Lecture 4: Mixture of Experts (MoE)

Marshall Meng and Dylan Fang

Stanford CS336 — Spring 2025

1 Mixture of Experts (MoE): The Big Idea

Definition

A *Mixture of Experts (MoE)* layer replaces a single dense feed-forward network (FFN) in a transformer block with a bank of E parallel FFNs (*experts*) and a small *router* that selects the top-K experts for each token. Only those K experts are executed; the rest are idle (sparse activation).

- **Why**: With the same per-token FLOPs, MoEs let us increase total parameters (more experts), often improving quality while keeping runtime manageable.
- **Where**: Typically replaces the MLP/FFN sub-layer in a transformer block.
- **How**: For each token representation $\mathbf{h} \in \mathbb{R}^d$, a router outputs scores over experts; the model dispatches $\mathbf{h}$ to the top-K experts and merges their outputs.

Example

Toy picture. Think of E specialized tutors. For each word (token), a tiny selector chooses K tutors best suited for that word. Only those tutors work, saving time while using many total tutors overall.

2 Three Routing Strategies

2.1 Token-chooses-expert (Top-K)

In this strategy, each token independently looks at all experts and chooses the top-K ones with the highest router logits. This is by far the **most common approach in practice** — used in models like Switch Transformer, GShard, Mixtral, DBRX, and DeepSeek.

- Each token x_t is passed through a small router.
- That router outputs scores for each expert, e.g. [0.2, .1, 0.8, 0.3, 0.5]
- Pick the top K experts and send the token there.
- Those experts process the token and the outputs are weighted by the scores.

2.2 Expert-chooses-token

Each expert examines all tokens in the batch and selects a subset to process, such as the top-M tokens whose router scores for that expert are highest.

- Pro: Can directly control load per expert
- Con: Harder to parallelize because experts need to negotiate who gets what.

2.3 Global routing via assignment

Instead of each token or expert deciding locally, we globally optimize the assignment of all tokens to experts at once. Think of it as: "Let's solve one big optimization problem that minimizes loss and balances experts."

This is conceptually elegant but it's too slow and complex for large-scale models.

3 Top-K Routing: Step-by-Step Explanation

The goal of Top-K routing is to *route each token* (a hidden vector output from the previous Transformer layer) to a few experts out of many available ones.

This process is controlled by a small component called the **router**, which:

1. scores how suitable each expert is for the current token,
2. selects the top-K highest-scoring experts, and
3. sends the token through only those experts' feed-forward networks (FFNs).

Only the selected experts perform computation for that token, making the operation sparse and efficient.

Step 1 — Compute Similarity Scores

The router first computes how compatible each expert is with each token. Formally, for token index t and expert index i:

$$s_{i,t} = \text{Softmax}_i \left(\mathbf{u}_t^\top \mathbf{e}_i^{(l)} \right), \tag{1}$$

where:

- t — token index (which word or subword we are routing),
- i — expert index ($i \in \{1, \ldots, N\}$ for N experts),
- $\mathbf{u}_t$ — the hidden vector of token t coming from the previous transformer sublayer,
- $\mathbf{e}_i^{(l)}$ — the learned *router embedding* for expert i in layer l.

The dot product $\mathbf{u}_t^\top \mathbf{e}_i^{(l)}$ measures the similarity between the token's representation and the expert's embedding: larger values mean "this expert seems like a good match for this token."

The Softmax over all experts i turns these raw similarity scores into a probability distribution over experts:

$$\sum_{i=1}^{N} s_{i,t} = 1.$$

Intuition. Each token "looks" at all experts and decides, with some probability, which experts are most relevant for processing it.

Step 2 — Keep Only the Top-K Experts

After computing all similarity scores, we keep only the K largest ones per token, setting the rest to zero:

$$g_{i,t} = \begin{cases} s_{i,t}, & \text{if } s_{i,t} \in \text{TopK}\left(\{s_{j,t} \mid 1 \le j \le N\}, K\right), \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

The variable $g_{i,t}$ is called the **gate weight** for expert i and token t. It is nonzero only for the selected experts $\mathcal{K}_t = \text{TopK}(\mathbf{s}_{\cdot,t})$.

- The top-K experts are said to be **activated** for that token.
- The remaining $N - K$ experts are skipped, saving compute and memory.
- In some implementations (e.g., Mixtral, DBRX, DeepSeek v3), a second softmax is applied only over the selected experts for normalization.

