G-TUNE: MEMORY-EFFICIENT FINE-TUNING VIA COMPONENT-LEVEL GRADIENT DITCHING

Anonymous authors
Paper under double-blind review

ABSTRACT

Fine-tuning large language models is essential for adapting them to downstream tasks and user preferences, but it is prohibitively memory-hungry. Intermediate activations, account for the dominant share of this memory, making memory-efficient fine-tuning critical. We perform memory-efficient fine-tuning at the granularity of individual components, selecting for each input only the attention heads and MLP blocks that matter, rather than treating entire tokens uniformly. This requires overcoming two challenges: (1) scoring and selecting the right components for each input, for which we introduce a gradient-guided importance score computable in a single forward-backward pass, and (2) keeping training stable despite per-instance selection, which we address via a balanced mask assignment procedure that casts sparsity-level assignment as a multi-objective search using NSGA-II. G-TUNE, a memory-efficient fine-tuning method via component-level gradient ditching. Across three models, Llama3.2-3B, Llama2-7B, and Qwen2.5-7B, G-TUNE surpasses the state of the art in both accuracy and memory efficiency, and remains effective when combined with LoRA and QLoRA.

1 INTRODUCTION

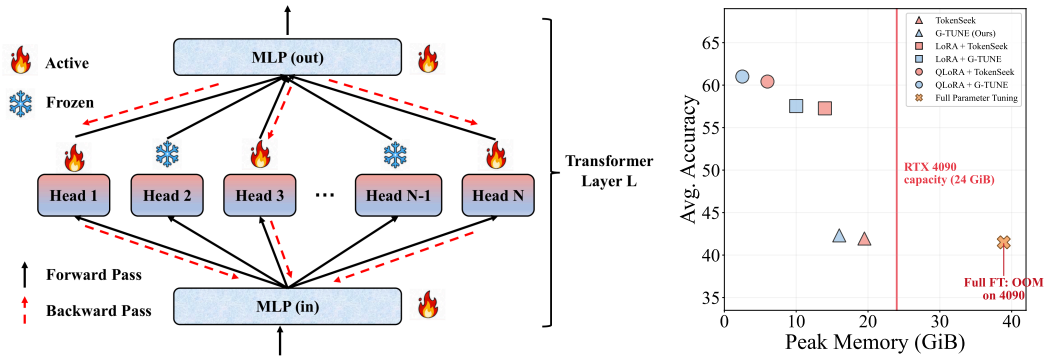

Figure 1: **Overview of G-TUNE.** (a) Within each Transformer layer, only a sparse, input-specific subset of attention heads and MLP blocks (Active) participates in backpropagation, while the remaining components (Frozen) are skipped and their activations are never cached. (b) At a 10% component selection ratio, full parameter tuning on Llama3.2 (3B) requires memory far beyond the capacity of a single RTX 4090 (24 GiB) and cannot be run, whereas combining LoRA or QLoRA with G-TUNE keeps peak memory well within budget while matching or exceeding the accuracy of TokenSeek at comparable memory cost.

Fine-tuning pre-trained large language models (LLMs) on curated instruction data has become the dominant paradigm for adapting general-purpose models to downstream tasks and user preferences Taori et al. (2023a); Wang et al. (2023), but this adaptation is prohibitively memory-hungry: fine-tuning Llama3.2-3B with full parameters already requires 38.9 GiB of peak GPU memory (Fig. 1), exceeding the 24 GiB capacity of a single consumer-grade RTX 4090 and making full

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.

fine-tuning infeasible without multi-GPU sharding or offloading, even at this comparatively modest model scale. The root cause is not the model parameters themselves but the intermediate activations retained for backpropagation, an imbalance that worsens with batch size: prior measurements on Llama3-8B show activations rising from 50% to 87% of training memory as batch size grows from 1 to 8, while parameters shrink from 12% to just 2% Zeng et al. (2026). **Reducing activation memory, not trainable parameters, is therefore the key lever for memory-efficient fine-tuning.**

Recent work has begun to exploit this by making activation retention input-aware: TokenSeek Zeng et al. (2026) and TokenTune Simoulin et al. (2024) selectively ditch a subset of input tokens from gradient computation for each instance. However, their granularity remains coarse: every attention head and MLP block within a kept token is treated identically, regardless of which components actually carry the information relevant to that input – **token-level selection never engages with the model's internal structure.** Motivated by findings in mechanistic interpretability showing that predictions can be attributed to sparse computational circuits Ou et al. (2025); Yao et al. (2024), we instead pursue memory-efficient fine-tuning that is driven by the relationship between each input and the model's fine-grained internal structure (attention heads and MLP blocks) rather than its input tokens alone.

Turning this idea into a practical fine-tuning method, however, requires overcoming two concrete challenges: **(1) how to score and select the right components for each input**, since components lack any cheap, obvious notion of importance; and **(2) how to keep training stable**, since components rarely selected across training receive too few gradient updates to adapt – a failure mode we term component starvation. We address first challenge with a gradient-guided importance score combining each component's forward activation with the backward gradient of its downstream components, computable in a single forward-backward pass, and second challenge by casting per-instance sparsity-level assignment as a design-space exploration problem, using NSGA-II to search for a Pareto-optimal trade-off between update balance and average memory cost. Together, these form **G-TUNE**, a memory-efficient fine-tuning method via component-level gradient ditching.

We evaluate G-TUNE on three backbones spanning different scales – Llama3.2-3B, Llama2-7B, and Qwen2.5-7B – and show that it is complementary to existing parameter-efficient fine-tuning techniques, remaining effective when combined with LoRA Hu et al. (2021) and QLoRA Dettmers et al. (2023) in addition to full parameter tuning. Our contributions are:

- We are the **first** to perform memory-efficient fine-tuning at component-level granularity, selecting input-specific attention heads and MLP blocks to directly reduce activation memory, at a finer granularity than existing token-level approaches.
- We propose a gradient-guided component importance score, computable with a single forward-backward pass, that identifies the Top-K components to retain for backpropagation on a per-input basis.
- We identify component starvation, an optimization instability caused by naive per-instance Top-K selection, and resolve it via a balanced mask assignment procedure based on multi-objective search.
- We show that G-TUNE surpasses the state of the art in both accuracy and memory efficiency across three backbones and multiple scales, and remains effective when combined with LoRA and QLoRA.

