DYNAMIC FFNS IMPROVE REPRESENTATION LEARNING IN TRANSFORMER PRETRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Modular architectures allow different inputs to invoke different computations, but existing approaches often couple this flexibility with increased model capacity. We ask how far input-dependent modularity can take us when parameter and compute budgets are held fixed. We introduce **D**ynamic **F**eed**f**orward **N**etworks (DFFNs), which reorganize a standard Transformer feedforward block into a shared static FFN and a set of routed low-rank Submodules (SMs). A learned router selects an SM for each token, providing input-dependent computation without increasing the parameter or compute budget of the dense FFN. Across multiple model scales, DFFN consistently improves language modeling and downstream performance over matched dense Transformers. Controlled experiments further show that these gains depend on learned input-dependent routing rather than simply the presence of the submodules. Together, our results show that modularity can improve Transformer pretraining without relying on increased model capacity.

1 INTRODUCTION

Scaling model size, data, and compute has driven substantial progress in language modeling (Kaplan et al., 2020; Hoffmann et al., 2022), but continued gains through scaling require correspondingly greater resources. By contrast, better architectures can improve performance without simply making models larger or more expensive. In this work, we focus on modular architectures, where different inputs can invoke different computations.

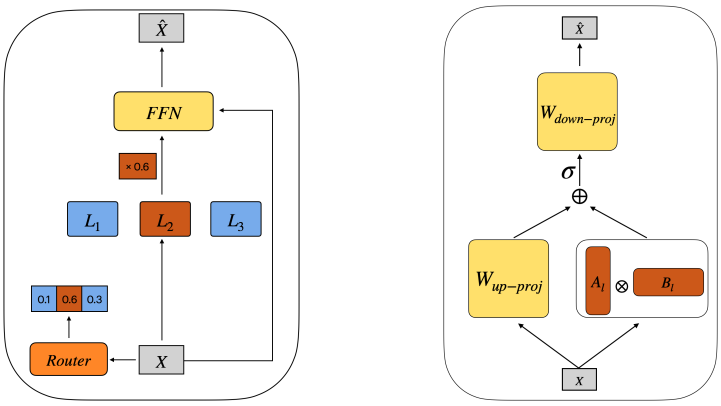
Mixture-of-Experts (MoE) models are a prominent example of this approach. By routing each token to a subset of expert feedforward networks, MoEs introduce input-dependent computation while allowing the model to contain many more parameters than are activated for each token (Shazeer et al., 2017; Fedus et al., 2022). Their success therefore combines input-dependent modularity with sparse computation and increased model capacity. However, this makes it difficult to disentangle the effect of modularity from the additional capacity and sparsity that accompany it.

We therefore ask ***How far can input-dependent modularity take us without scaling the parameter or compute budget?*** Rather than using modularity to increase model capacity, we study whether it can improve representation learning while keeping the resources available to the model fixed. This allows us to examine the effect of input-dependent computation without simultaneously changing the model's parameter or compute budget.

To investigate this question, we introduce **D**ynamic **F**eedforward **N**etworks (DFFNs). The central idea is to reorganize the Transformer feedforward block. Instead of allocating the entire parameter budget to a single static transformation, DFFN retains a shared static FFN and reallocates part of its parameter budget to a collection of small low-rank Submodules (SMs). A learned router selects an SM for each token, allowing different inputs to receive different feedforward transformations. The parameters allocated to the SMs are taken from the static FFN rather than added on top of it, preserving the parameter and compute budget of the dense FFN. The static FFN, router, and SMs are trained jointly from scratch during pretraining.

We find that DFFN consistently improves both language modeling and downstream performance over dense Transformers with the same parameter and compute budget. These improvements are observed across all model scales we study, while controlled experiments show that learned input-dependent routing is important to the gains. Our analysis further finds that the SMs learn distinct, largely non-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Overall DFFN layer. (b) Inner workings in the up-projection.

Figure 1: **DFFN architecture. Left.** For each token x, a learned router produces scores over a set of low-rank Submodules (SMs) and selects the top-k subset $\Omega_k(x)$. The selected SMs provide an input-dependent modification to the shared static FFN. **Right.** Each SM is parameterized by $A_\ell \in \mathbb{R}^{H\times r}$ and $B_\ell \in \mathbb{R}^{r\times D'_{\text{ff}}}$. The routed update is added to the static up-projection before the activation, giving $z = xW'_1 + \sum_{\ell\in\Omega_k(x)} s_\ell(x) x A_\ell B_\ell$, followed by the activation and down-projection $\text{DFFN}(x) = \sigma(z)W'_2$.

overlapping transformations. Together, these results demonstrate the effectiveness of input-dependent modularity within a parameter- and compute-matched setting.

Our main contributions are:

- We introduce DFFNs, a feedforward architecture that adds input-dependent modularity to a Transformer FFN without increasing its parameter or compute budget.
- We show that reallocating a fixed FFN budget between shared and routed components consistently improves language modeling and downstream performance over matched dense Transformers.
- Through controlled experiments, we show that the improvements depend on learned input-dependent routing rather than simply the presence of the submodules, providing evidence that input-dependent modularity is central to the gains observed with DFFN.

