RankMoD: Replacing Skips with Low-Rank Computation in Mixture of Depths

Maksim Vasilev [1] **Xiuying Wei** [1] **Caglar Gulcehre** [1]

Abstract

Transformers have achieved strong performance across a wide range of tasks, but their growing size makes efficient training and inference increasingly demanding. Mixture-of-Depths (`MoD`) reduces this cost by using a per-token router to select a top-k subset of tokens for standard block computation, while the remaining tokens bypass the layer through the residual stream. In this work, we investigate this idea specifically in the feed-forward network (FFN) module, a major source of computational cost that often accounts for over 60% of model parameters and FLOPs. We argue that the all-or-nothing design of `MoD` is suboptimal for FFN computation. Lower-scored tokens may still carry useful signal, and bypassing the FFN entirely removes the hidden-state mixing that could support more effective representation learning. We propose `RankMoD`, which processes these tokens with a low-rank FFN branch while reserving the full dense FFN for the top-routed tokens. Thus, every token receives an FFN-based update, but only a subset pays the full computational cost. We further adopt self-guided training from prior work, where a dense counterpart guides the low-rank branch during early training and improves performance. Scaling-law experiments show that, under the same FLOP training budget, `RankMoD` achieves a lower validation loss than both `MoD` and the dense baseline. With 80% training FLOPs, it still outperforms. It also exhibits a steeper scaling curve than dense FFN, while `MoD` scales more slowly, suggesting a stronger computing potential for `RankMoD`.

[1]CLAIRE lab at EPFL, Lausanne, Switzerland.. Correspondence to: Maksim Vasilev <maksim.vasilev@epfl.ch>, Xiuying Wei <xiuying.wei@epfl.ch>, Caglar Gulcehre <caglar.gulcehre@epfl.ch>.

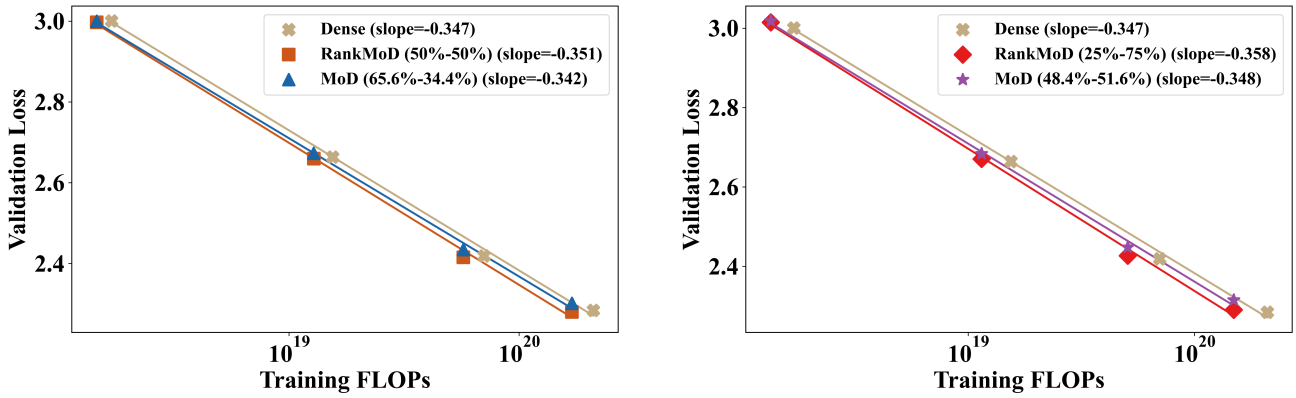
Preprint. June 8, 2026.

1. Introduction

In recent years, Transformer-based language models (Vaswani et al., 2017) have become the dominant architecture, while their model sizes have increased dramatically. This rapid scaling has imposed substantial pressure on both training budgets and downstream deployment. Each Transformer block contains two main computational components: attention and feed-forward networks (FFNs). Although attention efficiency has been extensively studied through mechanisms such as grouped-query attention, state-space models, KV caching, and sparse attention (Ainslie et al., 2023a; Gu & Dao, 2023; Tang et al., 2024; Beltagy et al., 2020), the FFN efficiency has received comparatively less diverse treatment despite its substantial cost. Most existing approaches reduce FFN computation through input-dependent sparsity, selectively activating parameters, experts, or layers according to the current token or sequence, as in mixture-of-experts (Shazeer et al., 2017) and mixture-of-depths (MoD) (Raposo et al., 2024). Some works instead investigate structured parameterizations that sparsify the FFN directly and train the resulting architecture from scratch (Dao et al., 2022a; Wei et al., 2024).

Motivated by the dominant cost of FFNs, we focus on improving their efficiency in this paper. A natural adaptation of MoD is to equip the feed-forward module with a per-token router, which selects the top-β fraction of tokens, e.g., 50%, for complete computation. The remaining $(1-\beta)$ fraction is carried forward only through the residual stream, without passing through the module's nonlinear transformation. Although this strategy substantially reduces computation, it assigns zero feed-forward computation to all tokens below the routing threshold. We argue that this design is overly restrictive: these tokens may be less critical, but they are not necessarily irrelevant, and excluding them entirely removes any opportunity for intermediate feature mixing.

Motivated by this limitation of `MoD`, we propose `RankMoD`, a low-rank token-adaptive feed-forward module that preserves feed-forward transformation for all tokens while reducing computation. Rather than completely bypassing the low-scoring tokens, `RankMoD` processes them with a lightweight low-rank FFN that factorizes each dense projection into two smaller matrices, while applying the original



(a) 50% — 50% mode

(b) 25% — 75% mode

Figure 1. **Scaling curves.** Validation loss as a function of training FLOPs for `Dense`, `MoD`, and `RankMoD` under two routing settings: (a) 50%–50% mode and (b) 25%–75% mode. Under the same training FLOPs, **`RankMoD` achieves the lowest validation loss**, followed by `MoD` and then `Dense`. We also report the fitted scaling slope for each method; **the steeper slope of `RankMoD` indicates stronger compute scaling, while `MoD` shows weaker scaling than the dense baseline** .

dense FFN only to top-routed tokens. As a result, every token receives an FFN-based update, while only a selected fraction incurs the full dense FFN cost. To improve the training of low-rank parameterization, we further adopt the guided training strategy from (Wei et al., 2024), where a dense guide layer supervises the low-rank parameterization during the early stage of training.