This step is the core of **sparse routing**: only a few experts "wake up" for each token.

Step 3 — Combine Expert Outputs

Each selected expert now processes the token through its own feed-forward network (FFN). The individual expert output for token t and expert i is:

$$\mathbf{y}_{i,t} = \text{FFN}_i(\mathbf{u}_t). \tag{3}$$

The router then mixes these expert outputs back together, weighted by the corresponding gate weights $g_{i,t}$:

$$\mathbf{h}_t^{(l)} = \sum_{i=1}^{N} g_{i,t}\ \text{FFN}_i(\mathbf{u}_t) + \mathbf{u}_t. \tag{4}$$

The addition of $\mathbf{u}_t$ at the end represents a **residual connection**, which helps stabilize training (as in standard transformer blocks).

Interpretation.

- Experts perform the heavy computation (the FFN transformations).
- The router determines which experts contribute, and by how much.
- The residual adds stability and allows the MoE layer to behave like a standard MLP when routing is uncertain.

A Toy Workflow Example

To make the routing process concrete, let's walk through a toy example.

Setup. Assume we have:

- $N = 4$ experts (indexed $i = 1, 2, 3, 4$),
- $K = 2$ (only two experts are activated per token),
- one token t with hidden vector $\mathbf{u}_t \in \mathbb{R}^d$.

Each expert i has its own router embedding $\mathbf{e}_i^{(l)} \in \mathbb{R}^d$. For simplicity, imagine that the router has already computed the dot products $\mathbf{u}_t^\top \mathbf{e}_i^{(l)}$ for each expert.

Step 1 — Compute similarity scores. The raw dot products (unnormalized logits) might be:

$$\mathbf{u}_t^\top \mathbf{e}_i^{(l)} = \big[0.3,\ 1.2,\ 0.9,\ 0.4\big] .$$

After applying the softmax across all four experts:

$$s_{i,t} = \mathrm{Softmax}_i(\mathbf{u}_t^\top \mathbf{e}_i^{(l)}) = \big[0.14,\ 0.42,\ 0.31,\ 0.13\big] .$$

Interpretation: - Expert 2 is the most compatible with this token ($s_{2,t} = 0.42$). - Expert 3 is the next most compatible ($s_{3,t} = 0.31$). - Experts 1 and 4 have lower scores and will likely be skipped.

Step 2 — Keep only Top-K. We keep the $K = 2$ highest scores and zero out the rest:

$$g_{i,t} = \big[0,\ 0.42,\ 0.31,\ 0\big] .$$

Here:

- Only experts 2 and 3 are *activated*.
- Experts 1 and 4 are *inactive* (no computation, saving FLOPs).

Optionally, we can renormalize $g_{2,t}$ and $g_{3,t}$ so that they sum to 1:

$$\tilde{g}_{2,t} = \frac{0.42}{0.42 + 0.31} = 0.575, \qquad \tilde{g}_{3,t} = \frac{0.31}{0.42 + 0.31} = 0.425.$$

Step 3 — Combine expert outputs. Each active expert processes the token through its FFN:

$$\mathbf{y}_{2,t} = \mathrm{FFN}_2(\mathbf{u}_t), \qquad \mathbf{y}_{3,t} = \mathrm{FFN}_3(\mathbf{u}_t).$$

The final output (including residual connection) is:

$$\mathbf{h}_t^{(l)} = 0.575\, \mathbf{y}_{2,t} + 0.425\, \mathbf{y}_{3,t} + \mathbf{u}_t.$$

4 Training MoEs: The Main Challenge

Problem. We want *sparse* routing for efficiency, but discrete top-K choices are non-differentiable. Without any preventative measures, the Top-K routing mechanism in a MoE model leads to a problematic feedback loop. The core issue can be summarized as follows:

1. **Selective Gradient Flow.** For any given input token, the training signal (gradient) only flows to the experts that were selected by the router. Experts that were not chosen receive zero gradient updates.

2. **Uneven Learning.** Because only the active experts are updated, they continue to improve over time. In contrast, the "idle" experts that are rarely (or never) selected do not learn at all.

3. **Reinforcing Bias.** This creates a loop: experts that are initially favored by the router become progressively stronger and more likely to be chosen again in the future, while idle experts fall further behind. Over time, this imbalance worsens, leading to poor overall utilization of the expert pool and degraded model performance.

