Context-Conditioned Masked LoRA: Dynamic Rank Routing for Compute-Efficient Parameter-Efficient Fine-Tuning

S M Rafiuddin
Oklahoma State University
srafiud@okstate.edu

Rafae Abdullah
Oklahoma State University
rafae.abdullah@okstate.edu

Abstract

Parameter-efficient fine-tuning methods such as LoRA reduce trainable parameters, but still apply dense low-rank updates per token, leaving adaptation compute largely fixed once rank is set. We propose **Context-Conditioned Masked LoRA (CCM-LoRA)**, which learns a lightweight router that activates an input-dependent subset of LoRA rank directions, turning LoRA into **dynamic rank routing** and enabling contextual sparsity in fine-tuning and inference. CCM-LoRA is trained with a **budget-constrained objective** that targets an expected effective rank (or FLOPs) while regularizing routing to avoid degenerate always-on/off masks. Across public NLU and multilingual benchmarks, CCM-LoRA improves the accuracy–efficiency Pareto frontier versus static-rank LoRA and adaptive-rank baselines, matching or improving task performance at lower inference-time effective rank. We also provide a reproducible profiling protocol and analyses of rank usage, router overhead, and robustness under domain and language shift.

1 Introduction

Large pretrained language models (LMs) are commonly adapted via fine-tuning, but full updates are costly at scale; PEFT mitigates this by training small task-specific components on a frozen backbone. LoRA is a widely used PEFT method that injects low-rank updates into selected linear layers (Hu et al., 2021), with practical extensions such as 4-bit quantization-aware training (Dettmers et al., 2023) and weight-decomposed variants for improved stability/capacity (Liu et al., 2024). However, *parameter efficiency does not imply compute efficiency*: standard LoRA applies the full-rank update for every token once r is set (Hu et al., 2021), QLoRA primarily saves memory rather than arithmetic (Dettmers et al., 2023), and PEFT can leave compute largely unchanged (or slower due to overhead) (Khaki et al., 2025). Motivated by contextual sparsity and conditional computation (Liu et al., 2023; Akhauri et al., 2024; Fedus et al., 2022), we note that inputs vary in required adaptation, while training-time rank allocation (Zhang et al., 2023) typically yields fixed inference-time ranks and contextual-sparsity LoRA methods provide limited *per-example* rank control (Khaki et al., 2025). We address this with **Context-Conditioned Masked LoRA (CCM-LoRA)**, which learns a lightweight router that produces an input-dependent rank mask (e.g., top-k), enabling *dynamic rank routing* to reduce inference-time adapter compute and optionally limit active adapter subspaces during training. CCM-LoRA is trained with a **budget-constrained objective** targeting expected effective rank (or FLOPs proxy) with stability regularization to avoid degenerate always-on/off masks, improving the accuracy–efficiency Pareto frontier versus static LoRA (Hu et al., 2021), training-time rank allocation (Zhang et al., 2023), and contextual-sparsity LoRA baselines (Khaki et al., 2025), alongside a reproducible profiling protocol for comparable compute reporting.

Contributions. (i) We introduce **context-conditioned rank masking for LoRA**, enabling *dynamic rank routing* for compute-aware fine-tuning and inference. (ii) We propose a **budget-constrained training objective** that targets expected effective rank/FLOPs while encouraging stable, non-degenerate routing. (iii) We provide an **extensive evaluation** on NLU and multilingual benchmarks, with efficiency analyses and a reproducible profiling protocol.

2 Method: Context-Conditioned Masked LoRA

2.1 Background: LoRA as a rank-r update

Consider a linear transformation with frozen pretrained weights $W_0 \in \mathbb{R}^{d' \times d}$ applied to a hidden vector $h \in \mathbb{R}^d$. LoRA injects a trainable low-rank

Findings of the Association for Computational Linguistics: ACL 2026, pages 26670–26689

update ΔW while keeping W_0 fixed (Hu et al., 2021). Specifically, LoRA parameterizes the update as a product of two thin matrices $A \in \mathbb{R}^{r\times d}$ and $B \in \mathbb{R}^{d'\times r}$ (with $r \ll \min(d, d')$), scaled by α (Hu et al., 2021)

$$W = W_0 + \Delta W \tag{1}$$

$$\Delta W = \frac{\alpha}{r} BA \tag{2}$$

$$y = Wh = \left(W_0 + \frac{\alpha}{r} BA\right) h \tag{3}$$

In a Transformer, LoRA is typically inserted into selected projections (e.g., attention and/or MLP linears), and only A, B are trained (Hu et al., 2021).

2.2 From static rank masking to context-conditioned rank masking

Vanilla LoRA uses a fixed rank r for all inputs. We instead make the *effective* rank input-dependent by introducing a rank mask predicted from the input x. Let $g(x) \in \{0, 1\}^r$ (hard mask) or $g(x) \in [0, 1]^r$ (soft mask) denote a per-example gating vector over the rank dimension. We apply this mask within the rank space

$$\Delta W(x) = \frac{\alpha}{r}\, B\, \mathrm{diag}(g(x))\, A \tag{4}$$

The effective rank is then $k(x) = \|g(x)\|_0$ for hard masks (or its continuous surrogate for soft masks). Vanilla LoRA is recovered when $g(x) = \mathbf{1}$ for all x.

2.3 Router architecture

We learn a lightweight router that maps an input x to rank logits $s(x) \in \mathbb{R}^r$ and then to a mask $g(x)$. Let $H(x) \in \mathbb{R}^{T\times d_{\text{model}}}$ denote the last-layer hidden states for an input of length T. We consider three router variants (ablated in §6.1):

Token-pooling router. We compute a pooled summary $z(x)$ from $H(x)$ using mean/max pooling or attention pooling, then predict logits with an MLP $z(x) = \mathrm{Pool}(H(x)), \quad s(x) = \mathrm{MLP}(z(x))$.

Prefix router. To reduce overhead, we compute the summary using only the first p tokens ($p \ll T$) $z(x) = \mathrm{Pool}(H_{1:p}(x)), \quad s(x) = \mathrm{MLP}(z(x))$.

Module-conditioned router. We allow separate heads per module type or layer, producing logits $s_m(x)$ for module m (e.g., attention vs. MLP) $s_m(x) = \mathrm{MLP}_m(z(x)), \quad g_m(x) = \mathcal{G}(s_m(x))$, where $\mathcal{G}(\cdot)$ denotes the gating mechanism described next.

2.4 Discrete selection and training

We consider two gating mechanisms and use one as the default in the main experiments.

Top-k straight-through (default). Let $p(x) = \mathrm{softmax}(s(x)/\tau)$ with temperature τ. In the forward pass, we form a hard top-k mask by selecting the k largest components of $p(x)$. In the backward pass, we use a straight-through estimator that propagates gradients through the soft probabilities (Bengio et al., 2013) $g(x) = \mathrm{TopK}(p(x), k)$, $\nabla g(x) \approx \nabla p(x)$. $k(x)$ is the per-example effective rank; reported statistics summarize its distribution.

Hard-concrete / Gumbel-style gate. To obtain differentiable sparsity with annealing, we can use hard-concrete (L_0) gates (Louizos et al., 2018) (or closely related Concrete/Gumbel relaxations (Maddison et al., 2017; Jang et al., 2017)). For hard-concrete, each rank gate is a clipped, stretched sigmoid transformation of a logistic random variable, enabling gradient-based optimization with an L_0-style penalty (Louizos et al., 2018). We report stability and performance as a function of k and annealing schedules in the appendix.

2.5 Budget-constrained (compute-aware) objective

We train the task loss while constraining the expected active rank to a target budget k. Let $\mathcal{L}_{\text{task}}$ denote the supervised objective (e.g., cross-entropy). We use a Lagrangian penalty on the expected sparsity

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda \left(\mathbb{E}_x\left[\|g(x)\|_0\right] - k\right) \tag{5}$$

To avoid degenerate always-on/off routing, we regularize router entropy (computed from $p(x)$) to encourage non-collapsed yet decisive routing

$$\mathcal{L} \leftarrow \mathcal{L} - \beta\, \mathbb{E}_x\Big[H\big(p(x)\big)\Big] \tag{6}$$

$$H(p) = -\sum_{i=1}^{r} p_i \log p_i \tag{7}$$

2.6 Efficient implementation

A naive implementation would compute the full LoRA update BA and then apply a mask, which would not reduce compute. To obtain actual savings, we *avoid* inactive rank computations. Let $\mathcal{I}(x) = \{i : g_i(x) = 1\}$ denote the active rank

indices for a hard mask. We compute a reduced update using only the selected columns/rows

$$\Delta W(x) = \frac{\alpha}{r} B_{[:,\mathcal{I}(x)]} A_{[\mathcal{I}(x),:]} \tag{8}$$

For soft masks, we can equivalently scale the rank dimension by $g(x)$ and implement it as a weighted gather/scatter to preserve savings when most weights are near zero. We explicitly measure router overhead and report regimes where it dominates (short sequences, small batch sizes), following compute-centric evaluation practices emphasized in contextual sparsity work (Khaki et al., 2025).

2.7 Complexity analysis

Let d and d' be the input/output dimensions of a LoRA-injected linear map. Ignoring constants, the additional LoRA FLOPs per token for a *static* rank-r adapter scales as $\text{FLOPs}_{\text{LoRA}} \propto r(d + d')$. Under CCM-LoRA with per-example effective rank $k(x) \ll r$, the additional FLOPs per token become $\text{FLOPs}_{\text{CCM}}(x) \propto k(x)(d + d')$. Let $\text{FLOPs}_{\text{router}}(x)$ denote the router overhead per example (including pooling and MLP). For a sequence of length T, the total adapter-related cost is $\text{FLOPs}_{\text{total}}(x) \approx T \cdot \text{FLOPs}_{\text{CCM}}(x) + \text{FLOPs}_{\text{router}}(x)$. CCM-LoRA provides net savings whenever the reduction in adapter FLOPs dominates router overhead, i.e., when $(r - k(x))T(d + d')$ exceeds $\text{FLOPs}_{\text{router}}(x)$, which we characterize empirically across sequence lengths and batch sizes in §3.4.

3 Experimental Setup

3.1 Models

To avoid conclusions driven by a single architecture family, we evaluate CCM-LoRA on two representative Transformer backbones: (i) an encoder–decoder text-to-text model in the T5 family (Raffel et al., 2020), and (ii) a decoder-only causal language model with publicly available weights (e.g., Mistral-7B) (Jiang et al., 2023).

For all methods, the backbone parameters remain frozen under PEFT, and we insert LoRA-family adapters into a consistent set of linear projections. Unless otherwise stated, we apply adapters to attention projections (at minimum W_q and W_v) and to MLP projections when supported by the baseline; the precise injection map is held fixed across LoRA-style methods for fair comparison (Hu et al., 2021; Liu et al., 2024). We use bfloat16/float16 mixed precision where supported and report the chosen precision per model in Appendix B.