We evaluate `RankMoD` through scaling-law experiments on the FineWeb-Edu dataset (Penedo et al., 2024), training models from scratch with sizes ranging from 110M to 1.3B parameters. Our experiments compare `RankMoD` against dense baselines of FFN and `MoD` under matched FLOP training budgets. Across model scales, `RankMoD` consistently achieves lower validation loss than `MoD` and the dense baseline under the same training compute budget. Moreover, the fitted scaling curves show that `RankMoD` exhibits a steeper loss–compute trend than the dense baseline, whereas `MoD` scales more slowly. In particular, when routing 50% of tokens to the lightweight module, the fitted power-law curve gives an absolute slope of 0.351 for `RankMoD`, compared with 0.347 for the dense baseline and 0.342 for `MoD`. These results suggest that preserving lightweight feed-forward transformations for low-scoring tokens leads to better compute efficiency and stronger scaling potential. Our further experiments show that self-guided training can further improve `RankMoD`, reducing validation loss by around 0.01 with only a marginal increase in training FLOPs, especially in more challenging settings. It also achieves lower loss and better scaling under the same model FLOPs, which are directly tied to per-token inference efficiency.

1. We propose `RankMoD`, a low-rank parameterization to replace the skipped path in `MoD` for feedforward networks. `RankMoD` provides low-scoring tokens with a lightweight transformation, while only top-routed tokens pay the full FFN cost.
2. We conduct scaling-law experiments following the Chinchilla setting, with models up to 1.3B parameters. We show that `RankMoD` achieves lower validation loss than both the dense baseline and `MoD` across model scales under the same training-FLOP budget. Moreover, the fitted compute–loss slope is steeper for `RankMoD` than for the dense baseline, while `MoD` scales more slowly.
3. We further investigate `RankMoD` with a self-guided training scheme (Wei et al., 2024), which uses a dense parameterization to guide the low-rank matrices during the early stage of training. We conduct extensive ablations to support the effectiveness of the current design in our low-rank mixture-of-depths method.

2. Method

To motivate our design, we first describe how `MoD` can be applied to FFNs and discuss its limitations. We then introduce `RankMoD` as a lightweight solution to address the problem, followed by the usage of self-guided training strategy used to improve low-rank optimization.

2.1. Mixture of Depths

In Transformers, a standard FFN module is defined as

$$\mathrm{FFN}(x) = W_2\, f(W_1 x), \tag{1}$$

where $x \in \mathbb{R}^N$ is the hidden state, $W_1 \in \mathbb{R}^{M\times N}$ and $W_2 \in \mathbb{R}^{N\times M}$ are dense projection matrices, and $f(\cdot)$ is the activation function.

FFN layers account for a large fraction of model parameters, often exceeding 60% in earlier GPT and LLaMA-style (Rad-

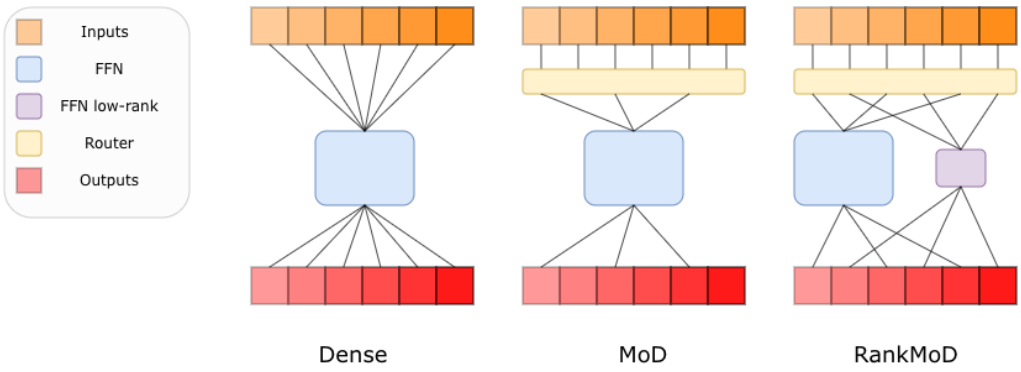


Figure 2. **Architecture designs:** FFN sublayer architectures. Dense (left) applies the full FFN to every token. `MoD` (middle) routes a $\beta\%$ of tokens through the FFN and skips the rest. `RankMoD` (right) replaces `MoD`'s skip path with a low-rank FFN, so every token receives some computation while only $\beta\%$ tokens pay the full FFN cost.

ford et al., 2019; Touvron et al., 2023b) architectures, and their cost further increases in recent models that adopt sparse expert layers (Shazeer et al., 2017). As a result, FFNs are among the most expensive components in terms of both FLOPs and parameters.

To reduce this cost, prior work on Mixture-of-Depths (MoD) skips Transformer block computation for a subset of tokens (Raposo et al., 2024). Following this idea, we adapt the `MoD` layer that reduces only FFN computation by routing a fraction of tokens through Equation (1). A per-token router $r_\theta : \mathbb{R}^N \to \mathbb{R}$ produces a scalar score $s_t = r_\theta(x_t)$. Only the top $\beta\%$ tokens according to this score, with $P_\beta(s)$ denoting the corresponding percentile threshold, are passed through the FFN and gated by $\sigma(s_t)$. Tokens below the threshold bypass the FFN transformation and are carried forward solely through the residual shortcut.

$$\mathrm{MoD}(x_t) = \begin{cases} \sigma(s_t)\,\mathrm{FFN}(x_t), & s_t > P_\beta(s), \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

The outer residual connection of the Transformer ensures that skipped tokens still propagate through the network, while the multiplicative gate $\sigma(s_t)$ enables gradients to flow into the router despite the discrete top-k selection.

`MoD` allocates FFN compute in a binary manner: a token either receives the full FFN update or bypasses the computation entirely. We argue that this all-or-nothing design can be suboptimal. Although the bottom $1-\beta$ tokens may be less demanding, they are not necessarily unimportant; skipping them removes any opportunity for lightweight refinement and hidden-state mixing within the layer. To address this limitation, we replace the skip path with a cheap, parameter-efficient FFN, ensuring that every token is updated by an FFN module while preserving most of the compute savings.