2 Related Work

Mixture-of-Experts. Mixture-of-Experts (MoE) architectures replace the dense feedforward network with a collection of experts and a learned router that selects a small subset for each input (Shazeer et al., 2017; Lepikhin et al., 2021; Fedus et al., 2022; Du et al., 2022; Zoph et al., 2022; Jiang et al., 2024). By activating only a few experts per token, MoEs increase total model capacity without proportionally increasing per-token computation. Each expert, however, is typically a full FFN, making MoEs a relatively coarse-grained form of conditional computation whose primary scaling advantage comes from increasing the number of parameters available to the model. In contrast, we study whether input-dependent computation is beneficial when introduced within a fixed FFN parameter budget. Rather than adding a pool of full FFNs, DFFN reallocates part of the dense FFN into small low-rank SMs while preserving the overall parameter and compute budget.

PEFT and Low-Rank Adaptation. Parameter-efficient fine-tuning (PEFT) methods introduce small trainable components for adapting pretrained models while leaving most model parameters fixed. Adapter methods insert bottleneck modules into the Transformer (Houlsby et al., 2019; Pfeiffer et al., 2020), while LoRA represents weight updates through low-rank matrices (Hu et al., 2022). Related work has combined low-rank adaptation with routing among multiple adapters. Mixture-of-LoRA Experts (Wu et al., 2024) routes inputs among multiple LoRA modules during fine-tuning, while LoRAMoE (Dou et al., 2023) similarly combines routed LoRA experts with a frozen pretrained backbone. These methods demonstrate that routing among low-rank transformations can be useful for downstream adaptation. DFFN instead incorporates routed low-rank transformations into the FFN during pretraining, with the static FFN, router, and SMs learned jointly from scratch.

FFN Variants. A separate line of work modifies the Transformer FFN itself. Gated FFNs such as GLU, GEGLU, and SwiGLU introduce multiplicative gating within the feedforward transformation (Dauphin et al., 2017; Shazeer, 2020). These architectures improve the form of the shared FFN computation, but the same parameterized transformation remains available to every input. DFFN instead introduces token-dependent variation within the FFN by routing each token through one of several low-rank transformations while retaining a shared static component.

Fine-Grained Expert Retrieval. Mixture-of-a-Million-Experts (MoME) (He, 2024) explores conditional computation at a much finer granularity than conventional MoEs, retrieving small experts from a very large expert pool using product-key routing. This demonstrates that useful conditional computation need not be organized around full FFN experts. MoME nevertheless follows the sparse scaling paradigm of providing a large pool of parameters from which only a small subset is activated for each input. Our question is complementary. Rather than expanding the available parameter pool, DFFN asks whether fine-grained conditional computation remains beneficial when the same parameter budget as a dense FFN is redistributed into shared and routed components.

3 METHOD

3.1 DYNAMIC FEEDFORWARD NETWORKS

The standard Transformer feedforward block maps a hidden state $x \in \mathbb{R}^H$ through an up projection, a nonlinear activation, and a down projection

$$\mathrm{FFN}(x) = \sigma(xW_1)W_2, \qquad W_1 \in \mathbb{R}^{H \times D_{\mathrm{ff}}},\ W_2 \in \mathbb{R}^{D_{\mathrm{ff}} \times H}. \tag{1}$$

DFFNs augment this block with a set of Submodules (SMs) and a learned router. Each feedforward layer contains L SMs

$$A_\ell \in \mathbb{R}^{H \times r}, \qquad B_\ell \in \mathbb{R}^{r \times D'_{\mathrm{ff}}}, \qquad \ell = 1, \ldots, L, \tag{2}$$

where r is the submodule rank. Each SM defines a low-rank transformation $\Delta W_{1,\ell} = A_\ell B_\ell$ that modifies the static up projection.

The router maps each input to a distribution over SMs

$$s(x) = \mathrm{softmax}(xW_R), \qquad W_R \in \mathbb{R}^{H \times L}, \tag{3}$$

and selects the top-k indices $\Omega_k(x)$.

The selected SMs modify the static up projection before the activation function

$$z(x) = xW'_1 + \sum_{\ell \in \Omega_k(x)} s_\ell(x)\, xA_\ell B_\ell, \qquad W'_1 \in \mathbb{R}^{H \times D'_{\mathrm{ff}}}. \tag{4}$$

The output of the DFFN block is

$$\mathrm{DFFN}(x) = \sigma(z(x))W'_2, \qquad W'_2 \in \mathbb{R}^{D'_{\mathrm{ff}} \times H}. \tag{5}$$

The static FFN, SMs, and router are trained jointly during pretraining. Unless otherwise specified, we use $L = 16$ rank-16 SMs and Top-1 routing, so that one SM is selected for each token.

3.2 ROUTING MECHANISM AND LOAD BALANCING

The learned router determines which low-rank component is applied to each token using the scores $s(x)$ defined above. Without additional regularization, routing can concentrate on a small subset of SMs, leaving other SMs rarely selected. We therefore use the Switch Transformer auxiliary load-balancing loss (Fedus et al., 2022) to encourage usage across the SMs. We use a load-balancing coefficient of 0.01 in our main experiments.