2 PRELIMINARIES

2.1 EMPIRICAL OBSERVATIONS

2.1.1 OBSERVATION 1: INPUT-SPECIFIC SPARSE COMPONENTS IN LLMS

Large language models achieve remarkable performance across diverse tasks through the collaborative computation of attention heads and MLP components. Despite their dense architectures, recent studies have shown that model predictions can often be represented by only sparse subsets of model components Ou et al. (2025); Yao et al. (2024). Inspired by this finding, we investigate whether different inputs activate the same sparse components or require distinct component subsets.

To investigate input-specific sparse components, we independently identify the sparse component set associated with each input using Edge Attribution Patching with Integrated Gradients (EAP-IG) Ou et al. (2025), which assigns an importance score to every connection between model components according to its contribution to the target prediction. We retain the highest-scoring connections after connectivity filtering and collect their associated attention heads and MLP blocks as the sparse component set of the current input. (The detailed formulation of EAP-IG is provided in **Appendix D**).

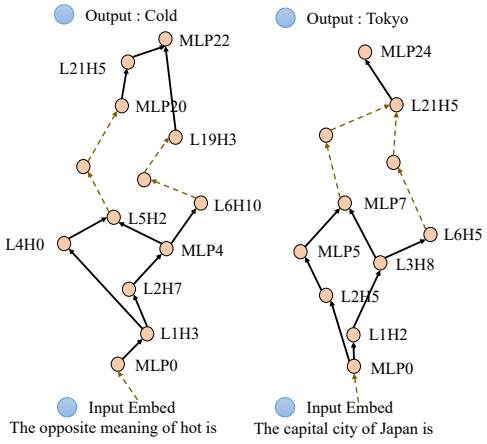

Figure 2: Representative sparse component sets identified for two different inputs. LxHy denotes the y-th attention head in the x-th Transformer layer, and MLPx denotes the MLP block in the x-th layer. Dashed arrows indicate additional selected components omitted for clarity.

As illustrated in Fig. 2, different inputs exhibit substantially different sparse component sets in terms of component compositions, layer distributions. To further quantify this diversity, we compute the pairwise Jaccard similarity of $\mathcal{S}(x)$ across 50 pairs of inputs, obtaining an average of 0.12 (The detailed information is provided in **Appendix E**). This observation suggests that **sparse components in large language models are inherently input-specific**. Different inputs require distinct sparse component sets rather than a universally shared component subset, motivating instance-aware component selection during sparse optimization.

2.1.2 Observation 2: Sparse Components Preserve Predictive Capability

We further investigate whether sparse component sets are sufficient to preserve the predictive capability. For each test input, we retain only its corresponding sparse component set and perform inference without modifying the model parameters, where non-selected components are zero-ablated, and random component sets are constructed with the same size while preserving graph connectivity. We compare its predictive capability with that of the full model and randomly selected components under the same component budget.

Type	Full	Random	Selected
Linguistic	0.68	0.09	0.57
Commonsense	0.61	0.07	0.51
Factual	0.72	0.11	0.60
Bias	0.56	0.08	0.47
Average	**0.64**	**0.09**	**0.54**

Table 1: Hit@10 of the full model, input-specific sparse component sets, and randomly selected component sets across different knowledge types on GPT2-Medium.

We adopt Hit@10 as the evaluation metric, which measures whether the target token appears among the model's top-10 predictions (Detailed definitions are provided in **Appendix F**). FollowingYao et al. (2024), we adopt the dataset from Hernandez et al. (2024), covering four knowledge types—Linguistic, Commonsense, Factual, and Bias—with validation and test sets split in a 1:1 ratio. As shown in Table 1, input-specific sparse component sets preserve most of the predictive capability of the original model across different knowledge types while substantially outperforming randomly selected component sets under the same component budget. This observation suggests that **sparse components preserve the core predictive information required for each input**. The substantial performance gap between selected and random further demonstrates that input-specific sparse components capture the most critical computation patterns for accurate prediction, highlighting the feasibility of sparse optimization.

2.2 Activation Memory Bottleneck in Fine-tuning

During fine-tuning, intermediate activations dominate GPU memory consumption because they must be retained after the forward pass until the corresponding backward propagation for gradient com-

putation. Consider the output activation of a Transformer component c in layer l:

$$\mathbf{a}_c^l = f_c^l\big(\mathbf{h}^{l-1}; \theta_c^l\big)\,, \tag{1}$$

where $\mathbf{h}^{l-1}$ denotes the input hidden representation, θ_c^l denotes the trainable parameters of component c, and $\mathbf{a}_c^l$ denotes its output activation. During backpropagation, the corresponding parameter gradient is computed according to the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_c^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_c^l} \cdot \frac{\partial \mathbf{a}_c^l}{\partial \theta_c^l}, \tag{2}$$

where $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_c^l}$ denotes the upstream gradient propagated from subsequent layers. Therefore, computing parameter gradients requires access to the corresponding forward activation $\mathbf{a}_c^l$. Each optimized component must therefore retain its forward activation until the corresponding backward computation is completed. Consequently, the total activation memory can be approximated as:

$$M_{\text{act}} \propto \sum_{l=1}^{L} \sum_{c=1}^{N_l} \text{size}\big(\mathbf{a}_c^l\big)\,, \tag{3}$$

where N_l denotes the number of computational components in the l-th Transformer layer. Therefore, activation memory grows approximately linearly with the number of optimized components, making activation storage the dominant memory bottleneck during fine-tuning. Section 2.1 reveals that model predictions are primarily governed by only a small subset of input-aware sparse components, while the remaining components contribute only marginally to the final prediction. This observation motivates us to incorporate input-aware sparse components into the fine-tuning process by selectively restricting backward propagation to important components: during the forward pass, the activations of non-selected components are not retained, and their gradient computation is skipped entirely during backpropagation, thereby eliminating their contribution to activation memory. **However, achieving components-aware fine-tuning requires addressing two key challenges: (1) accurately identifying the important components for each input, and (2) maintaining stable optimization under sparse component updates.**