2.2. `RankMoD`

Instead of leaving low-scored tokens unchanged as in Equation (2), we process them with a lightweight approximation of the FFN, while reserving the full FFN for top-routed tokens. Motivated by the effectiveness of low-rank parameterization for approximating dense linear layers in efficient LLM training and finetuning (Hu et al., 2021; Wei et al., 2024), we propose `RankMoD`. In `RankMoD`, the lightweight branch uses a low-rank FFN: each dense transformation in the original FFN, W_1 and W_2, is factorized into two smaller matrices, yielding a cheaper approximation of the full FFN.

Technically, at each feed-forward block, a single per-token router assigns each token a score $s_t = r_\theta(x_t)$, and we calculate a $P_\beta(s)$ percentile over all of the tokens. Tokens with $s_t > P_\beta(s)$ are processed by the regular FFN, while tokens with $s_t \le P_\beta(s)$ are processed by a low-rank $\mathrm{FFN}_{\mathrm{lr}}$ with R as the rank in each parameterization:

$$\texttt{RankMoD}(x_t) = \begin{cases} \sigma(s_t)\,\mathrm{FFN}(x_t) & s_t > P_\beta(s), \\ \mathrm{FFN}_{\mathrm{lr}}(x_t) & s_t \le P_\beta(s), \end{cases} \tag{3a}$$

$$\mathrm{FFN}_{\mathrm{lr}}(x_t) = B_2 A_2\, f(B_1 A_1 x_t), \tag{3b}$$

Low-rank factorization reduces both parameters and computation, while retaining a structured approximation to the dense transformation. The dense projection $W \in \mathbb{R}^{M\times N}$ costs $\mathcal{O}(MN)$ per token, while the low-rank factorization costs only $\mathcal{O}(R(M+N))$, with $R \ll \min(M,N)$. We now derive the expected per-token FLOPs for the different FFNs. The dense original has $2\cdot N\cdot M$ with M usually set as $4\cdot N$. `MoD` and `RankMoD` have the following:

$$\texttt{MoD}: \beta\cdot 2\cdot N\cdot M$$
$$\texttt{RankMoD}: \beta\cdot 2\cdot N\cdot M + (1-\beta)\cdot 2\cdot (N+M)\cdot R$$

Therefore, RankMoD interpolates between standard MoD (when $R=0$) and dense FFN (when $\beta=1$).

The FLOP formula interpolates between MoD ($R=0$) and a dense FFN ($\beta = 1$): every token receives an FFN-style update, but only top-routed tokens pay full price. One detail distinguishes the low-rank branch $B_2A_2f(B_1A_1x)$ from a parameter-matched smaller-dense FFN: the activation f is still applied at the full intermediate width $4N$, with the rank-R bottleneck only on the projections, whereas the smaller-dense branch applies f at a narrower intermediate width. Section 3.3 finds the low-rank branch slightly outperforms this alternative at matched parameters and FLOPs, suggesting the wider intermediate is the better use of the budget. RankMoD can thus be viewed as a two-expert module — a full-rank expert paired with a low-rank approximation — with the router deciding which path each token takes.

2.3. Self-guided training (SG)

As shown in (Wei et al., 2024), low-rank parameterizations can suffer from optimization difficulties when trained from scratch, such as slower convergence in the early training stage, especially for small rank values. Since it is designed to approximate a heavier dense transformation, we can use its dense counterpart to guide the optimization, following the idea of self-guided training in (Wei et al., 2024). Concretely, for a low-rank parameterization $B(Ax_t)$, we introduce a dense guidance matrix G with matching input and output dimensions. During the first 50% of training, we blend the dense and low-rank outputs as

$$\alpha G x_t + (1-\alpha)B(Ax_t),$$

where α follows a cosine annealing schedule and gradually decays to zero. After this stage, the dense guidance path is removed and only the low-rank matrices are used. We further adopt the reduced-FLOPs variant. At each forward pass, we sample $\xi \sim \mathrm{Uniform}(0,1)$ independently for each layer and apply the blended output only when $\xi < \alpha$; otherwise, we return the low-rank output alone:

$$\hat{y}_t = \begin{cases} \alpha G x_t + (1-\alpha)B(Ax_t), & \xi < \alpha, \\ B(Ax_t), & \text{otherwise.} \end{cases} \tag{4}$$

This stochastic guidance increases the training FLOPs of the `RankMoD` branch by only 25% and it does not affect inference FLOPs.

We further ablate several guidance strategies for low-rank parameterization in Section B, and find that the current self-guidance variant performs best.

3. Experiments

3.1. Set up

We use decoder-only Transformer language models as our main experimental backbone. The models largely follow a standard GPT-style architecture (Vaswani et al., 2017; Radford et al., 2019), except that we adopt rotary positional embeddings (Su et al., 2024) and the LLaMA tokenizer (Touvron et al., 2023a). Each Transformer block contains a self-attention module followed by a feed-forward network (FFN), implemented with two linear layers and a GELU activation. We evaluate four model scales: Transformer-S (110M), Transformer-M (335M), Transformer-L (729M), and Transformer-XL (1.3B). The corresponding dense Transformer models are denoted as `Dense` in our tables. For `MoD`, we apply token-level sparsification only to the FFN modules: at each layer, a fixed ratio of tokens skips the FFN computation. For `RankMoD`, tokens routed to the low-rank branch are processed by a low-rank FFN. We choose the rank such that the low-rank branch contains approximately 32% of the dense FFN parameters. The corresponding ranks are 192, 256, 384, and 512 for the four model sizes, respectively. The low-rank matrices are initialized with SVD. We always keep the first layer as dense in all experiments. We align compute budgets of `RankMoD` and `MoD` experiments, so they stay the same.