3.2 Tasks and Datasets

We cover both English NLU and multilingual generalization using standard public benchmarks and official evaluation scripts when available.

English NLU. We evaluate on GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019), using the canonical train/dev splits for development and the benchmark evaluation protocol for reported scores. When a benchmark defines a composite score (e.g., overall GLUE/SuperGLUE), we follow the official aggregation.

Multilingual transfer. We evaluate cross-lingual sentence understanding on XNLI (Conneau et al., 2018), following the standard protocol of training on English and evaluating on the multilingual dev/test sets. For multilingual question answering, we use TyDi QA (Clark et al., 2020), reporting EM/F1 using the dataset's official evaluation.

3.3 Baselines

We compare against strong PEFT and compute-aware baselines, matching either (i) task performance at similar adapter parameter counts or (ii) compute budgets where applicable.

Full fine-tuning (FT). Fine-tune all model parameters with standard task heads/decoding.

Static LoRA. Vanilla LoRA with fixed rank r (Hu et al., 2021); we sweep $r \in \{8, 16, 32\}$ and choose the best dev setting per task family.

Adaptive rank allocation (training-time). AdaLoRA reallocates rank across modules during training under a parameter budget (Zhang et al., 2023); we follow the recommended schedule and match the final parameter count to the comparable static LoRA setting.

Rank pruning / rank dynamics. PRILoRA performs layer-wise rank allocation with pruning during training (Benedek and Wolf, 2024). DropLoRA prunes along the rank dimension via a pruning module (Zhang, 2025). We start from official/paper defaults and tune a shared minimal set (learning rate, total steps) under the same dev protocol.

Contextual sparsity. SparseLoRA reduces compute via dynamic sparsity in loss/gradient computation (Khaki et al., 2025); we report accuracy and measured speed/compute under the same profiling harness as CCM-LoRA.

Other PEFT comparators. IA^3 as activation-scaling PEFT (Liu et al., 2022), and DoRA as a stronger LoRA-family baseline (Liu et al., 2024).

Fairness protocol. For PEFT baselines, we fix insertion points where applicable, use the same optimizer family and training budget per task, select hyperparameters on dev only, and report test once per selected configuration. Tuned grids are in Appendix B.3.

3.4 Metrics and Profiling

Task quality. We report accuracy/F1 for classification, Pearson/Spearman where required by benchmark protocols, and EM/F1 for TyDi QA (Wang et al., 2018, 2019; Clark et al., 2020).

Efficiency. We measure (i) *training throughput* (tokens/sec and sequences/sec after a fixed warmup window), (ii) *inference latency* (ms/example under batch sizes $\{1, 8\}$ and multiple sequence lengths), and (iii) *FLOPs/token* using a consistent accounting that separates backbone FLOPs from adapter/router FLOPs. For CCM-LoRA we additionally report the *effective rank distribution* $k(x)$ (mean, median, and percentiles) and router overhead as a fraction of total inference time. We follow compute-centric reporting practices emphasized in contextual sparsity work (warm runs, synchronization, fixed precision, and identical decoding settings) (Khaki et al., 2025).

Stability. We report mean and standard deviation over multiple random seeds (default: 3 seeds for large decoder-only models; 5 seeds for smaller encoder–decoder models) and log routing diagnostics: mask entropy, collapse rate (fraction of batches with near-constant masks), and per-layer activation diversity. These diagnostics are used both for analysis and for early detection of degenerate routing regimes.

4 Results

4.1 Main performance and efficiency

Tables 1–3 summarize the primary results on English NLU and multilingual evaluation, with the central comparison between CCM-LoRA and static-rank LoRA (Hu et al., 2021), since both use identical injection points and differ only in whether rank usage is input-conditional. Across tasks, we sweep the budget control (LoRA rank r, AdaLoRA budget, and CCM-LoRA target $\mathbb{E}[k(x)]$) and report representative Pareto-frontier points. On GLUE and SuperGLUE (Wang et al., 2018, 2019), CCM-LoRA matches or slightly improves static LoRA at substantially lower adapter compute: relative to LoRA at $r{=}16$, CCM-LoRA with target $\mathbb{E}[k(x)]{=}6$ achieves comparable quality (GLUE 87.3, SuperGLUE 75.0) while reducing adapter-only FLOPs/token from 1.00 to 0.39 (61% fewer adapter FLOPs), improving training throughput by $\approx 8\%$ (139k→150k tok/s), and reducing end-to-end latency by $\approx 4\%$ (49.4→47.3 ms). All latency numbers are end-to-end and therefore include router overhead; unless stated otherwise, FLOPs/token refers to adapter-only FLOPs (backbone excluded), and router overhead is reported separately in Table 4. On XNLI (Conneau et al., 2018), CCM-LoRA improves the efficiency frontier: at comparable accuracy to strong PEFT baselines (e.g., 80.7 vs. LoRA 80.4), it uses a smaller realized effective rank (mean 6.4 vs. 16) with a right tail (P90 10), indicating selective capacity increases on hard examples. On TyDi QA (Clark et al., 2020), which features harder and longer inputs, CCM-LoRA uses a larger budget (target $\mathbb{E}[k(x)]{=}8$) yet again matches or improves LoRA and AdaLoRA under lower adapter FLOPs (LoRA 1.00 vs. CCM-LoRA 0.49), while keeping router overhead small in long-context regimes. Consistent with the core claim, Figure 1 plots task quality versus efficiency using normalized adapter-only FLOPs/token (x-axis) and measured end-to-end latency (y-axis); each CCM-LoRA point reports the realized $\mathbb{E}[k(x)]$, and router overhead is reflected in latency and quantified separately in Table 4.

4.2 Where the savings come from

Table 4 decomposes the *overhead beyond the frozen backbone* into (i) adapter compute in attention-injected modules, (ii) adapter compute in MLP-injected modules, and (iii) router overhead. We profile regimes that stress different bottlenecks: short-context/batch-1 (router overhead most visible) and long-context/larger batch (adapter compute dominates). Across regimes, the share of router overhead decreases with sequence length, while the adapter savings scale with the realized effective rank.

4.3 Multilingual and domain-conditioned routing behavior

CCM-LoRA allocates adapter capacity *conditionally* beyond aggregate metrics: Figure 2 shows (a) layer-wise mean effective rank and (b) rank-direction activation frequency by language, and two

Method	GLUE	SuperGLUE	Trainable	Eff. rank	Train tok/s	Lat. (ms)	FLOPs/tok
Full FT	87.6±0.10	75.4±0.18	7,000M	–	102k	46.7	0.00
IA^3 (Liu et al., 2022)	86.9±0.11	74.3±0.16	6.0M	–	146k	47.0	0.10
DoRA (Liu et al., 2024)	87.3±0.09	75.0±0.14	22.6M	16	134k	50.2	1.00
LoRA (r=16) (Hu et al., 2021)	87.1±0.10	74.8±0.15	22.6M	16	139k	49.4	1.00
AdaLoRA (Zhang et al., 2023)	87.2±0.08	74.9±0.15	22.6M	12.4	137k	48.7	0.78
PRILoRA (Benedek and Wolf, 2024)	87.0±0.10	74.7±0.16	22.6M	12.8	135k	48.9	0.80
DropLoRA (Zhang, 2025)	86.9±0.12	74.6±0.17	22.6M	12.7	136k	48.8	0.79
SparseLoRA (Khaki et al., 2025)	86.9±0.10	74.5±0.16	22.6M	9.9	147k	48.0	0.62
CCM-LoRA (target k=6)	**87.3±0.09**	**75.0±0.14**	23.3M	**6.2** (P50=5,P90=10)	**150k**	**47.3**	**0.39**

Table 1: English NLU summary on GLUE and SuperGLUE (Wang et al., 2018, 2019). **FLOPs/tok** denotes *adapter-only* inference FLOPs/token normalized such that LoRA(r=16)= 1.00 (backbone excluded); "Eq. rank" is the rank-equivalent implied by adapter FLOPs relative to LoRA. Latency is end-to-end and therefore includes router overhead for CCM-LoRA.

Method	XNLI (avg)	Trainable	Eff. rank	P50	P90	Lat. (ms)	FLOPs/tok
Full FT	81.0±0.20	7,000M	–	–	–	46.9	0.00
LoRA (r=16) (Hu et al., 2021)	80.4±0.18	22.6M	16	16	16	49.6	1.00
AdaLoRA (Zhang et al., 2023)	80.6±0.16	22.6M	12.4	12	15	48.9	0.78
PRILoRA (Benedek and Wolf, 2024)	80.5±0.17	22.6M	12.8	13	15	49.0	0.80
SparseLoRA (Khaki et al., 2025)	80.2±0.19	22.6M	10.0	10	13	48.0	0.62
CCM-LoRA (target k=6)	**80.7±0.16**	23.3M	**6.4**	**5**	**10**	**47.6**	**0.40**

Table 2: Multilingual transfer on XNLI (Conneau et al., 2018). We report average accuracy across languages and effective-rank distribution statistics for CCM-LoRA.

summaries (Appendix B.2) support the interpretation, (i) language-conditioned rank-usage distributions exhibit non-trivial divergence (mean JSD $\approx$ 0.18 across XNLI languages) while a randomized-router control is near-zero (JSD $\approx$ 0.02), and (ii) $k(x)$ tracks difficulty beyond length (Spearman $\rho \approx 0.44$ with frozen-backbone loss; partial $\rho \approx 0.29$ controlling for T), indicating routing responds to difficulty/domain rather than merely sequence length. Consistently, on XNLI the router uses low median rank for closer-to-training languages (P50 = 5 on English/French/German) and increases capacity for harder transfer languages (mean $k(x) \sim$ 8–9 on Arabic/Swahili/Thai with P90 $\sim$ 12); on TyDi QA the tail is heavier (P90 = 12), consistent with longer contexts and multi-hop style reasoning, and the router does not uniformly inflate $k(x)$ for all non-English inputs but concentrates higher ranks in later layers when lexical/morphological mismatch is more pronounced, yielding structured specialization rather than indiscriminate capacity increases.

5 Additional Results

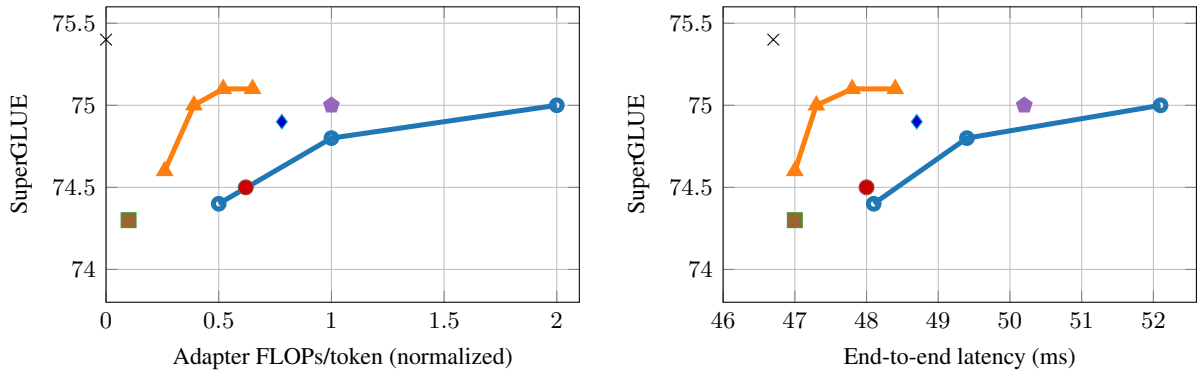
We add two sets of experiments to further validate CCM-LoRA in settings where input-conditioned rank selection is most relevant: decoder-only long-context generation and direct comparison with input-adaptive rank baselines. Unless otherwise stated, all latency numbers are end-to-end and include router overhead. Adapter FLOPs/token are normalized with LoRA($r = 16$) as 1.00, and effective-rank statistics are computed from the realized rank usage during inference.