Variant	Layers	Hidden H	Attn. Heads	FFN Dim	Num SMs	SM Rank	Parameters	Tokens
Dense	8	2048	32	6664	–	–	0.88B	345B
DFFN	8	2048	32	6144	16	16	0.88B	345B
Dense	24	2048	32	6664	–	–	1.58B	400B
DFFN	24	2048	32	6144	16	16	1.58B	400B
Dense	56	2048	32	6664	–	–	2.99B	95B
DFFN	56	2048	32	6144	16	16	2.99B	95B

Table 1: **Model configurations.** DFFN exactly matches the dense FFN parameter count at each scale by reallocating parameters from the static FFN to the routed submodules and router.

3.3 Matching Static FFN Parameters and Compute

Parameters. A dense FFN with hidden size H and expansion dimension D_{ff} contains

$$P_{\text{dense}} = 2HD_{\text{ff}}, \tag{6}$$

parameters, omitting biases. DFFN reduces the static expansion to D'_{ff} and reallocates the freed parameters to L rank–r SMs and the router, giving

$$P_{DFFN} = 2HD'_{\text{ff}} + Lr(H + D'_{\text{ff}}) + HL. \tag{7}$$

Setting $P_{DFFN} = P_{\text{dense}}$ yields

$$D'_{\text{ff}} = \frac{2HD_{\text{ff}} - HL(r+1)}{2H + Lr}. \tag{8}$$

In our experiments, $H = 2048$, $D_{\text{ff}} = 6664$, $D'_{\text{ff}} = 6144$, $L = 16$, and $r = 16$, giving exactly $27{,}295{,}744$ FFN parameters per layer for both models.

Compute. Under dense DFFN execution, the reduced static FFN, router projection, and low-rank projections require

$$2HD'_{\text{ff}} + Lr(H + D'_{\text{ff}}) + HL \tag{9}$$

linear-layer multiply-accumulates per token. By the parameter-matching construction above, this equals the $2HD_{\text{ff}}$ cost of the dense FFN. Full parameter and compute accounting is provided in Appendix B.

4 Design Intuition

The motivation behind DFFN is that a fixed feedforward parameter budget may be more effective when it provides multiple input-dependent transformations rather than a single transformation shared across all tokens. DFFN realizes this idea by retaining most of the FFN capacity in a shared static component while reallocating a smaller portion to routed low-rank SMs. Low rank makes it possible to introduce multiple alternative transformations while preserving most of the capacity of the shared FFN. The static width is chosen so that this reallocation exactly preserves the FFN parameter count and linear-layer compute of the dense baseline, as detailed in Section 3.3.

This design leads to two testable predictions. First, if this modular allocation is beneficial, DFFN should outperform a parameter- and compute-matched dense FFN. Second, if input-dependent selection is important, learned routing should outperform controls that remove this selection. We test these predictions in Section 7 and analyze the transformations learned by the SMs in Section 6.

5 Evaluation

Model Configurations and Training Data. We evaluate DFFNs at three model scales: 0.88B, 1.58B, and 2.99B parameters. All models are decoder-only Transformers with hidden dimension 2048 and

	Final Validation Perplexity				Perplexity @ Compute-Optimal			
Scale	**Tokens (B)**	Dense	DFFN	%↓	**Tokens (B)**	Dense	DFFN	%↓
0.88B	345	13.60	**13.40**	**1.47%**	18	18.28	**17.72**	**3.06%**
1.58B	400	11.09	**10.48**	**5.50%**	32	18.38	**16.17**	**12.02%**
2.99B	95	12.182	**11.080**	**9.05%**	60	12.769	**11.811**	**7.50%**

Table 2: **Validation perplexity on FineWeb-Edu.** The left section reports validation perplexity at the final training checkpoint for each model pair. The right section reports perplexity near the Chinchilla compute-optimal training length (approximately 20 tokens per parameter). DFFN achieves lower validation perplexity than the dense baseline at every model scale and training horizon shown. Percentages denote relative reductions in perplexity.

