Deputy: Accelerating Large Language Model Inference with Dynamic Low-Rank Substitution

Yuhua Zhou[1]*, Shichao Weng[2]*, Changhai Zhou[2]*, Yuhan Wu[1], Qian Qiao[4], Jun Gao[1], Fei Yang[3]†, Aimin Pan[3]†

[1]Zhejiang University, [2]Fudan University, [3]Zhejiang Lab, [4]OpenWPLab

zhouyuhua@zju.edu.cn, {yangf,panaimin}@zhejianglab.org

Abstract

While the massive scale of modern LLMs enables remarkable performance, their static, input-agnostic computational graph incurs substantial resource wastage and high latency during inference. Existing dynamic schemes, such as *early-exit* and *layer-drop* reduce FLOPs but break batch processing or introduce KV-cache inconsistency. We propose *Deputy*, a dynamic low-rank substitution framework that employs a lightweight decision module at each layer to dynamically determine the execution branch for different tokens: Attention layers choose between full and low-rank computation to mitigate the KV cache issue, while FFN layers additionally support skipping to further reduce computation. We fine-tune the LLM with LoRA and then derive an additional low-rank matrix C via a least-squares fit $\mathbf{BC} \approx \mathbf{W}_{\text{pre}}$, where B is the shared LoRA matrix, so that only one extra low-rank matrix is introduced, effectively reducing memory overhead. Moreover, a hybrid KV cache strategy stores KV values generated by the low-rank branch, achieving a 38% reduction in cache storage. Experiments on Llama models demonstrate that Deputy reduces computation by approximately 40% compared to the original dense model while outperforming existing baseline methods. The code is available at https://github.com/yuhua-zhou/Deputy.

1 Introduction

Recent advances in large language models (LLMs) have led to remarkable improvements in various text understanding and generation tasks (Chang et al., 2024; Zhou et al., 2025b, 2026b,a; Gao et al., 2025a,b,c). Despite these successes, the growing scale of LLMs, often containing billions of parameters, poses significant challenges in terms of computational cost and memory footprint during inference. To address this issue, numerous model compression techniques have been proposed, including quantization (Yang et al., 2023; Yao et al., 2024; Zhou et al., 2024), knowledge distillation (Wang et al., 2023; Li et al., 2024), and pruning (Ma et al., 2023; Kim et al., 2024; Zhou et al., 2025a). However, such static compression approaches typically cause model performance degradation and compromise generalizability across tasks of varying complexity.

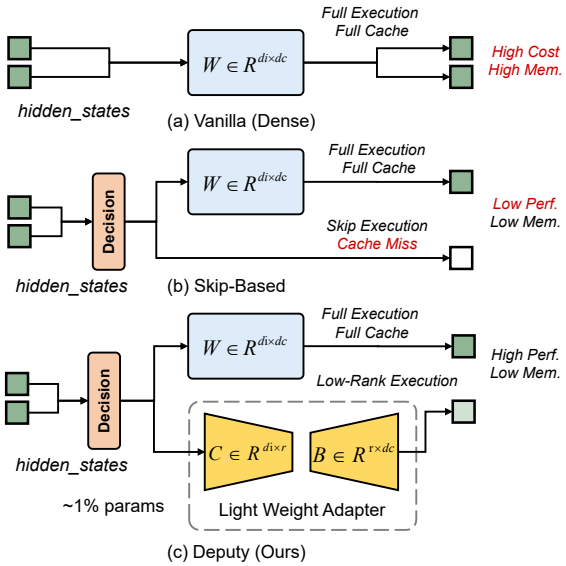
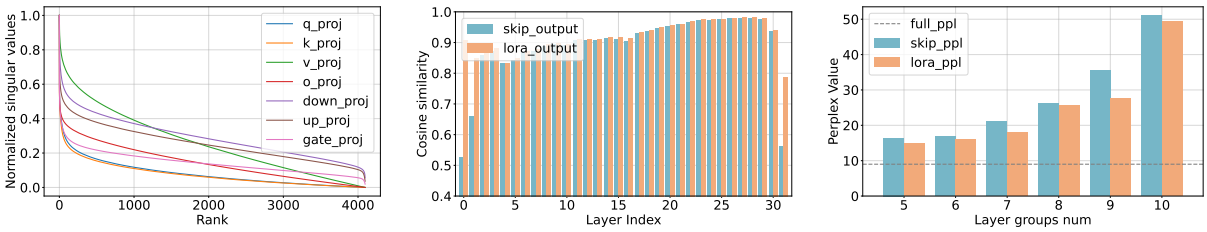


Figure 1: (a) Vanilla model executes full computation; (b) Skip-based model selectively skips some computation; (c) Deputy selectively light-weight computation.

To mitigate these drawbacks, recent work has focused on conditional computation methods that selectively activate only a subset of parameters based on the inputs, so as to decrease computational cost and preserve model performance. Methods such as *early-exiting* (Elhoushi et al., 2024) or selective *layer-dropping* (Fan et al., 2024; Raposo et al., 2024) have shown promising results, but they suffer from the problematic "KV cache miss" issue and batch process problem. They proposed that copy-

*Equal contribution.
†Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 19791–19810
July 2-7, 2026



(a) Normalized average singular values (b) Cosine similarity (low-rank) (c) PPL under different layer groups.

Figure 2: Left: Normalized average singular values. Middle: Cosine similarity (low-rank). Right: PPL performance under skip/low-rank execution on layer groups.

ing the KV caches from earlier layers (Schuster et al., 2022), evict (Jiang et al., 2024), or recomputation (Chen et al., 2024) to solve these issues, but inevitably risk losing valuable contextual information or introducing recomputation latency. Besides, attempts such as skipping only the Feed-Forward Network (FFN) layers will ignore the speedup potential gained from skipping part of the attention computation. Consequently, further optimization is required to achieve efficient LLM inference.

Low-rank approximation (Hsu et al., 2022; Li et al., 2023; Lv et al., 2023) offers another avenue for model compression. This technique significantly reduces computational overhead while maintaining the intrinsic dimension (Hu et al., 2022) and information of the original parameters. Compared to directly skipping layers, low-rank factorization can provide a more balanced solution: it reduces computation without sacrificing contextual information, while storing the low-rank KV values as the KV cache, thereby avoiding the KV cache issues introduced by layer skipping and improving performance (Section 3). However, designing an approach to dynamic low-rank approximation has two technical challenges: (1) **Token Selection:** *How to select which tokens should be executed by the low-rank branch.* (2) **Overhead Management:** *how to mitigate the performance degradation and memory overhead introduced by additional low-rank weights.*

In this paper, we introduce **Deputy**, a novel framework that introduces a new low-rank execution branch and dynamically selects a different execution branch for each token. To address the first challenge, we design separate decision modules for Attention and FFN layers in each Transformer block to provide an optimal execution branch for each token: (1) **Full-Parameters** branch executed with the pre-trained weights; (2) **Low-Rank** branch employs two low-rank weights to approximate the pre-trained weight; (3) **Skipping** branch skips the computation (FFN layers only). To tackle the second challenge, we finetune the LLM with LoRA (Hu et al., 2022). Afterwards, we apply a Least Squares Estimation (LSE) technique to compute a matrix C that, together with B, approximates the pre-trained weights as $\mathbf{BC} \approx \mathbf{W}_{\text{pre}}$. This strategy allows us to introduce only one additional matrix, $\mathbf{C}$, minimizing memory overhead and preserving efficiency. Additionally, we design a hybrid KV cache strategy, where KV cache would be stored as a low-rank value to reduce the storage footprint.