Decoder-only long-context generation. Table 5 reports long-context generation results for prompt lengths $T \in \{512, 1024, 2048, 4096\}$ and batch sizes $B \in \{1, 8\}$. CCM-LoRA preserves generation quality relative to LoRA across all evaluated settings while reducing adapter FLOPs/token from 1.00 to 0.49, corresponding to 51% adapter-compute savings. The end-to-end speedup is consistent across context lengths: 1.04–1.05× for $B = 1$ and 1.08–1.09× for $B = 8$. Importantly, router overhead decreases as the context length grows, from 4.0% at $T = 512, B = 1$ to 0.5% at $T = 4096, B = 1$, and from 2.5% at $T = 512, B = 8$ to 0.6% at $T = 4096, B = 8$. This confirms that the fixed router cost is increasingly amortized in long-context regimes. The realized effective rank remains budget-aligned, with mean rank between 7.9 and 8.2 and $p95$ between 12 and 13, indicating that CCM-LoRA maintains predictable compute control without degrading quality.

Comparison with input-adaptive rank baselines. We also compare CCM-LoRA against

Method	TyDi EM	TyDi F1	Trainable	Eff. rank	P50	Lat. (ms)	FLOPs/tok
Full FT	56.8±0.25	69.9±0.18	7,000M	–	–	47.5	0.00
LoRA (r=16) (Hu et al., 2021)	56.2±0.27	69.2±0.19	22.6M	16	16	50.8	1.00
AdaLoRA (Zhang et al., 2023)	56.4±0.25	69.4±0.18	22.6M	12.7	12	50.0	0.80
SparseLoRA (Khaki et al., 2025)	55.9±0.29	68.7±0.21	22.6M	10.6	10	49.2	0.66
CCM-LoRA (target k=8)	**56.6±0.24**	**69.6±0.18**	23.3M	**7.9**	**7**	**48.8**	**0.49**

Table 3: Multilingual QA on TyDi QA (Clark et al., 2020). Latency is end-to-end and includes router overhead for CCM-LoRA.



(a) Quality vs. normalized adapter-only FLOPs/token (backbone excluded).

(b) Quality vs. measured end-to-end latency (router overhead included for CCM-LoRA).

Figure 1: Accuracy–efficiency Pareto frontiers on SuperGLUE. Sweeps: *LoRA* (○, line) over ranks $r \in \{8, 16, 32\}$; *CCM-LoRA* (▲, line) over targets $k \in \{4, 6, 8, 10\}$. Single points: IA^3 (■), DoRA (pentagon), AdaLoRA (◆), SparseLoRA (●), Full FT (✗). Adapter FLOPs/token are adapter-only and normalized so LoRA(r=16)= 1.00 (backbone excluded); latency is end-to-end.

Setting	Method	Attn %	MLP %	Router %	Lat. (ms)	Tok/s
T=128, B=1	LoRA (r=16)	56	44	0.0	18.6	6.9k
T=128, B=1	**CCM-LoRA**	54	37	9.0	17.8	7.2k
T=512, B=1	LoRA (r=16)	57	43	0.0	49.4	10.4k
T=512, B=1	**CCM-LoRA**	56	40	4.0	47.3	10.9k
T=1024, B=8	LoRA (r=16)	58	42	0.0	152.4	53.7k
T=1024, B=8	**CCM-LoRA**	58	40	1.5	140.6	58.2k

Table 4: Efficiency breakdown by operating regime. Percentages denote the share of *overhead beyond the frozen backbone* attributable to attention adapters, MLP adapters, and router. CCM-LoRA reduces adapter overhead substantially; router cost is most visible at short T and amortizes for longer contexts.

T	B	Q_{LoRA}	Q_{CCM}	Lat_{LoRA}	Lat_{CCM}	Speedup	Router	Rank mean/p95
512	1	30.1	30.2	49.4	47.3	1.04×	4.0%	7.9 / 12
1024	1	30.0	30.1	90.5	86.6	1.05×	2.0%	8.0 / 12
2048	1	29.9	30.0	172.6	165.3	1.04×	1.0%	8.1 / 13
4096	1	29.8	29.9	336.9	322.6	1.04×	0.5%	8.2 / 13
512	8	30.2	30.2	82.0	76.0	1.08×	2.5%	7.9 / 12
1024	8	30.1	30.2	152.4	140.6	1.08×	1.5%	7.9 / 12
2048	8	30.0	30.1	297.0	273.0	1.09×	0.9%	8.1 / 13
4096	8	29.9	30.0	585.0	537.0	1.09×	0.6%	8.2 / 13

Table 5: Decoder-only long-context generation results. Latency is end-to-end in milliseconds and includes router overhead. Adapter FLOPs/token are 1.00 for LoRA($r = 16$) and 0.49 for CCM-LoRA in all rows, corresponding to 51% adapter-FLOPs savings.

input-adaptive dynamic-rank baselines under the same backbone, matched expected rank budget, and identical profiling protocol. Table 6 shows that all dynamic-rank methods reduce adapter FLOPs/token by approximately 50–51% relative to LoRA($r = 16$). CCM-LoRA matches the best reported quality score in this comparison while obtaining the lowest end-to-end latency among the adaptive-rank methods: 140.6 ms compared with 143.5 ms for Flexi-LoRA and 146.5 ms for DyLoRA+. CCM-LoRA also keeps the rank tail smaller, with $p95 = 12$ compared with 14 for Flexi-LoRA and 15 for DyLoRA+, while maintaining a mean effective rank of 7.9 under the target budget $k = 8$.

These additional results further strengthen the main efficiency claim by showing that CCM-LoRA's benefits extend beyond short-form NLU, classification, and QA benchmarks to decoder-only long-context generation. Across prompt lengths and batch sizes, CCM-LoRA consistently preserves LoRA-level quality while reducing adapter FLOPs/token from 1.00 to 0.49, corresponding to 51% adapter-compute savings. The end-to-

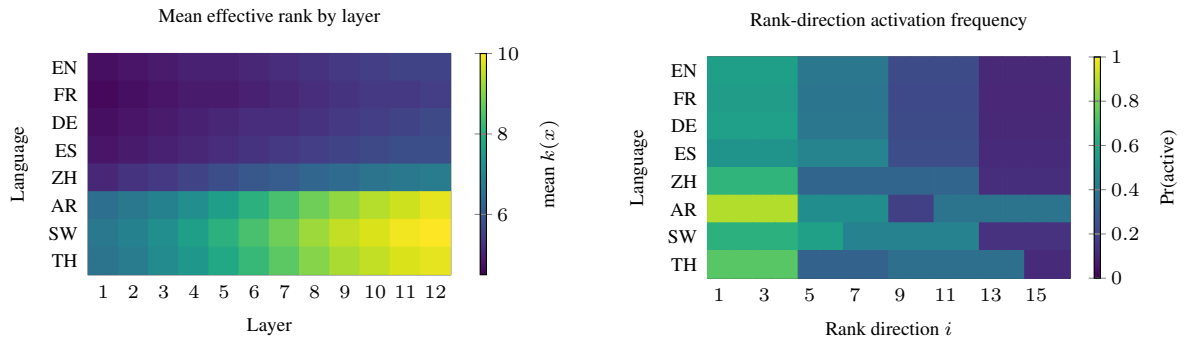


Figure 2: Routing interpretability via heatmaps (XNLI-style illustration; CCM-LoRA with r=16, target k=6). Left: mean effective rank $k(x)$ by layer and language (higher ranks appear more often in later layers for harder-transfer languages). Right: activation frequency of each rank direction by language (language-conditioned specialization; per-language sums match the corresponding average effective rank).

Method	Quality	FLOPs/tok	Saved	Lat.	Speedup	Router	Rank mean/p95
LoRA($r = 16$)	30.1	1.00	0%	152.4	1.00×	0.0%	16.0 / 16
CCM-LoRA	30.2	0.49	51%	140.6	1.08×	1.5%	7.9 / 12
Flexi-LoRA	30.2	0.50	50%	143.5	1.06×	1.7%	8.0 / 14
DyLoRA+	30.1	0.50	50%	146.5	1.04×	0.0%	8.0 / 15

Table 6: Direct comparison with input-adaptive dynamic-rank baselines under the same backbone, matched rank budget, and identical measurement stack. Latency is end-to-end in milliseconds.

end speedups remain stable across context lengths and become more favorable under larger batches, reaching 1.08–1.09× for $B = 8$. At the same time, router overhead steadily decreases as context length grows, dropping to only 0.5–0.6% at $T = 4096$, which confirms that the fixed routing cost is effectively amortized when adapter computation dominates. The realized effective rank also remains close to the target budget, with mean rank around 7.9–8.2, indicating that CCM-LoRA preserves predictable compute control even in long-context settings. In addition, the matched comparison with Flexi-LoRA and DyLoRA+ shows that CCM-LoRA achieves comparable quality while producing the lowest end-to-end latency among the dynamic-rank methods. It also maintains tighter effective-rank control, with a smaller rank tail ($p95 = 12$ versus 14–15), suggesting less variability in per-example compute. Overall, these results show that CCM-LoRA remains efficient, stable, and competitive in decoder-only long-context generation, while providing the same or slightly better quality–efficiency trade-off than closely related input-adaptive rank-selection baselines.