Key Equation

$$\frac{\partial \mathcal{L}}{\partial e_{\text{idle}}} = \frac{\partial \mathcal{L}}{\partial \text{Final Output}} \times \frac{\partial \text{Final Output}}{\partial (\text{Expert}_{\text{idle}}\text{'s contribution})} \times \frac{\partial (\text{Expert}_{\text{idle}}\text{'s contribution})}{\partial e_{\text{idle}}}. \quad (5)$$

The problem lies in the middle term:

$$\frac{\partial \text{Final Output}}{\partial (\text{Expert}_{\text{idle}}\text{'s contribution})} = 0.$$

4.1 Reinforcement Learning

Treat the router as a stochastic policy $\pi_\theta(e \mid \mathbf{h})$ over experts and maximize expected return. RL is principled but can be high-variance and complex in practice.

4.2 Stochastic Approximation

Stochastic approximations introduce randomness into the routing decision to help train the Mixture of Experts model more effectively. The core idea is to encourage exploration by preventing the router from always picking the same few "favorite" experts, which makes the learned experts more robust.

For each token representation x and each expert i, the router computes a noisy pre-activation score:

Key Equation

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \times \text{Softplus}\big((x \cdot W_{\text{noise}})_i\big). \quad (6)$$

- x: The input token's hidden vector representation from the previous transformer sublayer.
- W_g: The main gating weight matrix. The term $(x \cdot W_g)_i$ is the deterministic routing score (logit) for expert i.

- StandardNormal(): A random variable drawn from a standard normal (Gaussian) distribution $\mathcal{N}(0, 1)$. This introduces stochasticity into the routing decision.
- W_{noise}: A second, learnable matrix that controls how strongly noise affects each token. Intuitively, this determines how "uncertain" the router should be for a given input.
- Softplus$(\cdot)$: An activation function defined as $\text{Softplus}(z) = \log(1 + e^z)$. It ensures the noise scale is always positive.
- $H(x)_i$: The final noisy logit for expert i — a combination of a deterministic score and scaled Gaussian noise.

Note

How $\mathbf{W_{noise}}$ Learns: Controlling the "Exploration Dial" The key idea is that W_{noise} learns to control *how much randomness* the router should inject, depending on how **confident or uncertain** the router is about its own decisions.
Think of W_{noise} as a *learned exploration dial* for the router:

- **When the router is uncertain:** Suppose the initial gating scores for the top experts are very close, e.g. $[8.5, 8.4, 8.3]$. The router isn't sure which expert is truly best. In this case, adding noise encourages exploration — letting the router occasionally try a different expert. If this exploration leads to a lower loss (a good outcome), gradients flow back to *reward* the use of noise. Consequently, W_{noise} is updated to **increase noise** for similarly uncertain inputs in the future.
- **When the router is confident:** Imagine the top expert's score is clearly higher than the others, e.g. $[9.5, 6.1, 5.8]$. Here, the router is already confident about which expert is best. Adding too much noise might accidentally select a worse expert, increasing the loss. In this case, the gradient will *penalize* excessive noise, causing W_{noise} to **reduce the noise scale** for future confident inputs.

After adding noise, we still perform sparse routing by keeping only the top-K noisy scores:

Key Equation

$$G(x) = \text{Softmax}\big(\text{KeepTopK}(H(x),\, K)\big). \tag{7}$$

- KeepTopK$(H(x), K)$: This function looks at all the noisy scores in $H(x)$ and preserves only the top-K entries. For all other experts not in the top K, their logits are replaced by $-\infty$.
- Softmax$(\cdot)$: Converts the remaining logits into a probability distribution. Because $\exp(-\infty) = 0$, experts not in the top-K receive exactly 0 probability.

4.3 Heuristic balancing losses (widely used in practice)

The heuristic balancing loss solves the "rich get richer" problem by adding a penalty to the model's main loss function. This penalty increases when the workload across experts is uneven, effectively forcing the router to distribute tokens more evenly to ensure all experts are utilized and trained.

A widely used form of balancing loss comes from the **Switch Transformer** (Fedus et al., 2021). It penalizes experts that are used too heavily and encourages the router to distribute tokens more evenly. The auxiliary loss is defined as:

Key Equation

$$\text{loss} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \, P_i, \tag{8}$$

- $\boldsymbol{\alpha}$ — A small, constant hyperparameter. It acts as a *penalty knob* controlling how much weight this auxiliary loss has compared to the main training loss. A larger α means the model prioritizes expert balancing more strongly.
- $\boldsymbol{N}$ — The total number of experts in the layer.
- $\boldsymbol{f_i}$ — The **actual workload** (usage) of expert i. This measures the fraction of tokens in a batch that were actually dispatched to expert i. A high f_i means that expert i is very busy.
- $\boldsymbol{P_i}$ — The **average routing score** for expert i. This is the fraction of the router's total probability mass allocated to expert i across all tokens in the batch. A high P_i means the router is very confident about sending tokens to that expert.