We validate the effectiveness of Deputy using Llama-based models across diverse zero-shot tasks. Our results demonstrate that Deputy achieves superior accuracy, perplexity, and FLOPs reduction compared to baseline methods. Moreover, when utilizing the proposed hybrid KV cache method, Deputy can reduce storage overhead by an average of 38%. In summary, our contributions are as follows:

- We propose *Deputy*, a dynamic inference framework that adaptively routes tokens to Full-Parameter, Low-Rank, or Skip branches. This granularity balances computational efficiency with strict KV-cache consistency. Moreover, we derive a matrix $\mathbf{C}$ via LSE to optimize the memory overhead.
- We tackle the memory bottleneck of LLM inference by introducing a **Hybrid KV Cache** strategy. By explicitly storing low-rank compressed states for non-critical tokens, we break the dependency between sequence length and memory usage.
- We conduct comprehensive experiments on Llama models across various tasks, validating that Deputy markedly reduces computational cost, while achieving better or comparable accuracy relative to the baselines.

2 Related Work

2.1 Model Compression for LLM

Model compression for LLMs aims to reduce model size and computational demands (Zhou et al., 2026c). Current *static* methods permanently alter the model's structure or weights, including pruning, which removes redundant weights (Frantar and Alistarh, 2023; Sun et al., 2023) or entire structures (Xia et al., 2023; Ma et al., 2023); quantization, which reduces numerical precision (Yang et al., 2023; Yao et al., 2024); and knowledge distillation, where a smaller student model learns from a larger teacher (Wang et al., 2023; Li et al., 2024). Low-rank factorization is another popular approach that decomposes large weight matrices into smaller low-rank matrices, so that improve computational efficiency (Lv et al., 2023; Hsu et al., 2022).

However, these static approaches fundamentally modify the original model, which would limit their capacity for different tasks, leading to some performance and flexibility degradation. Deputy diverges from this paradigm by performing dynamic computation for different input tokens.

2.2 Conditional Computation for LLM

Conditional computation offers a more flexible alternative by adapting its computational graph based on inputs, thus enabling a superior trade-off between performance and efficiency (Han et al., 2022; Li et al., 2021).

Recently, some works focus on dynamically selecting/skipping some module weights (e.g., heads, channels, experts, or layers) based on the inputs (Cai et al., 2024; Liu et al., 2023). Another research attempt proposes the token-level compression techniques (Elbayad et al., 2020; Liu et al., 2021), which skip some redundant tokens to reduce computation (Elbayad et al., 2020; Liu et al., 2021). The token-level compression includes *early-exit* technique (Fan et al., 2024; Elhoushi et al., 2024) that enables the LLMs to output from a mid-level layer while omitting subsequent layers, and *token-dropping* methods (Raposo et al., 2024), which utilize a router or decision module to determine whether to drop some tokens individually. Nevertheless, these methods encounter challenges regarding the Key-Value (KV) cache issue. Some attempt to share or reconstruct the caches from adjacent layers (Liu et al., 2023) or discard them (Jiang et al., 2024), both risking the loss of vital contextual information. A different line of work circumvents this issue by only skipping FFN blocks (Peroni and Bertsimas, 2024; Jaiswal et al., 2024), but it limits the potential efficiency gains for skipping the computation of attention.

This work focuses on dynamic token skipping and adjusts the model structure via introducing additonal computation path to provide a more holistic conditional computation framework, to dynamically skip computation of *both* FFN and attention blocks, while handling the KV cache issue.

3 Motivation

We premise our approach on the observation that weight matrices in LLMs exhibit intrinsic low-rank structures. As shown in Figure 2a, the singular values of pretrained weights decay rapidly, suggesting that the bulk of the computational energy can be captured by low-rank matrices $A \in \mathbb{R}^{d_{out} \times r}$ and $B \in \mathbb{R}^{r \times d_{in}}$ (where $r \ll d$).

To validate the potential of this property for dynamic inference, we compare low-rank approximation against standard layer skipping. Our preliminary analysis reveals a critical distinction:

- **Representation Fidelity:** As illustrated in Figure 2b, hidden states produced by low-rank execution maintain consistently high cosine similarity to the original full-rank outputs. In contrast, skipping layers causes a sharp divergence in representation, effectively severing the semantic continuity.

- **Performance Stability:** This representational gap directly impacts model perplexity (PPL). Figure 2c demonstrates that replacing layer groups with low-rank approximations yields PPL significantly closer to the dense model compared to skipping.

These findings indicate that layer skipping suffers from a "discontinuity problem," disrupting the Key-Value (KV) cache chain required for coherent generation. Low-rank execution, however, acts as a **semantic bridge**, drastically reducing FLOPs while preserving the essential structural information needed to maintain KV cache consistency. This motivates *Deputy*: instead of a binary choice between expensive execution and destructive skipping, we introduce an adaptive low-rank branch to trade off fine-grained details for speed only when semantic redundancy permits.

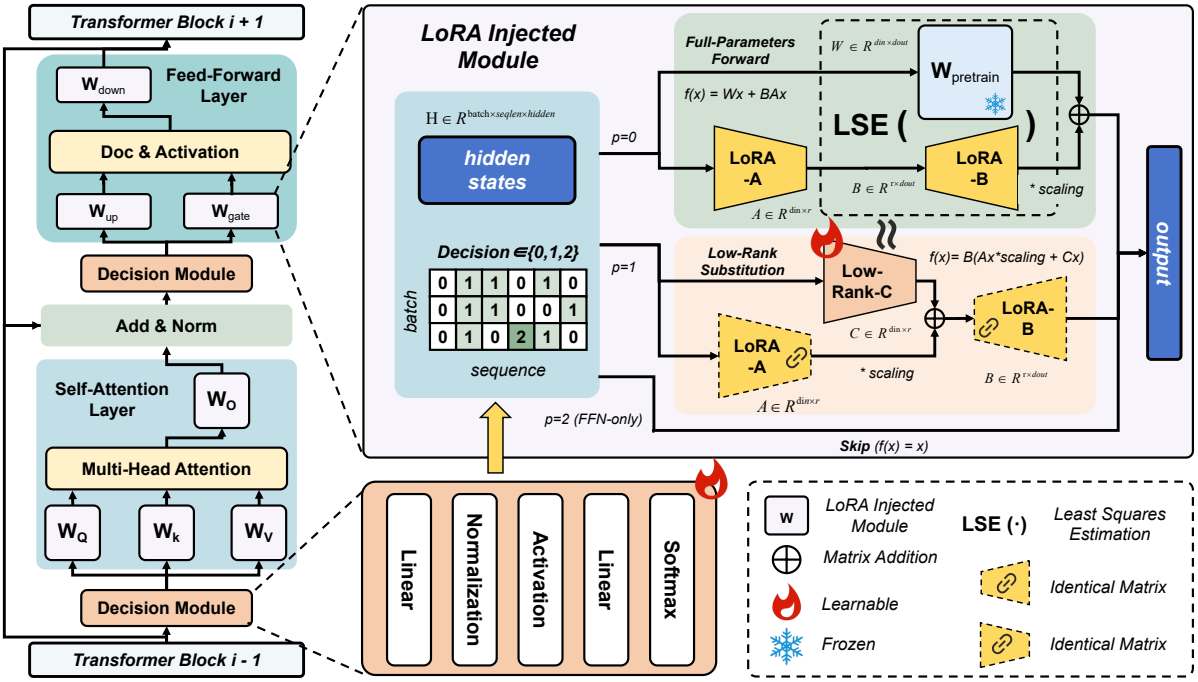


Figure 3: Overview of Deputy: Two individual decision modules decide the execution branch for attention and FFN layers. The low-rank execution branch introduces an additional low-rank matrix, initialized by the up-projection of LoRA with the pretrained weight using the LSE method, to down-projection the hidden states and generate the output combined with the LoRA weights.

4 Methodology

We present *Deputy*, a dynamic execution framework that selectively activates different computational branches in LLMs to optimize efficiency. *Deputy* intelligently chooses between full parameter execution, low-rank approximation, or layer skipping based on input characteristics.

4.1 Framework Overview

Deputy employs lightweight decision modules for both Attention and Feed-Forward Network (FFN) layers in transformers (Figure 3). For each input token $x_i \in \mathbb{R}^{1\times d}$, these modules dynamically route computation through the most efficient path while maintaining model performance.