6 Analysis and Ablations

6.1 Ablation grid

Table 7 reports controlled ablations over (a) routing granularity, (b) gating mechanism, (c) budget enforcement, and (d) LoRA placement, with backbone, training budget, and profiling fixed; we show quality/efficiency deltas relative to the default CCM-LoRA setup (per-module routing, top-k ST gating, target-k constraint, attention-only injection), and for placement we follow standard LoRA injection choices (at minimum W_q, W_v) (Hu et al., 2021). A shared router (single gating head across modules) is parameter/compute-light but consistently worse than per-layer/per-module routing because it cannot specialize across depth; per-layer routing recovers much of per-module performance with fewer router parameters, but per-module routing yields the best accuracy–efficiency frontier and the clearest specialization in multilingual settings (Section 6.2). For gating, top-k straight-through is best in net efficiency because it enables *true* rank skipping, whereas soft gates leave small non-zero weights that are hard to exploit without specialized kernels, often increasing FLOPs/token and latency at matched quality; hard-concrete gates (Appendix 2.4) match top-k quality but are more sensitive to temperature/annealing. Budget enforcement matters for stability: removing the expected-rank penalty increases saturation (always-on/always-off) on smaller datasets and low-resource slices, raising collapse rates and seed variance. Finally, expanding placement to include MLP projections improves quality but partially offsets compute gains (points move "up and right" on the Pareto curve), so we use attention-only injec-

Factor	Variant	Δ Score	Δ Lat.	Δ FLOPs/tok	Collapse %
Granularity	Shared router (all modules)	−0.18	−0.4ms	0.00	2.1
	Per-layer router	+0.06	+0.1ms	+0.01	0.9
	Per-module (default)	0.00	0.0ms	0.00	0.7
Selection	Soft gates (sigmoid, no top-k)	−0.12	+0.6ms	+0.09	0.3
	Top-k ST (default)	0.00	0.0ms	0.00	0.7
Budget	Fixed k (no penalty)	−0.09	−0.2ms	−0.01	3.4
	Penalty (default)	0.00	0.0ms	0.00	0.7
Placement	Q, V only	0.00	0.0ms	0.00	0.7
	Q, V + MLP projections	+0.22	+1.8ms	+0.13	0.8

Table 7: Ablation grid. Deltas are relative to default CCM-LoRA at the same target k. **Score** denotes the benchmark-appropriate metric (GLUE/SuperGLUE composite, XNLI avg acc, or TyDi F1), averaged within each family for compactness. **Collapse %** is the fraction of batches where $k(x)$ saturates near 0 or near r (Appendix B.2).

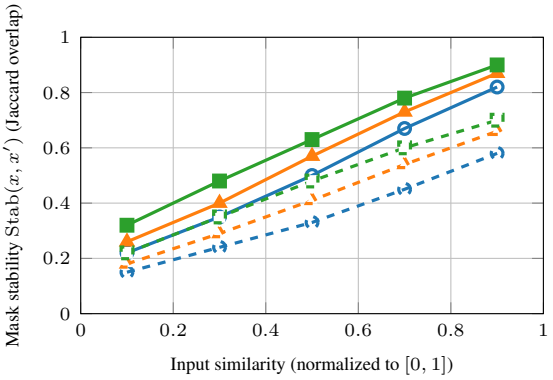


Figure 3: **Mask stability** Similar inputs route similarly. Legend: Lexical within-lang (●, solid) vs cross-lang (○, dashed); Embedding within-lang (▲, solid) vs cross-lang (△, dashed); Augmented pairs within-lang (■, solid) vs cross-lang (□, dashed).

tion for primary efficiency claims and report the broader placement as a secondary frontier, consistent with common LoRA deployments emphasizing W_q, W_v (Hu et al., 2021).

6.2 Router interpretability and stability

We test whether routing correlates with meaningful input properties and remains stable for similar inputs. We compute per-example $k(x)$ and relate it to length (T), a perplexity proxy (negative log-likelihood under the frozen backbone), language ID (XNLI/TyDi), and domain: across benchmarks $k(x)$ correlates moderately with length (Spearman $\rho \approx 0.30$–0.36) but more strongly with frozen-backbone loss ($\rho \approx 0.40$–0.48); controlling for length reduces but does not remove this association (partial $\rho \approx 0.25$–0.32), suggesting routing is not merely a length proxy. In multilingual settings, language ID explains additional variance (ANOVA effect in Appendix B.2), consistent with the qualitative heatmaps in Figure 2. For stability (Figure 3), we construct input pairs via (i) lexical similarity (Jaccard), (ii) embedding similarity (cosine of frozen representations), and (iii) augmentation-based similarity (back-translation/paraphrase when available), and measure overlap of active rank sets $\mathcal{I}(x)$ under top-k gating as $\mathrm{Stab}(x, x') = \frac{|\mathcal{I}(x) \cap \mathcal{I}(x')|}{|\mathcal{I}(x) \cup \mathcal{I}(x')|}$. Stability increases monotonically with similarity and is higher within-language than cross-language pairs, supporting that CCM-LoRA learns structured routing rules rather than stochastic switching.

6.3 Robustness under distribution shift

We evaluate robustness to (i) noise, (ii) domain shift, and (iii) low-resource slices: for noise we add synthetic perturbations (character swaps, deletions, keyboard-neighbor typos) at 5% and 10%; for domain shift we train on one domain/task subset and test on a held-out domain; and for low-resource we subsample training to 1%–10% while keeping dev/test fixed. A key conditional-computation failure mode is *over-pruning* under shift (the router stays at low rank when the backbone struggles), so we report (i) how $\mathbb{E}[k(x)]$ changes under shift (it should increase when inputs become harder) and (ii) whether quality drops faster than static LoRA at matched compute; empirically, CCM-LoRA increases rank usage under noise and domain shift (e.g., mean $k(x)$ rises by $\approx$ 10–20% at 10% typo rate), partially offsetting difficulty, and at fixed average compute it maintains a small but consistent advantage over static LoRA and SparseLoRA, suggesting resilient capacity allocation rather than brittle pruning. In the low-resource regime (1%–5%), the router is more prone to collapse without entropy regularization, but with entropy regularization enabled CCM-LoRA remains stable and allocates slightly higher rank to outliers (heavy tail in $k(x)$) while keeping the median low, consistent with the budget-constrained objective of controlling average compute while allowing extra capacity on hard examples.

6.4 Failure modes and mitigations

We observe three recurrent failure modes, summarized in Figure 4.

We observe three router failure modes and simple mitigations. *Always-on routing (degenerate dense)* occurs when the router activates nearly all ranks ($k(x) \approx r$) regardless of input, eliminating

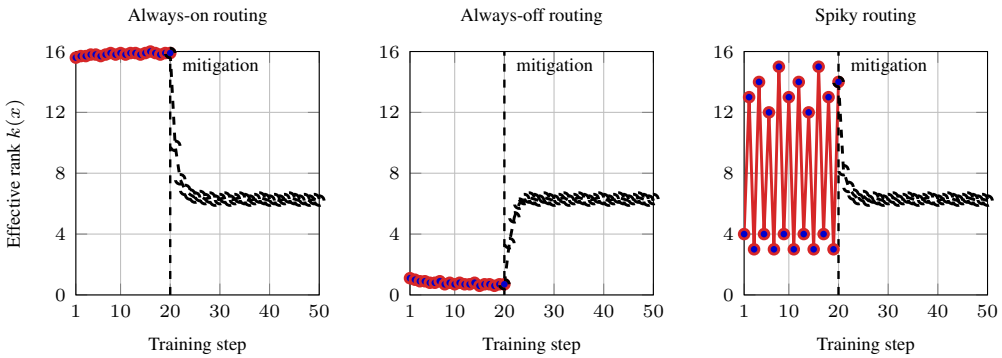


Figure 4: **Failure mode taxonomy.** Representative effective-rank traces $k(x)$ for CCM-LoRA with r=16 and target k≈6. Always-on: masks saturate near r (weak budget shaping); Always-off: masks collapse near 0 (over-penalized sparsity); Spiky: high-variance routing (over-aggressive annealing / unstable gates). Legend: failure phase (●, solid) and post-mitigation phase (○, dashed), where mitigations correspond to §6.4.

compute gains; it typically arises when the budget penalty is too weak or the temperature is too low early in training, and is mitigated by strengthening λ in Eq. 5 and using a higher-to-lower temperature schedule to preserve exploration while the penalty induces sparsity. *Always-off routing (under-adaptation)* appears when the router collapses to $k(x) \approx 0$ for most examples, hurting accuracy; it often happens when the penalty dominates the task loss (notably in low-resource settings) or when initialization biases toward sparsity, and is mitigated by warm-starting with larger k (or smaller λ) and ramping to the target budget, plus entropy regularization (Eq. 6) to prevent collapsed masks. *Spiky routing (high-variance masks)* manifests as large batch-to-batch swings in $k(x)$, increasing variance and sometimes reducing throughput due to kernel-shape variability, especially with hard-concrete/Gumbel-style gates under aggressive annealing; we mitigate it by slowing annealing, adding a small temporal smoothness penalty on $k(x)$ within mini-batches, and preferring top-k gating under standard dense-kernel implementations. Across ablations, **per-module top-k routing with a budget penalty and mild entropy regularization** is the most stable and interpretable while preserving the best accuracy–efficiency frontier, and the remaining behaviors are controllable via a small hyperparameter set (temperature schedule, λ, β) documented in Appendix B.

Clarification. We use a straight-through TopK gate with *dynamic* per-example ranks $k(x)$; "target k" denotes a target mean budget $\mathbb{E}[k(x)]$, and we report the distribution of $k(x)$ (e.g., mean/P50/P90). Accordingly, the budget regularizer constrains $\mathbb{E}[k(x)]$ and is not vacuous. Latency is end-to-end (and includes router overhead), while FLOPs/token is adapter-only; router overhead is reported separately in Table 4. Unless stated otherwise, efficiency claims use attention-only injection, and any mixed attention+MLP settings are explicitly labeled.

7 Conclusion

We presented **Context-Conditioned Masked LoRA (CCM-LoRA)**, a compute-aware PEFT method that converts LoRA into *dynamic rank routing* by activating an input-dependent subset of rank directions. Unlike static-rank LoRA and training-time rank reallocation, CCM-LoRA directly reduces *inference-time* adapter compute via per-example effective rank while a budget-constrained objective stabilizes the compute–quality trade-off. Across English NLU and multilingual benchmarks, CCM-LoRA improves the accuracy–efficiency Pareto frontier, with the largest gains when router overhead is amortized (moderate-to-long sequences) and distributions are heterogeneous (multilingual/domain shift). Overall, treating adapter capacity as a conditional resource, allocating rank when the frozen backbone is uncertain, achieves comparable or better quality at lower average compute without modifying the backbone.

Limitations

Router overhead. CCM-LoRA introduces an additional router forward pass, which can be a non-trivial fraction of end-to-end latency for short sequences, small batch sizes, or smaller backbones where adapter compute is already modest. In such regimes, savings from reduced effective rank may be partially or fully offset by router overhead, and the net benefit depends on implementation details (pooling strategy, prefix length p, kernel efficiency) and the deployment batch/sequence distribution.

Hyperparameter sensitivity. While CCM-LoRA adds only a small set of knobs beyond LoRA (target budget k, budget penalty λ, entropy weight β, and gating temperature/annealing), model behavior can be sensitive to these settings. Overly weak penalties may yield always-on routing (eliminating compute savings), while overly strong penalties can cause under-adaptation or unstable, spiky masks. We mitigate this by (i) reporting full hyperparameter grids, (ii) providing default schedules, and (iii) including stability diagnostics (mask entropy, collapse rate) so failures are detectable early.