	0.88B			1.58B			2.99B		
Benchmark	Dense	DFFN	**Diff. (%)**	Dense	DFFN	**Diff. (%)**	Dense	DFFN	**Diff. (%)**
ARC-Challenge	$31.8_{\pm 1.4}$	$33.2_{\pm 1.4}$	+4.40	$34.6_{\pm 1.4}$	$36.6_{\pm 1.4}$	+5.78	$34.7_{\pm 1.4}$	$36.9_{\pm 1.4}$	+6.34
ARC-Easy	$64.4_{\pm 1.0}$	$66.6_{\pm 1.0}$	+3.42	$70.6_{\pm 0.9}$	$72.1_{\pm 0.9}$	+2.12	$70.5_{\pm 0.9}$	$72.1_{\pm 0.9}$	+2.27
BoolQ	$61.7_{\pm 0.9}$	$61.9_{\pm 0.9}$	+0.32	$62.8_{\pm 0.8}$	$64.1_{\pm 0.8}$	+2.07	$62.5_{\pm 0.8}$	$63.3_{\pm 0.8}$	+1.28
HellaSwag	$38.3_{\pm 0.5}$	$38.7_{\pm 0.5}$	+1.04	$57.4_{\pm 0.5}$	$59.6_{\pm 0.5}$	+3.83	$53.6_{\pm 0.5}$	$57.9_{\pm 0.5}$	+8.02
LAMBADA	$35.2_{\pm 0.7}$	$37.9_{\pm 0.7}$	+7.67	$45.5_{\pm 0.7}$	$48.5_{\pm 0.7}$	+6.59	$42.8_{\pm 0.7}$	$49.0_{\pm 0.7}$	+14.49
MNLI	$35.3_{\pm 0.5}$	$35.9_{\pm 0.5}$	+1.70	$36.4_{\pm 0.5}$	$37.8_{\pm 0.5}$	+3.85	$34.5_{\pm 0.5}$	$36.4_{\pm 0.5}$	+5.51
OpenBookQA	$26.0_{\pm 2.0}$	$25.0_{\pm 1.9}$	-3.85	$27.0_{\pm 2.0}$	$30.2_{\pm 2.1}$	+11.85	$26.8_{\pm 2.0}$	$30.0_{\pm 2.1}$	+11.94
PubMedQA	$53.6_{\pm 2.2}$	$56.4_{\pm 2.2}$	+5.22	$55.6_{\pm 2.2}$	$65.0_{\pm 2.1}$	+16.91	$52.2_{\pm 2.2}$	$65.0_{\pm 2.1}$	+24.52
SciQ	$77.7_{\pm 1.3}$	$79.7_{\pm 1.3}$	+2.57	$83.6_{\pm 1.2}$	$86.7_{\pm 1.1}$	+3.71	$81.8_{\pm 1.2}$	$84.6_{\pm 1.1}$	+3.42
Winogrande	$53.1_{\pm 1.4}$	$52.8_{\pm 1.4}$	-0.56	$58.0_{\pm 1.4}$	$59.8_{\pm 1.4}$	+3.10	$57.8_{\pm 1.4}$	$57.9_{\pm 1.4}$	+0.17
Average	47.71	48.81	**+2.31**	53.15	56.04	**+5.44**	51.72	55.31	**+6.94**

Table 3: **Downstream benchmark performance across model scales.** We report zero-shot accuracy (higher is better) on ten downstream tasks. Diff. denotes the relative percentage change of DFFN with respect to the corresponding dense baseline. Green indicates an improvement and red a regression.

32 attention heads. We vary depth across scales, using 8, 24, and 56 layers, respectively. Dense models use an FFN intermediate dimension of 6664, while DFFN reduces this dimension to 6144 and reallocates the corresponding parameter budget to 16 rank-16 submodules. This construction exactly matches the FFN parameter count between the dense and DFFN models. Full model configurations are summarized in Table 1.

All models are pretrained on FineWeb-Edu (Lozhkov et al., 2024). The 0.88B, 1.58B, and 2.99B model pairs are trained for 345B, 400B, and 95B tokens, respectively. Within each scale, the dense and DFFN models use the same data, optimization settings, and training duration.

DFFN Design Choices. We use $L{=}16$ SMs of rank $r{=}16$ and apply the routed low-rank transformations to the FFN up-projection. These choices provide multiple alternative transformations while keeping each SM small relative to the shared FFN, so that only a modest portion of the fixed FFN parameter budget is reallocated from the static component. We found this configuration to perform well among the parameter allocations we evaluated, including different numbers and ranks of SMs, as shown in Appendix Fig. 5. We apply SMs only to the up-projection as this is sufficient to introduce input-dependent variation before the FFN nonlinearity.

We use top–$k{=}1$ routing so that each token selects a single SM, giving the routed component an explicitly modular structure without increasing the number of active SMs per token. This choice is also supported by our routing ablations (Section 7.1), where activating all 16 SMs for every token provides no improvement over learned Top-1 routing. We use an auxiliary load-balancing coefficient of 0.01 to prevent routing collapse.

Benchmark	Dense	DFFN	Δ%
ARC-Challenge	$35.4_{\pm 1.4}$	$\mathbf{36.9}_{\pm 1.4}$	+4.24
ARC-Easy	$70.9_{\pm 0.9}$	$\mathbf{73.2}_{\pm 0.9}$	+3.24
BoolQ	$62.6_{\pm 0.8}$	$\mathbf{62.9}_{\pm 0.8}$	+0.48
HellaSwag	$53.7_{\pm 0.5}$	$\mathbf{57.8}_{\pm 0.5}$	+7.64
LAMBADA	$44.1_{\pm 0.7}$	$\mathbf{49.4}_{\pm 0.7}$	+12.02
MNLI	$35.3_{\pm 0.5}$	$\mathbf{36.2}_{\pm 0.5}$	+2.55
OpenBookQA	$27.0_{\pm 2.0}$	$\mathbf{29.4}_{\pm 2.0}$	+8.89
PubMedQA	$53.6_{\pm 2.2}$	$\mathbf{64.6}_{\pm 2.1}$	+20.52
SciQ	$83.6_{\pm 1.2}$	$\mathbf{85.6}_{\pm 1.1}$	+2.39
Winogrande	$57.4_{\pm 1.4}$	$\mathbf{58.6}_{\pm 1.4}$	+2.09
Average	**52.36**	**55.46**	**+5.92**

Table 4: **Performance after supervised fine-tuning.** The pretrained 2.99B dense and DFFN models are fine-tuned for one epoch on Alpaca and evaluated on the same ten downstream benchmarks used in the main evaluation. Δ% denotes the relative percentage change from Dense to DFFN.