4.2 Formulation

For each transformer block we design decision modules to decide the execution path for each token. We design distinct decision modules for the attention and FFN layers to assess each input token and determine the appropriate execution branch. For layer l and token i, we define a generic branching operator:

$$\mathcal{F}^l(h_i, G_i^l) = \begin{cases} h_i + \mathcal{F}^l_{\text{full}}(h_i) & \text{if } G_i^l = 0, \\ h_i + \mathcal{F}^l_{\text{lowrank}}(h_i) & \text{if } G_i^l = 1, \\ h_i & \text{if } G_i^l = 2, \end{cases} \quad (1)$$

where h_i denotes the input hidden state for token i at layer l, and G_i^l is the corresponding discrete routing decision. The function $\mathcal{F}^l_{\text{full}}$ applies the original full-parameter transformation at layer l, while $\mathcal{F}^l_{\text{lowrank}}$ applies its low-rank approximation. The case $G_i^l = 2$ corresponds to skipping the layer and forwarding h_i through the residual connection, thereby saving computation.

Attention layer decision. The key–value cache is crucial for maintaining contextual information, and we store cache entries for every token. Skipping attention for a subset of tokens would create KV-cache misses and inconsistent attention contexts. Therefore, in attention layers we only allow full-parameter and low-rank execution:

$$\mathcal{F}^l_{\text{attn}}(h_i) = \mathcal{F}^l(h_i, G^l_{\text{attn},i}), \quad G^l_{\text{attn},i} \in \{0, 1\}, \quad (2)$$

where $G^l_{\text{attn},i} = 0$ selects the full-parameter attention and $G^l_{\text{attn},i} = 1$ selects the low-rank attention. Both branches produce valid KV states and thus preserve KV-cache consistency.

FFN layer decision. FFN layers typically contain more parameters and incur higher computational cost than attention. Since they do not interact with the KV cache, we can safely introduce an additional skip branch for FFNs:

$$\mathcal{F}^l_{\text{ffn}}(h_i) = \mathcal{F}^l(h_i, G^l_{\text{ffn},i}), \quad G^l_{\text{ffn},i} \in \{0, 1, 2\}, \quad (3)$$

where $G^l_{\text{ffn},i} = 0$ selects the full-parameter FFN, $G^l_{\text{ffn},i} = 1$ selects the low-rank FFN, and $G^l_{\text{ffn},i} = 2$ skips the FFN layer.

This conditional execution strategy allows *Deputy* to adaptively allocate expensive full-parameter computation only to tokens that benefit from it, while routing others through cheaper low-rank or skip branches, thereby balancing computational efficiency and model performance.

4.3 Optimization via Least Squares

A naive implementation of low-rank approximation (e.g., via SVD) would decompose W_{pre} into two new matrices, doubling the parameter storage requirement for the alternative branch. To address this memory bottleneck, we propose a constrained optimization approach that leverages the existing LoRA adapter. Since the LoRA matrix $B \in \mathbb{R}^{r \times d_{out}}$ already captures the task-specific low-rank subspace during fine-tuning, we reuse B as a shared basis. We then solve for a single projection matrix $C \in \mathbb{R}^{d_{in} \times r}$ such that the product BC approximates the frozen pretrained weight W_{pre}. This formulation transforms the approximation into a linear least-squares problem, minimizing the reconstruction error under memory constraints via Least Squares Estimation (LSE):

$$\begin{aligned} &\arg\min_{\mathbf{C}} \|\mathbf{BC} - \mathbf{W}_{\text{pre}}\|_F \ , \\ &\mathbf{C} = \left(\mathbf{B}^T\mathbf{B}\right)^{-1}\mathbf{B}^T\mathbf{W}_{\text{pre}} \ . \end{aligned} \quad (4)$$

In this way, we only need to introduce a single additional low-rank matrix to support our dynamic inference scheme. Moreover, the LSE initialization takes only a few seconds and yields a small reconstruction error, thereby preserving more information from the pretrained parameters (discussed in the Appendix). Consequently, the computation in the low-rank execution branch of layer l is defined as:

$$F^l_{\text{lowrank}}(x_i) = s\,\mathbf{BA}x_i + \mathbf{BC}x_i \ , \quad (5)$$

where s denotes the scaling factor used in LoRA. By exploiting the associativity of matrix multiplication, this can be rewritten as:

$$F^l_{\text{lowrank}}(x_i) = \mathbf{B}\left(s\mathbf{A} + \mathbf{C}\right)x_i \ . \quad (6)$$

Since both $\mathbf{C}$ and $\mathbf{A}$ participate in the down-projection of the low-rank branch and share the same shape, we can further reduce the computational overhead during inference by merging the weights of $\mathbf{A}$ into $\mathbf{C}$. Specifically, we define:

$$\begin{aligned} &\hat{\mathbf{C}} = \mathbf{C} + s\,\mathbf{A} \ , \\ &F^l_{\text{lowrank}}(x_i) = \mathbf{B}\hat{\mathbf{C}}x_i \ . \end{aligned} \quad (7)$$

4.4 Implementation

We design the decision module via a lightweight MLP layer (see Figure 3) for each layer. During training, we only mark the additional $\mathbf{ABC}$ matrix and the parameters of decision modules as trainable parameters, and we jointly optimize these parameters through the backward propagation method (details see details in Appendix).

We introduce a cost term $\mathcal{L}_{\text{cost}}$ that measures computational cost during the forward process. We introduce factors $\alpha = (\alpha_f, \alpha_l)$ and $\beta = (\beta_f, \beta_l, \beta_s)$ to customize our acceleration ratio. It is formulated as follows:

$$\begin{aligned} &\mathcal{L}_{\text{decision}} = \tfrac{1}{n}\textstyle\sum_{l=1}^{n}\sum_{i=1}^{5}\left|G_i^{(l)} - f_i\right| \ , \\ &\mathbf{G}^{(l)} = \left(G^{f,(l)}_{\text{attn}}, G^{l,(l)}_{\text{attn}}, G^{f,(l)}_{\text{ffn}}, G^{l,(l)}_{\text{ffn}}, G^{s,(l)}_{\text{ffn}}\right) \ , \\ &\mathbf{f} = \left(\alpha_f, \alpha_l, \beta_f, \beta_l, \beta_s\right) \ , \\ &G^{f,(l)}_{\text{attn}} + G^{l,(l)}_{\text{attn}} = 1, G^{f,(l)}_{\text{ffn}} + G^{l,(l)}_{\text{ffn}} + G^{s,(l)}_{\text{ffn}} = 1 \ , \end{aligned} \quad (8)$$

where n denotes the number of layers, $\mathbf{G}^{(l)}$ means the decision output ratio for each layer, $\mathbf{f}$ refers to the given customized acceleration ratio for each execution path. We calculate the mean absolute error (MAE) between $\mathbf{G}^{(l)}$ and $\mathbf{f}$. We add a penalty factor λ to balance cross-entropy loss $\mathcal{L}_{\text{cls}}$ and computational cost $\mathcal{L}_{\text{cost}}$. Our optimization objective is to minimize total loss:

$$\mathcal{L} = \mathcal{L}_{cls} + \lambda\mathcal{L}_{cost} \quad (9)$$

4.5 Hybrid KV Cache

To fully exploit low-rank approximation, we introduce a Hybrid KV cache that jointly optimizes the storage and computation of key–value states to further reduce GPU memory usage.For tokens routed to the low-rank execution branch, we cache the key_states and $value\_states$ after the down-projection, instead of storing their full-dimensional

