Architecture of DeepSeek-V3

Molnár Botond

June 5, 2025

Transformer Model Origins

Transformer models were first introduced in the Paper "Attention is all you need" in 2017, which arguably launched the artificial intelligence revolution. It introduced the **self-attention** mechanism which enables the model to understand the whole context of the input without using convolution or sequence based RNNs. [9]

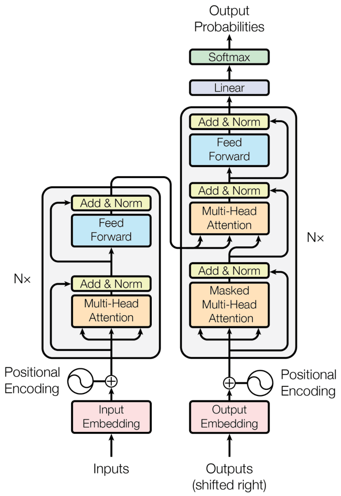
Why not Convolution?

- Convolution is excellent at capturing local patterns, but since they have a limited view on the whole context it falls short of capturing global context. [8]
- The weights of CNNs are constant and learned during training, hence applying the same transformation across the same input. (In transformer models these weights come from the self-attention head $\rightarrow$ calculated dynamically)

Why not Recurrent Neural Networks (RNN)?

- However, the problem of vanishing/exploding gradients has been solved by newer models (LSTM, GRU) they still struggle to maintain "attention". For example, when the model goes through the sequence it might "forget" the early parts of the message or might become insignificant. [6]
- Its sequential processing nature prevents efficient parallelization (the t-th step depends on the $t-1$-th step), which makes it time consuming to train the model. [4]
- It has a fixed sized hidden state (fixed size vector). As the sequence progresses, this vector must continuously update to summarize all relevant past information. For very long sequences, this leads to an information compression problem, where the hidden state cannot retain all the nuances of the entire history. [7]

Transformer Model Architecture



Source: [9]

Input Processing

- The input text must first be converted into numerical representation, usually using byte-pair encoding done on subword level.
- Then each token gets assigned a numerical ID.

Example:

Raw text: *The quick brown fox jumps.*
$\rightarrow$ ["The", "quick", "brown", "fox", "jump", "##s", "."] (## signals the continuation of a word)
$\rightarrow$ [101, 234, 567, 890, 123, 456, 789]

Word Embedding

- **Learned Embeddings**: The transformer uses learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. It learns these representation during the training process.
- **Embedding Table**: $V \times d_{model}$ matrix, where each row represents a unique token in the vocabulary
- **Lookup Process**: After the token enters the embedding layer the model simply "looks up" the corresponding row in this embedding table. The entire row is the vector representation for that token.

Output:

$$E \in \mathbb{R}^{n \times d_{model}}$$

V: vocabulary size
d_{model}: dimension of the model
n: number of tokens

Position Encoding

Since transformer models do not have recurrence or convolution, the order of words is not inherently captured, therefore positional encodings must be used. [9] uses sine and cosine functions to achieve this:

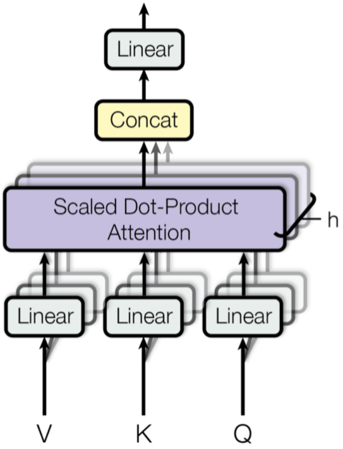
$$PE_{(\mathrm{pos},2i)} = \sin(\mathrm{pos}/10000^{2i/d_{\mathrm{model}}})$$

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

$$Input_{model} = E + EP \in \mathbb{R}^{n \times d_{model}}$$

Where *pos* is the position of the token and i is the dimension.