3 METHODOLOGY

3.1 INPUT-AWARE COMPONENT SELECTION

Problem Definition. Given a Transformer consisting of N computational components, we denote the complete component set as $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$, where each component corresponds to an attention head or an MLP block. For each input sample x, our goal is to identify an input-specific sparse component subset $\mathcal{S}(x) \subseteq \mathcal{C}$ satisfying $|\mathcal{S}(x)| = K$. During fine-tuning, only the selected components participate in backward optimization.

Gradient-guided Component Importance Estimation. We estimate component importance from the information flow over the computational graph. A standard forward pass first records the output representation $\mathbf{z}_c(x)$ of every component. A subsequent backward pass computes the gradient of the training loss with respect to the input residual representation of every downstream component. Combining the forward activation and backward gradient, the importance of component c is computed as:

$$s(c; x) = \frac{1}{|\mathcal{D}(c)|} \sum_{j \in \mathcal{D}(c)} \left| \left\langle \mathbf{z}_c(x), \frac{\partial \mathcal{L}(x)}{\partial \mathbf{r}_j(x)} \right\rangle \right|, \tag{4}$$

where $\mathcal{D}(c)$ denotes the downstream components receiving the output of component c, and $\mathbf{r}_j(x)$ denotes the input residual representation of component j. The forward activation characterizes the information propagated along each computational edge, whereas the backward gradient measures the sensitivity of the loss at the edge destination. Their inner product therefore provides a first-order estimate of edge contribution. Absolute values prevent cancellation across different computation paths, while averaging avoids favoring components with more outgoing edges. This formulation can be viewed as a first-order approximation of edge ablation while requiring only a single forward pass and a single backward pass.

Top-K Component Selection. After computing the importance score of every component, we rank all components and retain the Top-K ones:

$$\mathcal{S}(x) = \mathrm{TopK}_{c \in \mathcal{C}}\big(s(c; x), K\big), \tag{5}$$

The resulting subset $\mathcal{S}(x)$ serves as the input-specific sparse component mask. During fine-tuning, gradients are propagated only through the selected components, allowing different inputs to optimize different sparse computational paths under the same sparsity budget.

3.2 Stable Sparse Optimization

Selective Activation Caching. Let $m_c(x) \in \{0, 1\}$ be a mask, where $m_c(x) = 1$ if component c is selected ($c \in \mathcal{S}(x)$) and $m_c(x) = 0$ otherwise. During the forward pass, every component still computes its output representation $\mathbf{z}_c(x)$ as usual, but only the masked representation is cached for backpropagation:

$$\tilde{\mathbf{z}}_c(x) = m_c(x) \cdot \mathbf{z}_c(x), \tag{6}$$

When $m_c(x) = 0$, $\tilde{\mathbf{z}}_c(x)$ is not stored, so only K activations are retained instead of the full N components. In practice, this masking is realized via a stop-gradient operation rather than explicit multiplication, so that non-selected activations are never allocated in the computation graph. The residual stream still carries the forward contribution of every component, so gradients still reach earlier layers regardless of masking.

Component Starvation Undermines Optimization Stability. We observe a phenomenon illustrated in the left of Fig. 3: since $\mathcal{S}(x)$ only selects the Top-K components for each input, a component that repeatedly falls outside this range receives very few updates throughout training. (More examples are provided in **Appendix G**).

Effective fine-tuning requires every component to receive sufficient gradient updates in order to adapt to the new task; a component that is updated too rarely remains close to its pre-trained state and fails to properly adapt. Since model predictions depend on the joint computation of many components (Section 2.1), the presence of such persistently under-updated components leaves part of the model insufficiently trained, forcing the remaining, well-adapted components to compensate for their outdated behavior. This imbalance in optimization progress across components is a source of instability during fine-tuning. We refer to this phenomenon as **component starvation**.

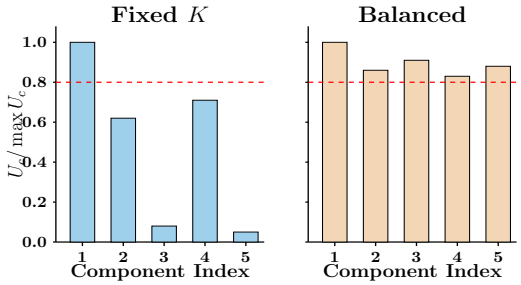

Figure 3: Normalized update count $U_c / \max_c U_c$ across 5 representative components. Left (Fixed K): coverage is highly uneven. Right (Balanced): our method narrows this gap.