Dataset	Shortened (rm=11)		Adainfer (svm)		MoD		D-LLM		Deputy	
Accuracy Tasks										
Com. Sen.	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓
OBQA	30.80	1121.52 (0.66)	**35.00**	1666.62 (0.98)	25.60	1102.65 (0.65)	29.60	1549.59 (0.91)	32.80	**1007.72 (0.59)**
PIQA	58.38	1121.52 (0.66)	62.13	1512.51 (0.89)	55.11	1056.08 (0.62)	58.81	1547.15 (0.91)	**68.61**	**1020.93 (0.60)**
BoolQ	62.60	1121.52 (0.66)	**72.78**	1404.38 (0.83)	50.43	**977.07 (0.58)**	61.28	1536.40 (0.91)	71.01	1079.90 (0.64)
SIQA	39.51	1121.52 (0.66)	39.76	1683.91 (0.99)	35.98	1076.07 (0.63)	35.67	1549.07 (0.91)	**42.86**	**1016.06 (0.60)**
Hellaswag	48.20	1121.52 (0.66)	43.00	1413.77 (0.83)	29.82	**975.18 (0.57)**	31.97	1533.95 (0.90)	**54.37**	1090.04 (0.64)
ARC-E	41.79	1121.52 (0.66)	**52.99**	1636.93 (0.96)	28.79	1056.36 (0.62)	33.63	1548.73 (0.91)	52.40	**1021.93 (0.60)**
ARC-C	33.62	1121.52 (0.66)	**34.98**	1609.28 (0.95)	26.96	1034.14 (0.61)	27.56	1547.68 (0.91)	31.66	**1029.25 (0.61)**
Winogrande	57.14	1121.52 (0.66)	57.62	1508.90 (0.89)	47.67	1123.03 (0.66)	50.43	1550.91 (0.91)	**61.80**	**993.39 (0.59)**
Math	Acc. ↑	FLOPs ↓	Acc. ↑	FLOPs ↓	Acc. ↑	FLOPs ↓	Acc. ↑	FLOPs ↓	Acc. ↑	FLOPs ↓
GSM8K	**2.27**	1121.52 (0.66)	1.29	1460.25 (0.86)	0.00	**979.40 (0.58)**	0.68	1531.00 (0.90)	2.05	1074.26 (0.63)
Other Tasks	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓	Acc. ↑	FLOPs↓
MedQA	**32.91**	1121.52 (0.66)	28.99	1491.28 (0.88)	25.61	**969.37 (0.57)**	26.32	1533.83 (0.90)	28.44	1100.97 (0.65)
COPA	66.00	1121.52 (0.66)	59.00	1539.79 (0.91)	63.00	1096.94 (0.65)	58.00	1544.57.83 (91)	**75.00**	**1086.10 (0.64)**
LogiQA	**29.65**	1121.52 (0.66)	26.27	1375.91 (0.81)	25.19	**969.28 (0.57)**	25.35	1545.84 (0.91)	**28.73**	1099.77 (0.65)
CoQA	44.43	1121.52 (0.66)	42.45	1453.36 (0.86)	2.18	**969.45 (0.57)**	0.55	1533.83 (0.90)	**44.53**	**1102.64 (0.65)**
Avg.	42.10	1121.52 (0.66)	42.79	1519.75 (0.89)	32.03	**1029.54 (0.61)**	33.83	1542.51 (0.91)	**45.70**	1055.61 (0.62)
PPL Tasks										
Text	PPL ↓	FLOPs↓	PPL ↓	FLOPs↓	PPL ↓	FLOPs↓	PPL ↓	FLOPs↓	PPL ↓	FLOPs↓
Wikitext2	93.36	1121.52 (0.66)	126.10	1459.17 (0.86)	1171.11	**969.43 (0.57)**	253.17	1549.07 (0.91)	**43.55**	1102.96 (0.65)
PTB	**109.02**	1121.52 (0.66)	315.87	1454.54 (0.86)	4931.32	**972.14 (0.57)**	340.22	1541.00 (0.91)	297.39	1069.37 (0.63)

Table 1: Results of Zero-shot performance and PPL on Llama 2-7B in Deputy and baselines on various benchmarks. We report the accuracy and average FLOPs. **Boldface** indicates the best performance, and the underline refers to the sub-optimal performance.

counterparts, which substantially decreases the cache footprint. During attention computation, the current $query\_states$ attend to both the full-rank cache and the low-rank cache. For the low-rank cache, we first project $query\_states$ into the low-rank subspace to compute the attention scores with the low-rank keys, and then aggregate the corresponding low-rank values. The outputs from the full-rank and low-rank branches are finally combined to obtain the updated $hidden\_states$. This design is illustrated in Figure 4, and further implementation details are provided in the Appendix.

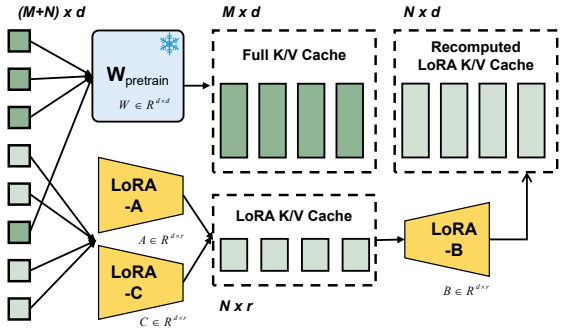


Figure 4: Hybrid KV Cache design: stores the full cache from the full-parameter execution branch and the low-rank cache from the low-rank execution branch. The low-rank cache is reconstructed into a full cache during inference.

5 Experiments

5.1 Experiments Settings

LLMs and Benchmarks. To evaluate Deputy across diverse tasks, we conducted experiments using the Llama 2-7B model (Touvron et al., 2023) on **Commonsense Reasoning** (OpenbookQA (Mihaylov et al., 2018), PIQA (Bisk et al., 2020), BoolQ (Clark et al., 2019), SIQA (Sap et al., 2019), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2021), ARC-easy/challenge (Clark et al., 2018)), **GSM8K** for math reasoning (Cobbe et al., 2021), and **Other tasks** (including MedQA (Jin et al., 2020), COPA (Gordon et al., 2012), LogiQA (Liu et al., 2020), and CoQA (Reddy et al., 2019)). We performed zero-shot evaluation using the lm-eval-harness (Gao et al., 2023) for open prompts and scoring. Additionally, we measured perplexity on Wikitext2 (Merity et al., 2016) and PTB (Marcus et al., 1993). Model latency was quantified using FLOPs (floating-point operations) (via the *thop* library), providing a hardware-agnostic measure of computational cost.

Baselines. We compared state-of-the-art methods as our baselines including **Shortened Llama** (Kim et al., 2024), **AdaInfer** (Fan et al., 2024), **MoD** (Raposo et al., 2024), and **D-LLM** (Jiang

et al., 2024). Details of the official repositories or our reproductions can be found in the Appendix.

Implementation Details. We adopted the popular LoRA method to fine-tune the LLMs. We used Alpaca (Taori et al., 2023), the comprehensive instruction tuning dataset, to fine-tune the models. We set the learning rate to $1e-4$ and converted the model precision to BFloat16. We set the penalty factor to 0.01. We set an initial temperature of 1.0 for each decision model and employed the exponential decay τ to 0.1 during training. We set the acceleration factors $\alpha = \{0.6, 0.4\}$, $\beta = \{0.6, 0.2, 0.2\}$. For more details, see Appendix.

5.2 Main Results

Table 1 summarizes the zero-shot results of Deputy against strong dynamic-computation baselines. Overall, Deputy achieves the best average accuracy (45.70) while using only 62% of the full-model FLOPs, substantially improving the performance–efficiency trade-off. Adainfer attains best or second-best accuracy on several benchmarks, but typically requires $\sim 89\%$ FLOPs on average, indicating a considerably higher compute budget. In contrast, MoD reduces computation most aggressively ($\sim 61\%$ FLOPs on average) but suffers from pronounced accuracy degradation, limiting its practical utility.

For language modeling, Deputy achieves the lowest PPL on Wikitext2 (43.55), which is a 53% reduction compared to the next-best baseline, demonstrating its strength in generative settings. While Shortened remains competitive on PTB, Deputy consistently provides a stronger accuracy/PPL–compute balance than fixed-computation schemes via dynamic, token-level resource allocation.