Encoder Attention Head



Source: [9]

Encoder Attention Head

Each encoder attention head's input consists of the *Query* (Q), *Key* (K), *Value* (V). These values are linearly transformed from the embeddings using learned weights into lower dimensions:

$$Q_i = QW_i^Q$$
$$K_i = KW_i^K$$
$$V_i = VW_i^V$$

Where Q_i, K_i, $V_i \in \mathbb{R}^{(d_{model}/h)}$,
$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$
$W_i^K \in \mathbb{R}^{d_{model} \times d_k}$
$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$
h is the number of attention heads.
i is the index of the attention head. [9]

Encoder Attention Head

- **Query (Q)**: Represents what each token attends to
- **Key (K)**: Represents what each token offers as context
- **Value (V)**: The actual content that gets aggregated

Scaled Dot-Product Attention

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

Where
$Q \in \mathbb{R}^{n \times d_k}$,
$K \in \mathbb{R}^{n \times d_k}$,
$V \in \mathbb{R}^{n \times d_v}$.
(In a transformer $d_k = d_v = d_{model}/h$ [9])

Scaled Dot-Product Attention

- $QK^T \in \mathbb{R}^{n \times n}$: Similarity between each *Query* and *Key* is computed. Result of the operation is often called the *"attention scores"*. A large positive dot product indicates high similarity or relevance between the query at position i and the key at position j. This matrix essentially tells us, for each token in the sequence (represented by a query), how much it relates to every other token in the sequence (represented by keys).
- Scaling by $\sqrt{d_k}$ $\left(\frac{QK^T}{\sqrt{d_k}}\right)$: he raw dot products can become very large in magnitude, especially with larger d_{model}. This can push the softmax function into regions where its gradients are extremely small (vanishing gradient). Dividing by $\sqrt{d_k}$ (the square root of the key dimension) normalizes these dot products, preventing them from becoming too large and ensuring more stable gradients for the softmax.

Scaled Dot-Product Attention

- Softmax function ($\mathit{softmax}(\cdot)$): The softmax function converts the scores into a probability distribution. Let $A_{ij} = \frac{(QK^T)_{ij}}{\sqrt{d_k}}$. Then, the softmax is applied row-wise: $\text{softmax}(A_i) = \frac{e^{A_{i,j}}}{\sum_{k=1}^{n} e^{A_{i,k}}}$ for each row i. Output of the $\mathit{softmax}(\cdot)$ is the *"attention weight matrix"*: $W \in \mathbb{R}^{n \times n}$.
- Multiplication with the *Value (V)* matrix ($W \cdot V$): This multiplication effectively takes a weighted sum of the *Value* vectors, where the weights are given by the attention weight matrix W. This step aggregates the information from the relevant parts of the sequence into a single context-aware representation for each position.

$$head_j = W \cdot V$$

$$head_j \in \mathbb{R}^{n \times d_v}$$

Where j is the index of the attention head.

Concatenate Outputs From Attention Heads

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \in \mathbb{R}^{n \times d_{model}}$$

where $head_j = Attention(QW_i^Q, KW_i^K, VW_i^Q) \in \mathbb{R}^{n \times (h \cdot d_k)}$,
(since $h \cdot d_k = h \cdot (d_{model}/h) = d_{model}$)
$W^O \in \mathbb{R}^{hd_v \times d_{model}}$: Learned linear projection
n: sequence length

Add and Norm Layer

The output of the Multi-Head Attention is added to its input ($Z^{(l-1)}$) and then layer normalized.

$$Z_{attention\_normalized} = LayerNorm(Z^{(l-1)} + MultiHead(\cdot)) \in \mathbb{R}^{n \times d_{model}}$$

Layer Normalization

For an input vector x (a row in the matrix), it normalizes over the features [2]:

$$\mu = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i \text{ (mean)}$$

$$\sigma^2 = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_i - \mu)^2 \text{ (variance)}$$

$$\text{LayerNorm}(x_i) = \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

γ and β are learned scaling and shifting parameters, respectively.
ϵ is a small constant for numerical stability (to prevent division by zero).