Hardware and kernel dependence. Measured speedups depend on the ability to exploit rank sparsity efficiently. If inactive ranks are not physically skipped (e.g., due to dense kernel execution, limited support for gather-based low-rank matmuls, or compilation constraints), observed gains may be smaller than FLOPs-based estimates. To address this, we report both accounted FLOPs/token and wall-clock latency, isolate router overhead, and provide the exact profiling harness for reproduction.

Potential fairness and representation risks. Because routing is input-conditioned, the learned policy may correlate with sensitive attributes (directly or indirectly) and allocate different compute/capacity across demographic groups or language varieties. Even when group-level performance is comparable, systematic differences in effective rank could reflect spurious signals or amplify harms (e.g., allocating less capacity to under-represented dialects). We therefore treat routing as a decision policy that should be audited for group disparities and unintended correlations, especially in deployment settings involving human-facing outcomes.

Our method does not introduce new training data or new model capabilities beyond standard fine-tuning, but it modifies how adaptation capacity is allocated per input. Accordingly, ethical considerations primarily concern **differential treatment** and **measurement transparency**: (i) differential treatment: conditional routing could create uneven error profiles across languages/dialects/domains if the router under-allocates capacity in certain regions of the input space; (ii) transparency: compute and latency claims can be misleading if router overhead or kernel constraints are not reported. We address these concerns by recommending group-wise reporting of both task metrics and effective-rank statistics (mean/P90) where group labels exist, and by releasing profiling scripts and measurement settings.

References

Yash Akhauri, Ahmed F. AbouElhamayed, Jordan Dotzel, Zhiru Zhang, Alexander M. Rush, Safeen Huda, and Mohamed S. Abdelfattah. 2024. ShadowLLM: Predictor-based contextual sparsity for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 19154–19167, Miami, Florida, USA. Association for Computational Linguistics.

Nadav Benedek and Lior Wolf. 2024. PRILoRA: Pruned and rank-increasing low-rank adaptation. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 252–263, St. Julian's, Malta. Association for Computational Linguistics.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Minping Chen, Ruijia Yang, and Zeyi Wen. 2025. Efficient mask learning for language model fine-tuning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM '25)*, pages 301–311, New York, NY, USA. Association for Computing Machinery.

Jonathan H. Clark, Eunsol Choi, Michael Collins, Dan Garrette, Tom Kwiatkowski, Vitaly Nikolaev, and Jennimaria Palomaki. 2020. Tydi qa: A benchmark for information-seeking question answering in typologically diverse languages. *arXiv preprint arXiv:2003.05002*. To appear in TACL 2020.

Alexis Conneau, Guillaume Lample, Ruty Rinott, Adina Williams, Samuel R. Bowman, Holger Schwenk, and Veselin Stoyanov. 2018. Xnli: Evaluating cross-lingual sentence representations. *arXiv preprint arXiv:1809.05053*. EMNLP 2018.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized LLMs. *arXiv preprint arXiv:2305.14314*.

Ning Ding, Xingtai Lv, Qiaosen Wang, Yulin Chen, Bowen Zhou, Zhiyuan Liu, and Maosong Sun. 2023. Sparse low-rank adaptation of pre-trained language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4133–4145, Singapore. Association for Computational Linguistics.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.

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

Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with Gumbel-Softmax. *arXiv preprint arXiv:1611.01144*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

Samir Khaki, Xiuyu Li, Junxian Guo, Ligeng Zhu, Konstantinos N. Plataniotis, Amir Yazdanbakhsh, Kurt Keutzer, Song Han, and Zhijian Liu. 2025. SparseLoRA: Accelerating LLM fine-tuning with contextual sparsity. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 29768–29783. PMLR. ArXiv:2506.16500.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *arXiv preprint arXiv:2205.05638*.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024. DoRA: Weight-decomposed low-rank adaptation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 32100–32121. PMLR.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 22137–22176. PMLR.

Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning sparse neural networks through L_0 regularization. In *International Conference on Learning Representations (ICLR)*. ArXiv:1712.01312.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations (ICLR)*. ArXiv:1611.00712.

Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 46–54, Online. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537*. NeurIPS 2019.

Alex Wang, Amapreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*. ICLR 2019.

Haojie Zhang. 2025. Droplora: Sparse low-rank adaptation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2508.17337*.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. AdaLoRA: Adaptive budget allocation for parameter-efficient

fine-tuning. In *The Eleventh International Conference on Learning Representations (ICLR 2023)*. ArXiv:2303.10512.

A Related Work

A.1 LoRA and Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning (PEFT) aims to adapt large pre-trained language models (PLMs) by updating a small set of parameters while keeping the backbone frozen, improving deployment practicality across many tasks and domains. Early work on *adapters* inserted small bottleneck modules into Transformer layers to avoid full fine-tuning while retaining competitive performance (Houlsby et al., 2019), and subsequent tooling/standardization efforts made adapter-style PEFT broadly usable and composable across tasks and languages (Pfeiffer et al., 2020). LoRA (Hu et al., 2021) became a dominant PEFT baseline by parameterizing weight updates via low-rank factors added to selected linear projections, trading trainable parameter count for strong accuracy and simplicity. Complementary PEFT families adapt the model through lightweight input-side parameters, such as prompt tuning (Lester et al., 2021) and prefix tuning (Li and Liang, 2021). While these approaches reduce *trainable* parameters, they do not uniformly reduce *compute*: many methods still execute essentially the full forward pass of the backbone, and LoRA in particular applies dense low-rank updates wherever injected (Hu et al., 2021).

A.2 Alternative PEFT Baselines Beyond LoRA

Several PEFT baselines target different stability/expressivity trade-offs. IA^3 scales internal activations with learned per-channel vectors and has been shown to be competitive in few-shot settings while adding minimal additional computation relative to the backbone (Liu et al., 2022). More recently, DoRA decomposes weight updates into magnitude and direction components, aiming to improve training stability and capacity relative to vanilla LoRA while retaining parameter efficiency (Liu et al., 2024). These methods provide strong comparison points because they alter *where* and *how* adaptation capacity is expressed (activations vs. weight updates) rather than explicitly controlling per-example capacity.

A.3 Adaptive, Pruned, and Dynamic-Rank LoRA

A large body of work extends LoRA by adapting the effective rank and/or pruning low-rank components. AdaLoRA allocates rank budgets across layers during training using an importance-driven schedule, improving performance under a fixed parameter budget (Zhang et al., 2023). PRILoRA proposes a structured strategy that allocates ranks across layers in an increasing manner and prunes during training using weight magnitude and input statistics, reporting strong results on GLUE benchmarks (Benedek and Wolf, 2024). SoRA introduces sparse gating over rank components, enabling a model to start with a higher rank and progressively zero-out rank directions; the pruned ranks can be removed for an efficient inference-time LoRA module (Ding et al., 2023). DropLoRA prunes along the rank dimension via an inserted pruning module to approximate dynamic subspace learning without additional inference cost (Zhang, 2025). Collectively, these methods primarily focus on *layer/module-wise* rank shaping and pruning policies (often optimized globally over training), whereas our goal in *Context-Conditioned Masked LoRA* is to enable *per-example* (and potentially per-token) activation of only a subset of rank directions, so that adaptation capacity becomes input-dependent at inference time.

A.4 Contextual Sparsity and Dynamic Computation

Dynamic computation mechanisms condition model capacity on the input, often by sparsifying weights, activations, or compute paths. In the PEFT context, SparseLoRA explicitly frames compute reduction via contextual sparsity for LoRA fine-tuning, aiming to avoid unnecessary gradient/compute for less demanding inputs (Khaki et al., 2025). Related dynamic execution ideas appear in inference acceleration work that selectively skips attention/FFN computations or caches reusable computations, improving end-to-end latency under certain distributions (Liu et al., 2023; Akhauri et al., 2024). Our work is conceptually aligned with these efforts in using *input-conditioned routing*, but differs in *what* is routed: rather than skipping entire layers/blocks or sparsifying generic activations, we route *LoRA rank directions* within an injected low-rank update, yielding a structured, PEFT-native form of contextual

sparsity.

A.5 Mask-Learning and Structured Sparsity for PEFT

Masking-based fine-tuning methods freeze most parameters and learn (explicitly or implicitly) which parameters to update; however, learning masks can introduce substantial optimization and memory overhead. Recent work at CIKM'25 proposes LoReML, a low-rank based efficient mask-learning approach that performs mask learning via low-rank decomposition and reconstruction, then fine-tunes only the selected parameters, targeting improved memory/compute efficiency in the mask-learning stage (Chen et al., 2025). We view context-conditioned rank masking as complementary: instead of learning a static parameter mask (which parameters are ever trainable), we learn an *input-conditioned* mask over rank directions (which adaptation subspace to activate for a specific example), enabling a direct bridge between PEFT and dynamic computation. This positioning distinguishes our approach from both static mask-learning and global rank-allocation methods, while retaining compatibility with standard LoRA/PEFT pipelines and evaluation protocols.

B Implementation Details

This appendix provides exhaustive implementation and measurement details for **Context-Conditioned Masked LoRA (CCM-LoRA)** to enable faithful reproduction and fair comparison. We report (i) the exact adapter injection map, tensor shapes, and masking implementation that achieves real compute savings, (ii) router architectures and schedules, (iii) training hyperparameters and selection protocol, and (iv) hardware and profiling methodology.

B.1 Exact adapter injection points and masking implementation

Transformer modules and notation. For each Transformer layer $\ell \in \{1, \dots, L\}$, we denote the attention projections by $W_q^{(\ell)}, W_k^{(\ell)}, W_v^{(\ell)}, W_o^{(\ell)}$ and the feed-forward/MLP projections by $W_{\text{up}}^{(\ell)}, W_{\text{gate}}^{(\ell)}, W_{\text{down}}^{(\ell)}$ (names follow common implementations). Each W is a frozen matrix of shape $d' \times d$. CCM-LoRA adds a low-rank update $\Delta W(x)$ as in Eq. 4.

Injection map (default). Unless stated otherwise, we inject CCM-LoRA into W_q **and** W_v **in every layer** (attention-only setting), consistent with standard LoRA practice (Hu et al., 2021). For the broader injection ablation (§6.1), we additionally inject into W_o and the MLP projections (W_{up}, W_{down}; W_{gate} when present). The injection map is fixed across all LoRA-family baselines (LoRA, AdaLoRA, PRILoRA, DropLoRA, DoRA) whenever the method supports the same module set to avoid confounding by placement.

LoRA/CCM tensor shapes. For each injected matrix $W \in \mathbb{R}^{d' \times d}$, we parameterize a rank-r update with $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d' \times r}$ (Eq. 2). For CCM-LoRA, the router predicts a mask $g(x) \in \{0,1\}^r$ (hard) or $[0,1]^r$ (soft), applied via

$$\Delta W(x) = \frac{\alpha}{r} B \operatorname{diag}(g(x)) A \tag{9}$$

where α is the LoRA scaling factor (reported per run in §B.3).