We train on the FineWeb-Edu dataset (Penedo et al., 2024). We randomly reserve 0.5B tokens for validation and use the remaining tokens for training. Following the Chinchilla scaling law (Hoffmann et al., 2022), each baseline model is trained with a token budget equal to 20 times its parameter count. Hyperparameters such as learning rate and global batch size are set following recent scaling-law studies (Gu & Dao, 2023; Zhang et al., 2022; Wei et al., 2024). We report two types of computational cost. First, *training FLOPs* measure the total compute consumed during pretraining. For routed models, this quantity is computed using the number of processed training tokens and the effective computation activated by the routing decisions, rather than assuming dense computation for every token. Second, *model FLOPs* measure the per-token forward-pass cost and are computed from the fraction of tokens routed through each computational branch. Training FLOPs therefore reflect total training efficiency, while model FLOPs are more directly related to inference speed. Additional implementation details are provided in Section A. Owing to the substantial compute required to train across this range of FLOP budgets, all scaling-law results are reported from a single seed per configuration.

3.2. Main results

Scaling law studies Table 1 compares dense Transformers, MoD-style FFN skipping, and our low-rank FFN routing variants across different model scales. Overall, replacing skipped FFN computation with a low-rank branch provides a consistently better compute-performance trade-off than directly skipping FFNs. In particular, the 50%-50% `RankMoD` setting achieves lower validation loss than the

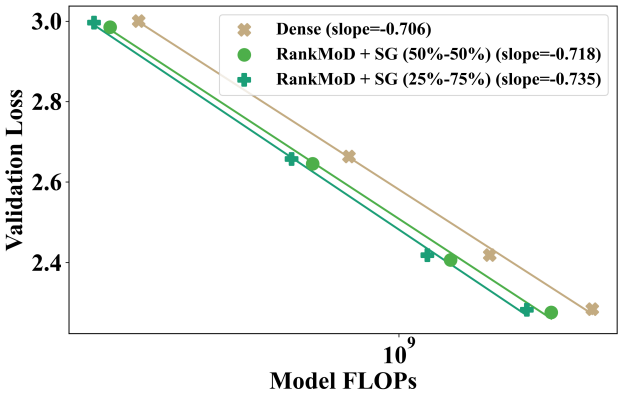


Figure 3. **Scaling curves.** Validation loss as a function of *model* FLOPs for `Dense` and `RankMoD` with *self-guidance* under two routing settings: 50%–50% mode and 25%–75% mode. Under the same forward pass FLOPs, **`RankMoD` achieves the lowest validation loss with steeper slopes**.

dense baseline at every model scale, while using fewer training FLOPs. For example, at the 1.3B scale, `RankMoD` reduces the validation loss from 2.2840 to 2.2799, while decreasing training FLOPs from 2.102×10^{20} to 1.697×10^{20}. In contrast, both `MoD` with the sa variants at the 1.3B scale underperform the dense baseline under the same budget as `RankMoD`.

We further analyze the fitted scaling trends in Figure 1. The fitted slope of `RankMoD` is slightly steeper than that of `Dense` (-0.347 vs -0.351), while the `MoD` curve is flatter (-0.347 vs -0.351). This indicates that `RankMoD` achieves a more favorable scaling trend, whereas pure FFN skipping provides a weaker compute-quality trade-off. Extrapolating these slopes, `RankMoD`'s compute-efficiency margin over `Dense` should widen at larger scales, while `MoD` gap to `Dense` is expected to grow rather than close. Combined with the absolute-loss results above, this supports low-rank routing — rather than pure FFN skipping — as the more scalable form of conditional FFN computation.

Measured in forward-pass FLOPs Figure 3, RankMoD+SG dominates Dense at every scale with steeper slopes (-0.718 / -0.735 vs -0.706), and the 25%–75% setting — which routes more tokens through the low-rank branch — exhibits the strongest scaling, showing that the inference-cost advantage of low-rank routing widens with model size.

Self-guided training Since the low-rank branch has fewer parameters than the dense FFN, its optimization may become a bottleneck when a large fraction of tokens is routed to it. We therefore adopt the existing self-guided training strategy to provide additional supervision from a dense counterpart, encouraging the low-rank branch to better approximate the behavior of the full FFN. As shown in Table 1, self-guided training improves `RankMoD` across most model scales and routing ratios. The gains are particularly visible when the low-rank branch is used more heavily, such as in the 25%-75% configuration, and in the 50%-50% configuration at smaller model scales. Self-guided training introduces a slight increase in training FLOPs, but does not change the model FLOPs at inference time. Therefore, `RankMoD` +SG preserves the inference-time efficiency of `RankMoD` and still appears lower training FLOPs than `Dense` while achieving the best overall validation loss across all reported settings. We also ablate other self-guidance schemes in Section B and show that this scheme provides the best performance out of the proposed variants.

Downstream performance We benchmark the 1.3B-parameter models on a standard suite of zero-shot tasks—Hellaswag (Zellers et al., 2019), ARC (Clark et al., 2018), PIQA (Bisk et al., 2020), Winogrande (Sakaguchi et al., 2021), BoolQ (Clark et al., 2019), and LAMBADA (Paperno et al., 2016)—using the LM-Evaluation-Harness; results are reported in Table 5. The main takeaway is that `RankMoD` tracks the dense baseline closely on accuracy: at 50%–50% it averages 51.81% across the seven benchmarks against 51.91% for dense—within noise—and outperforms dense outright on Hellaswag (47.37 vs. 47.19), ARC-C (33.53 vs. 31.74), and LAMBADA accuracy (40.87 vs. 40.35). Tightening the routing budget to 25%–75% costs only about a point of average accuracy (51.00%), so the architecture stays near-dense even when just a quarter of tokens take the full-rank path. The token-dropping `MoD` baseline, by contrast, fails to reach dense parity at either of the routing budgets we tested (50.84% at 65%–35% and 49.62% at 48%–52%), suggesting that the low-rank "skip" branch contributes useful signal that hard token-dropping throws away. Self-guided initialization (SG) preserves `RankMoD`'s downstream profile, scoring 51.26% and 50.98% at the two budgets.

3.3. Ablation studies

Routing mechanism `RankMoD` uses a small learned router to decide which tokens take the full-rank FFN path and which take the low-rank branch. To verify that the router actually learns a useful token-level distinction—rather than the model effectively averaging two parallel paths regardless of which token sees which—we replace its trained decisions with a random token-to-path assignment at the same active fraction, holding the rest of the architecture and training recipe fixed. Table 3 reports the result at two scales: random routing loses roughly 2 PPL on both Transformer-S and Transformer-M, with the gap essentially identical across scale. The router therefore learns a meaningful token-level partition: *which* tokens are routed to the full-rank path matters, not just how many. We also ablate usage of sigmoid on a low-rank path in Section C.