Position-wise Feed-Forward Network

This sub-layer is applied to each position independently and identically. It consists of two linear transformations with a ReLU activation in between. Let the input be $Z_{attention\_normalized}$

$$\mathrm{FFN}(Z_{attention\_normalized}) = \max(0, xW_1 + b_1)W_2 + b_2$$

- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ (e.g., 512×2048)
- $b_1 \in \mathbb{R}^{d_{ff}}$
- $max(0, \cdot)$ is the ReLU activation function
- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ (e.g., 2048×512)
- $b_2 \in \mathbb{R}^{d_{\text{model}}}$
- Let the output be $FFN_{output} \in \mathbb{R}^{n \times d_{model}}$ $\rightarrow$ This will be passed to another **Add and Norm** sublayer whose output will be passed to the next encoder layer.

Final Encoder Output

After passing through all N encoder layers, the output of the last layer ($Z^{(N)}$) is a sequence of n vectors, each of dimension d_{model}. These vectors are dense, contextualized representations of the input sequence. This final output of the encoder stack is then passed to the decoder's "encoder-decoder attention" mechanism. [9]

Decoder Layer

The decoder in the Transformer model is responsible for generating the output sequence one token at a time, incorporating context from both the input sequence (via the encoder) and the tokens it has already generated. It operates in an auto-regressive fashion.
Each of the N identical layers within the decoder stack is composed of three main sub-layers, each followed by a residual connection and layer normalization.
The main difference to the encoder layer is a new sub-layer, the multi-head self-attention.

Decoder Layer

- Let the input be $Y^{(l-1)}$ for the current decoder layer l. (output from the previous decoder layer or the combined embedding and positional encoding of the partial output sequence for the first layer)
- The encoder's final output $Z_{\text{enc}}^{(N)}$, serves as the memory for the encoder-decoder attention.
- $Y^{(l-1)} \in \mathbb{R}^{m \times d_{model}}$
- m: current length of the partial output sequence being generated
- $Z_{\text{enc}}^{(N)} \in \mathbb{R}^{n \times d_{\text{model}}}$

Masked Multi-Head Self-Attention Sub-layer

1. **Input Projections for Q, K, V:**
   - The input to this sub-layer is $Y^{(l-1)}$.
   - For each of the h attention heads ($j = 1, \ldots, h$), $Y^{(l-1)}$ is linearly projected into Query (Q_j), Key (K_j), and Value (V_j) matrices using learned weight matrices $W_j^Q, W_j^K, W_j^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (where $d_k = d_v = d_{\text{model}}/h$).
   - $Q_j = Y^{(l-1)} W_j^Q$
   - $K_j = Y^{(l-1)} W_j^K$
   - $V_j = Y^{(l-1)} W_j^V$
   - $Q_j, K_j, V_j \in \mathbb{R}^{m \times d_k}$.

Masked Attention

2. Masked Scaled Dot-Product Attention:

- The core attention calculation is performed:

$$\text{head}_j = \text{softmax}\left(\frac{Q_j K_j^T + \text{Mask}}{\sqrt{d_k}}\right) V_j$$

- **The crucial difference here is the "Mask."** This is a look-ahead mask (upper triangular matrix of negative infinities) that is applied to the attention scores before the softmax function.
- It prevents each position in the decoder from attending to subsequent positions. When predicting the i-th token, the mask ensures that the calculation for head_j only considers information from tokens at positions $1, \ldots, i-1$. This is essential for maintaining the auto-regressive property of the decoder, ensuring that predictions for future tokens do not influence current token generation.
- Everithing else works the same as in the encoder layers.
- Output: $\text{head}_j \in \mathbb{R}^{m \times d_k}$.
- $Mask \in \mathbb{R}^{m \times m}$

Decoder Multi-Head Attention

Acts as the bridge between the input sequence (processed by the encoder) and the output sequence (being generated by the decoder).
Inputs:

- $K_{\text{enc}} =$ Encoder Output
- $V_{\text{enc}} =$ Encoder Output
- $Q_{\text{dec}} =$ Output of decoder's masked self-attention layer

Encoder-Decoder Attention: Core Mechanism

- Now, the Q_{dec} (the decoder's current understanding of the partial output) is used to **query** the K_{enc} (the encoder's representation of the entire input).
- The attention mechanism calculates compatibility scores between each part of the decoder's query and all parts of the encoder's keys.
- This determines which parts of the input sentence are most relevant to predicting the *next* token in the output sequence.