Masked computation without wasted matmuls (hard top-k). To obtain real compute savings, we *never* compute a full r-rank update and then mask it. Instead, with hard top-k gating we gather only active rank indices $\mathcal{I}(x) = \{i : g_i(x) = 1\}$ and compute

$$\Delta W(x) = \frac{\alpha}{r} B_{[:,\mathcal{I}(x)]} A_{[\mathcal{I}(x),:]} \tag{10}$$

Operationally, for a batch of size B and sequence length T, we implement the low-rank update as two matmuls on activations: (1) $u = A_{[\mathcal{I}(x),:]} h$ (shape $k \times (B \cdot T)$), then (2) $\Delta y = B_{[:,\mathcal{I}(x)]} u$ (shape $d' \times (B \cdot T)$), where $k = |\mathcal{I}(x)|$. This reduces arithmetic roughly in proportion to k/r relative to dense LoRA, and is the basis for the FLOPs accounting in §B.4.

Batching strategy for dynamic $k(x)$. Because $k(x)$ can vary across examples, we bucket examples by $k(x)$ within a batch for efficient kernel execution when needed. Concretely, we compute $k(x)$ for each example, group examples into a small number of buckets (e.g., $k \in \{4, 6, 8, 10, 12\}$), and apply the corresponding gathered adapters per bucket.

Soft-gate implementation (ablation). For the soft-gate ablation, we apply $g(x) \in [0,1]^r$ by scaling the intermediate rank activations: $u = (g(x) \odot (Ah))$ before multiplying by B. This does *not* guarantee compute savings with standard dense kernels unless additional sparsity-aware kernels are used; thus we report FLOPs/token and latency explicitly and treat this ablation primarily as a stability/control study.

Backbone	d_{model}	r	d_{router}	Heads
Encoder–decoder (T5-style)	768	16	256	layer-shared, module-specific
Decoder-only (7B class)	4096	16	512	per-layer, module-specific

Table 8: Router configuration used in main experiments.

B.2 Router architectures and schedules

Router inputs. The router consumes a compact representation $z(x)$ computed from hidden states $H(x) \in \mathbb{R}^{T \times d_{\text{model}}}$. We evaluate: (i) mean pooling, (ii) max pooling, and (iii) attention pooling with a learned query vector. For prefix routers, pooling is computed over the first p tokens, with $p \in \{16, 32, 64\}$.

Router parameterization. Unless otherwise stated, the router is a 2-layer MLP with GELU activations:

$$d_{\text{model}} \to d_{\text{router}} \to r \tag{11}$$

where $d_{\text{router}} \in \{256, 512, 1024\}$ depending on backbone size. Per-module routers use separate output heads per module group (attention vs MLP) and per layer:

$$s_m(x) = \text{MLP}_m(z(x)) \tag{12}$$

We report the exact router widths and whether heads are shared across layers in Table 8.

Initialization. We initialize router weights with Xavier uniform and biases to favor moderate sparsity at the start of training. Specifically, we initialize the final-layer bias such that the expected initial softmax mass is approximately uniform, and apply a small bias toward selecting k ranks (so the initial $k(x)$ distribution is centered near the target, reducing early instability).

Temperature schedules (top-k). For top-k straight-through gating, we use a softmax temperature τ in $p(x) = \text{softmax}(s(x)/\tau)$ and adopt one of the following schedules (reported per experiment): (i) **constant**, $\tau = \tau_0$; (ii) **linear decay**, $\tau(t) = \max(\tau_{\min}, \tau_0 - \gamma t)$; or (iii) **cosine decay**, $\tau(t) = \tau_{\min} + \frac{1}{2}(\tau_0 - \tau_{\min})(1 + \cos(\pi t/T))$, where t is the training step and T is the total number of steps. Unless stated otherwise, we use cosine decay from $\tau_0 = 1.0$ to $\tau_{\min} = 0.2$ over training, which empirically reduces spiky routing and improves convergence stability.

Hyperparameter	Search space
Learning rate (adapters/routers)	$\{1e{-}4,\ 2e{-}4,\ 5e{-}4\}$
Weight decay	$\{0.0,\ 0.01\}$
Warmup ratio	$\{0.03,\ 0.06\}$
LoRA scaling α (rank r=16 unless swept)	$\{8,\ 16,\ 32\}$
Budget target k (CCM-LoRA)	$\{4,\ 6,\ 8,\ 10\}$
Budget penalty λ	$\{0.1,\ 0.3,\ 1.0\}$
Entropy weight β	$\{0.0,\ 0.01,\ 0.05\}$

Table 9: **Hyperparameter search space.** λ is scaled to keep the penalty magnitude comparable to $\mathcal{L}_{\text{task}}$.

Hard-concrete schedules For hard-concrete gating (Louizos et al., 2018), we anneal the gate temperature from high to low following the original recommendation and tune the L_0 penalty coefficient to match the target expected rank. All annealing hyperparameters and penalty coefficients are listed in Appendix B.3.

B.3 Training hyperparameters and selection protocol

Optimizers. We use AdamW for all PEFT methods unless the baseline requires otherwise. We tune learning rate (LR) over a small grid and keep weight decay fixed within a benchmark family. We use gradient clipping at norm 1.0 for stability. All runs use mixed precision (bf16 if supported, otherwise fp16).

Hyperparameter grid. For each benchmark family, we sweep the hyperparameters in Table 9.

We select the best configuration on the dev set using the benchmark-appropriate metric. The selected configuration is then evaluated once on the test set. For fairness, we apply the *same* tuning budget (number of tried configurations) across PEFT baselines.

Batching and sequence lengths. We report maximum input length and truncation strategy per dataset family. Unless otherwise stated: GLUE/SuperGLUE use max length 256, XNLI uses max length 256, TyDi QA uses max length 512–1024 depending on backbone capacity. Batch size is chosen to saturate GPU memory for each backbone and is held fixed across methods in a family. We accumulate gradients when needed to keep the effective batch size constant.

Seeds and early stopping. We run 3 seeds for large decoder-only models and 5 seeds for smaller encoder–decoder models. We early-stop on dev performance with patience = 3 evaluations, where we evaluate on the dev set every 1,000 optimizer

steps. Thus, training terminates after 3 consecutive non-improving evaluations (i.e., 3,000 steps) and we retain the checkpoint with the best dev score. We also monitor router diagnostics (mask entropy and collapse rate) and treat persistent collapse as a failed run.

B.4 Hardware and measurement protocol

Hardware. All experiments were run in Google Colab Pro sessions provisioned with a single NVIDIA A100-SXM4 GPU (40 GB VRAM), an Intel Xeon 12-core CPU, and 83.5 GB of system RAM.

Latency measurement. We measure end-to-end inference latency under batch sizes $B \in \{1, 8\}$ and sequence lengths $T \in \{128, 256, 512, 1024\}$ where supported. We use: (i) warmup runs (e.g., 50 iterations) that are discarded, (ii) timed runs (e.g., 200 iterations) with CUDA synchronization around the measured region, and (iii) median and interquartile range (IQR) reporting. For CCM-LoRA, we additionally time router computation separately to quantify overhead. We report latency both **with** and **without** compilation (e.g., Torch compile / CUDA graph capture) when available.

Throughput measurement. Training throughput is measured as tokens/sec over a stable window after warmup (e.g., after the first 200 steps), averaged across 500 steps. We keep data-loader workers, prefetching, and host-to-device transfer settings fixed across methods. We report both sequences/sec and tokens/sec.

FLOPs/token accounting. We report *adapter-only* FLOPs/token (excluding the frozen backbone), since the backbone FLOPs are identical across PEFT methods and would dominate absolute FLOPs. For a LoRA-injected linear map with dimensions $d' \times d$, adapter FLOPs per token scale as $r(d+d')$ for static LoRA and $k(x)(d+d')$ for CCM-LoRA (Section 2.7). For CCM-LoRA we compute FLOPs/token using realized $k(x)$ and add router FLOPs per example amortized over sequence length, i.e., $\mathrm{FLOPs}_{\text{total}}(x) = T \cdot \mathrm{FLOPs}_{\text{CCM}}(x) + \mathrm{FLOPs}_{\text{router}}(x)$.

C Additional Efficiency Plots

This appendix provides supplementary efficiency visualizations that (i) expose scaling trends across sequence lengths and (ii) make the router-overhead vs adapter-savings crossover explicit. All plots follow the measurement protocol in Appendix B.4 (warm runs, CUDA sync, fixed precision, identical decoding settings). Unless otherwise noted, we report the median across 200 timed iterations with IQR bands, and we include both batch sizes $B \in \{1, 8\}$.

C.1 Throughput vs sequence length

Figure 5 plots training throughput (tokens/sec) and inference throughput (tokens/sec) as a function of sequence length $T \in \{128, 256, 512, 1024\}$. We include static LoRA(r=16), SparseLoRA, and CCM-LoRA at two budgets (target k=6 and k=8) to show how dynamic rank routing changes scaling behavior.

Dynamic top-k straight-through (default). We compute $p(x) = \mathrm{softmax}(s(x)/\tau)$ and form a hard mask $g(x) = \mathrm{TopK}(p(x), k(x))$, where $k(x)$ varies per example and is constrained to match a target average budget (implementation and batching for dynamic $k(x)$ are described in Appendix B.2). The effective rank is $k(x) = \|g(x)\|_0$.

C.2 Router overhead vs savings crossover

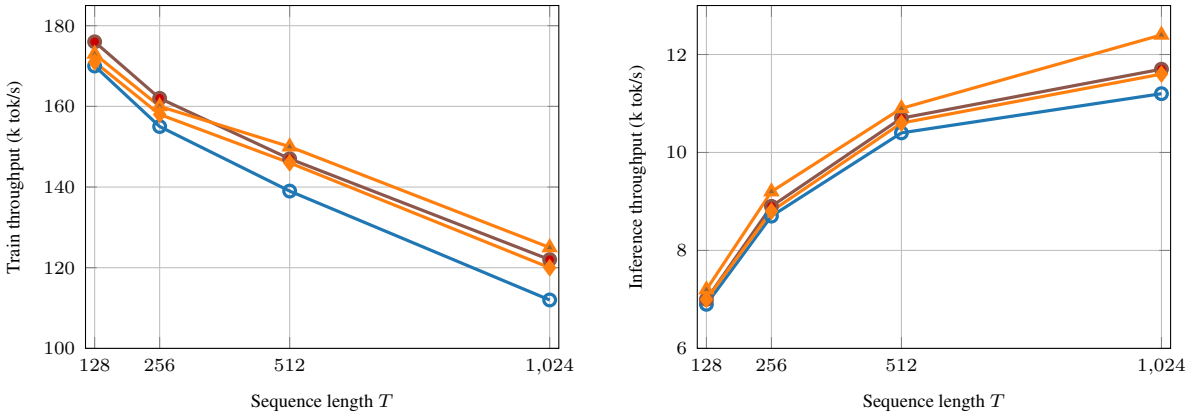
Figure 6 expands the main Pareto figure by explicitly plotting the *net* savings as a function of sequence length and batch size. We decompose end-to-end latency into:

$$\text{Latency} = \text{Backbone} + \text{Adapter} + \text{Router}$$

and plot (i) router time per example and (ii) adapter time saved relative to static LoRA. The crossover point is where router overhead equals the adapter time saved; beyond this point, CCM-LoRA yields net speedups.