Balanced Mask Assignment via Multi-Objective Search. To address component starvation, we assign each input a mask without breaking its input-specific relevance. For each input x_i, the profiling ranking (Eq. 5) gives P candidate masks $\{\mathcal{S}_i^{(1)}, \ldots, \mathcal{S}_i^{(P)}\}$, each retaining a different percentage of components, with the same P percentage choices shared across all inputs and fluctuating around the target budget K. An assignment $\pi : i \mapsto p_i$ picks one candidate mask per input. We optimize two objectives:

$$\begin{aligned} f_1(\pi) &= \mathrm{Var}_c\big(U_c(\pi)\big), \\ f_2(\pi) &= \frac{1}{M} \sum_{i=1}^{M} \big|\mathcal{S}_i^{(p_i)}\big|, \end{aligned} \tag{7}$$

where $U_c(\pi)$ is the number of times component c is selected across all M inputs under π, f_1 is the variance of $\{U_c(\pi)\}_{c=1}^{N}$ across all N components, and f_2 is the average mask size across the training set, reflecting the average memory cost. These objectives conflict: better balance requires a wider percentage range, which increases storage cost. We search over a population of N_p assignments

using NSGA-II Deb et al. (2002) for N_{iter} generations, with crossover mixing parent assignments and mutation reassigning positions p_i with probability p (Detail in **Algorithm 1**). A single point on the resulting Pareto front is selected as the fixed mask assignment, resolving component starvation as shown in the right of Fig. 3.

4 EXPERIMENT

4.1 EXPERIMENTAL SETUP

We evaluate our approach in the common instruction-tuning setting, where a pre-trained LLM is adapted to downstream tasks via fine-tuning on curated instruction data (Wang et al., 2023; Taori et al., 2023b). All fine-tuning runs are performed on NVIDIA A100 GPUs. We select three backbones spanning different architectures and parameter scales: Llama3.2 3B (Grattafiori et al., 2024), Llama2 7B (Touvron et al., 2023), and Qwen2.5 7B (Hui et al., 2024), and train each on Open-Platypus (Lee et al., 2023), an aggregation of 11 publicly available instruction-tuning corpora. To measure downstream capability, we evaluate the resulting models under the few-shot protocol of the LM Evaluation Harness on five widely used benchmarks: MMLU (Hendrycks et al., 2020), ARC-Easy and ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), TruthfulQA (Lin et al., 2022), and WinoGrande (Sakaguchi et al., 2021). Each task is scored by ranking the candidate answers according to their model-assigned likelihoods and selecting the top-ranked option.

Algorithm 1: Balanced Mask Assignment

Input: candidate masks $\{\mathcal{S}_i^{(p)}\}_{p=1}^{P}$, population size N_p, iterations N_{iter}, budget K
Output: balanced mask assignment π^{best}
Initialize population $\mathcal{P}$ and Pareto archive $\mathcal{A}$
for $t = 1, \ldots, N_{iter}$ **do**
  for $\pi \in \mathcal{P}$ **do**
    Evaluate $f_1(\pi)$ and $f_2(\pi)$ by Eq. 7
  end for
  Update $\mathcal{A}$ with non-dominated assignments
  $\mathcal{P} \leftarrow \emptyset$
  repeat
    $\pi' \leftarrow$ CROSSOVER($\mathcal{A}$)
    Add π' to $\mathcal{P}$ if $f_2(\pi') \leq K$
  until $|\mathcal{P}| = N_p/2$
  repeat
    $\pi' \leftarrow$ MUTATE($\mathcal{A}$)
    Add π' to $\mathcal{P}$ if $f_2(\pi') \leq K$
  until $|\mathcal{P}| = N_p$
end for
$\pi^{best} \leftarrow$ SELECTBEST($\mathcal{A}$)
return π^{best}

4.2 PERFORMANCE RESULTS

Tab. 2 reports few-shot performance across different model families, scales, and PEFT configurations, where the token selection ratio is fixed at around 0.1 for both G-TUNE and TokenSeek to ensure a fair comparison. Several observations emerge. First, G-TUNE consistently outperforms TokenSeek: on Llama3.2 3B with QLoRA, G-TUNE attains an average score of 61.01, exceeding both TokenSeek (60.42) and the QLoRA baseline itself (60.39); a similar pattern holds on Llama2 7B, where G-TUNE reaches 63.16 compared to 62.14 for TokenSeek and 62.55 for QLoRA alone. Second, this advantage is not tied to a specific PEFT configuration: G-TUNE improves over the corresponding adapter-only baseline whether paired with LoRA, QLoRA, or applied without any adapter, and in every case it also edges out TokenSeek under the same configuration. Third, the improvement generalizes across different backbones and scales, with Qwen2.5 7B following the same trend, reaching 65.69 under QLoRA + G-TUNE compared to 65.47 for QLoRA + TokenSeek. Overall, these results indicate that G-TUNE delivers consistently strong downstream performance across diverse model families, scales, and fine-tuning settings.

4.3 GPU MEMORY RESULTS

To better understand the GPU memory results of G-TUNE, we further investigate the impact of the component selection ratio K, which controls the percentage of model components participating in optimization for each input instance. Unless otherwise specified, all experiments are conducted on Llama3.2-3B. Following the setup of TokenSeek, we vary K from 10% to 50% with a step size of 10%. For each ratio, we evaluate both LoRA and QLoRA settings and compare the peak memory consumption of G-TUNE against the corre-

Table 2: **Few-shot evaluation on question-answering benchmarks.** This includes ARC (25-shot), MMLU (5-shot), HellaSwag (10-shot), TruthfulQA (0-shot), and WinoGrande (0-shot). We report the average accuracy on five MMLU ethics tasks and WinoGrande, the normed accuracy on ARC and HellaSwag, and the MC2 score on TruthfulQA. We highlight the better result between TokenSeek and G-TUNE in **bold**.