5.3 Analysis

Rank Selection. The introduced $\mathbf{C}$ matrix has the same dimensionality as the LoRA parameters used for fine-tuning. Therefore, we conduct experiments on Llama 2-7B to analyze the performance recovery ratio and computational cost under different fine-tuning ranks. We report the performance recovery ratio as the normalized accuracy. As shown in Figure 5, the results indicate that as the rank increases, the $\mathbf{C}$ matrix can preserve more information from the pretrained weights, leading to better performance. Besides, as the rank increased to 64 or 128, the computational cost decreased, meaning that the router prefers the low-rank execution branch when the performance is good enough.

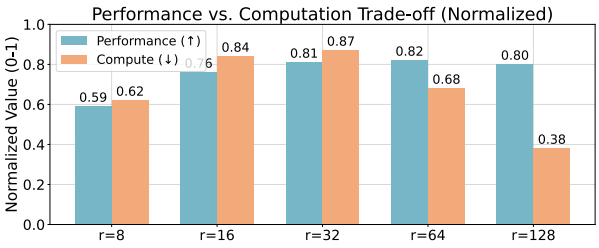


Figure 5: Performance across different rank selections.

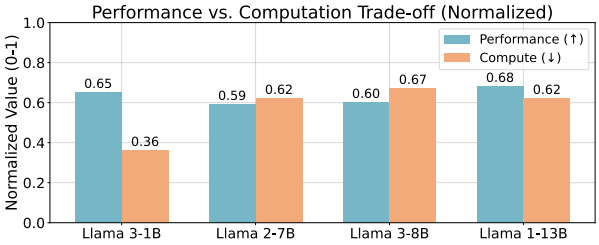


Figure 6: Performance across different model scales.

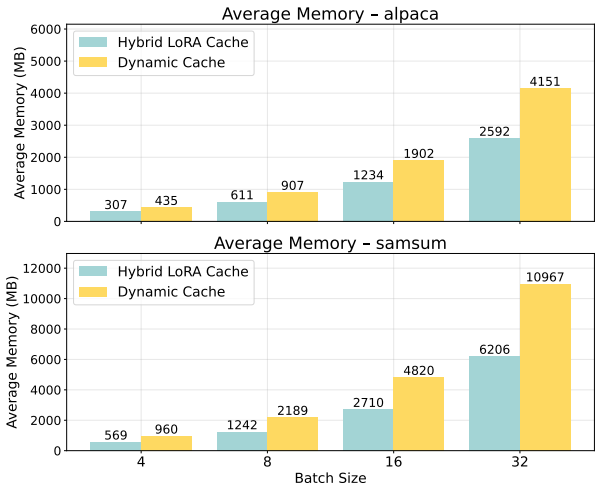


Figure 7: Storage overhead of hybrid KV cache strategy.

Model Scale. To validate effectiveness across different scale models, we also systematically evaluated Deputy across Llama 3.2-1B/2-7B/3-8B/1-13B models via the normalized accuracy and computational cost. Results (Figure 6) demonstrate that the performance recovery is closely related to the original performance of the base model. For example, Llama 3.2-1B, the most recent model, can achieve the best performance and computational cost trade-off.

Hybrid KV Cache. We tested our hybrid KV Cache strategy on the Llama 2-7B model and used different batch sizes to perform inference on the Alpaca and Samsum (Gliwa et al., 2019) datasets to

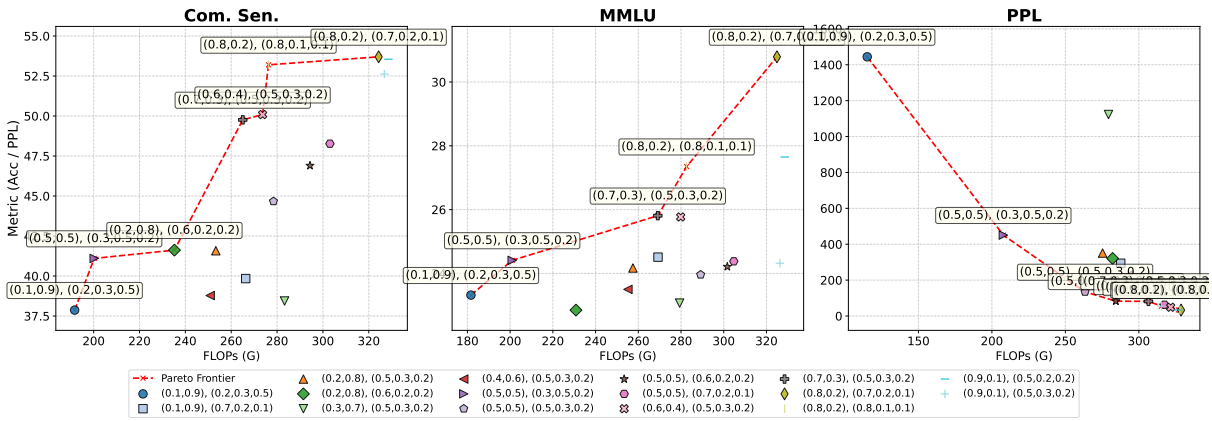


Figure 8: Ablation study on Common Sense Reasoning and MMLU tasks, and the PPL under different acceleration factor settings.

analyze the memory overhead. We only include the memory used for KV cache. The results are shown in the Figure 7, which shows that kV cache has more savings under larger batch size. Meanwhile, for tasks with longer context (Samsum dataset), the savings of ratio will be greater. On average, our hybrid key value cache strategy achieves a 38.24% storage overhead savings in KV cache storage.

Acceleration Factors. We performed an analysis study on the acceleration factors $\{\alpha_f, \alpha_l\}$ and $\{\beta_f, \beta_l, \beta_s\}$ to analyze their impact on the performance-efficiency trade-off. Figure 8 visualizes these results and plots the Pareto frontier for each task, elucidating a clear positive correlation between computational budget and model performance. Specifically, performance-oriented configurations (e.g., $\alpha = \{0.8, 0.2\}$) achieve peak accuracy and lowest perplexity at the expense of higher computational overhead, whereas efficiency-oriented settings (e.g., $\alpha = \{0.1, 0.9\}$) maximize acceleration by minimizing FLOPs, albeit with a slight trade-off in performance. The smooth transition observed in intermediate configurations confirms that the Deputy framework allows for flexible navigation of the performance-efficiency landscape, enabling tailored optimization to meet the specific latency and accuracy constraints of diverse deployment scenarios.

5.4 Ablation Study

Penalty Factor. We conducted an ablation study to investigate the impact of the penalty term λ in our custom loss function on model performance. Using the average accuracy of Common Sense Reasoning and MMLU as the primary metric, the results are presented in Table 2. The results indicate that setting λ to 5.0 achieves the best performance.

λ	0.1	0.5	1.0	2.0	5.0	10.0
Com. Sens. ↑	35.40	35.53	36.54	36.96	**38.71**	38.63
MMLU ↑	**24.69**	24.10	22.95	23.75	24.22	23.82

Table 2: Com. Sen. and MMLU performance under different λ settings.

Low-Rank Approximation Strategies. We compare two approximate methods to quantify the impact of our low-rank approximation design: **(1) Down-projection: $\mathbf{BC}_{down} \approx \mathbf{W}_{\text{pre}}$**, where $\mathbf{C}_{down} \in \mathbb{R}^{r \times d}$ first projects the hidden state to a lower dimension and B reconstructs it; **(2) Up-projection: $\mathbf{C}_{up}\mathbf{A} \approx \mathbf{W}_{\text{pre}}$**, where $\mathbf{A}$ produces a low-rank representation that is expanded back to the original dimension by $\mathbf{C}_{up} \in \mathbb{R}^{d \times r}$. The results, listed in Table 3, show that modeling $\mathbf{C}$ as a *down-projection* matrix consistently achieves lower loss and higher accuracy. This suggests that the down-projection formulation preserves the essential input features more effectively than the up-projection alternative.