Encoder-Decoder Attention: Formula

Mathematically, for each attention head j in the encoder-decoder attention:

$$\text{head}_j = \text{softmax}\left(\frac{Q_{\text{dec},j} K_{\text{enc},j}^T}{\sqrt{d_k}}\right) V_{\text{enc},j}$$

This formula is similar to the self-attention formula, but uses different sources for Q, K, and V.

- $Q_{dec,j} \in \mathbb{R}^{n \times d_k}$: Query from the masked multi-head self-attention sub-layer.
- $K_{enc,j} \in \mathbb{R}^{n \times d_k}$: Key
- $V_{enc,j} \in \mathbb{R}^{n \times d_v}$
- $head_j \in \mathbb{R}^{m \times d_v}$

Token Prediction

After the input went through all the decoder layers, the output takes the hidden vector ($h_{last} \in \mathbb{R}^{d_{model}}$ the next generated token's representation) applies a linear layer which projects h_{last} into the a vector of the vocabulary's space V_{vocab}. Afterwards the model runs *softmax* on this result which will transform the vector into a probability distribution.
Input:

- $Y_{dec}^{(N)} \in \mathbb{R}^{m \times d_{model}}$: Output of the last encoder layer.

Linear Layer

$$Logits = h_{last} W_{out} + b_{out}$$

- $W_{out} \in \mathbb{R}^{d_{model} \times V_{vocab}}$: Learned weight matric for the output linear layer.
- $b_{out} \in \mathbb{R}^{V_{vocab}}$: This is the learned bias vector for the output linear layer.
- $Logits \in \mathbb{R}^{V_{vocab}}$: The resulting vector. Each element in this vector is a raw, unnormalized score (a "logit") corresponding to a specific token in the vocabulary.

Vocabulary Probability Distribution

The *Logits* vector represents a raw likelihood values, to acquire the probabilities of these tokens we apply softmax function on the vector, which gives the probability of each token in the vocabulary.

$$P = Softmax(Logits) \in \mathbb{R}^{V_{vocab}}$$

Token Selection

Multiple strategies for choosing the next token:

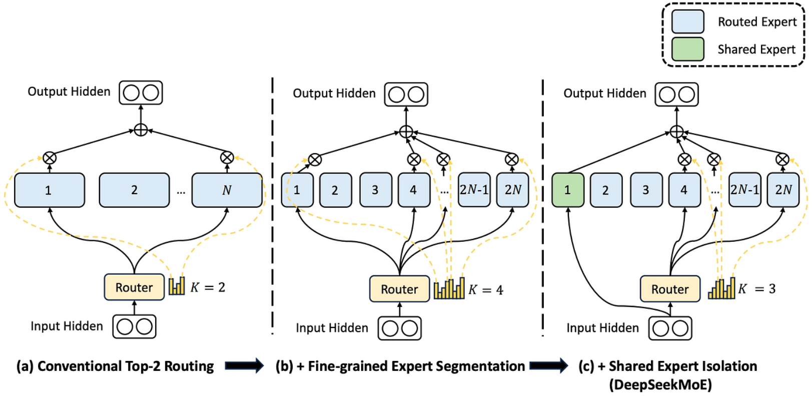
- **Greedy Decoding**: The simplest method, where the token with the absolute highest probability is always chosen. This is deterministic but can sometimes lead to sub-optimal or repetitive sequences.
- **Beam Search**: A more sophisticated method that explores multiple promising sequences simultaneously. It keeps track of the 'k' most probable partial sequences (the "beam") at each step and extends them, ultimately selecting the most probable complete sequence.[1]
- **Sampling (e.g., Top-K, Nucleus Sampling)**: Instead of always picking the highest probability token, these methods introduce an element of randomness.
  - **Top-K Sampling** involves randomly sampling from only the 'k' most probable tokens.
  - **Nucleus Sampling (Top-P Sampling)** samples from the smallest set of tokens whose cumulative probability exceeds a predefined threshold 'p'. These methods help generate more diverse and less predictable outputs, which can be desirable for creative tasks.