Table 1. **Validation loss and perplexity across model scales and routing budgets.** `Dense`, `MoD`, `RankMoD`, and `RankMoD` +SG (self-guided) trained on a Chinchilla 20× token budget over FineWeb-Edu and evaluated on a 0.5B-token held-out split. **FFN routing** is the per-token top%–bottom% split between the full FFN and the secondary path (a skip for `MoD`, a low-rank FFN with ≈32% of the dense parameters for `RankMoD`). **FFN / Model / Training FLOPs** are, respectively, the per-token FFN cost, the per-token full-Transformer forward cost, and the total pretraining compute under the realized routing. At every scale, `RankMoD` beats `Dense` at fewer Training FLOPs and beats `MoD` at matched routing; self-guided training improves it further, with 50%–50% `RankMoD + SG` the best configuration throughout.

Architecture	FFN routing	FFN FLOPs	Model FLOPs	Training FLOPs	Loss	PPL
`Dense` (110M)	100%	1.132e+8	2.568e+8	1.695e+18	3.0007	20.24
`MoD`	65.6%-34.4%	7.755e+7	2.211e+8	1.459e+18	3.0001	20.11
`MoD`	48.4%-51.6%	5.97e+7	2.032e+8	1.341e+18	3.0185	20.46
`RankMoD`	50%-50%	7.758e+7	2.211e+8	1.459e+18	2.9967	20.02
`RankMoD`	25%-75%	5.974e+7	2.033e+8	1.341e+18	3.0149	20.39
`RankMoD` + SG	50%-50%	7.758e+7	2.211e+8	1.545e+18	2.9851	19.79
`RankMoD` + SG	25%-45%	5.974e+7	2.033e+8	1.470e+18	2.9963	20.01
`Dense` (335M)	100%	4.027+e8	7.702e+8	1.548e+19	2.6637	14.34
`MoD`	65.6%-34.4%	2.7e+8	6.375e+8	1.281e+19	2.6728	14.48
`MoD`	48.4%-51.6%	2.036e+8	5.711e+8	1.148e+19	2.6838	14.64
`RankMoD`	50%-50%	2.701e+8	6.376e+8	1.282e+19	2.6597	14.29
`RankMoD`	25%-75%	2.037e+8	5.713e+8	1.148e+19	2.6708	14.45
`RankMoD` + SG	50%-50%	2.701e+8	6.376e+8	1.378e+19	2.6456	14.09
`RankMoD` + SG	25%-75%	2.036e+8	5.713e+8	1.294e+19	2.6571	14.25
`Dense` (729M)	100%	9.060e+8	1.608e+9	7.034e+19	2.4187	11.23
`MoD`	65.6%-34.4%	6.073e+8	1.310e+9	5.728e+19	2.4344	11.41
`MoD`	48.4%-51.6%	4.580e+8	1.160e+9	5.075e+19	2.4482	11.57
`RankMoD`	50%-50%	6.076e+8	1.310e+9	5.729e+19	2.4150	11.19
`RankMoD`	25%-75%	4.584e+8	1.161e+9	5.077e+19	2.4269	11.32
`RankMoD` + SG	50%-50%	6.076e+8	1.310e+9	6.204e+19	2.4063	11.09
`RankMoD` + SG	25%-75%	4.584e+8	1.161e+9	5.789e+19	2.4177	11.22
`Dense` (1.3B)	100%	1.611e+9	2,748e+9	2.102e+20	2.2840	9.82
`MoD`	65.6%-34.4%	1.08e+9	2.217e+9	1.696e+20	2.3018	9.99
`MoD`	48.4%-51.6%	8.143e+8	1.953e+9	1.493e+20	2.3150	10.12
`RankMoD`	50%-50%	1.080e+9	2.218e+9	1.697e+20	2.2799	9.78
`RankMoD`	25%-75%	8.148e+8	1.953e+9	1.494e+20	2.2900	9.87
`RankMoD` + SG	50%-50%	1.080e+9	2.218e+9	1.844e+20	2.2756	9.73
`RankMoD` + SG	25%-75%	8.148e+8	1.953e+9	1.715e+20	2.2825	9.80

Table 2. Comparison between `RankMoD` and a compact dense FFN branch in the second branch. `RankMoD` slightly outperforms the compact dense alternative, since this parameterization approximates the behavior of a larger dense FFN. Moreover, the performance of `RankMoD` can be further improved by SG.

Method	FFN Routing	Loss	PPL
Transformer-L			
`RankMoD`	25%–75%	2.4269	11.32
`RankMoD` +SG	25%–75%	**2.4177**	**11.22**
smaller dense	25%–75%	2.4306	11.36
Transformer-XL			
`RankMoD`	25%–75%	2.2900	9.87
`RankMoD` +SG	25%–75%	**2.2825**	**9.80**
smaller dense	25%–75%	2.2996	9.97

Usage of `RankMoD` A natural alternative to `RankMoD`'s low-rank branch is to keep the standard dense FFN shape on the unselected tokens but to shrink its hidden width. We call this baseline *smaller dense*: the rank-r pair (A, B) is replaced by a full-rank FFN whose intermediate width is chosen to match the FLOP and parameter budget of the low-rank branch. Everything else — the router, the gate, and the 25%–75% split — is unchanged. Table 2 shows that `RankMoD` is slightly ahead on both scales. The intuition is that a low-rank pair acts as the top-R approximation of a *full-width* FFN, while the smaller-dense branch is locked into a narrower hidden space from the start; given the same compute, approximating the wide FFN is the better use of it. Self-guided training further widens the gap, the dense guide pulls (A, B) toward the subspace of an unconstrained FFN that the smaller-dense branch cannot reach by construction.

Self-guidance techniques. We also ablate different self-guidance techniques which don't use extra parameters during the self-guidance phase. Exact definitions can be found in Section B. Table 4 gives us two considerations. First, both parameter-free recipes (Warm; SG with $G \equiv W_i$) perform almost at the same level the no-SG baseline — within

Table 3. Effect of the learned router. Replacing the trained router with random token-to-path assignment degrades validation PPL $\sim$2 at both Transformer-S and Transformer-M, indicating that the router learns a meaningful token-level partition rather than collapsing to a noisy or uniform assignment.