Method	Best Loss ↓	Com. Sens. ↑	MMLU ↑
$\mathbf{C}_{up}$	3.23	38.41	22.77
$\mathbf{C}_{down}$	**2.87**	**38.63**	**23.82**

Table 3: The performance comparison between different low-rank approximation designs.

6 Conclusion

This paper introduces *Deputy*, a novel framework for efficient LLM inference. By integrat-

ing lightweight decision modules for attention and FFN layer within each Transformer block, Deputy dynamically routes tokens to full-parameter, low-rank, or skipped computations, achieving significant computational savings while preserving the original model's performance. Quantitative evaluations on Llama demonstrate Deputy's effectiveness: it achieves the optimal performance than baseline methods on several benchmarks with only 60% of the computational cost. For long-context scenarios, Deputy's hybrid KV cache strategy reduces storage overhead by 38%.

7 Limitations

While our approach reduces computational costs via low-rank layer execution, it does not address the quadratic complexity of attention ($O(n^2d)$), which remains a bottleneck. Future work will develop a plug-and-play module for flexible switching between inference modes. Second, although Deputy reduces FLOPs by skipping layers, the current GPU implementation sees no throughput gain because divergent paths in a batch force each layer to handle full, low-rank, and skipped tokens, adding control-flow and memory overhead that cancels the savings. Future work will explore hardware-aware optimizations (e.g., token grouping, expert sharding, load balancing) to better align Deputy with GPU architectures.

Acknowledgements

This project is based upon work supported by National Natural Science Foundation of China Grant No. U22A6001, "Pioneer" and "Leading Goose" R&D Program of Zhejiang No.2025SSYS0005, Zhejiang Provincial Natural Science Foundation of China NO. LQK26F020007. We would like to sincerely thank all our supervisors and co-authors for their hard work, valuable input, and continued support throughout this project. We also greatly appreciate the reviewers and the Area Chair for their careful reading and constructive feedback, which helped us improve both the quality and presentation of the paper.

References

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, and 1 others. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439.

Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. 2024. A survey on mixture of experts. *Preprint*, arXiv:2407.06204.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, and 1 others. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45.

Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. EE-LLM: Large-scale training and inference of early-exit large language models with 3d parallelism. In *Proceedings of International Conference on Machine Learning*.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2924–2936.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. 2020. Depth-adaptive transformer. In *International Conference on Learning Representations*.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. 2024. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 12622—-12642.

Siqi Fan, Xin Jiang, Xiang Li, Xuying Meng, Peng Han, Shuo Shang, Aixin Sun, Yequan Wang, and Zhongyuan Wang. 2024. Not all layers of llms are necessary during inference. *Preprint*, arXiv:2403.02181.

Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337.

Jun Gao, Yongqi Li, Ziqiang Cao, and Wenjie Li. 2025a. Interleaved-modal chain-of-thought. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19520–19529.

Jun Gao, Qi Lv, Zili Wang, Tianxiang Wu, Ziqiang Cao, and Wenjie Li. 2025b. Uniicl: An efficient icl framework unifying compression, selection, and generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 500–510.

Jun Gao, Qian Qiao, Tianxiang Wu, Zili Wang, Ziqiang Cao, and Wenjie Li. 2025c. Aim: Let any multimodal large language models embrace efficient in-context learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 3077–3085.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, and 5 others. 2023. A framework for few-shot language model evaluation.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Association for Computational Linguistics.

Andrew Gordon, Zornitsa Kozareva, and Melissa Roemmele. 2012. SemEval-2012 task 7: Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *The First Joint Conference on Lexical and Computational Semantics –*, pages 394–398.

Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2022. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456.

Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. 2022. Language model compression with weighted low-rank factorization. In *Proceedings of International Conference on Learning Representations*.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *Proceedings of International Conference on Learning Representations*.

Ajay Jaiswal, Bodun Hu, Lu Yin, Yeonju Ro, Shiwei Liu, Tianlong Chen, and Aditya Akella. 2024. Ffn-skipllm: A hidden gem for autoregressive decoding with adaptive feed forward skipping. *Preprint*, arXiv:2404.03865.

Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with gumbel-softmax. In *Proceedings of International Conference on Learning Representations*.

Yikun Jiang, Huanyu Wang, Lei Xie, Hanbin Zhao, Chao Zhang, Hui Qian, and John C.S. Lui. 2024. D-LLM: A token adaptive computing resource allocation strategy for large language models. In *Proceedings of The Annual Conference on Neural Information Processing Systems*.

Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. 2020. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *arXiv preprint arXiv:2009.13081*.

Bo-Kyeong Kim, Geonmin Kim, Tae-Ho Kim, Thibault Castells, Shinkook Choi, Junho Shin, and Hyoung-Kyu Song. 2024. Shortened LLaMA: A simple depth pruning for large language models. In *ICLR 2024 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

Fanrong Li, Gang Li, Xiangyu He, and Jian Cheng. 2021. Dynamic dual gating neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5330–5339.

Shiyang Li, Jianshu Chen, yelong shen, Zhiyu Chen, Xinlu Zhang, Zekun Li, Hong Wang, Jing Qian, Baolin Peng, Yi Mao, Wenhu Chen, and Xifeng Yan. 2024. Explanations from large language models make small reasoners better. In *Proceedings of Workshop on Sustainable AI*.

Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2023. LoSparse: Structured compression of large language models based on low-rank and sparse approximation. In *Proceedings of the International Conference on Machine Learning*, volume 202, pages 20336–20350.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2020. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *Preprint*, arXiv:2007.08124.

Yijin Liu, Fandong Meng, Jie Zhou, Yufeng Chen, and Jinan Xu. 2021. Faster depth-adaptive transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13424–13432.

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 International Conference on Machine Learning*, volume 202, pages 22137–22176.

Xuan Luo, Weizhi Wang, and Xifeng Yan. 2025. Adaptive layer-skipping in pre-trained llms. *Preprint*, arXiv:2503.23798.

Xiuqing Lv, Peng Zhang, Sunzhu Li, Guobing Gan, and Yueheng Sun. 2023. LightFormer: Light-weight transformer using SVD-based weight transfer and parameter sharing. In *Findings of the Association for Computational Linguistics*, pages 10323–10335.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. In *Proceedsing of Advances in Neural Information Processing Systems*, volume 36, pages 21702–21720.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *Preprint*, arXiv:1609.07843.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2381–2391.

Matthew Peroni and Dimitris Bertsimas. 2024. Skip transformers: Efficient inference through skip-routing. In *Proceedings of NeurIPS Workshop on Fine-Tuning in Modern Machine Learning: Principles and Scalability*.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. 2024. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *Preprint*, arXiv:2404.02258.

Siva Reddy, Danqi Chen, and Christopher D. Manning. 2019. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266.

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

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. 2019. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 4463–4473.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems*, volume 35, pages 17456–17472.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2023. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 13484–13508.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. 2023. Sheared llama: Accelerating language model pre-training via structured pruning. In *Proceedings of International Conference on Learning Representations*.

Guo Yang, Daniel Lo, Robert D. Mullins, and Yiren Zhao. 2023. Dynamic stashing quantization for efficient transformer training. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. 2024. Exploring post-training quantization in llms from comprehensive study to low rank compensation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19377–19385.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800.

Changhai Zhou, Shijie Han, Lining Yang, Yuhua Zhou, Xu Cheng, Yibin Wang, and Hongguang Li. 2025a. RankAdaptor: Hierarchical rank allocation for efficient fine-tuning pruned LLMs via performance model. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5796–5810.

Changhai Zhou, Qian Qiao, Yuhua Zhou, Yuxin Wu, Shichao Weng, Weizhong Zhang, and Cheng Jin. 2026a. Large language model compression with global rank and sparsity optimization. *Preprint*, arXiv:2505.03801.