Method	ARC	HellaSwag	MMLU	TruthfulQA	WinoGrande	Avg. Score
Llama3.2 (3B)						
TokenSeek Zeng et al. (2026)	27.30	**25.96**	58.14	**48.65**	49.72	41.95
G-TUNE (Ours)	**28.12**	25.54	**58.98**	48.12	**50.85**	**42.32**
LoRA Hu et al. (2021)	45.12	68.34	61.78	44.56	63.21	56.60
LoRA + TokenSeek	**46.01**	68.92	62.10	**45.33**	63.98	57.27
LoRA + **G-TUNE (Ours)**	45.78	**69.87**	**63.04**	44.89	**64.12**	**57.54**
QLoRA Dettmers et al. (2023)	51.37	75.88	63.95	42.31	68.43	60.39
QLoRA + TokenSeek	50.00	**76.30**	63.43	**43.37**	68.98	60.42
QLoRA + **G-TUNE (Ours)**	**52.18**	75.92	**64.82**	42.98	**69.12**	**61.01**
Llama2 (7B)						
TokenSeek	52.22	**78.96**	**65.28**	39.95	**68.43**	60.97
G-TUNE (Ours)	**53.67**	78.45	65.01	**40.88**	68.22	**61.25**
LoRA	54.12	77.89	64.56	41.23	68.54	61.27
LoRA + TokenSeek	**54.98**	78.34	**65.02**	42.01	**68.92**	61.85
LoRA + **G-TUNE (Ours)**	54.56	**79.01**	64.89	**42.89**	68.54	**61.98**
QLoRA	56.06	78.60	65.08	43.64	69.38	62.55
QLoRA + TokenSeek	53.50	**78.82**	65.26	**44.62**	68.51	62.14
QLoRA + **G-TUNE (Ours)**	**57.23**	78.45	**65.98**	44.01	**70.12**	**63.16**
Qwen2.5 (7B)						
TokenSeek	**55.89**	81.67	**67.98**	42.56	71.93	64.01
G-TUNE (Ours)	55.34	**82.45**	67.54	**42.89**	**72.56**	**64.16**
LoRA	56.23	80.12	66.89	43.12	70.89	63.45
LoRA + TokenSeek	**57.01**	**80.89**	67.45	44.01	**71.23**	64.12
LoRA + **G-TUNE (Ours)**	56.78	80.34	**68.22**	**44.89**	70.89	**64.22**
QLoRA	58.34	81.76	67.89	45.14	71.89	65.02
QLoRA + TokenSeek	**58.89**	82.01	**68.12**	**46.01**	72.34	65.47
QLoRA + **G-TUNE (Ours)**	58.12	**83.22**	67.89	45.78	**73.45**	**65.69**

sponding TokenSeek baseline. The dashed horizontal line in Fig. 4 represents the peak memory consumption of full parameter tuning, serving as the reference point for memory reduction. Several observations can be drawn from the results. First, the peak memory consumption increases monotonically as K becomes larger. This behavior is expected because more selected components require additional activation storage and gradient computation during backpropagation. Nevertheless, even at $K = 50\%$, both G-TUNE and TokenSeek consume substantially less memory than full parameter tuning, indicating that sparse optimization remains highly effective across a wide range of selection budgets. Second, G-TUNE consistently achieves lower peak memory consumption than TokenSeek under the same configuration and selection ratio. This trend holds for both LoRA and QLoRA, demonstrating that component-level selection is more memory-efficient than token-level selection when operating under an equivalent sparsity budget. Third, the memory savings become even more pronounced when combined with QLoRA. For example, at $K =$

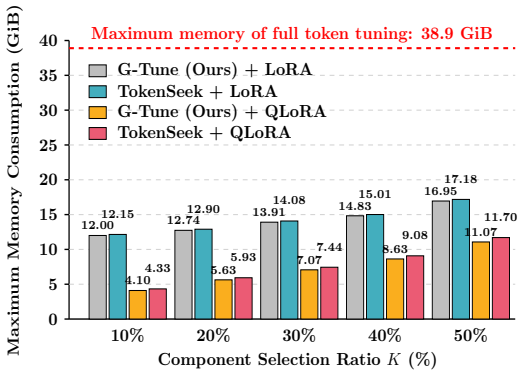

Figure 4: Impact of component selection ratio K on peak memory. Results are measured on Llama3.2-3B with K varied from 10% to 50%. We compare G-TUNE and TokenSeek under LoRA and QLoRA configurations. The dashed line denotes the peak memory consumption of full parameter tuning.

10%, the peak memory consumption is reduced to only 10.5% of full parameter tuning, while remaining below the corresponding TokenSeek configuration. These results confirm that G-TUNE is complementary to existing parameter-efficient fine-tuning techniques and provides a flexible mechanism for controlling memory consumption through a single interpretable hyperparameter K.

4.4 EFFECTIVENESS OF INPUT-AWARE COMPONENT SELECTION

To verify that selecting components by importance is necessary for effective sparse fine-tuning, we compare G-TUNE against random component selection on Llama3.2 3B under full parameter tuning, sweeping the selection ratio K from 10% to 50% while keeping all other settings identical. As shown in Fig. 5, G-TUNE consistently outperforms random selection across the entire range of K, and the gap remains stable rather than closing as the budget increases. This indicates that the advantage of G-TUNE does not simply come from having more components available to update, but from correctly identifying which components matter for the current input: even when random selection is given the same budget, choosing components without regard to their importance fails to make effective use of the additional capacity. These results confirm that **importance-guided component selection is essential to the effectiveness of sparse fine-tuning, and that this benefit holds robustly across different selection budgets.**

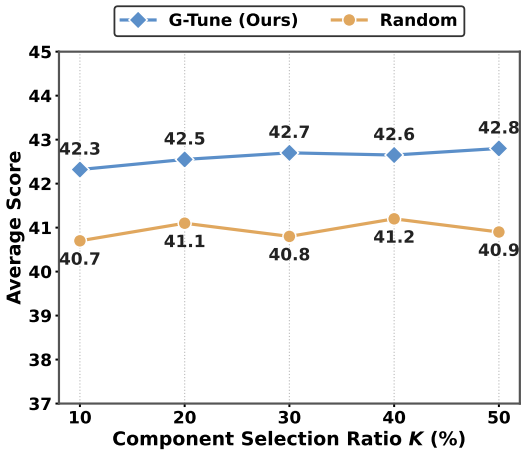

Figure 5: Average score on Llama3.2 3B under full parameter tuning, comparing G-TUNE against random component selection across selection ratios $K = 10\%$–50%.