5.1 Main Results

Table 2 reports validation perplexity across all three model scales. At the final training checkpoint, DFFN achieves lower perplexity than the corresponding dense baseline at every scale, reducing perplexity by 1.47%, 5.50%, and 9.05% at 0.88B, 1.58B, and 2.99B parameters, respectively. Due to the substantially greater computational cost of the 2.99B models, we train this model pair for 95B tokens, compared with 345B and 400B tokens for the 0.88B and 1.58B models.

Perplexity at compute-optimal training length. We follow the Chinchilla scaling laws Hoffmann et al. (2022) and additionally evaluate models near the predicted compute-optimal training length of roughly 20 tokens per parameter: 18B, 32B, and 60B tokens for the 0.88B, 1.58B, and 2.99B models, respectively. At these checkpoints, DFFN reduces perplexity relative to the dense baseline by 3.06%, 12.02%, and 7.50%, respectively. Thus, the perplexity improvement is also observed when the comparison is made at approximately compute-optimal training budgets across all three scales.

Benchmark Performance. Table 3 reports zero-shot accuracy across ten general language understanding benchmarks. The 0.88B and 1.58B results use their final checkpoints after 345B and 400B training tokens, respectively, while the 2.99B results use its 95B-token checkpoint. At 0.88B, DFFN improves average benchmark accuracy from 47.71 to 48.81, a relative improvement of 2.31%. At 1.58B, average accuracy improves from 53.15 to 56.04 (+5.44%), and at 2.99B from 51.72 to 55.31 (+6.94%). DFFN improves over the corresponding dense baseline on eight of ten benchmarks at 0.88B and on all ten benchmarks at both 1.58B and 2.99B.

5.2 Do Pretraining Improvements Persist After Fine-Tuning?

As a post-training robustness check, we test whether the improvements obtained during pretraining persist after supervised fine-tuning. Starting from the pretrained 2.99B dense and DFFN checkpoints, we fine-tune both models for one epoch on the Alpaca instruction-following dataset (Taori et al., 2023) using the same training setup. We then evaluate the fine-tuned models on the same ten downstream benchmarks used in our main evaluation.

As shown in Table 4, the advantage of DFFN persists after fine-tuning. Average accuracy improves from 52.36 for the dense model to 55.46 for DFFN, corresponding to a 5.92% relative improvement. DFFN achieves higher accuracy on all ten evaluated benchmarks. This experiment is intended as a robustness check of the pretraining result, showing that the observed advantage is retained after a standard supervised fine-tuning stage rather than disappearing during subsequent adaptation.

5.3 Training Efficiency

We measure the practical training overhead of DFFN using the 1.58B models. Table 5 reports average iteration time, training throughput, total runtime, GPU hours, and achieved throughput per GPU. Despite introducing input-dependent routing and low-rank submodules, DFFN has nearly

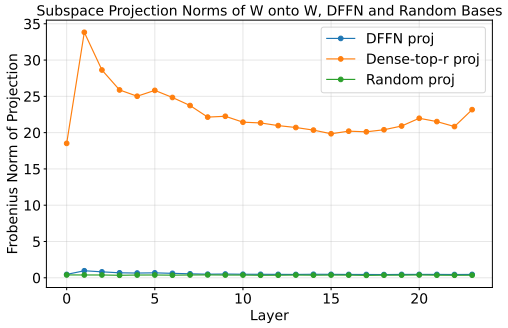


Figure 2: **Alignment of SMs with the static FFN.** Projection of the static FFN up-projection W_1' onto each SM's learned rank–r subspace. Projection norms remain small across layers, indicating low alignment between the SMs and the directions most strongly emphasized by the static FFN.

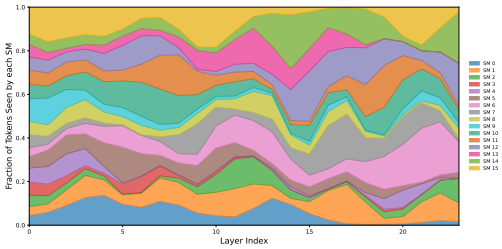


Figure 3: **Layerwise SM usage.** Fraction of token assignments to each SM across Transformer layers. Multiple SMs remain active throughout the network, with the distribution of assignments varying across depth.

identical training cost to the dense baseline. Average iteration time increases from 4.8511 to 4.8788 seconds, corresponding to only 0.57% overhead, while throughput decreases by 0.58%. Over the full 400B-token training run, total GPU usage increases from 7,330 to 7,372 GPU hours.