The objective is to minimize the product $f_i P_i$. This product becomes large when an expert is both frequently used (high f_i) and the router is very confident about using it (high P_i). This is the exact "rich get richer" scenario. By minimizing this loss, the optimizer forces the router to a state where both the actual workload and the routing confidence are spread evenly across all N experts.

The Deepseek V1/V2 Variation

- Per-expert balancing: This is the same principle as the Switch Transformer loss described above.
- Per-device balancing: This applies the same balancing objective but aggregates it by the physical device (e.g., GPU). This is crucial for large-scale training to ensure that not only the experts are balanced, but the computational load is also evenly distributed across the hardware, preventing network bottlenecks.

The DeepSeek V3 Variation DeepSeek v3 introduced a simple yet powerful modification to the routing process: a learnable bias term b_i for each expert i. This bias is added to the router's original score for that expert before the Top-K selection is made.

Formally, the adjusted gating rule is:

$$g'_{i,t} = \begin{cases} s_{i,t}, & \text{if } s_{i,t} + b_i \in \text{TopK}(\{s_{j,t} + b_j\}, K), \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

- $s_{i,t}$: the original router score (logit) for expert i on token t.
- b_i: a new, learnable bias specific to expert i. Conceptually, it acts as a "handicap" or "head start" for underutilized experts.

- $s_{i,t} + b_i$: the adjusted score used by the router when performing Top-K selection. The router now bases its routing decision on this modified score, not on $s_{i,t}$ alone.

Note

How the bias b_i is trained (online balancing). Unlike a fixed learned bias, DeepSeek v3 continuously updates b_i *during training* through an online learning mechanism that monitors expert utilization in near real-time.
The update process works as follows:

1. During each training step, the system measures which experts are being overutilized and which are underutilized.
2. It then makes a small, immediate adjustment:
   - If an expert is underutilized, its bias b_i is **increased**, giving it a higher chance of being selected next time.
   - If an expert is overutilized, its bias b_i is **decreased**, slightly lowering its selection probability.
3. This feedback loop runs continuously, ensuring that all experts receive roughly balanced workloads over time.

5 System Considerations and Parallelism

5.1 Efficient Expert Computation: The Systems Perspective

Definition

Training Mixture-of-Experts (MoE) models is not only a problem of architecture design, but also of **systems optimization**. Because each expert is an independent feed-forward network (FFN), how we perform matrix multiplications for multiple experts directly determines the model's runtime efficiency and GPU utilization.

In practice, MoE routing enables massive parallelism but also introduces non-trivial systems challenges:

- **Imbalanced routing:** Some experts may receive many tokens while others receive few, causing load imbalance.
- **Variable tensor shapes:** Each expert's input batch size can differ per step, depending on routing.
- **Kernel efficiency:** GPUs favor large, contiguous matrix multiplications (GEMMs). Small or irregular GEMMs waste compute.

To address these, modern MoE implementations rely on increasingly sophisticated strategies for expert computation. Three common approaches are outlined below.

(A) Batched Matrix Multiplication

Example

Each expert independently processes its assigned tokens in parallel using a single `batched GEMM` kernel.

Let each expert i receive input tokens stacked as $\mathbf{X}_i \in \mathbb{R}^{B_i \times d_{\text{model}}}$, and have a feed-forward weight matrix $\mathbf{W}_i \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ffn}}}$.

We can compute all expert outputs in parallel via:

$$\mathbf{Y}_i = \mathbf{X}_i \mathbf{W}_i, \qquad \forall i \in \{1, \ldots, E\}. \tag{10}$$

These can be efficiently computed with one batched matrix multiplication:

$$\mathbf{Y} = \text{BatchedGEMM}(\{\mathbf{X}_i\}, \{\mathbf{W}_i\}). \tag{11}$$

Advantages.

- Straightforward and parallel: all experts execute in one kernel call.
- Effective when each expert has a similar input size ($B_i \approx B_j$).

Limitations.

- If some experts receive fewer tokens, GPU cores underutilize.
- All experts must have identical dimensions ($d_{\text{model}}, d_{\text{ffn}}$).

