experts but only small subset of them are activated for any given input.

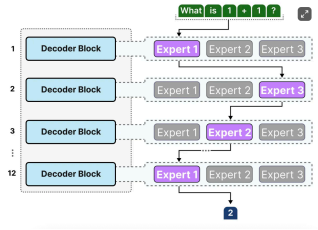


What do these experts learn? https://arxiv.org/pdf/2202.08906

Expert specialization	Expert position	Routed tokens
Sentinel tokens	Layer 1	been <extra_id_4><extra_id_7>floral to <extra_id_10><extra_id_12><extra_id_15> <extra_id_17><extra_id_18><extra_id_19>..
	Layer 4	<extra_id_0><extra_id_1><extra_id_2> <extra_id_4><extra_id_6><extra_id_7> <extra_id_12><extra_id_13><extra_id_14>...
	Layer 6	<extra_id_0><extra_id_4><extra_id_5> <extra_id_6><extra_id_7><extra_id_14> <extra_id_16><extra_id_17><extra_id_18>..
Punctuation	Layer 2	, , , , , , , , , - , , , , ,).)
	Layer 6	, , , , : : , & , & & ? & - , , ? , , , . <extra_id_27>
Conjunctions and articles	Layer 3	The the the the the the the the the The the the the the The the the the the a and and and and and and and or and a and . the the if ? a designed does been is not
	Layer 6	
Verbs	Layer 1	died falling identified fell closed left posted lost felt

		left said read miss place struggling talking signed shed falling designed based disagree submitted develop
Visual descriptions *color, spatial position*	Layer 0	her over her know dark upper dark outer center upper blue inner yellow raw mama bright bright over open your dark blue
Proper names	Layer 1	A Mart Gr Mart Kent Med Cor Tri Ca Mart R Mart Lorraine Colin Ken Sam Ken Gr Angel A Dou Now Gu GT Q Gu C Ko C Ko Gu G
Counting and numbers *written and numerical forms*	Layer 1	after 37 19. 6. 27 11 Seven 25 4, 54 1 two dead we Some 2012 who we few lower each

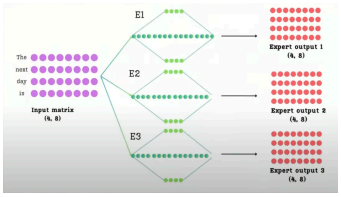
How the token travels till end?



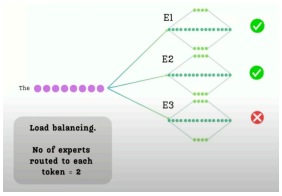
Now, We'll go through hands on example to see how MOE actually works:

Step 1: We start with input matrix of size 4x8 (4 token with embedding dimension 8)

We are going to pass this input matrix to MOE with 3 experts. Note that each expert is nothing but simple FNN.



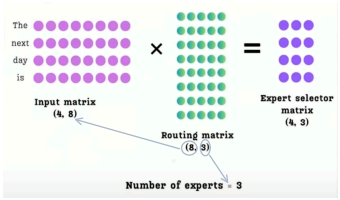
Step 2: When we pass input matrix through these 3 experts, we get 3 expert outputs matrices each of 4x8.
The challenge is to figure out how to merge these to get single resulting matrix of 4x8



Step 3: to do the merging, we use a method called load balancing.
Every token will be sent to only selected number of experts. This is called load balancing.
As shown in figure, lets say that we decide to route token only via 2 experts.

Step 4: Once we have decided how many experts will be assigned to each token, the next question is
"**How much weightage to give to each expert**?"
"Which Expert to select?"

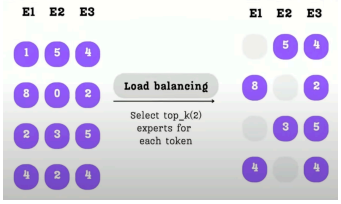
This is decided using **Routing mechanism**.



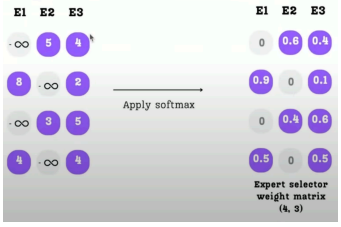
As shown in figure above, we multiply the input matrix 4x8 with a routing matrix 8x3 and this leads to a expert selector matrix 4x3
Note that every column of expert selector corresponds to each expert.
Since we only decide to keep 2 experts, for each token, we take expert selector matrix and make sure each row has only 2 active experts.