Problems with Traditional Transformer Architecture

Problems with regular transformer LLMs include [**shazeer2017outrageously**] [3]:

- **Limited Scalability of Total Parameters**: To increase the model's capacity the traditional way is to increase the number of parameters in the model.
- **Proportional Computational Cost Increate with Parameter Count**: The computational costs linearly increase with the parameter count, making scaling up very expensive and computationally prohibitive beyond a certain point.
- **Less Efficient Parameter Utilization and Specialization**: In a dense Transformer, every parameter in every FFN layer processes every token. This means the network has to find a "one-size-fits-all" set of parameters for all inputs. This can lead to redundancy or inefficiency.

Mixture of Experts Architecture



Source: [3]

Mixture of Experts Architecture

- **A Router (Gating Network)**: This small neural network takes the token's representation and determines which experts are most relevant for processing it.
- **A Pool of Experts**: These are typically individual FFNs, each specializing in different aspects of the input.
- **Sparse Activation**: Instead of activating all experts, the router selects and activates only a small subset (e.g., top-K) of experts for each token. Their outputs are then combined, usually as a weighted sum based on the router's scores.

[**shazeer2017outrageously**]

$$y = \sum_{i=1}^{n} G(x)_i E_i(x)$$

Mixture of Experts Architecture I

$$\mathbf{h}_t^l = \sum_{i=1}^{N} \left(g_{i,t} \mathrm{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l$$

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{Topk}(\{s_{j,t} | 1 \leq j \leq N\}, K), \\ 0, & \text{otherwise}, \end{cases}$$

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^T \mathbf{e}_i^l \right),$$

- N: Total number of experts
- $FFN_i(\cdot)$: i-th expert FFN
- $g_{i,t}$: Gate value for the i-th expert
- $s_{i,t}$: Token-to-expert affinity
- $TopK(\cdot, K)$: set comprising K highest affinity scores among those calculated for the t-th token and all N experts.

Mixture of Experts Architecture II

- $\mathbf{h}_t^l$: Output hidden state
- $\mathbf{u}_{1:T}^l \in \mathbb{R}^{T \times d_{model}}$: hidden states of all tokens after the l-th attention module
- $\mathbf{e}_i^l \in \mathbb{R}^{d_{model}}$

[3]

DeepSeek-V3's MoE Architecture

Core innovations in DeepSeek's MoE architecture [3]:

- **Fine-Grained Expert Segmentation**: DeepSeek segments each conceptual "expert FFN" into m smaller, distinct experts. This is achieved by reducing the intermediate hidden dimension of each expert's FFN to $1/m$ times its original size.
- **Shared Experts**: DeepSeek introduces a fixed number of "shared experts" that are always active.
- **Auxiliary-Loss-Free Load Balancing**: DeepSeek's router design and routing algorithm are inherently constructed to promote balanced expert utilization without needing a separate auxiliary loss term during training. This simplifies the training objective and can lead to more robust balance.

Shared Experts

Problem: In a purely sparse MoE, all experts are specialized. However, much of the knowledge required by an LLM is universal (e.g., basic grammar, common facts). Forcing specialized experts to repeatedly learn this common knowledge is redundant. [3]
The shared expert mechanism solves knowledge redundancy between experts, meaning an expers more diverse more in depth knowledge, since the shared expert takes care of common knowledge. [3]

Shared Experts

Mechanism:

1. **Always-On Activation**: For every token processed by an MoE layer, its representation is simultaneously sent to:
   - The router, which then selects a subset of "routed experts."
   - All of the pre-designated shared experts.
2. **Output Integration**: The output of the shared experts is directly added to the weighted sum of the outputs from the selected routed experts. They do not have a gating coefficient applied to their output (or effectively have a fixed coefficient of 1), as they are universally important.