6 Understanding DFFNs

To understand how DFFNs differ from a standard FFN and why they provide empirical gains, we analyze the structure and usage of the Submodules. Our goal is to determine whether the network develops the fine–grained modularity that the architecture is designed to encourage.

6.1 Do SMs Learn Transformations Complementary to the Static FFN?

We first ask whether the learned SMs reproduce directions already emphasized by the static FFN or learn complementary transformations. For each SM, the pair (A_i, B_i) defines a rank–r transformation $\Delta W_i = A_i B_i$, where $A_i \in \mathbb{R}^{H \times r}$ and $B_i \in \mathbb{R}^{r \times D'_{\text{ff}}}$. We compute the truncated SVD $\Delta W_i \approx U_i \Sigma_i V_i^\top$ and measure its alignment with the static up-projection W_1' as $\|U_i^\top W_1' V_i\|_F$. Larger values indicate stronger alignment with directions emphasized by the static FFN.

Averaged across layers and SMs, the projection norm is 0.54 (Figure 2). For comparison, projecting W_1 onto its own top–r singular subspace gives 22.7, while projection onto a random r-dimensional subspace gives 0.37. The SM projection is therefore much closer to the random baseline than to the dominant rank–r subspace of W_1, suggesting that the SMs do not simply reproduce the directions most strongly represented by the static FFN, but instead learn largely complementary low-rank transformations.

6.2 Do SMs Learn Distinct Transformations?

Next, we ask whether different SMs learn distinct transformations or collapse to redundant ones. For each SM, we use the top–r left singular vectors U_i of its learned update and measure the Grassmann similarity between pairs of SMs as $\psi(i, j) = \|U_i^\top U_j\|_F^2 / r$, following Hu et al. (2021). Values near 1 indicate strongly overlapping subspaces, while values near 0 indicate nearly orthogonal subspaces.

Across layers, the mean pairwise similarity is only 0.019. Figure 4 shows that off-diagonal similarities remain close to zero across SM pairs. Thus, the SMs learn largely non-overlapping low-rank subspaces rather than converging to redundant transformations. This provides the router with a set of distinct transformations to choose from for each token.

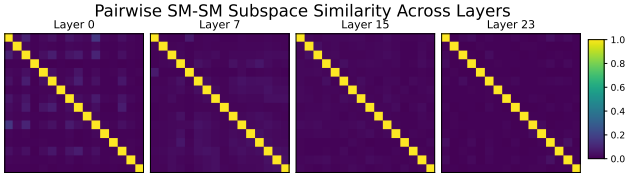


Figure 4: **Pairwise SM subspace similarity.** Grassmann similarities between learned SM subspaces are low (mean 0.019), indicating little overlap between their dominant transformation directions.

Metric	Dense	DFFN
Iteration time (s)	4.8511	4.8788
Tokens / sec	486,393	483,581
Total runtime (days)	9.54	9.59
Total GPU hours	7,330	7,372
TFLOPs / sec / GPU	196.0	194.9

Table 5: **Measured training efficiency at the 1.58B scale.** Runtime statistics are measured over the 400B-token training runs for the dense and DFFN models.

6.3 Does the Router Learn to Use Multiple SMs?

Finally, we examine whether routing collapses onto a small number of SMs or continues to use multiple submodules across the network. Figure 3 shows the fraction of tokens assigned to each SM across layers. Multiple SMs remain active throughout the network, although usage varies substantially across depth and some layers concentrate more strongly on a subset of SMs. Thus, routing does not collapse to a single submodule and instead maintains heterogeneous SM usage across layers.

7 Ablations

We use controlled experiments to test whether DFFN's gains are driven by learned input-dependent modularity rather than simply the presence of the submodules. We additionally test whether the performance advantage is reproduced with a different random seed.

7.1 Does Input-Dependent Routing Drive the Improvements?

An important question is whether DFFN's improvements arise from learning to route different inputs to different SMs, or whether the SMs themselves provide the benefit regardless of how they are routed. To distinguish these possibilities, we train two additional controls. Random Top-1 assigns each token uniformly at random to one SM, preserving the same number of active SMs per token as DFFN while removing input-dependent routing. Top-16 activates all 16 SMs for every token, testing whether simply making all of the submodules available to every input improves performance. We train both controls, together with the standard learned Top-1 configuration, from scratch for 10B tokens at the 0.88B scale.

As shown in Table 6, learned Top-1 routing achieves a validation perplexity of 22.39, compared with 24.35 for Random Top-1. Thus, the presence of the SMs alone is not sufficient to obtain the performance of DFFN; learning which SM to apply to each input matters. Top-16 reaches a perplexity of 22.73, showing no improvement from applying all 16 SMs to every token. Together, these controls demonstrate that input-dependent routing contributes to the improvements of DFFN, rather than the gains arising simply from the presence of the low-rank submodules.