This is shown in below figure.

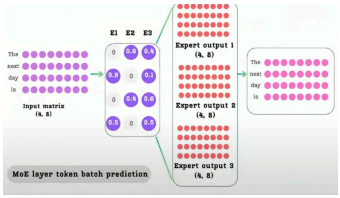
To select expert.



To select weightage



Step 5:
Once we have expert selector weight matrix. We can use then to merge expert matrices.



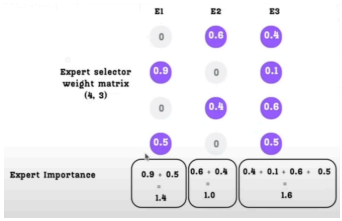
There are two major issues with Mixture of Experts. We'll discuss those in next two steps.

Step 6: Auxillary loss version 1

In MOE models, the routing mechanism selects a subset of experts for each input.
If some expert is chosen too often while others remain unutilized, this creates inefficient learning and performance bottlenecks.

An Auxiliary loss term is added to main training loss to penalize imbalanced expert selection, pushing the routing function towards more uniform distribution.
To calculate auxiliary loss, we first start with expert selector weight matrix which consists of experts assigned to every token, and probabilities assigned to every expert.

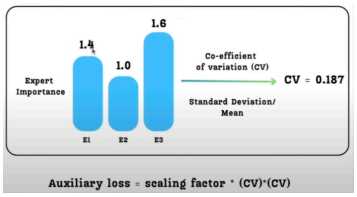
Each column of expert selector weight matrix gives us probability assigned to each expert. Based on this we calculate term called expert importance.
Expert importance is just sum of all probabilities for that expert.



We want a Balanced MOE. What does balance mean?
We want roughly same number of token routed to each expert. How to quantify it? We just saw above. Expert importance.

We want to penalize model if expert importance has more variance. This would mean some experts are more important than others.
The coefficient variation is given by formula:

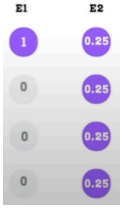
Coefficient Variation = Standard Deviation / Mean



We add this auxilary loss to llm loss.

Step 7: Load Balancing

While previously discussed importance loss is useful. Assigning equal importance to experts does not necessarily lead to uniform token routing.



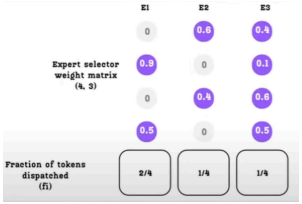
Here both experts adds up to 1 and hence have same importance but expert 2 is getting selected way more than expert 1.

We need another way: load balancing.

The first step of load balancing is to calculate probability that the router choose given expert.
We can calculate it by dividing expert importance by number of tokens

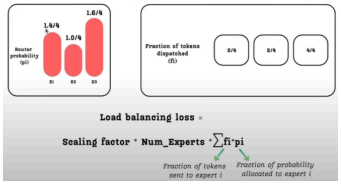
- Probability that the router will choose Expert 1 (p1) = 1.4/4 = 0.35
- Probability that the router will choose Expert 2 (p2) = 1/4 = 0.25
- Probability that the router will choose Expert 3 (p3) = 1.6/4 = 0.4

The next step is to calculate the fraction of tokens dispatched to each expert.
Again, this can be calculated from the expert selector weight matrix.
Fraction of tokens dispatched is simply the number of tokens that have non-zero activation for each expert divided by total tokens (in this case = 4).



- The fraction of tokens dispatched to expert 1 (f1) = 2/4 = 0.5
- The fraction of tokens dispatched to expert 2 (f2) = 1/4 = 0.25
- The fraction of tokens dispatched to expert 3 (f3) = 1/4 = 0.25

$$\text{Load balancing loss} = \text{Scaling factor} * \text{Number of experts} * \sum_{i=1}^{\text{num-experts}} f_i p_i$$



pi is the fraction of total router probability (importance) given to expert i.
fi is the fraction of tokens dispatched to expert i.

By minimizing the auxiliary loss, you mathematically enforce the model to distribute tokens proportionally to how much each expert is valued or "trusted."
resulting in more efficient and balanced use of the MoE architecture.