Shared Experts

$$\mathbf{h}_t^l = \sum_{i=1}^{K_s} \mathrm{FFN}_i(\mathbf{u}_t^l) + \sum_{i=K_s+1}^{mN} \left(g_{i,t} \mathrm{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l$$

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{Topk}(\{s_{j,t} | K_s + 1 \leq j \leq mN\}, mK - K_s), \\ 0, & \text{otherwise}, \end{cases}$$

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^T \mathbf{e}_i^l \right).$$

- K_s: Number of shared experts
- m: Number of fine-grained experts (later)

[3]

Shared Experts Benefits Compared to Treaditional Method

- **Deeper Specialization for Routed Experts**: By offloading universal knowledge to shared experts, the vast pool of fine-grained, routed experts can truly focus on highly specific, nuanced, or conditional patterns.
- **Enhanced Model Robustness**: Shared experts ensure that fundamental capabilities and common knowledge are always present and contribute to the model's output. This makes the model more reliable and robust, especially when dealing with inputs that might not perfectly align with any specific niche expert.
- **Improved Generalization**: By consolidating general knowledge, the model can generalize better to unseen data, as the core understanding is consistently applied. [3]

Fine-Grained Expert Segmentation

Problem:

- **Coarse Granularity of Expertise**: A large FFN expert might still learn a broad range of knowledge, leading to overlap or redundancy if different experts acquire similar functionalities.
- **Limited Combinatorial Flexibility**: If you have N experts and select K of them, the number of unique combinations, while large, is constrained by the size of the individual experts. Finer distinctions in expertise are often desirable for complex tasks.

[3]

Fine-Grained Expert Segmentation

$$\mathbf{h}_t^l = \sum_{i=1}^{mN} \left(g_{i,t} \mathrm{FFN}_i(\mathbf{u}_t^l) \right) + \mathbf{u}_t^l$$

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{Topk}(\{s_{j,t} | 1 \leq j \leq mN\}, mK), \\ 0, & \text{otherwise}, \end{cases}$$

$$s_{i,t} = \mathrm{Softmax}_i \left(\mathbf{u}_t^T \mathbf{e}_i^l \right)$$

Fine-Grained Expert Segmentation

Mechanism:

- **Segmentation Factor (m)**: A chosen factor (e.g., m=4) by which the FFN's intermediate hidden dimension is divided.
- **Creating "Smaller Experts"**: If a conceptual "parent" FFN expert would have had a hidden dimension of $D_{\text{hidden\_original}}$, then DeepSeek effectively segments it into m "fine-grained experts," each of which has an intermediate hidden dimension of $D_{\text{hidden\_original}}/m$.
- **Total Expert Pool**: If the model conceptually starts with N experts, this segmentation process means the actual pool of distinct, routable experts becomes $N \times M$ fine-grained experts.

Fine-Grained Expert Segmentation I

1. **First Linear Layer + Activation**:

$$\mathbf{h}_j = \text{Activation}(\mathbf{x}W_{1,j} + \mathbf{b}_{1,j})$$

   - $\mathbf{x} \in \mathbb{R}^{D_{\text{model}}}$ is the input token representation.
   - $W_{1,j} \in \mathbb{R}^{D_{\text{model}} \times (D_{\text{hidden\_original}}/m)}$ is the weight matrix for the first linear transformation.
   - $\mathbf{b}_{1,j} \in \mathbb{R}^{(D_{\text{hidden\_original}}/m)}$ is the bias vector.
   - $\mathbf{h}_j \in \mathbb{R}^{(D_{\text{hidden\_original}}/m)}$ is the intermediate hidden state for this specific fine-grained expert.

2. **Second Linear Layer**:

$$\mathbf{y}_j = \mathbf{h}_j W_{2,j} + \mathbf{b}_{2,j}$$

Fine-Grained Expert Segmentation II

- $W_{2,j} \in \mathbb{R}^{(D_{\text{hidden\_original}}/m) \times D_{\text{model}}}$ is the weight matrix for the second linear transformation.
- $\mathbf{b}_{2,j} \in \mathbb{R}^{D_{\text{model}}}$ is the bias vector.
- $\mathbf{y}_j \in \mathbb{R}^{D_{\text{model}}}$ is the output contribution of this fine-grained expert.

3. **Gating Coeficient**: The output of each expert then multiplied by the gating coef $g_{j,t}$.