4.5 STABILITY BENEFIT OF BALANCED MASK ASSIGNMENT.

We further examine whether component starvation, identified in Section 3.2, actually translates into a measurable performance cost during fine-tuning, and whether balanced mask assignment is an effective remedy. We compare Fixed K selection against three representative points sampled from the Pareto front produced by NSGA-II, all evaluated on Llama3.2 3B under full parameter tuning with the selection budget fixed at $K = 10\%$. Under Fixed K, the average score is only 39.8, noticeably lower than what the same budget can achieve once starvation is addressed. Balanced Point A, which sits at the end of the Pareto front closest to Fixed K in terms of storage cost, already reaches 41.9, a gain of 2.1 points from correcting the imbalance alone. Moving further along the front toward more balanced solutions, Point B reaches 42.3 and Point C reaches 42.2, showing that the gain grows as the assignment becomes more balanced, but that even the least aggressive point on the front already substantially outperforms Fixed K. Taken together, these results provide direct evidence that component starvation is not a theoretical artifact but a real source of performance degradation, and that balanced mask assignment reliably closes this gap: **every point we evaluate on the Pareto front, regardless of its specific balance-storage trade-off, consistently and comfortably outperforms Fixed K.**

Method	Avg. Score
Fixed K	**39.8**
Balanced (Point A)	**41.9**
Balanced (Point B)	**42.3**
Balanced (Point C)	**42.2**

Table 3: Fixed K vs. three points on the Balanced Mask Assignment Pareto front (Llama3.2 3B, full parameter tuning, $K = 10\%$).

4.6 LAYER-WISE COMPONENT ANALYSIS

To understand how the selected component set $\mathcal{S}(x)$ is distributed across network depth, we profile its layer-wise composition on Llama3.2 (3B) (28 Transformer layers, 24 attention heads per layer, i.e., 700 components in total) fine-tuned on Open-Platypus. For each of 500 randomly sampled training instances x, we compute the component importance score $s(c; x)$ following Eq. 4 and retain the Top-K components as $\mathcal{S}(x)$ following Eq. 5, with K set to 10% of all components. We record the layer index of every component in $\mathcal{S}(x)$ and aggregate across all instances, grouping the 28 layers into four equal depth ranges (L1–7, L8–14, L15–21, and L22–28).

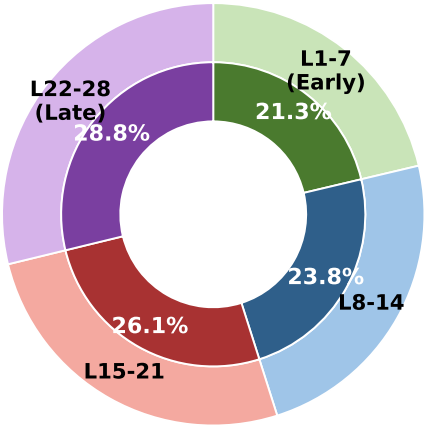

Figure 6: Layer-wise distribution of the selected component set $\mathcal{S}(x)$ (attention heads and MLP blocks combined) on Llama3.2 (3B).

As shown in Fig. 6, component selection is not uniform across depth: the last quarter of layers (L22–28) accounts for 28.8% of selected components, versus 21.3% for the first quarter (L1–7), increasing steadily in between. This matches Eq. 4: gradients weaken as they backpropagate through more layers, so components closer to the output receive a stronger signal and score higher. The gap is small and the increase is smooth across all four ranges, with no single layer standing out. **This shows $\mathcal{S}(x)$ is genuinely instance-aware rather than a fixed depth-based rule, and the effect is mild enough to remain compatible with the balanced mask assignment in Section 3.2.**

5 Related Work

Memory-efficient fine-tuning methods fall into three paradigms. Recomputation, i.e., gradient checkpointing (Chen et al., 2016), trades compute for memory by recomputing activations, with extensions to checkpointing schedules (Jain et al., 2020), runtime strategies (Kirisame et al., 2020), and offloading (Rajbhandari et al., 2020). Compression reduces stored-state size via sparsification, e.g., LoRA (Hu et al., 2021) and diff pruning (Guo et al., 2021), or quantization, e.g., mixed-precision training (Micikevicius et al., 2017) and QLoRA (Dettmers et al., 2023); TokenTune (Simoulin et al., 2024) and TokenSeek (Zeng et al., 2026) instead sparsify token activations during backpropagation. Reversible networks (Gomez et al., 2017; Kitaev et al., 2020), originally proposed for image classification and later extended to Transformer architectures, avoid caching activations altogether by reconstructing them from later layers. Orthogonal to these, parameter-efficient fine-tuning (PEFT) reduces trainable parameters via partial tuning (Ding et al., 2023), adapters (Houlsby et al., 2019), low-rank reparameterization (Hu et al., 2021; Dettmers et al., 2023), or hybrid combinations (Hendrycks et al., 2020). However, because PEFT only restricts which parameters are updated rather than which activations are retained, it does little for activation memory, which remains over 75% of full fine-tuning even with under 1% trainable parameters (Liao et al., 2023). Despite this progress, memory-efficient and parameter-efficient methods have largely been studied independently, and neither line of work has explored sparsity at the level of individual model components – attention heads and MLP blocks – as a mechanism for reducing activation memory during fine-tuning.

6 Conclusion

We presented G-TUNE, a memory-efficient fine-tuning method that scores and selects input-specific components for backpropagation while resolving component starvation via balanced mask assignment. Across three backbones and multiple PEFT configurations, G-TUNE consistently surpasses prior token-level sparsity methods in both accuracy and memory efficiency. These results demonstrate that exploiting component-level redundancy provides an effective complement to token-level sparsity, enabling more selective gradient computation without sacrificing model adaptation. We believe G-TUNE offers a practical direction for reducing the memory overhead of LLM fine-tuning and facilitating efficient adaptation under resource-constrained settings.