Changhai Zhou, Shiyang Zhang, Yuhua Zhou, Qian Qiao, Jun Gao, Shichao Weng, Weizhong Zhang, and Cheng Jin. 2026b. Balancing fidelity and plasticity: Aligning mixed-precision fine-tuning with linguistic hierarchies. *Preprint*, arXiv:2505.03802.

Changhai Zhou, Yuhua Zhou, Shijie Han, Qian Qiao, and Hongguang Li. 2024. Qpruner: Probabilistic decision quantization for structured pruning in large language models. *Preprint*, arXiv:2412.11629.

Yuhua Zhou, Ruifeng Li, Changhai Zhou, Fei Yang, and Aimin PAN. 2025b. BSLoRA: Enhancing the parameter efficiency of loRA with intra-layer and inter-layer sharing. In *Proceedings of International Conference on Machine Learning*.

Yuhua Zhou, Changhai Zhou, Shiyang Zhang, Fei Yang, Yi Zhang, and Aimin Pan. 2026c. Lara: Layer-wise rank allocation for efficient fine-tuning of pruned large language models. *Information Processing & Management*, 63(3):104538.

Appendix

A Low-Rank Approximation

We compare three representative low-rank decomposition methods to evaluate their approximation effectiveness of the pretrained model parameters: **(1) Singular Value Decomposition (SVD)** decomposes a matrix into the product of three matrices (U, Σ, and V), where Σ contains the singular values. **(2) Randomized SVD (RSVD)** is a randomized variant of SVD that leverages random projections to accelerate computation and significantly reduce computational complexity for large matrices. **(3) Least Squares Estimation (LSE)** is an optimization-based method that minimizes the sum of squared residuals to solve for linear model parameters. For low-rank approximation, LS refines the factorized matrices by minimizing the reconstruction error and is often used in matrix completion tasks.

We evaluate the above methods by comparing their total decomposition time and the average L2 norm of the reconstructed parameters. Specifically, we conduct experiments on the Llama3.2-1B and Llama2-7B models, using each method under different rank settings (i.e. 8, 16, and 32). The results are presented in Table 4, showing that SVD achieves the lowest L2 norm, indicating the highest approximation accuracy. RSVD, on the other hand, is the fastest among the three in terms of decomposition speed. LSE strikes a balance between the two: it achieves a similar MSE to RSVD while maintaining comparable computational efficiency. Notably, LSE computes the low-rank matrix LoRA-C directly based on existing LoRA parameters, which reduces memory usage by nearly half compared to SVD and RSVD, while maintaining competitive performance and speed.

Method	Rank	Llama3.2-1B		Llama2-7B	
		Time (s)	L2 Norm.	Time (s)	L2 Norm.
SVD	8	127.2464	51.0261	1339.2912	98.2881
RSVD	8	1.7750	51.9624	12.4437	99.4124
LSE	8	4.9447	52.4378	26.9648	99.8048
SVD	16	130.3211	50.2135	1325.8284	97.3657
RSVD	16	3.9565	51.4564	17.0903	98.9753
LSE	16	6.7804	52.3236	40.4506	99.7291
SVD	32	131.8951	48.8511	1328.9537	95.8690
RSVD	32	12.4437	50.5329	32.6724	98.1833
LSE	32	12.1155	52.0943	78.6971	99.5780

Table 4: Low-rank Approximation Method Performance Comparison

B Computational Efficiency

Given a large language model with L transformer layers, a hidden size of d, a sequence length of s, a vocabulary size of V, and a batch size of B, the computational cost of each layer can be quantified in terms of floating-point operations (FLOPs). In the Attention layer, the projection of each matrix (i.e., Query (Q), Key (K), Value (V), and Output (O)) requires $2sd^2$ FLOPs, and the multi-head attention operation requires $4s^2d$ FLOPs. Consequently, the total FLOPs required for the Attention layer is given by: $C_{\text{attn}}^{\text{full}} = 8sd^2 + 4s^2d$. For the feed-forward network (FFN) layer, the Up and Gate projections require $2sd^2d_c$ FLOPs, and the Down projection requires $2sdd_c^2$ FLOPs, where d_c denotes the expanded intermediate dimension. Thus, the total FLOPs required for the FFN layer is: $C_{\text{ffn}}^{\text{full}} = 6sdd_c$. When employing Low-Rank Adaptation (LoRA), the computational costs are reduced. Specifically, the attention layer requires $C_{\text{attn}}^{\text{lora}} = 16sdr$ FLOPs, and the FFN layer requires $C_{\text{ffn}}^{\text{lora}} = 12sr(d + d_c)$ FLOPs. Therefore, the total FLOPs of the dense model with LoRA adapters are $C_{dense} = Ls(8d^2 + 4sd + 16dr + 6dd_c + 12r(d + d_c))$.

For low-rank execution, the attention layer introduces $C_{\text{attn}}^{\text{lowrank}} = 8sdr$ FLOPs, while the FFN layer requires $C_{\text{ffn}}^{\text{lowrank}} = 6sr(d + d_c)$ FLOPs. Let $\{\alpha_f, \alpha_l\}$ denote the average full and low-rank execution rate for tokens in the attention layer, $\{\beta_f, \beta_l, \beta_s\}$ denote the full/low-rank/skip execution rate in the FFN layer. Note that, according to Eq. (8), we can merge the weights of LoRA-A into LoRA-C, so that we can save α_l FLOPs of LoRA-A for attention and β_l FLOPs of LoRA-A for FFN. The total FLOPs required for the forward pass in the Deputy model can be expressed as: $C_{lorarank} = L(((1 - /\frac{\alpha_l}{2})C_{attn}^{lora} + (1 - /\frac{\beta_l}{2})C_{ffn}^{lora}) + [(\alpha_f C_{attn}^{full} + \alpha_l C_{attn}^{lowrank}) + (\beta_f C_{ffn}^{full} + \beta_l C_{ffn}^{lowrank})]) = Ls((16((1 - \frac{\alpha_l}{2})dr + 12r(1 - \frac{\beta_l}{2}(d + d_c)) + [(\alpha_f 8d^2 + \alpha_l 8dr + 4sd) + (\beta_f 6dd_c + \beta_l 6r(d + d_c))]$ in the forward pass.

For example, consider the length of input $s = 1024$, and the base model is Llama 2-7B model ($L = 32, d = 4096, d_c = 11008$) finetuned with LoRA ($r = 8$). The FLOPs of the dense model are 12924.25 GFLOPS. And, the FLOPs of the Deputy model with $\alpha = \{0.5, 0.5\}, \beta = \{0.5, 0.3, 0.2\}$ are 6748.25 GFLOPs, saving about 47.79% computation cost.

C Overhead of Decision Modules

Training overheads. The proposed decision module and LoRA-C module demonstrate remarkable parameter efficiency. Specifically, the decision module accounts for merely 0.74% of the pretrained model's parameters, while the LoRA-C module introduces an even smaller proportion at 0.13%. During training with a batch size of 2, the standard LoRA approach requires 27.97GB of GPU memory, whereas our Deputy framework increases this requirement to 37.99GB, representing an additional memory overhead of approximately 10GB.

Inference overheads. During inference, the additional decision module and LoRA-C module introduce only 7.5 GB of extra GPU memory consumption. Particularly noteworthy is that the decision module contributes a negligible computational overhead during forward propagation, adding 2.0 GFLOPs, which only takes 0.02% additional FLOPs.

D Visualization of Decision Making

D.1 Instance Analysis

To gain qualitative insight into our routing mechanism, we probe the model with the arithmetic question "*Which one is greater in math, 9.9 or 9.11?*". The model correctly replies that **9.9** $>$ **9.11**, even though its intermediate difference computation is imprecise, indicating robust relational reasoning.