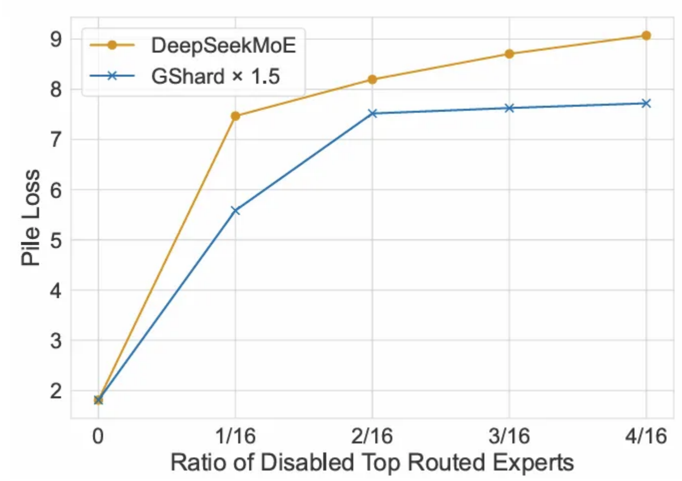
Fine-Grained Expert Segmentation Benefits

- **Increase in Combinatorial Flexibility**: Allows the model to learn highly custized groups of expertise.
- **Enhanced and Deeper Specialization**: Since each fine-grained expert is smaller it is naturally pushed to specialize in a much narrower and more specific aspect of knowledge or transformation.
- **Reduced Redundancy Across Experts**: More efficient utilization of the model's vast parameter count.
- **Effective Scaling to Unprecedented Sizes**: This mechanism is crucial for enabling the creation of models with hundreds of billions or even trillions of parameters. It allows the model to absorb and organize an immense amount of information by segmenting it into manageable, highly specialized chunks, which can then be sparsely activated.

[3] [5]

Is there truly an improvement?

The authors measured specialization by disabling the top routed experts, since experts should be less replaceble when they are more specialized $\rightarrow$ disabling the top routed expert should impact a bigger impact on performance.
They disabled top routed experts in both DeepSeekMoE and GShard x 1.5 (serving as baseline) and looked at the Pile loss at each number of disabled experts and the results suggested that disabling experts has a bigger effect on DeepSeek than on Gshard therefore DeepSeeekMoE is more specialized. [3] [5]


DeepSeekMoE
GShard × 1.5
Pile Loss
9
8
7
6
5
4
3
2
0
1/16
2/16
3/16
4/16
Ratio of Disabled Top Routed Experts

Source: [3]

References I

[1] Analytics Vidhya. *What is Beam Search in NLP Decoding?* Jan. 2025. URL: https://www.analyticsvidhya.com/blog/2025/01/beam-search-in-nlp-decoding/ (visited on 06/04/2025).

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[3] Damai Dai et al. "Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models". In: *arXiv preprint arXiv:2401.06066* (2024).

References II

[4] EITCA (European IT Certification Academy). *What are the main challenges faced by RNNs during training and how do Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) address these issues?* 2023. URL: https://eitca.org/artificial-intelligence/eitc-ai-adl-advanced-deep-learning/recurrent-neural-networks-eitc-ai-adl-advanced-deep-learning/sequences-and-recurrent-networks/examination-review-sequences-and-recurrent-networks/what-are-the-main-challenges-faced-by-rnns-during-training-and-how-do-long-short-term-memory-lstm-networks-and-gated-recurrent-units-grus-address-these-issues/ (visited on 06/04/2025).

[5] GoPubby AI. *DeepSeek-V3 Explained (2): DeepSeekMoE.* URL: https://ai.gopubby.com/deepseek-v3-explained-2-deepseekmoe-106cffcc56c1 (visited on 06/05/2025).

References III

[6] HeyCoach.in. *Recurrent Neural Networks (RNNs) and their Applications*. 2024. URL: https://blog.heycoach.in/recurrent-neural-networks-rnns-and-their-applications/ (visited on 06/04/2025).

[7] IBM. *Recurrent Neural Networks (RNNs)*. 2023. URL: https://www.ibm.com/think/topics/recurrent-neural-networks (visited on 06/04/2025).

[8] Zhijie Lin et al. “Adaptive RoI-aware network for accurate banknote recognition using natural images”. In: *Soft Computing* (May 2025), pp. 1–11. DOI: 10.1007/s00500-025-10649-1.

[9] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).