Method	FFN Routing	Loss	PPL
Transformer-S			
`RankMoD`	50% - 50%	3.0007	20.24
`RankMoD` +RandomRouting	50% - 50%	3.11827	22.6
Transformer-M			
`RankMoD`	50% - 50%	2.6637	14.34
`RankMoD` +RandomRouting	50% - 50%	2.7854	16.21

Table 4. Validation loss and perplexity on a 0.5B-token validation set for `RankMoD` at the 50%–50% routing setting under three optimization recipes. All runs use Transformer-M $R = 256$, trained on FineWeb-Edu for 6.7B tokens with AdamW at peak learning rate 3×10^{-4} on a cosine schedule. Best in **bold**.

Method	Recipe	Loss	PPL
`RankMoD`	—	2.6597	14.29
`RankMoD` + Warm	$\beta : 1 \to 0.5$ over 50% of steps	2.6574	14.26
`RankMoD` + SG, $G \equiv W_i$	$\alpha : 1 \to 0$ cosine, 50%	2.6585	14.28
`RankMoD` + SG, separate G	$\alpha : 1 \to 0$ cosine, 50%	**2.6456**	**14.09**

seed noise for this setup. Second, the canonical self-guided recipe with a separate G improves the loss by an order of magnitude larger gap. We read this as evidence that the gain from self-guidance is bound to the *extra parameter* G acting as an independently optimized smoothing surrogate for the structured pair: once the G is parameter-tied to the dense path or replaced by a curriculum that exposes only the dense path early, the benefit collapses to noise.

4. Related work

4.1. Mixture-of-Depths and conditional FFN compute

Conditional computation has become an increasingly popular approach for improving the efficiency of large models (Fedus et al., 2022; Shazeer et al., 2017). Rather than applying the same computation uniformly to all tokens, they dynamically allocate computation according to token, input, or layer difficulty.

In this work, we focus on Mixture-of-Depths (Raposo et al., 2024), a conditional-depth method. Instead of changing which expert parameters are used, as in MoE, Mixture-of-Depths changes whether each token is processed by a given Transformer block. A per-token capacity-based router selects a top-k subset of tokens for block computation, while the remaining tokens bypass the block through the residual stream. Building on this idea, follow-up work has extended MoD-style routing to parameter-shared recursive Transformers (Bae et al., 2025) and to vision tokens in video-language models (Wu et al., 2024). In this paper, given the large amount of FFN prameters and FLOPs, we investigate the MoD idea only skipping FFN module. Mixture-of-Depths belongs to a longer line of work on token- and layer-adaptive computation in transformers. Early-exit methods such as CALM (Schuster et al., 2022) and LayerSkip (Elhoushi et al., 2024) exit the transformer stack at intermediate layers on a per-token basis, while Universal Transformers (Dehghani et al., 2018) and PonderNet (Banino et al., 2021) use a learned halting mechanism to vary the number of iterative computation steps. These methods share with MoD the premise that uniform compute allocation is wasteful; they differ in granularity— exiting the layer stack early, halting iterative refinement, or skipping a single block.

The closest structural neighbor to `RankMoD` is CoLT5 (Ainslie et al., 2023b), which adds an always-on light branch to a top-k heavy branch in an encoder; `RankMoD` instead routes a decoder-only LM, partitions tokens strictly between the two paths, and uses a low-rank (rather than small-dense) light branch trained with self-guidance (Section 2.3), which we confirm in Table 2.

4.2. Structured and low-rank FFN parametrizations

The cheap branch of `RankMoD` is a low-rank FFN, building on a body of work showing that FFN weights tolerate aggressive structured compression. LoRA (Hu et al., 2021) established that low-rank updates capture much of the expressivity of full fine-tuning at a small fraction of the parameter cost. Structured parametrizations such as Monarch (Dao et al., 2022a) investigates other sub-quadratic factorizations like butterfly matrices. (Wei et al., 2024) compares among different structured parameterization and finds that the low-

Table 5. Zero-shot performance on lm-evaluation-harness for the 1.3B-parameter models. Accuracy (%) for all tasks; Avg. is the mean over Hellaswag, ARC-E, ARC-C, PIQA, Winogrande, BoolQ, and LAMBADA. Note that MoD and RankMoD all use fewer training FLOPs than Dense.

Architecture	FFN routing	Training FLOPs	Hellaswag	ARC-E	ARC-C	PIQA	Winogrande	BoolQ	LAMBADA	Avg.
Dense (1.3B)	100%	2.102e+20	47.19	59.18	31.74	69.70	54.14	61.07	40.35	51.91
MoD	65%-35%	1.696e+20	46.25	58.33	31.06	69.59	53.59	57.77	39.26	50.84
MoD	48%-52%	1.493e+20	45.60	54.55	29.27	68.50	53.04	56.91	39.49	49.62
RankMoD	50%-50%	1.697e+20	47.37	58.46	33.53	69.42	53.67	59.33	40.87	51.81
RankMoD	25%-75%	1.494e+20	46.15	56.86	30.89	68.61	53.75	60.73	39.98	51.00

rank matrices are still the most effective one. Futhermore, it proposes using a guide-based optimization scheme that we also adopt for RankMoD (Section 2.3). Compared to them, RankMoD does not commit to a single parametrization for the entire FFN: the low-rank branch serves only as a token-conditional fallback, while router-selected tokens retain a full dense computation.

5. Conclusion

In this work, we study conditional FFN computation in Transformers through the lens of Mixture-of-Depths. While MoD reduces computation by routing only a subset of tokens through the FFN, its binary design leaves the remaining tokens without an FFN-based update. To address this limitation, we propose RankMoD, a two-branch FFN module that combines a full-rank FFN for top-routed tokens with a low-rank FFN for the remaining tokens. Our scaling-law experiments show that RankMoD achieves lower validation loss than both MoD and the dense baseline under the same training FLOP budget. Moreover, RankMoD exhibits a steeper scaling curve, suggesting stronger compute-scaling potential. These results indicate that replacing hard skipping with lightweight low-rank computation is a promising direction for efficient Transformer FFN design.