(B) Block-Diagonal Matrix Multiplication

Example

We can represent all experts' matrices as a single large **block-diagonal** matrix, so one big GEMM handles every expert simultaneously.

Define a block-diagonal weight matrix:

$$\mathbf{W}_{\text{block}} = \begin{bmatrix} \mathbf{W}_0 & 0 & 0 \\ 0 & \mathbf{W}_1 & 0 \\ 0 & 0 & \mathbf{W}_2 \end{bmatrix}. \tag{12}$$

If we stack the corresponding token inputs $\mathbf{X}_{\text{stacked}} = [\mathbf{X}_0; \mathbf{X}_1; \mathbf{X}_2]$, then the combined output is simply:

$$\mathbf{Y}_{\text{stacked}} = \mathbf{X}_{\text{stacked}} \mathbf{W}_{\text{block}}. \tag{13}$$

Interpretation. This is mathematically identical to executing all experts separately but implemented as one unified matrix multiplication kernel. Because GPUs are extremely efficient at large GEMMs, this approach can dramatically improve throughput.

Limitation. All block sizes must be equal; if some experts are underloaded, computation is wasted on zero-padding. This method also cannot easily accommodate experts with different FFN dimensions.

(C) Block-Sparse Matrix Multiplication

Example

To handle variable expert loads and sizes, we use **block-sparse matrix multiplication**, which computes only for active (non-zero) expert blocks.

Instead of a full block-diagonal matrix, we now define a sparse structure:

$$\mathbf{W}_{\text{sparse}} = \begin{bmatrix} \mathbf{W}_0 & 0 & \cdots \\ 0 & 0 & \mathbf{W}_1 \\ \vdots & & \ddots \end{bmatrix}, \tag{14}$$

where each non-zero block corresponds to an active expert for the current batch.

The computation is:

$$\mathbf{Y}_{\text{sparse}} = \mathbf{X}_{\text{packed}} \mathbf{W}_{\text{sparse}}, \tag{15}$$

where $\mathbf{X}_{\text{packed}}$ concatenates only the tokens that were actually routed to active experts.

Advantages.

- Fully supports variable token counts per expert.
- Enables heterogeneous experts (different FFN widths).
- Avoids wasting compute on inactive or empty experts.

5.2 Numerical Stability in MoE Routing

Definition

Stability problem. The MoE router applies a *softmax (soft maximum)* over expert logits to obtain gates. With low-precision arithmetic (e.g., `bfloat16 (BF16)` or `float16 (FP16)`), tiny rounding on large logits can swing the softmax output dramatically because it depends on the *log-sum-exp (LSE)* normalizer $Z = \sum_{j=1}^{N} e^{x_j}$. When the logits are large, a 0.5 rounding can shift Z enough to change the Top-K routing.

Remedies that work in practice

- **Compute the router in** `float32 (FP32)` while keeping experts in mixed precision. Only the tiny router MLP and softmax run in FP32; everything else remains BF16/FP16.
- **Normalize strictly over selected experts.** After `TopK`, renormalize the kept gates so their sum is exactly 1:

$$\tilde{g}_{i,t} = \frac{\mathbf{1}[i \in \mathcal{K}_t]\, e^{x_{i,t}}}{\sum_{j \in \mathcal{K}_t} e^{x_{j,t}}} \quad \Longrightarrow \quad \sum_{i=1}^{N} \tilde{g}_{i,t} = 1. \tag{16}$$

This keeps the *normalizer* at 1 by construction and prevents "lost mass" when dropping non-TopK experts.

- **Z-loss (logit norm regularizer).** Penalize the magnitude of the LSE normalizer to discourage extreme logits.

Key Equation

Auxiliary z-loss (zero-mean logit objective). For a batch of size B with router logits $x^{(i)} \in \mathbb{R}^N$ for example i:

$$L_z(x) = \frac{1}{B} \sum_{i=1}^{B} \left(\log \sum_{j=1}^{N} e^{x_j^{(i)}} \right)^2. \tag{17}$$

6 Issues with MoEs — Fine-Tuning Challenges

Definition

Observation. Sparse MoE models often **overfit when fine-tuned on small datasets**. Because only a few experts are active per token, fine-tuning on limited data can cause those same experts to dominate training, reducing generalization on unseen samples.