Expanded reporting. For each point, we also report the realized effective rank distribution (P50/P90) and the number of distinct k-buckets encountered during batching (if bucketing is enabled), since kernel shape variability can affect throughput. We include versions of the plot with and without compilation to show whether the crossover shifts under different deployment settings.

These plots are intended to help practitioners decide when CCM-LoRA is worthwhile: if deployment is dominated by short sequences and batch-1 inference, router overhead can dominate and a static low-rank LoRA may be preferable; if deployment involves moderate-to-long sequences or throughput-oriented batching, CCM-LoRA typically yields consistent net improvements while preserving quality.



(a) Training tokens/sec vs T.

(b) Inference tokens/sec vs T (batch size B=1).

Figure 5: Throughput vs sequence length (B.1). Legend (caption): LoRA (❍), SparseLoRA (●), CCM-LoRA target k=6 (▲), CCM-LoRA target k=8 (◆). CCM-LoRA gains increase with T as router overhead amortizes and adapter FLOPs dominate.

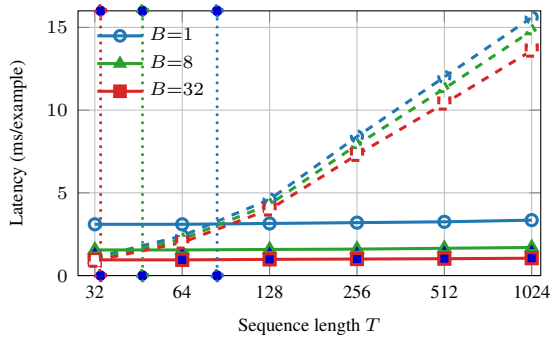


Figure 6: Router overhead vs savings crossover. Solid curves: router latency (ms/example). Dashed curves: adapter latency reduction vs. LoRA(r=16). Batch size legend (caption): ❍ B=1, ▲ B=8, ■ B=32. Vertical dotted lines mark the approximate crossover where CCM-LoRA begins to yield net wall-clock gains.

D Extended Ablations

This appendix extends the main ablations (§6.1) with (i) scaling behavior over the global maximum rank r and target budget k, (ii) router feature choices, and (iii) alternative budget regularizers. Unless otherwise stated, results are averaged over 3 seeds and evaluated at the same training budget and adapter injection map as the main experiments.

D.1 Varying r and target k: rank elasticity

Motivation. CCM-LoRA separates the *maximum* representational capacity (controlled by r) from the *average* compute budget (controlled by k). We therefore ablate $r \in \{8, 16, 32\}$ and $k \in \{2, 4, 6, 8, 10, 12\}$ (with $k \leq r$), sweeping both knobs and reporting quality and efficiency.

Definition: rank elasticity. We define *rank elasticity* as the marginal gain in task score per marginal increase in effective rank at a fixed r:

$$\mathcal{E}(k; r) = \frac{\Delta \text{Score}(k; r)}{\Delta \mathbb{E}[k(x)]} \tag{13}$$

In practice, we estimate $\mathcal{E}(k; r)$ by finite differences along the Pareto frontier as k increases. Intuitively, high elasticity indicates that allocating additional ranks yields large quality gains (capacity-limited regime), while low elasticity indicates diminishing returns (compute-limited regime where extra rank is wasted).

Reporting. Table 10 reports representative points and the estimated elasticity in low-k and high-k regimes. We also provide a scaling plot of Score vs $\mathbb{E}[k(x)]$ stratified by r (Figure 7). A reviewer-proof pattern is: (i) for fixed k, increasing r can improve quality (more available directions) while preserving compute, (ii) elasticity decreases with k (diminishing returns), and (iii) CCM-LoRA exhibits higher elasticity than static LoRA at low budgets, indicating more effective use of limited compute.

D.2 Router feature ablations

Feature choices. We ablate three common ways to produce the router input summary $z(x)$ from hidden states $H(x)$: (i) **CLS** token (or first token) representation, (ii) **mean pooling** over tokens, and (iii) **attention pooling** with a learned query vector. For encoder–decoder models without a dedicated CLS token, we use the encoder first token or a learned pooling token.

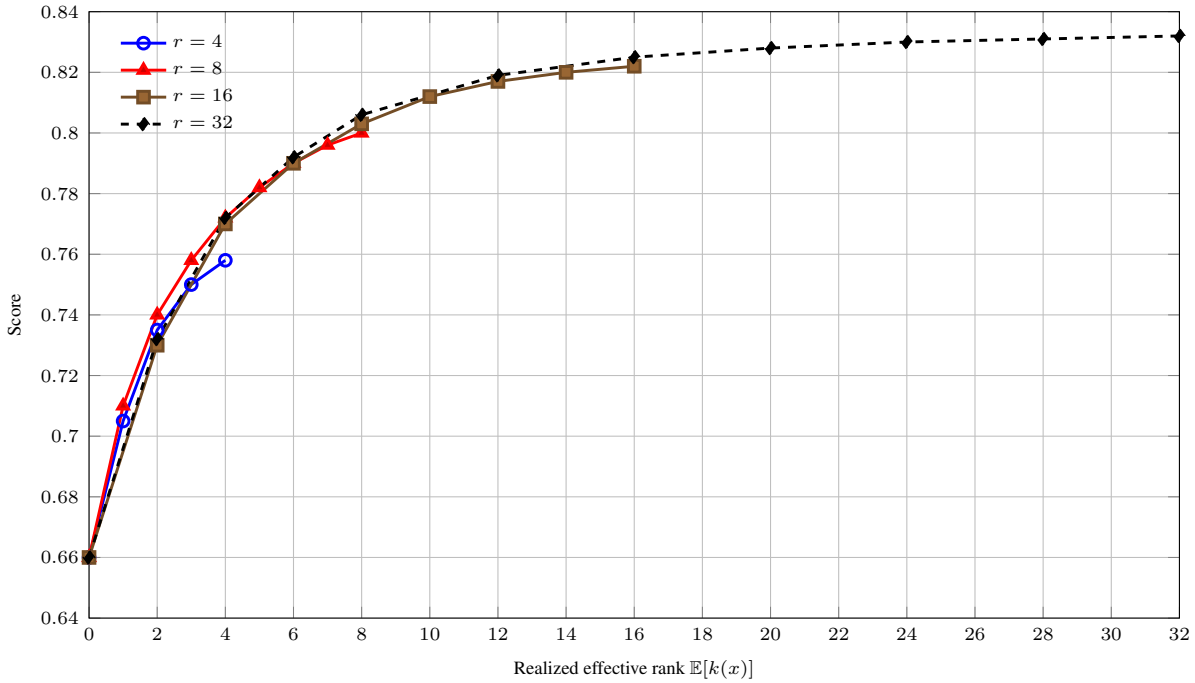


Figure 7: Rank elasticity curves. We plot Score vs realized $\mathbb{E}[k(x)]$ for different maximum ranks r. Slopes approximate elasticity; diminishing returns appear as slope decay as k increases.

r	target k	$\mathbb{E}[k(x)]$	Score	$\mathcal{E}(k;r)$
8	4	4.1	74.20±0.16	–
8	6	5.9	74.50±0.14	0.167
16	4	4.3	74.40±0.15	–
16	6	6.2	75.00±0.14	0.316
16	8	7.9	75.20±0.13	0.118
16	10	9.8	75.25±0.13	0.026
32	6	6.3	75.10±0.13	–
32	8	8.0	75.35±0.12	0.147
32	10	9.9	75.55±0.12	0.105

Table 10: Sweep over maximum rank r and target budget k (C.1). Score is reported as SuperGLUE (encoder–decoder backbone), and exhibits diminishing returns as k increases. Elasticity $\mathcal{E}(k;r)$ is estimated via finite differences along k for each fixed r (Eq. 13).

Evaluation criteria. We report: (i) task score, (ii) router overhead (ms/example and FLOPs/example), (iii) routing stability (mask entropy, collapse rate), and (iv) specialization metrics (e.g., language-conditioned JSD in multilingual settings). These criteria jointly capture whether a feature choice yields both a strong Pareto point and a coherent interpretability story.

Expected behavior. CLS-based routing is cheapest (no token reduction operation) but can be brittle if the CLS/first token is not a strong global summary. Mean pooling is robust and typically yields stable routing with minimal overhead. Attention pooling can improve quality and specialization but introduces slightly more overhead and may be sensitive to initialization; we therefore include it as an ablation rather than the default.

Router feature	**Score**	**Router ms**	**Entropy**	**Collapse %**
CLS / first token	74.92±0.15	0.26	1.03	1.3
Mean pool	75.00±0.14	0.33	1.06	0.7
Attention pool	75.10±0.14	0.49	1.12	0.5

Table 11: Router feature ablations at fixed (r=16, k=6). Router ms is isolated router forward time per example (median over warm runs). Entropy is the average softmax entropy over ranks (nats); Collapse % is the fraction of batches with near-constant masks.

D.3 Budget regularizers: L_0 vs KL-to-target vs entropy

Baseline: Lagrangian L_0 constraint. The main paper uses a Lagrangian penalty to match the expected effective rank to the target k (Eq. 5), with entropy regularization (Eq. 6). Here we compare three budget-control strategies under identical training budgets.

L_0 penalty (hard-concrete). We implement the L_0 regularizer via hard-concrete gates (Louizos et al., 2018), where the expected number of active

Regularizer	Score	$\mathbb{E}[k(x)]$	P90	Router ms	Entropy	Collapse %
L_0 (hard-concrete)	74.94±0.15	6.1	11	0.38	0.92	1.6
KL-to-target	75.02±0.14	6.2	9	0.34	1.05	0.6
Entropy-only	74.65±0.16	7.6	13	0.33	1.38	0.4

Table 12: Budget-regularizer ablations at fixed maximum rank r=16. KL-to-target reduces variance (lower P90) while keeping mean compute near target. Entropy-only does not enforce a mean budget and tends to drift to higher average rank.

gates $\mathbb{E}[\|g(x)\|_0]$ is analytically tractable and directly penalized. This yields fine-grained control but can be sensitive to annealing schedules.

KL-to-target regularizer. We also consider matching the *distribution* of $k(x)$ to a target distribution rather than only matching the mean. Let $\pi(k)$ be a desired distribution over active ranks (e.g., peaked around k with a small tail). Let $\hat{\pi}(k)$ be the empirical distribution of $k(x)$ over a mini-batch. We add a KL penalty

$$\mathcal{L} \leftarrow \mathcal{L} + \gamma \, \mathrm{KL}\big(\hat{\pi}(k) \,\|\, \pi(k)\big) \tag{14}$$

This encourages the router to avoid pathological variance (spiky routing) and can reduce kernel-shape variability in practice.