Limitations. This work has several limitations. First, our experiments focus on scaling-law behavior under controlled training settings. Although these experiments provide evidence that RankMoD improves compute efficiency, further evaluation on larger-scale models is needed to confirm its practical benefits across broader deployment scenarios. Second, although we explore different token routing ratios for both the baselines and RankMoD, we mainly set the low-rank branch to use 32% of the original dense FFN parameters. A very small rank may limit the expressive capacity of the lightweight branch, while a larger rank reduces the computational savings. A more systematic study of rank allocation across layers or training stages could further improve the trade-off between performance and efficiency. Third, the self-guided training strategy slightly increases training FLOPs. Several cheaper alternatives are worth investigating, such as better initialization or more efficient guidance strategies.

Impact Statement

Enhancing the efficiency of Large Language Models (LLMs) can significantly reduce computational resources and energy consumption, benefiting the environment and democratizing access to advanced AI technologies. However, increased efficiency could also lead to greater dissemination of disinformation and the creation of deepfakes, posing risks to public trust and security and potentially reinforcing existing biases that impact specific groups unfairly. This research aims to promote the responsible development and deployment of LLMs, maximizing societal benefits while acknowledging potential harms.

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023a.

Ainslie, J., Lei, T., de Jong, M., Ontañón, S., Brahma, S., Zemlyanskiy, Y., Uthus, D. C., Guo, M., Lee-Thorp, J., Tay, Y., et al. Colt5: Faster long-range transformers with conditional computation. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 5085–5100, 2023b.

Bae, S., Kim, Y., Bayat, R., Kim, S., Ha, J., Schuster, T., Fisch, A., Harutyunyan, H., Ji, Z., Courville, A., et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.

Banino, A., Balaguer, J., and Blundell, C. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Clark, C., Lee, K., Chang, M.-W., Kwiatkowski, T., Collins, M., and Toutanova, K. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers)*, pp. 2924–2936, 2019.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Dao, T., Chen, B., Sohoni, N. S., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pp. 4690–4721. PMLR, 2022a.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022b.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Elhoushi, M., Shrivastava, A., Liskovich, D., Hosmer, B., Wasti, B., Lai, L., Mahmoud, A., Acun, B., Agarwal, S., Roman, A., et al. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, 2024.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. Transformer engine. https://github.com/NVIDIA/TransformerEngine, 2022.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N.-Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 1525–1534, 2016.

Penedo, G., Kydlíček, H., Lozhkov, A., Mitchell, M., Raffel, C. A., Von Werra, L., Wolf, T., et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37: 30811–30849, 2024.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Raposo, D., Ritter, S., Richards, B., Lillicrap, T., Humphreys, P. C., and Santoro, A. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*, 2024.

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Schuster, T., Fisch, A., Gupta, J., Dehghani, M., Bahri, D., Tran, V., Tay, Y., and Metzler, D. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wei, X., Moalla, S., Pascanu, R., and Gulcehre, C. Building on efficient foundations: Effective training of llms with structured feedforward layers. *Advances in Neural Information Processing Systems*, 37:4689–4717, 2024.

Wu, S., Chen, J., Lin, K. Q., Wang, Q., Gao, Y., Xu, Q., Xu, T., Hu, Y., Chen, E., and Shou, M. Z. Videollm-mod: Efficient video-language streaming with mixture-of-depths vision computation. *Advances in Neural Information Processing Systems*, 37:109922–109947, 2024.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

A. Implementation details

We use 4 H100 94G GPUs for training and evaluation, employing mixed precision (`bfloat16` and `float32`) with `torch.cuda.amp` to accelerate training. Training FLOPs are calculated following Megatron (Narayanan et al., 2021), including all matrix multiplications. To enhance training and inference efficiency, our code is based on `PyTorch` but incorporates optimized CUDA kernels. We leverage Flash Attention (Dao et al., 2022b), fast LayerNorm, and rotary embeddings from TransformerEngine (NVIDIA, 2022), along with fused operations including bias and GeLU.

Architecture details are provided in Table 6. We consider four baseline Transformer sizes, ranging from 110M to 1.3B parameters, with widths from 768 to 2048. For `RankMoD`, we adopt configuration with around 32% of additional parameters of the FFN in a low-rank branch as described in the main paper.

Note that in all experiments, we do not apply structured matrices to the first FFN module, as doing so can lead to non-negligible performance loss in models on shallow networks. For dense models, we use Gaussian random weight initialization with a standard deviation of 0.02. For Low-Rank matrices spectral initialization is applied.

All models are trained with AdamW (β_1=0.9, β_2=0.999, weight decay 0.1) at peak learning rate in Table 7 on a cosine schedule with linear warmup over the first 10% of step. Gradients are clipped to a global norm of 1.0. We use up to 60 hours of compute for each of the experiments.

Table 6. Detailed configurations of our Transformers. **Width** denotes the model width or the input and output dimensions of the attention and FFN modules. **Intermediate dim.** refers to the intermediate dimension of the FFN. **Attention dim.** specifies the dimension used in scaled-dot product attention.

Model	Total Size (M)	Layers	Width	Intermediate dim.	Attention dim.	LR Dimension
Transformer-s	110.0	12	768	3072	768	-
`MoD`	110.0	12	768	3072	768	-
`RankMoD`	125.7	12	768	3072	768	192
Transformer-m	335.1	24	1024	4096	1024	-
`MoD`	335.1	24	1024	4096	1024	-
`RankMoD`	395.1	24	1024	4096	1024	256
Transformer-l	729.1	24	1536	6144	1536	-
`MoD`	729.1	24	1536	6144	1536	-
`RankMoD`	864.3	24	1536	6144	1536	384
Transformer-xl	1274.1	24	2048	8192	2048	-
`MoD`	1274.1	24	2048	8192	2048	-
`RankMoD`	1514.1	24	2048	8192	2048	512

Table 7. Training configuration used in all experiments. Note that we apply the same global batch size (**Batch**) and the same peak learning rates (**LR**) to both `Dense`, `MoD` and `RankMoD` to avoid hyperparameter search. The hyperparameter values are selected based on Zhang et al. (2022); Gu & Dao (2023).