Empirical Trend

The SuperGLUE CB benchmark (Figure 1) illustrates this behavior. The blue curve (sparse model) rises rapidly on the training metric but plateaus and even declines on validation performance, indicating clear overfitting. In contrast, the dense baseline (orange) trains more slowly but maintains stable generalization.

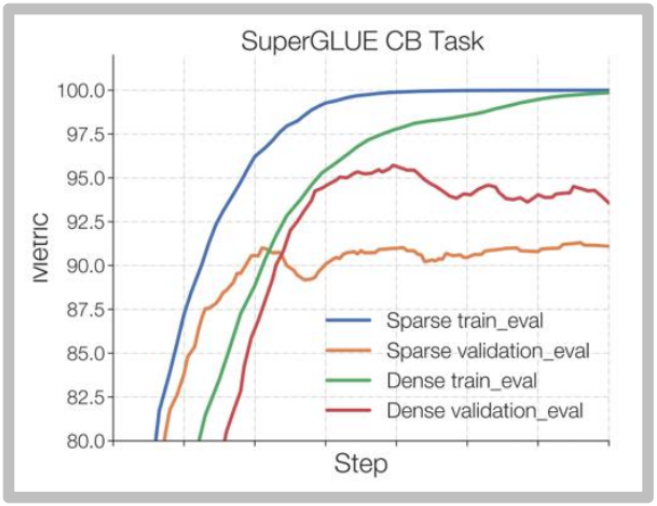

Figure 1: Sparse MoEs can overfit on smaller fine-tuning datasets. Dense models (orange) generalize better when data is limited.

Trend Summary.

- Sparse MoE models quickly minimize training loss but often fail to improve validation metrics when data is small.
- The overfitting stems from only a few experts updating during fine-tuning—effectively reducing model capacity.
- Dense FFNs (Feed-Forward Networks) spread gradients across all parameters, giving smoother adaptation.

7 Design Variations Seen in Practice

7.1 Other Training Methods — Upcycling (Detailed Guide for Students)

Goal. Turn a pre-trained *dense* Transformer into a *sparse* Mixture-of-Experts with minimal extra training. We **reuse** almost all weights, **duplicate** the MLP (feed-forward) into experts, add a small **router**, then **continue pretraining**.

Prerequisites (what we start with)

A standard dense block at layer l:

$$\text{Block}_l(x) \;=\; x \;+\; \underbrace{\text{FFN}\big(\,\text{LN}_2(x \;+\; \text{MHA}(\text{LN}_1(x)))\big)}_{\text{dense MLP}}$$

where **MHA** (multi-head attention) and **LN** (layer normalization) are already trained.

Note

We will *keep* the attention and normalization weights, and only change the MLP part into an MoE.

Step 1 — Copy shared layers

Copy parameters of $\text{LN}_1,\ \text{MHA},\ \text{LN}_2$ from the dense model to the MoE model *as is*. No changes needed here.

Step 2 — Make E expert copies of the dense MLP

Let the dense MLP be $\text{FFN}_{\text{dense}}(x) = \sigma(xW_1 + b_1)W_2 + b_2$. Create E experts:

$$\text{FFN}_i(x) \;=\; \sigma(xW_{1,i} + b_{1,i})\,W_{2,i} + b_{2,i}, \qquad i = 1, \ldots, E,$$

initialized by *copying*:

$$W_{k,i} \leftarrow W_k, \quad b_{k,i} \leftarrow b_k \quad (k \in \{1, 2\}).$$

So all experts begin identical to the dense MLP (a good warm start).

Step 3 — Add a small router (gating network)

For token hidden state $h \in \mathbb{R}^d$, compute router logits:

$$z = hW_r + b_r \in \mathbb{R}^E.$$

Select top-K experts (usually $K = 1$ or $K = 2$) and normalize gates *only* over the kept experts:

$$\mathcal{K} = \mathrm{TopK}(z, K), \qquad g_i = \frac{\exp(z_i)\,\mathbf{1}[i \in \mathcal{K}]}{\sum_{j \in \mathcal{K}} \exp(z_j)} \quad \Rightarrow \quad \sum_i g_i = 1.$$

MoE output replaces the dense MLP:

$$\mathrm{MoE}(h) \;=\; \sum_{i \in \mathcal{K}} g_i\, \mathrm{FFN}_i(h).$$

Note

Numerics. Compute router in **FP32 (float32)** and use the "subtract max + TopK + renormalize" recipe so gates sum to 1 (Section 5.2).