Step 8: Capacity Factor

There is one more issue which a MoE model faces: expert imbalance.
Expert overload essentially means some experts receiving disproportionally high number of tokens, while others not receiving any.
Although we have seen another ways to handle this, this is also a guardrail we can enforce.

Expert capacity is the maximum number of tokens that a single expert can process in a given batch.

$$\text{Expert capacity} = \text{Tokens per batch} / \text{Number of experts} * \text{Capacity Factor}$$

Tokens per batch = batch size * sequence length * top_k

DeepSeek Innovation 1: Auxiliaty Loss Free Load Balancing

$$\text{Load balancing loss} = \text{Scaling factor} * \text{Number of experts} * \sum_{i=1}^{\text{num-experts}} f_i p_i$$

This is the load balancing loss we saw earlier.
The use of this loss helps maintaining load balance in expert models but it also acts as a regularization term that interferes with language modelling.

If the scaling factor is too small or absent, it leads to poor balance and routing collapse, reducing model efficiency and limiting expert utilization.

A large scaling factor ensures load balance but degrades overall model performance.

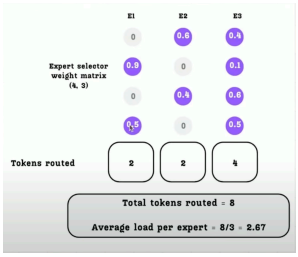
This trade-off presents a fundamental problem: balancing experts effectively without compromising training quality.

Can we do something better? (Answer is always, Yes.)

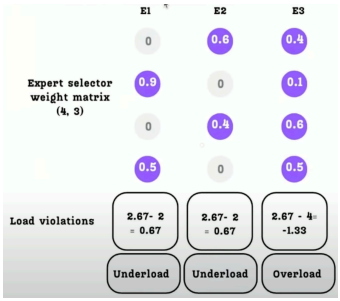
To resolve this problem, Deepseek, implemented Loss-Free Balancing, a method that directly enforces load balance without introducing additional gradients beyond those from the language modeling loss.

Here is how its done:

Based on the total tokens routed, we calculate the average load per token.



Based on this value, we know whether given expert is underloaded or overloaded.



From above figure expert 1 and 2 are underloaded and expert 3 is overloaded.

Now, we look at load violation error. = average load per expert - actual load per expert

In the next step, we introduce bias.
For each expert, we introduce a bias term b.
The bias is initialized as 0 for each expert.

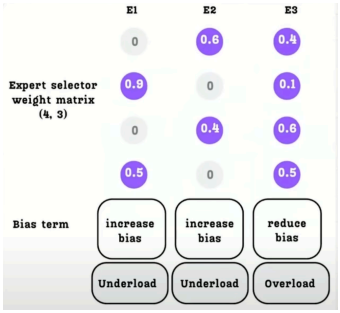
Here is how the bias term is updated for each expert:

$$b_i = b_i + u * sign(\text{load violation error})$$

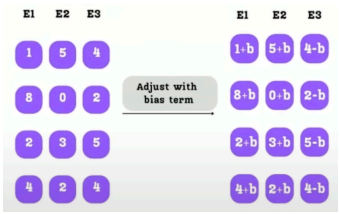
Here, u is predefined constant.

Note that if an expert is underloaded, it's load violation error is positive.
If an expert is overloaded, it's load violation error is negative.

The above formula ensures that if an expert has a heavy load on the previous batch, we will reduce its bias. otherwise, we will increase it.



Here is how its done



Since expert 1 and 2 are underloaded, adding these bias terms will make sure values of the expert 1 and expert 2 increase.

Through the dynamic adjustment for the biases, Deepseek achieved good expert load balance, without directly introducing noisy gradients into the model like what happens with the load balancing loss.
Deepseek showed that auxiliary loss free load balancing achieves both better performance and better load balance compared with traditional load balancing.

Deepseek Innovation 2: Shared Experts

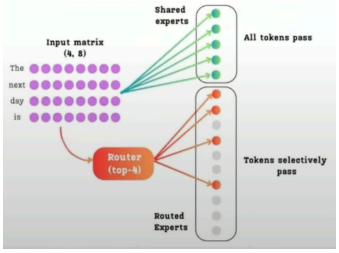
This architecture manifests two potential issues: (1) **Knowledge Hybridity**: existing MoE practices often employ a limited number of experts (e.g., 8 or 16), and thus tokens assigned to a specific expert will be likely to cover diverse knowledge. Consequently, the designated expert will intend to assemble vastly different types of knowledge in its parameters, which are hard to utilize simultaneously. (2) **Knowledge Redundancy**: tokens assigned to different experts may require common knowledge. As a result, multiple experts may converge in acquiring shared knowledge in their respective parameters, thereby leading to redundancy in expert parameters. These issues collectively hinder the expert specialization in existing MoE practices, preventing them from reaching the theoretical upper-bound performance of MoE models.