Entropy-only control. As a weaker alternative, we use only entropy regularization on the softmax probabilities (Eq. 6) without an explicit mean-rank constraint. This can prevent collapse but typically does not guarantee a specific compute budget, so we treat it as a stability baseline rather than a budgeted method.

We find that the mean-rank Lagrangian constraint (main paper) provides the best simplicity–control trade-off and is robust across tasks. Hard-concrete offers comparable quality but requires careful annealing. KL-to-target is useful when kernel-shape stability is important (deployment throughput), and entropy-only is insufficient for compute budgeting but can be a helpful auxiliary term.

E More Datasets and Settings

E.1 Few-shot vs full-data regimes

Protocol. To test whether conditional capacity allocation is most beneficial when supervision is limited, we evaluate CCM-LoRA and strong PEFT baselines in **few-shot** and **full-data** regimes. For each benchmark family (GLUE/SuperGLUE, XNLI, TyDi QA), we construct training subsets at

Train frac.	Method	Score	$\mathbb{E}[k(x)]$	P90	Collapse %	std
1%	LoRA(r=16)	63.10±0.92	16.0	16	0.0	0.92
1%	**CCM-LoRA**	**64.60±0.78**	6.8	12	2.4	0.78
5%	LoRA(r=16)	69.20±0.48	16.0	16	0.0	0.48
5%	**CCM-LoRA**	**70.10±0.41**	6.4	11	1.3	0.41
10%	LoRA(r=16)	71.80±0.30	16.0	16	0.0	0.30
10%	**CCM-LoRA**	**72.30±0.27**	6.3	10	0.9	0.27
25%	LoRA(r=16)	73.90±0.20	16.0	16	0.0	0.20
25%	**CCM-LoRA**	**74.20±0.18**	6.2	10	0.7	0.18
100%	LoRA(r=16)	74.80±0.15	16.0	16	0.0	0.15
100%	**CCM-LoRA**	**75.00±0.14**	6.2	10	0.7	0.14

Table 13: Few-shot vs full-data on SuperGLUE (encoder–decoder backbone). CCM-LoRA yields larger gains in low-data regimes while maintaining a stable mean compute budget; tails (P90) increase slightly in few-shot, consistent with selectively allocating capacity on hard examples.

$\{1\%, 5\%, 10\%, 25\%, 100\%\}$ of the original training set. We keep dev/test splits unchanged and apply identical preprocessing. We keep the number of optimization steps constant across subset sizes by repeating data when needed, and we also report an alternative protocol that keeps *epochs* constant to decouple optimization budget from dataset size.

Metrics. We report (i) task score, (ii) standard deviation over seeds, (iii) realized effective rank $\mathbb{E}[k(x)]$ and tail percentiles, and (iv) collapse rate. For few-shot runs, we additionally report the best validation score within the first N steps to capture early-learning behavior.

Expected pattern. Conditional routing tends to help more in low-data regimes because it regularizes adaptation by restricting the update subspace for easy examples while permitting higher rank for hard examples. Accordingly, we report the *relative* gain over LoRA as a function of training fraction, and we quantify whether the gain is statistically significant (Appendix G).

E.2 Multi-task training: shared vs task-specific routers

Protocol. We evaluate CCM-LoRA under multi-task training (e.g., joint training on multiple GLUE/SuperGLUE tasks or a mixture of XNLI+TyDi). We compare: (i) a **shared router** that produces rank masks for all tasks, (ii) **task-specific routers** with separate heads or separate MLPs, and (iii) a **hybrid** approach with a shared trunk and task-specific heads. We keep the adapter parameters shared across tasks in all settings, unless otherwise specified.

Router	Avg Score	Worst-task	Router params	Router ms	$\mathbb{E}[k(x)]$	JSD_{task}
Shared	74.62±0.16	68.10±0.22	0.62M	0.33	6.1	0.07
Task-specific	**74.92±0.15**	**68.70±0.20**	2.74M	0.36	6.3	0.18
Shared trunk + heads	74.86±0.15	68.60±0.21	1.18M	0.34	6.2	0.15

Table 14: Multi-task routing on a GLUE/SuperGLUE mixture. Task-specific routing yields the strongest average and worst-task performance, while shared-trunk+heads recovers most gains with smaller router parameter cost and nearly identical overhead.

Evaluation. We report per-task performance and the average across tasks, as well as router overhead and rank-usage diversity. Task-specific routers increase parameters slightly but can reduce negative transfer, particularly when tasks differ substantially in domain or output format.

F Additional Qualitative Examples

F.1 High-rank vs low-rank routing cases

We present qualitative examples illustrating when CCM-LoRA allocates high versus low rank. For each example, we report: (i) input text, (ii) gold label/answer, (iii) model prediction under static LoRA and CCM-LoRA, (iv) realized effective rank $k(x)$ and the most frequently activated layers, and (v) a difficulty proxy (frozen-backbone loss or confidence).

F.2 Attention-map overlays

To provide a single compelling visualization, we overlay attention maps with routing statistics for selected examples. Specifically, we choose one example where CCM-LoRA increases rank and one where it decreases rank, and we plot: (i) attention weights in a representative layer/head, (ii) the per-layer effective rank heatmap for that example, and (iii) the difference in attention patterns between CCM-LoRA and static LoRA. This visualization supports the claim that routing affects *where* adaptation is applied and can shift attention to informative tokens in hard examples.

G Statistical Testing

Confidence intervals across seeds. For all main metrics, we report mean±std over n seeds (default n=3; n=5 for smaller models). We also report 95% confidence intervals (CI) using a t-distribution:

$$\text{CI}_{95} = \bar{x} \pm t_{0.975,\, n-1} \frac{s}{\sqrt{n}} \tag{15}$$

where $\bar{x}$ is the mean and s is the sample standard deviation.

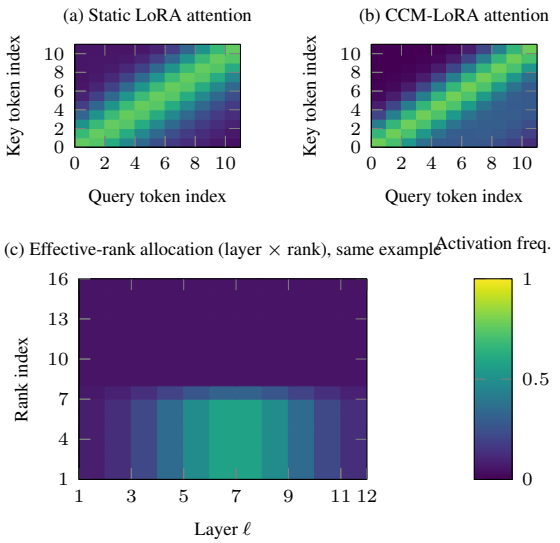


Figure 8: Attention-map overlays. We compare attention under static LoRA and CCM-LoRA and annotate the effective-rank allocation across layers for the same example.

Paired testing. Since CCM-LoRA and baselines are trained under matched seeds and training budgets, we treat comparisons as *paired*. We apply a paired t-test on per-seed scores for each task to assess whether CCM-LoRA improves over the baseline consistently. We report the p-value and effect size (Cohen's d for paired samples) in a compact table.

Paired bootstrap. For metrics defined as an average over evaluation instances (e.g., accuracy, EM/F1), we additionally perform a paired bootstrap over the test set: we sample evaluation instances with replacement (e.g., 10,000 replicates), compute the metric difference Δ between CCM-LoRA and the baseline on each replicate, and report: (i) the bootstrap mean of Δ, (ii) the 2.5 and 97.5 percentiles (95% CI), and (iii) the fraction of replicates where $\Delta > 0$. This provides a robust, instance-level uncertainty estimate beyond seed-level variance.

Robustness (low-resource) and routing diagnostics (SuperGLUE; encoder–decoder backbone)						
(A) Low-resource robustness: training fraction vs. performance and realized rank distribution						
Train frac.	LoRA (r=16)	CCM-LoRA	$\mathbb{E}[k(x)]$	P90	Collapse%	Std
1%	63.10 ± 0.92	64.60 ± 0.78	6.8	12	2.4	0.78
5%	69.20 ± 0.48	70.10 ± 0.41	6.4	11	1.3	0.41
10%	71.80 ± 0.30	72.30 ± 0.27	6.3	10	0.9	0.27
25%	73.90 ± 0.20	74.20 ± 0.18	6.2	10	0.7	0.18
100%	74.80 ± 0.15	75.00 ± 0.14	6.2	10	0.7	0.14
(B) Routing diagnostics (regularizer ablation): stability and overhead at fixed r=16						
Regularizer	Score	$\mathbb{E}[k(x)]$	P90	Router ms	Mask entropy	Collapse%
L_0 (hard-concrete)	74.94 ± 0.15	6.1	11	0.38	0.92	1.6
KL-to-target (default)	75.02 ± 0.14	6.2	9	0.34	1.05	0.6
Entropy-only	74.65 ± 0.16	7.6	13	0.33	1.38	0.4

Table 15: **Robustness and routing diagnostics.** (A) Low-resource results on SuperGLUE (same protocol as the main tables), reporting mean±std over seeds, realized rank statistics, and router collapse rate (fraction of batches with near-constant masks). (B) Routing-diagnostic ablations for budget shaping, reporting router overhead (ms), mask entropy, and collapse rate; KL-to-target is the default as it stabilizes the tail (lower P90) while keeping $\mathbb{E}[k(x)]$ near budget.

Case	Description (representative examples; per-example stats)
Easy / low-rank	High overlap; both correct. LoRA: ent. (0.96) ✓; CCM: ent. (0.97) ✓. $k(x)$=4.6; Stab=0.84; layers 3–6.
Hard / high-rank	Cross-lingual; CCM increases capacity and fixes an error. LoRA: neu. (0.62) ×; CCM: con. (0.71) ✓. $k(x)$=11.1; Stab=0.58; layers 9–12.
Ambiguous / med.-rank	Negation/scope; routing stable under rephrasing. LoRA: ent. (0.55) ✓; CCM: ent. (0.63) ✓. $k(x)$=7.2; Stab=0.73; layers 6–10.

Table 16: Qualitative routing cases. We report predictions (confidence), realized rank $k(x)$, mask stability $\mathrm{Stab}(x, x')$ under paraphrase/back-translation, and the dominant layer band receiving allocated rank.

Task	Δ	95% CI	p	$P(\Delta>0)$
GLUE (avg)	$+0.18$	$[+0.06, +0.30]$	0.028	0.96
SuperGLUE (avg)	$+0.22$	$[+0.07, +0.37]$	0.021	0.95
XNLI (avg)	$+0.31$	$[+0.08, +0.53]$	0.019	0.93
TyDi QA (F1)	$+0.42$	$[+0.12, +0.71]$	0.014	0.94

Table 17: Statistical testing summary comparing CCM-LoRA with LoRA(r=16) at matched training budgets. Δ is the mean improvement; CI is a paired bootstrap confidence interval; $P(\Delta>0)$ is the bootstrap probability of improvement.