REFERENCES

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2): 182–197, 2002.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature machine intelligence*, 5(3):220–235, 2023.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems*, 30, 2017.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Demi Guo, Alexander M Rush, and Yoon Kim. Parameter-efficient transfer learning with diff pruning. In *Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 1: Long papers)*, pp. 4884–4896, 2021.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Evan Hernandez, Arnab Sen Sharma, Tal Haklay, Kevin Meng, Martin Wattenberg, Jacob Andreas, Yonatan Belinkov, and David Bau. Linearity of relation decoding in transformer language models. In *International Conference on Learning Representations*, volume 2024, pp. 10504–10526, 2024.

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.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Ariel N Lee, Cole J Hunter, and Nataniel Ruiz. Platypus: Quick, cheap, and powerful refinement of llms. *arXiv preprint arXiv:2308.07317*, 2023.

Baohao Liao, Shaomu Tan, and Christof Monz. Make pre-trained model reversible: From parameter to memory efficient fine-tuning. *Advances in Neural Information Processing Systems*, 36:15186–15209, 2023.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 3214–3252, 2022.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Yixin Ou, Yunzhi Yao, Ningyu Zhang, Hui Jin, Jiacheng Sun, Shumin Deng, Zhenguo Li, and Huajun Chen. How do llms acquire new knowledge? a knowledge circuits perspective on continual pre-training. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19889–19913, 2025.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pp. 1–16. IEEE, 2020.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Antoine Simoulin, Namyong Park, Xiaoyi Liu, and Grey Yang. Memory-efficient fine-tuning of transformers via token selection. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 21565–21580, 2024.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7, 2023a.

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, 2023b.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 13484–13508, 2023.

Yunzhi Yao, Ningyu Zhang, Zekun Xi, Mengru Wang, Ziwen Xu, Shumin Deng, and Huajun Chen. Knowledge circuits in pretrained transformers. *Advances in Neural Information Processing Systems*, 37:118571–118602, 2024.

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

Runjia Zeng, Qifan Wang, Qiang Guan, Ruixiang Tang, Lifu Huang, Zhenting Wang, Xueling Zhang, Cheng Han, and Dongfang Liu. Tokenseek: Memory efficient fine tuning via instance-aware token ditching. *arXiv preprint arXiv:2601.19739*, 2026.

A AI USE STATEMENT

Large Language Models (LLMs) were used to assist with the writing and polishing of this manuscript, including language refinement, sentence rephrasing, grammar checking, and improving overall readability. All LLM-assisted content was carefully reviewed and verified by the authors, who take full responsibility for the final manuscript.

B ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. The research does not involve human subjects, personally identifiable information, or other privacy-sensitive data. We have carefully considered potential ethical concerns throughout the research process and found no significant ethical risks associated with this work.

C REPRODUCIBILITY STATEMENT

We have made every effort to ensure the reproducibility of this work. The methodology, experimental settings, model configurations, datasets, and evaluation metrics are described in detail in the paper and appendix. These details are provided to facilitate independent reproduction and verification of our experimental results.

D EDGE ATTRIBUTION PATCHING WITH INTEGRATED GRADIENTS

To analyze the computational mechanisms underlying different inputs, we employ Edge Attribution Patching with Integrated Gradients (EAP-IG)Ou et al. (2025), a gradient-based attribution method that efficiently estimates the contribution of every computational edge without explicitly ablating each edge. Compared with exhaustive edge patching, EAP-IG provides a faithful approximation of edge importance using only a small number of gradient evaluations, making it suitable for large language models.

D.1 TRANSFORMER COMPUTATION GRAPH

Given an input sequence, a Transformer can be represented as a directed computation graph $G = (V, E)$, where each node $v \in V$ corresponds to a computational component, including the input embedding, attention heads, MLP blocks, and the output node. Each directed edge $e = (u, v) \in E$ represents the information flow from component u to component v through the residual stream. For a given input, the objective of EAP-IG is to estimate the importance of every computational edge, providing edge-level attribution scores that characterize the information flow throughout the Transformer computation graph.

D.2 EDGE ATTRIBUTION PATCHING

Instead of evaluating every computational edge through repeated activation patching, EAP-IG estimates edge importance using gradient attribution. For each edge $e = (u, v)$, z denotes the clean activation obtained from the original input, while z' denotes the corresponding activation produced by a corrupted baseline input. The activation difference between the clean and corrupted runs is defined as

$$\Delta z = z - z', \tag{8}$$

A first-order Taylor approximation estimates the attribution score of an edge as

$$S(e) \approx \Delta z \cdot \frac{\partial L}{\partial z}, \tag{9}$$

where L denotes the target prediction objective. Intuitively, an edge receives a high attribution score if a perturbation of its activation causes a significant change in the prediction objective. However, computing gradients only at the clean activation may lead to unstable attribution because the gradient is evaluated at a single operating point. To obtain a more faithful estimate of edge importance, EAP-IG employs Integrated Gradients to accumulate attribution along the interpolation path.