Deepseek modified the existing MoE architecture by introducing the idea of shared experts.

This approach divides experts into two groups:

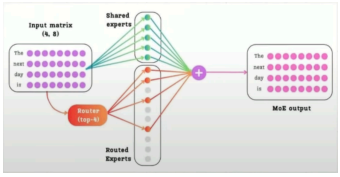
- Shared Experts: Experts that process every token, regardless of routing.
- Routed Experts: Experts that handle tokens selectively, based on the usual routing strategy.

While routed experts continue to be assigned tokens according to standard MoE procedures,
shared experts are always active and handle every token in the batch.



Typically, the number of shared experts should be smaller than the number of routed experts.

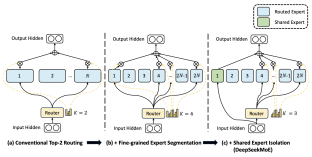
If there are too many shared experts, the model becomes less sparse, reducing the efficiency benefits of using an MoE architecture.



Deepseek Innovation 3. Fine-grained expert segmentation

If the number of experts is small, each expert is forced to learn a wide variety of knowledge types, reducing its specialization.
This leads to inefficient learning because the expert parameters need to generalize across too many diverse inputs.

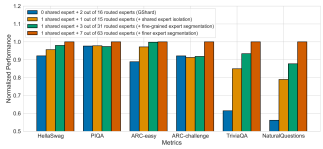
In fine-grained expert segmentation, each large expert FFN (Feed-Forward Network) is split,
into m smaller experts by reducing the hidden dimension of the FFN by a factor of 1/m.
To compensate for the reduced capacity per expert, the model activates m times more experts per token while keeping the same total computational cost.



It is noteworthy that across these three architectures, the number of expert parameters and computational costs remain constant.

Some results from Deepseek-MOE paper:

Metric	# Shot	Dense	Hash Layer	Switch	GShard	DeepSeekMoE
# Total Params	N/A	0.2B	2.0B	2.0B	2.0B	2.0B
# Activated Params	N/A	0.2B	0.2B	0.2B	0.3B	0.3B
FLOPs per 2K Tokens	N/A	2.9T	2.9T	2.9T	4.3T	4.3T
# Training Tokens	N/A	100B	100B	100B	100B	100B
Pile (Loss)	N/A	2.060	1.932	1.881	1.867	**1.808**
HellaSwag (Acc.)	0-shot	38.8	46.2	49.1	50.5	**54.8**
PIQA (Acc.)	0-shot	66.8	68.4	70.5	70.6	**72.3**
ARC-easy (Acc.)	0-shot	41.0	45.3	45.9	43.9	**49.4**
ARC-challenge (Acc.)	0-shot	26.0	28.2	30.2	31.6	**34.3**
RACE-middle (Acc.)	5-shot	38.8	38.8	43.6	42.1	**44.0**
RACE-high (Acc.)	5-shot	29.0	30.0	30.9	30.4	**31.7**
HumanEval (Pass@1)	0-shot	0.0	1.2	2.4	3.7	**4.9**
MBPP (Pass@1)	3-shot	0.2	0.6	0.4	0.2	**2.2**
TriviaQA (EM)	5-shot	4.9	6.5	8.9	10.2	**16.6**
NaturalQuestions (EM)	5-shot	1.4	1.4	2.5	3.2	**5.7**



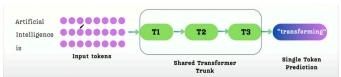
3. Multi Token Prediction

One of the major innovations in the Deepseek architecture is multi-token prediction.

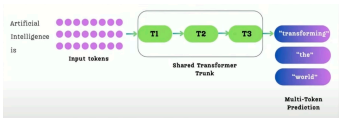
Multi-Token Prediction was a technique already introduced by Meta in their 2024 paper titled "Better & Faster Large Language Models via Multi-token Prediction".
(https://arxiv.org/pdf/2404.19737)

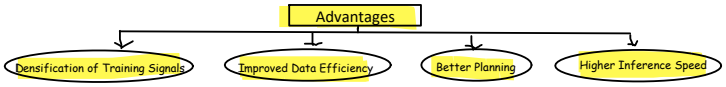
Deepseek implemented this same technique with slight modifications.
Let us understand what multi-token prediction is and how it was implemented by DeepSeek.