7.2 Does the Improvement Hold with a Different Random Seed?

We repeat the 1.58B comparison with a different random seed to test whether the improvement is consistent across independent training runs. Due to the computational cost of the full 400B-token experiments, we train this additional dense and DFFN pair for 30B tokens. As shown in Table 7,

Routing	Validation PPL	Δ%
Learned Top-1	**22.39**	–
Random Top-1	24.35	+8.75
Top-16	22.73	+1.52

Table 6: **Effect of input-dependent routing.** All variants are trained from scratch for 10B tokens at the 0.88B scale. Random routing substantially degrades performance, while activating all 16 SMs provides no improvement over learned Top-1 routing. Δ% is relative to Learned Top-1.

Model	Validation PPL	Δ%
Dense	14.73	–
DFFN	**14.01**	-4.89

Table 7: **Performance with a different random seed.** An additional 1.58B dense and DFFN pair is trained from scratch with a different random seed for 30B tokens. DFFN again achieves lower validation perplexity than the dense baseline. Δ% is relative to Dense.

DFFN again outperforms the dense baseline, reducing validation perplexity from 14.73 to 14.01, a 4.89% improvement. This result provides an independent replication of the performance advantage observed in our main experiments.

8 Limitations

Our experiments focus on decoder-only Transformers up to 2.99B parameters trained on FineWeb-Edu. While DFFN consistently improves over the parameter-matched dense baseline across the three model scales studied, experiments at larger scales, with other pretraining corpora, and across different model families would help establish the broader generality of these findings. Our study also focuses specifically on whether input-dependent modularity can improve representation learning under a fixed parameter budget, and does not attempt to characterize all possible forms of modular computation.

9 Discussion & Conclusion

We introduced DFFNs to study how far input-dependent modularity can improve Transformer pretraining when parameter and compute budgets are held fixed. Rather than increasing model capacity, DFFN reorganizes the feedforward block into a shared static FFN and routed low-rank Submodules, allowing different tokens to invoke different feedforward transformations within the same overall budget.

Across models ranging from 0.88B to 2.99B parameters, this reallocation consistently improves validation perplexity and downstream performance over parameter-matched dense Transformers. The improvement also persists after supervised fine-tuning, suggesting that the representations learned during pretraining remain useful during subsequent adaptation. Controlled routing experiments provide further evidence that the gains are not explained simply by introducing the low-rank submodules. Replacing learned routing with random assignment substantially degrades performance, while activating all submodules for every token provides no improvement over learned Top-1 routing. Together, these results indicate that learning which transformation to apply to each input is an important part of DFFN's advantage.

Our analysis provides a complementary view of what emerges from this architecture. The learned SMs occupy distinct, largely non-overlapping low-rank transformation subspaces and show little alignment with the dominant directions of the shared static FFN, while routing remains distributed across multiple SMs. These observations suggest that the routed component learns complementary transformations rather than simply reproducing the shared FFN.

More broadly, our results show that input-dependent computation need not rely on increasing total model capacity to improve representation learning. Reallocating part of an existing feedforward parameter budget toward learned modular computation can provide consistent gains while retaining the size and linear-layer compute of the corresponding dense architecture. This suggests a complementary direction for improving Transformer architectures in which existing capacity is organized more effectively rather than simply increased.

10 AI USE STATEMENT

AI-assisted tools were used for language editing and assistance with visualization code.

11 ETHICS STATEMENT

This work studies architectural modifications to Transformer language models using existing pretraining and evaluation datasets. We do not introduce new datasets containing personal or sensitive information, nor do we conduct experiments involving human subjects. The broader ethical considerations of the resulting language models are similar to those of other pretrained language models of comparable scale.

12 REPRODUCIBILITY STATEMENT

We provide the architectural specification, model configurations, training budgets, optimization setup, and evaluation protocol needed to reproduce our experiments. Additional implementation and experimental details are provided in the Appendix. We will also release the code used to train and evaluate the models.

REFERENCES

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pp. 933–941. PMLR, 2017.

Shihan Dou, Enyu Zhou, Yan Liu, Songyang Gao, Jun Zhao, Wei Shen, Yuhao Zhou, Zhiheng Xi, Xiao Wang, Xiaoran Fan, et al. Loramoe: Revolutionizing mixture of experts for maintaining world knowledge in language model alignment. *arXiv preprint arXiv:2312.09979*, 4(7), 2023.

Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pp. 5547–5569. PMLR, 2022.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL https://arxiv.org/abs/2101.03961.

Xu Owen He. Mixture of A million experts. *CoRR*, abs/2407.04153, 2024. URL https://doi.org/10.48550/arXiv.2407.04153.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pp. 2790–2799. PMLR, 2019.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. Mad-x: An adapter-based framework for multi-task cross-lingual transfer. *arXiv preprint arXiv:2005.00052*, 2020.

Noam Shazeer. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Stanford alpaca: An instruction-following llama model, 2023.

Xun Wu, Shaohan Huang, and Furu Wei. Mixture of lora experts. *arXiv preprint arXiv:2404.13628*, 2024.

Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*, 2022.