D.3 Integrated Gradients

To compute Integrated Gradients, the activation is interpolated between the corrupted activation z' and the clean activation z as

$$z(\alpha) = z' + \alpha(z - z'), \qquad \alpha \in [0, 1], \tag{10}$$

where α denotes the interpolation coefficient. When $\alpha = 0$, the interpolated activation is equal to the corrupted activation z', whereas $\alpha = 1$ recovers the clean activation z. Intermediate values of α define a continuous path between the two activations. Rather than evaluating the gradient only at the clean activation, Integrated Gradients accumulates gradients along the entire interpolation path, reducing the sensitivity of attribution to a single operating point and yielding a more faithful estimate of edge importance. Accordingly, the attribution score of edge $e = (u, v)$ is computed as

$$S(e) = (z'_u - z_u) \cdot \int_0^1 \frac{\partial L(z(\alpha))}{\partial z_v} d\alpha, \tag{11}$$

where z_u and z'_u denote the clean and corrupted activations associated with the source component u, respectively, and z_v denotes the activation received by the destination component v. The activation difference measures the magnitude of the perturbation introduced on the edge, while the integrated gradient measures the sensitivity of the prediction objective to this perturbation along the interpolation path. In practice, the integral is approximated using m uniformly sampled interpolation points,

$$S(e) \approx (z'_u - z_u) \cdot \frac{1}{m} \sum_{k=1}^{m} \frac{\partial L\big(z' + \frac{k}{m}(z - z')\big)}{\partial z_v}, \tag{12}$$

where m denotes the number of integration steps. Following the original EAP-IG implementation, we set $m = 5$ throughout this work. The resulting attribution score quantifies the contribution of each computational edge to the model prediction. Edges with larger absolute attribution values are regarded as more important for the current input.

D.4 Edge Selection

After obtaining attribution scores for all computational edges, the edges are ranked according to the absolute values of their attribution scores. Edges with larger attribution values are considered more important for the current input and are retained, while those with small attribution values are discarded. To identify the dominant information flow, the retained edges are further filtered according to their connectivity in the Transformer computation graph. Specifically, only connected computational paths that propagate information from the input toward the model prediction are preserved, while isolated edges that do not participate in such paths are removed. The resulting connected edge paths capture the major information flow responsible for the current prediction and provide an interpretable representation of the model's computation for a specific input.

E Quantifying Cross-Input Diversity of Sparse Component Sets

To further quantify the diversity of input-specific sparse component sets observed in Section 2.1, we compute the pairwise Jaccard similarity of $\mathcal{S}(x)$ across $N = 50$ inputs spanning the four knowledge types described in Section 2.1. For each pair of inputs (x_i, x_j), the Jaccard similarity is defined as

$$J\big(\mathcal{S}(x_i), \mathcal{S}(x_j)\big) = \frac{|\mathcal{S}(x_i) \cap \mathcal{S}(x_j)|}{|\mathcal{S}(x_i) \cup \mathcal{S}(x_j)|}, \tag{13}$$

We report the average Jaccard similarity across all pairs. As shown in Table 4, the average pairwise Jaccard similarity is only 0.12, confirming that sparse component sets exhibit limited overlap across different inputs and further supporting the input-specific nature of sparse components identified in Section 2.1.

F Definition of Hit@10

To evaluate the predictive performance of different component configurations, we adopt Hit@10 as the evaluation metric. Given an input sequence x, the language model predicts a probability

Metric	Value
Number of inputs (N)	50
Number of pairs	1225
Average Jaccard similarity	0.12

Table 4: Pairwise Jaccard similarity of input-specific sparse component sets $\mathcal{S}(x)$, computed across all pairs from N inputs spanning four knowledge types (Linguistic, Commonsense, Factual, Bias).

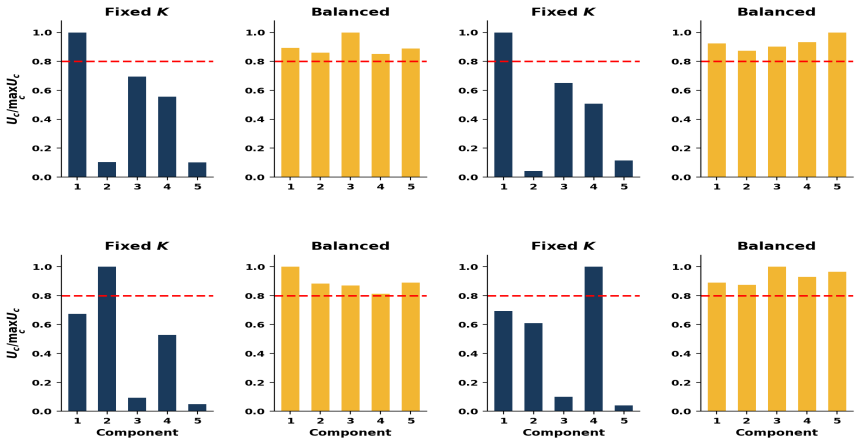

Figure 7: Four additional examples comparing normalized update coverage under Fixed K and our balanced mask assignment. Red dashed lines mark the 0.8 reference level.

distribution over the vocabulary for the next token. Let y^* denote the ground-truth target token, and let $\mathcal{V}_{10}(x)$ denote the set of the ten tokens with the highest predicted probabilities. The Hit@10 score for a single sample is defined as

$$\text{Hit@10}(x) = \begin{cases} 1, & \text{if } y^* \in \mathcal{V}_{10}(x), \\ 0, & \text{otherwise,} \end{cases} \tag{14}$$

where a prediction is regarded as successful if the ground-truth target token appears among the model's top-10 predictions. Given an evaluation dataset containing N samples, the overall Hit@10 is computed as

$$\text{Hit@10} = \frac{1}{N} \sum_{i=1}^{N} \text{Hit@10}(x_i), \tag{15}$$

which measures the fraction of evaluation samples whose ground-truth target tokens are included in the top-10 predicted tokens. In our experiments, Hit@10 is computed using the full model, the input-specific sparse component sets, and randomly selected component sets under the same component budget.

G Additional Examples of Balanced Mask Assignment

To further illustrate the effect of our balanced mask assignment, we present four additional representative examples in Fig. 7. For each example, we show the normalized update count $U_c / \max_c U_c$ across 5 representative components under Fixed K selection and under our balanced mask assignment. Across all four examples, Fixed K selection consistently leaves several components with near-zero coverage, while our method consistently narrows this gap toward the reference level, confirming that the balance improvement observed in Fig. 3 is not an isolated case.