Single Token Prediction



Multi Token Prediction





1. Densification of training signals

MTP provides richer(long range understanding) and denser training signals than single-token prediction.

Traditional next-token training only guides the model to predict a single immediate token.

MTP, however, instructs the model to simultaneously predict multiple future tokens, generating more informative gradient signals per training example.

2. Improved Data Efficiency

MTP-trained models achieved better results on standard benchmarks like HumanEval and MBPP with the same amount of training data, solving about 15% more code problems on average.

Training data	Vocabulary	n	MBPP			HumanEval			APPS/Intro		
			@1	@10	@100	@1	@10	@100	@1	@10	@100
		1	19.3	42.4	64.7	18.1	28.2	47.8	0.1	0.5	2.4

313B bytes (0.5 epochs)	bytes	8	**32.3**	**50.0**	**69.6**	**21.8**	**34.1**	**57.9**	**1.2**	**5.7**	**14.0**
		16	28.6	47.1	68.0	20.4	32.7	54.3	1.0	5.0	12.9
		32	23.0	40.7	60.3	17.2	30.2	49.7	0.6	2.8	8.8
200B tokens (0.8 epochs)	32k tokens	1	30.0	53.8	73.7	22.8	36.4	62.0	2.8	7.8	17.4
		2	30.3	55.1	76.2	22.2	38.5	62.6	2.1	9.0	21.7
		4	**33.8**	**55.9**	**76.9**	**24.0**	**40.1**	**66.1**	1.6	7.1	19.9
		6	31.9	53.9	73.1	20.6	38.4	63.9	**3.5**	**10.8**	**22.7**
		8	30.7	52.2	73.4	20.0	36.6	59.6	3.5	10.4	22.1
1T tokens (4 epochs)	32k tokens	1	40.7	65.4	83.4	31.7	57.6	83.0	**5.4**	**17.8**	**34.1**
		4	**43.1**	65.9	83.7	31.6	57.3	**86.2**	4.3	15.6	33.7

3. Better Planning

Greater importance to choice points.

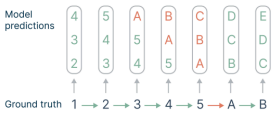


Figure 9: **Multi-token prediction loss assigns higher implicit weights to *consequential* tokens.** Shown is a sequence in which all transitions except "5 → A" are easy to predict, alongside the corresponding prediction targets in 3-token prediction. Since the consequences of the difficult transition "5 → A" are likewise hard to predict, this transition receives a higher implicit weight in the overall loss via its correlates "3 → A", ..., "5 → C".

4. Higher Inference speed

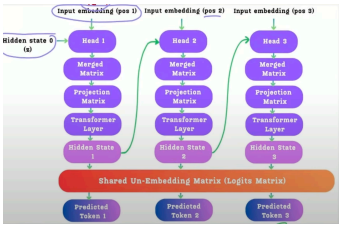
Upto 3x faster inference speed.

3.2. Faster inference

We implement greedy *self-speculative decoding* (Stern et al., 2018) with heterogeneous batch sizes using xFormers (Lefaudeux et al., 2022) and measure decoding speeds of our best 4-token prediction model with 7B parameters on completing prompts taken from a test dataset of code and natural language (Table S2) not seen during training. We observe a speedup of **3.0**× on code with an average of 2.5 accepted tokens out of 3 suggestions on code, and of

Deepseek only used it during pretraining and not in inference.

How Deepseek Rewrote Multi Token Prediction?



Inspired by Gloeckle et al. (2024), we investigate and set a Multi-Token Prediction (MTP) objective for DeepSeek-V3, which extends the prediction scope to multiple future tokens at each position. On the one hand, an MTP objective densifies the training signals and may improve data efficiency. On the other hand, MTP may enable the model to pre-plan its representations for better prediction of future tokens. Figure 3 illustrates our implementation of MTP. Different from Gloeckle et al. (2024), which parallelly predicts D additional tokens using independent output heads, we sequentially predict additional tokens and keep the complete causal chain at each prediction depth. We introduce the details of our MTP implementation in this section.