8 DeepSeek MoE Family (v1–v3)

Foundation Mechanism (Base Routing Logic)

DeepSeek builds upon the standard Mixture-of-Experts (MoE) routing mechanism but adds a few always-active **shared experts** for stability. The idea: most tokens go to routed experts, but every token also passes through shared experts to preserve global knowledge.

$$\mathbf{h}'_t = \mathbf{u}_t + \sum_{i \in \mathcal{S}} \mathrm{FFN}_i^{(s)}(\mathbf{u}_t) + \sum_{i \in \mathrm{TopK}(\mathcal{R}, K)} g_{i,t}\, \mathrm{FFN}_i^{(r)}(\mathbf{u}_t)$$

where:

- $\mathbf{u}_t$ — the input token's hidden vector.
- $\mathcal{S}$ — shared experts (always on).
- $\mathcal{R}$ — routed experts (selected by the router).
- $g_{i,t}$ — normalized router weights (how much token t trusts expert i).

—

DeepSeek MoE v1 — (16B total, 2.8B active)

Setup: Shared(2) + Fine-grained(64/4) experts, using **Top-K routing** and a standard **auxiliary balancing loss**.

Routing.

$$\mathbf{h}'_t = \mathbf{u}_t + \sum_{i=1}^{N_s} \mathrm{FFN}_i^{(s)}(\mathbf{u}_t) + \sum_{i=1}^{N_r} g_{i,t}\,\mathrm{FFN}_i^{(r)}(\mathbf{u}_t)$$

with gate weights:

$$g_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \mathrm{TopK}(s_{\cdot,t}, K_r), \\ 0, & \text{otherwise}, \end{cases} \qquad s_{i,t} = \mathrm{Softmax}_i(\mathbf{u}_t^\top \mathbf{e}_i).$$

- Each token computes similarity scores $s_{i,t}$ between its vector $\mathbf{u}_t$ and each expert's embedding $\mathbf{e}_i$.
- The router picks the top-K experts (highest $s_{i,t}$ values).
- $g_{i,t}$ are normalized weights controlling each selected expert's contribution.

Balancing.

$$\mathcal{L}_{\mathrm{ExpBal}} = \alpha_1 \sum_{i=1}^{N_r} f_i P_i, \quad f_i = \frac{N_r}{K_r T} \sum_t \mathbf{1}[\text{token } t \text{ selects } i], \quad P_i = \frac{1}{T} \sum_t s_{i,t}.$$

- f_i: fraction of tokens actually routed to expert i.
- P_i: average probability of selecting expert i.
- The loss penalizes overused experts to keep workload balanced across experts and devices.

—

DeepSeek MoE v2 — (236B total, 21B active)

Setup: Shared(2) + Fine-grained(160/10) experts, 6 active at once. Adds two main improvements to v1.

(1) Top-M Device Routing.

$$\text{Select devices } \mathcal{D}_t = \mathrm{TopM}(s_{\cdot,t}, M), \quad \text{then experts } \mathcal{K}_t = \mathrm{TopK}(\mathcal{R}_{\mathcal{D}_t}, K_r).$$

- Instead of sending tokens to any expert across all GPUs, tokens first choose the top-M devices with the highest routing scores.
- Only experts on these M devices are considered for Top-K selection.
- This reduces inter-device communication, improving speed and scalability.

(2) Communication Balancing Loss.

$$\mathcal{L}_{\mathrm{CommBal}} = \alpha_3 \sum_{i=1}^{D} f'_i P'_i, \quad f'_i = \frac{1}{MT} \sum_t \mathbf{1}[\text{token } t \text{ sent to device } i], \quad P'_i = \sum_{j \in \text{device } i} P_j.$$

- f'_i measures how many tokens are routed *to* device i.
- P'_i measures how confident the router is in sending to that device.
- This loss ensures each device sends and receives roughly equal data (balanced "in" and "out" traffic).

—

DeepSeek MoE v3 — (671B total, 37B active)

Setup: Shared(1) + Fine-grained(258) experts, 8 active. Builds on v2 with new routing and balancing refinements.

(1) Sigmoid–Softmax Hybrid Routing.

$$s_{i,t} = \sigma(\mathbf{u}_t^\top \mathbf{e}_i), \quad g'_{i,t} = \begin{cases} s_{i,t}, & s_{i,t} \in \text{TopK}(s_{\cdot,t}, K_r), \\ 0, & \text{otherwise.} \end{cases} \quad g_{i,t} = \frac{g'_{i,t}}{\sum_j g'_{j,t}}.$$

- Replaces pure softmax with a sigmoid gate for smoother selection.
- Prevents extreme probabilities and makes training more stable.
- After selecting top-K, the gate weights are re-normalized.