Model	Tokens	Batch	LR
-s size			
Dense	2.2B	512	6.0e-4
-m size			
Dense	6.7B	512	3.0e-4
-l size			
Dense	14.6B	512	2.5e-4
-xl size			
Dense	25.5B	512	2.0e-4

B. Self-guidance Methods

The self-guided training scheme adopted in Section 2.3 introduces, for each low-rank linear $B(Ax_t)$ in $\mathrm{FFN}_{\mathrm{lr}}$, a separately optimized dense matrix $G \in \mathbb{R}^{M\times N}$ that is convex-blended with the structured output as $\alpha G x_t + (1-\alpha)B(Ax_t)$ during the first 50% of training and discarded once $\alpha = 0$. The guide doubles the parameter count of the low-rank branch during the guided phase and runs an *auxiliary* dense matmul alongside the structured pair on every blended forward pass. Two cheaper alternatives — guide reuse and routing warmup — are worth examining: the first removes the parameter overhead, the second additionally removes the per-token double evaluation, so that no token is ever passed through both branches. We compare both against the canonical scheme.

Self-guidance with $G \equiv W_1, W_2$ To eliminate the per-linear parameter overhead of self-guidance, we *identify the guide with the existing dense path*: rather than allocating a fresh G for each low-rank linear, we set $G \equiv W_1$ for the first low-rank linear and $G \equiv W_2$ for the second. On low-rank tokens ($s_t \le P_\beta(s)$) the blend becomes

$$\hat{y}_t^{(1)} = \alpha\, W_1 x_t + (1-\alpha)\, B_1 A_1 x_t, \tag{5}$$

$$\hat{y}_t^{(2)} = \alpha\, W_2 z_t + (1-\alpha)\, B_2 A_2 z_t, \qquad z_t = f(\hat{y}_t^{(1)}), \tag{6}$$

with α following the same cosine schedule from 1 to 0 over the first 50% of training. The stochastic reduced-FLOPs variant of Equation (4) is applied unchanged. To keep the rank-R pair geometrically aligned with its guide at step 0, we initialize (A_i, B_i) as the rank-R truncated SVD of W_i. Once $\alpha = 0$ the parameter-tying drops and the layer reduces to the unblended form of Equation (2).

This variant trades parameters for cross-branch coupling. W_1 and W_2 now receive gradient signal from the low-rank tokens as well as the top-routed tokens, whereas in the canonical `RankMoD` layer the dense matrices see only tokens with $s_t > P_\beta(s)$.

Routing warmup A second alternative removes the need for *any* guide by deferring engagement of the routing partition entirely. We replace the fixed routing budget β with a linearly annealed schedule

$$\beta(\tau) = \beta_0 + \min\!\left(\tfrac{\tau}{W},\, 1\right)\,(\beta - \beta_0), \qquad \beta_0 = 1, \tag{7}$$

so that at step 0 every token takes the dense path ($\beta(0) = 1$) and the dense set $\{t : s_t > P_{\beta(\tau)}(s)\}$ shrinks linearly to its target size of $\lceil \beta N \rceil$ over W optimizer steps. While $\beta(\tau) = 1$ the low-rank branch receives no tokens, so the randomly initialized (A_i, B_i) contribute nothing to the forward pass; the optimizer develops W_1, W_2 and the router before the bottleneck closes on the low-rank tokens. We use $W = 0.5 \cdot \mathrm{max_step}$, matching the duration of the self-guided phase. Unlike the original self-guidance scheme, this setup introduces no extra parameters; moreover, every token is evaluated by exactly one of the two branches rather than twice — once through the structured pair and once through the full-rank guide — as canonical self-guidance requires.

C. Routing Ablation

In the `RankMoD` layer (Equation (3)) the multiplicative gate $\sigma(s_t)$ is applied only to the dense path: top-routed tokens ($s_t > P_\beta(s)$) are scaled by $\sigma(s_t)$, while the low-rank tokens ($s_t \le P_\beta(s)$) leave $B_i A_i$ *ungated*. Because the top-β selection is non-differentiable, this gate is the router's only source of gradient, and it carries signal from the dense partition alone. We ask whether also gating the low-rank branch — opening a second, low-rank-side gradient path to the router — helps. We multiply $B_i A_i x_t$ by either $\sigma(s_t)$ (the dense-path gate) or its complement $1 - \sigma(s_t) = \sigma(-s_t)$. The two differ in sign: low-rank tokens sit below threshold, so $\sigma(s_t) < \frac{1}{2}$ attenuates their output and pushes the router to *raise* their scores, whereas $1 - \sigma(s_t)$ keeps the gain near unity and rewards *lowering* the score — a routing-consistent, two-sided gate.

Table 8. Validation loss and perplexity (0.5B-token validation set) for at the 50%–50% routing setting when the low-rank branch is left ungated (default) versus gated by $\sigma(s_t)$ or its complement $1 - \sigma(s_t)$. Transformer-S ($N = 768$, $M = 3072$, 12 layers, $R = 192$, 2.2B tokens, peak LR 6×10^{-4}) and Transformer-M ($N = 1024$, $M = 4096$, 24 layers, $R = 256$, 6.7B tokens, peak LR 3×10^{-4}); both FineWeb-Edu, AdamW, cosine schedule. Best per column in **bold**.

		Transformer-S		Transformer-M	
Low-rank gate	**Multiplier**	**Loss**	**PPL**	**Loss**	**PPL**
ungated (default)	—	**2.9967**	**20.02**	2.6597	14.29
same-sign	$\sigma(s_t)$	3.0082	20.25	2.6763	14.53
complement	$1 - \sigma(s_t)$	2.9976	20.04	**2.6590**	**14.28**

The complement gate is indistinguishable from the ungated baseline (-0.0007 in loss at Transformer-M, $+0.0009$ at Transformer-S; within seed noise), so the second gradient path buys nothing — the dense-side gate already supplies sufficient routing signal. The same-sign $\sigma(s_t)$ gate is actively harmful ($+0.017$ in loss at M, $+0.012$ at S): it attenuates exactly the tokens the low-rank branch exists to serve and drives the router against its own partition. We therefore leave the low-rank branch ungated.

D. License information

- FineWeb-Edu (dataset): Open Data Commons License Attribution family. Link: https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu
- lm-evaluation-harness (code): MIT License.

  Link: https://github.com/EleutherAI/lm-evaluation-harness