A DFFN Design Ablations

A.1 Allocating the DFFN budget: rank vs. count

A routed DFFN layer introduces two hyperparameters that control how its parameter budget is allocated: the number of routed SMs L and their rank r. To study how these choices affect performance, we sweep several configurations while fixing the rank-count product at $Lr = 256$ and adjusting D'_{ff} to preserve exact parameter matching (cf. Eq. 8).

We evaluate four settings with $Lr = 256$:

- $r = 1,\ L = 256$
- $r = 8,\ L = 32$
- $r = 16,\ L = 16$
- $r = 32,\ L = 8$

Figure 5 shows the training curves for a small-scale run for each of these choices. Among the configurations tested, (r=16, L=16) achieves the lowest loss and therefore, we adopt this setting for all large-scale experiments.

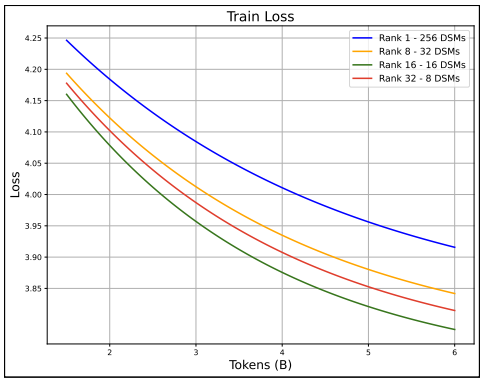


Figure 5: **DFFN rank–count sweep at constant budget.** Training loss vs. total training tokens for four configurations with fixed DFFN budget $L \times r = 256$ and matched overall parameters. Curves correspond to (r=1, L=256), (r=8, L=32), (r=16, L=16), and (r=32, L=8).

B Parameter and FLOP Matching

A. Parameter Matching

A dense FFN with hidden size H and expansion size D_{ff} has

$$P_{\text{dense}} = 2HD_{\text{ff}}. \tag{A.1}$$

A DFFN layer with reduced expansion D'_{ff}, L routed components, and rank r contains

$$P_{\text{DFFN}} = 2HD'_{\text{ff}} + Lr(H + D'_{\text{ff}}) + HL. \tag{A.2}$$

Setting $P_{\text{DFFN}} = P_{\text{dense}}$ and solving for D'_{ff} gives

$$D'_{\text{ff}} = \frac{2HD_{\text{ff}} - HL(r+1)}{2H + Lr}, \tag{A.3}$$

which determines the reduced expansion dimension needed for exact parameter matching.

B. FLOP Matching

Counting a multiply–add as two FLOPs, a dense FFN costs

$$F_{\text{dense}} = 4HD_{\text{ff}} \tag{A.4}$$

per token.

Under dense DFFN execution, the cost is

$$F_{\text{DFFN}} = 4HD'_{\text{ff}} + 2HL + 2Lr(H + D'_{\text{ff}}), \tag{A.5}$$

corresponding to the base FFN, the router projection, and the two low–rank projections.

Substituting the expression for D'_{ff} from Eq. A.3 into Eq. (A.5) yields

$$F_{\text{DFFN}} = 4HD_{\text{ff}} = F_{\text{dense}}, \tag{A.6}$$

which shows that FLOPs match exactly when the parameter counts match.

C Hyperparameters

This section lists the key hyperparameters used for training the DFFN models. We include only the values required to reproduce the architecture, routing setup, and training dynamics described in the main text. Less relevant implementation details such as logging configuration, filesystem paths, and backend flags are omitted for clarity.

All experiments use the same optimizer, learning–rate schedule, precision settings, and distributed training configuration unless noted otherwise. The hyperparameters for the 1.58B configuration are summarized in Table 8..

Category	Value
Model Architecture	
Hidden size (H)	2048
Number of layers	24
Attention heads	32
Sequence length	2048
MLP expansion	6664 (Dense), 6144 (DFFN)
Positional encoding	Rotary
Normalization	LayerNorm
Weight tying	Disabled
DFFN Configuration	
Number of DFFN components (L)	16
DFFN rank (r)	16
Top–k routing	1
Routing head dimensions	$H \times L$
Load balancing coefficient	0.01
Interaction type	Addition
Training Hyperparameters	
Optimizer	Adam
Learning rate	3×10^{-4}
Minimum learning rate	3×10^{-5}
LR schedule	Cosine decay
Warmup fraction	0.001
Batch size (per GPU)	36
Gradient accumulation	1
Weight decay	0.1
Gradient clipping	1.0
Precision	bfloat16
Regularization & Dropout	
Hidden dropout	0.0
Attention dropout	0.0
Training Duration	
Total training steps	170,000
LR decay steps	170,000
Eval interval	10,000
Eval iterations	100
Distributed Training Settings	
Model parallel size	1
Pipeline parallel size	0
ZeRO stage	1
Activation checkpointing	Enabled

Table 8: **Key hyperparameters for the 1.58B DFFN model.** Architectural, routing, and training settings for the 1.58B configuration.

D Computational Resources

Experiments were conducted on a cluster of four nodes, each equipped with eight NVIDIA A100 (80GB) GPUs. The total compute requirement for the 1.58B dense and DFFN training runs was approximately 7,330 and 7,372 GPU hours, respectively.