(2) Bias-based Balancing (Aux-Free).

$$g'_{i,t} = \begin{cases} s_{i,t} + b_i, & s_{i,t} + b_i \in \text{TopK}(s_{\cdot,t} + b, K_r), \\ 0, & \text{otherwise.} \end{cases}$$

- Each expert has a learnable bias b_i.
- If an expert is underused, its bias increases slightly to raise its chance of being selected.
- This replaces the auxiliary loss, allowing real-time balancing during training.

—

Summary of the Evolution

v1: Top-K routing + auxiliary load balancing (per expert + per device).
v2: Adds Top-M device routing + communication balance loss.
v3: Adds sigmoid gating + bias-based balancing (no explicit aux loss).

9 Additional Components in DeepSeek MoE v3

1. Multi-Head Latent Attention (MLA)

Concept: MLA stands for *Multi-Head Latent Attention*. It modifies the standard attention block by introducing a lower-dimensional **latent representation** for generating the Query (Q), Key (K), and Value (V) matrices.

$$\mathbf{c}_t^{KV} = W^{DKV}\mathbf{h}_t, \quad \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV}, \quad \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV}.$$
$$\mathbf{c}_t^{Q} = W^{DQ}\mathbf{h}_t, \quad \mathbf{q}_t^C = W^{UQ}\mathbf{c}_t^{Q}.$$

Explanation:

- $\mathbf{h}_t$ is the token hidden state.
- The model first projects $\mathbf{h}_t$ into a lower-dimensional latent space ($\mathbf{c}_t^{KV}$ or $\mathbf{c}_t^{Q}$).
- Then Q, K, and V are computed from these smaller latent vectors.
- This reduces memory and computation cost.

Benefits:

- When using **KV-caching** during inference, we only need to store the small latent vector $\mathbf{c}_t^{KV}$ instead of large K and V.
- The weight matrix W^{UK} can be merged into the query projection, further simplifying the architecture.
- Overall, MLA saves significant memory and bandwidth during both training and inference.

Challenge – RoPE conflict: Rotary Positional Embedding (RoPE) normally rotates Q and K directly:

$$\langle Q, K \rangle = \langle \mathbf{h} W^{Q}, W^{UK} W^{UK} \mathbf{c}_t^{KV} \rangle.$$

However, applying RoPE after the latent compression breaks positional alignment. DeepSeek solves this by keeping a few non-latent key dimensions that remain rotatable for RoPE-based position encoding.

—

2. MTP – Multi-Token Prediction

Concept: MTP (*Multi-Token Prediction*) introduces several lightweight models that predict multiple future tokens instead of just the next one. Each MTP module shares embeddings and normalization layers with the main model but predicts one or more steps ahead.

Structure: Each MTP module is an additional small Transformer block:

$$\mathbf{h}_i^k = M_k \left[\mathrm{RMSNorm}(\mathbf{h}_i^{k-1}); \mathrm{RMSNorm}(\mathrm{Emb}(t_{i+k})) \right],$$

$$\mathbf{h}_{1:T-k}^k = \mathrm{TRM}_k(\mathbf{h}_{1:T-k}^k), \quad P_{i+k+1}^k = \mathrm{OutHead}(\mathbf{h}_i^k).$$

Explanation:

- The main model predicts the next token t_{i+1}.
- Each MTP module predicts an additional token further ahead ($t_{i+2}, t_{i+3}, \dots$).
- These predictions are trained with their own cross-entropy losses $\mathcal{L}_{\mathrm{MTP}_k}$.

Benefits:

- Speeds up inference — the model can generate multiple tokens per forward pass.
- Encourages the network to model longer temporal dependencies.
- DeepSeek v3 uses one-step MTP, while extended frameworks (e.g., EAGLE) can predict several tokens ahead.

—

3. Putting It Together — What Makes DeepSeek MoE v3 Complete

- **Sparse routing (MoE):** Sigmoid + Top-K/M gating with bias-based balancing.
- **MLA:** Compresses Q/K/V projections to a lower-dimensional latent space, reducing memory cost.
- **MTP:** Adds multi-step token prediction for faster inference and richer training signals.

Together, these make DeepSeek v3 both **efficient** and **scalable**, combining sparse activation, memory compression, and multi-token prediction for state-of-the-art performance.