Figure 9 traces the token-level execution path across the 32-layer Llama2-7B model. In the self-attention stack, layers {2, 20, 21, 27, 29} predominantly select the low-rank approximation, while all other attention layers execute in the full configuration. Within the feed-forward network (FFN), the router skips entire computations in layers {13, 21, 26, 28, 32}, and prefers low-rank execution in layers {23, 29, 30}. These patterns clearly show an *importance decay*: the deeper the layer, the more aggressively the router gravitates toward cheaper branches, validating our hypothesis that late-stage computations contribute less to the final prediction for this instance.

Overall, the visualization confirms that our adaptive router can allocate computation dynamically—preserving accuracy on essential early layers while reducing redundancy in later layers—thereby achieving efficient, instance-wise inference.

D.2 Task Decision Analysis

We utilize the trained deputy model to investigate how different layers perform across various tasks, and present the results in a proportional stacked bar chart (as shown in Figure 10). The results demonstrate that the model's decision-making patterns are quite similar for most tasks, with layers predominantly favoring the same type of execution path. For instance, attention layers at layers {3, 4, 22, 23, 28, 31} exclusively employ full-parameter computation for inference. Furthermore, layers beyond the 20th tend to adopt more shortcut execution choices, specifically favoring low-rank approximation paths in attention layers while preferring both low-rank approximation paths and skip operations in FFN layers.

E Implementation Details

Since the output of the decision module is discrete and not differentiable during training. To tackle this issue, we utilize the Gumbel-Softmax (Jang et al., 2017) reparametrization technique with Straight-Through Gradient Estimator to ensure the transformer networks can be optimized in an end-to-end fashion during training. The Gumble Softmax method samples a random noise from the gumble distribution and adds it to the softmax function, making the output of the model discrete and differentiable at the same time, which is formulated as follows:

$$\tilde{b}_{l,i} = \frac{\exp\big(\big(\log(g_l(x_l))_i + \pi_i\big)/\tau\big)}{\sum_i \exp\big(\big(\log(g_l(x_l))_i + \pi_i\big)/\tau\big)}, i \in \{0, 1, 2\} \quad . \quad (10)$$

E.1 Training Strategy

Stage 1: Adapter Distillation. We first perform knowledge distillation on our LoRA-C parameters to ensure they acquire feature extraction and transformation capabilities as substitutes for pretrained parameters. Specifically, we randomly select certain layers for low-rank approximation while maintaining full-parameter execution in other layers, keeping the routing unchanged. Through multiple randomized iterations, we ensure the parameters can adapt to different path combinations, thereby reducing optimization difficulty.

Stage 2: Router Warmup. After the LoRA-C parameters are well-trained, we proceed to jointly optimize both the LoRA-C and decision modules. This dual optimization serves two purposes: further enhancing the representational capacity of LoRA-C while training the decision module to explore

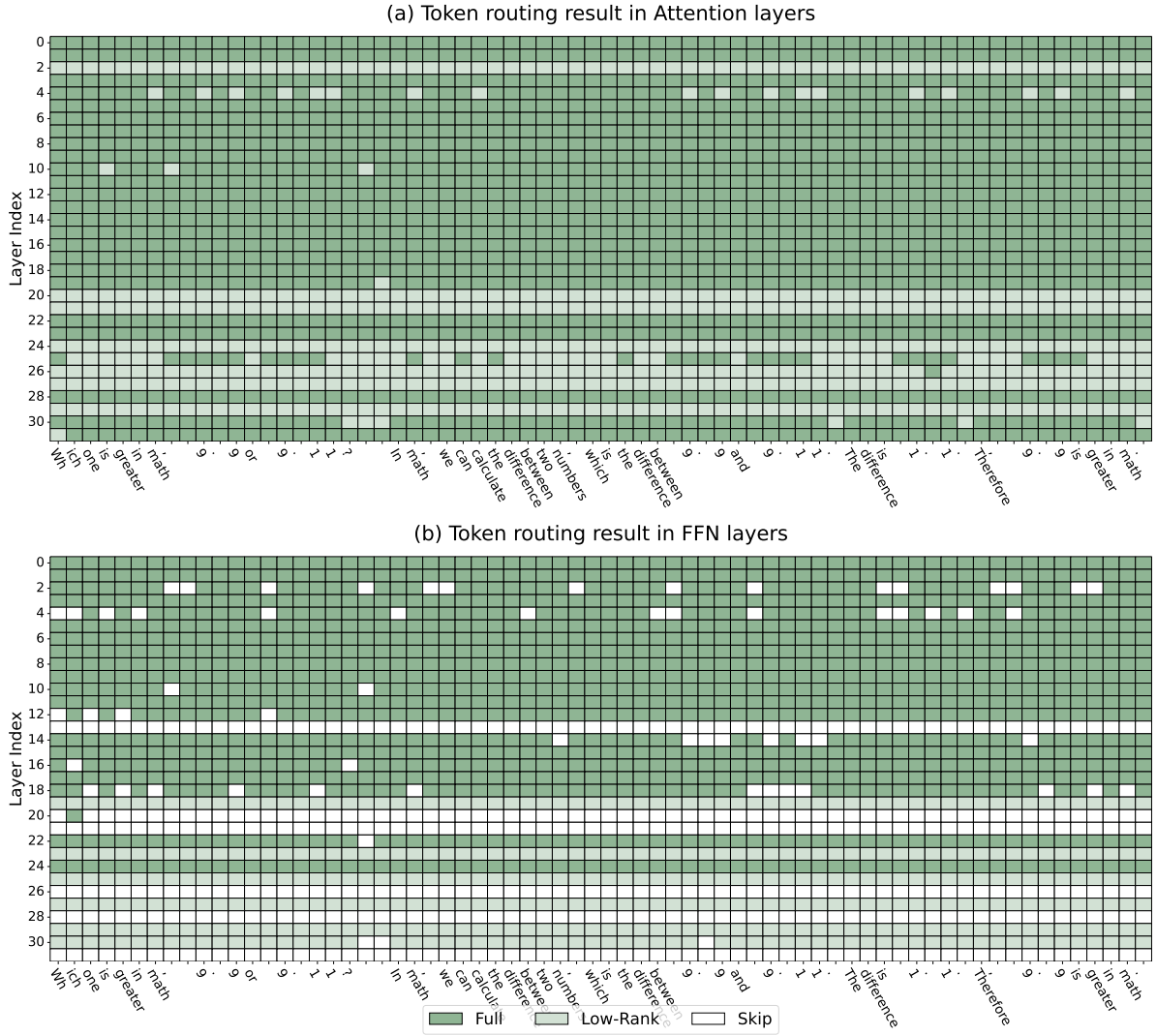


Figure 9: Token-level routing decisions for the illustrative arithmetic query. (a) Attention layers and (b) FFN layers are visualized as $32 \times |\text{tokens}|$ heatmaps, where each cell encodes the execution mode chosen by our router.

optimal path combinations for achieving peak performance.

Stage 3: Instruction Tuning. In this final stage, we incorporate the LoRA-AB parameters into training. This enables the complete model to learn diverse path selections while simultaneously optimizing the inference paths that combine full-path, LoRA-path, and skip-path configurations to maximize overall performance.

E.2 Experimental Details

Temperature. We initialize the Gumbel–Softmax temperature of each decision module at 1.0 to encourage extensive exploration during the early phase of training. The temperature is then exponentially annealed with a decay rate of 0.999 at every training step until it reaches 0.1.

Training Strategy. We employ a three-stage fine-tuning regimen: Stage 1 spans the first 400 steps, Stage 2 covers the subsequent 400 steps (up to step 800), and Stage 3 consists of all remaining steps.

Lambda Setting. The regularization coefficient λ is fixed at 0 throughout Stage 1 to facilitate the updating of LoRA-C parameters. From step 400 to step 2000, λ is gradually increased to its target value using a cosine-annealing schedule, after which it remains constant until the end of training to achieve the desired acceleration ratio.

Acceleration Factors. The acceleration factors $\alpha = (\alpha_f, \alpha_l)$ and $\beta = (\beta_f, \beta_l, \beta_s)$ are used to control the target compute budget of attention and FFN layers, respectively. To maintain similar accelera-

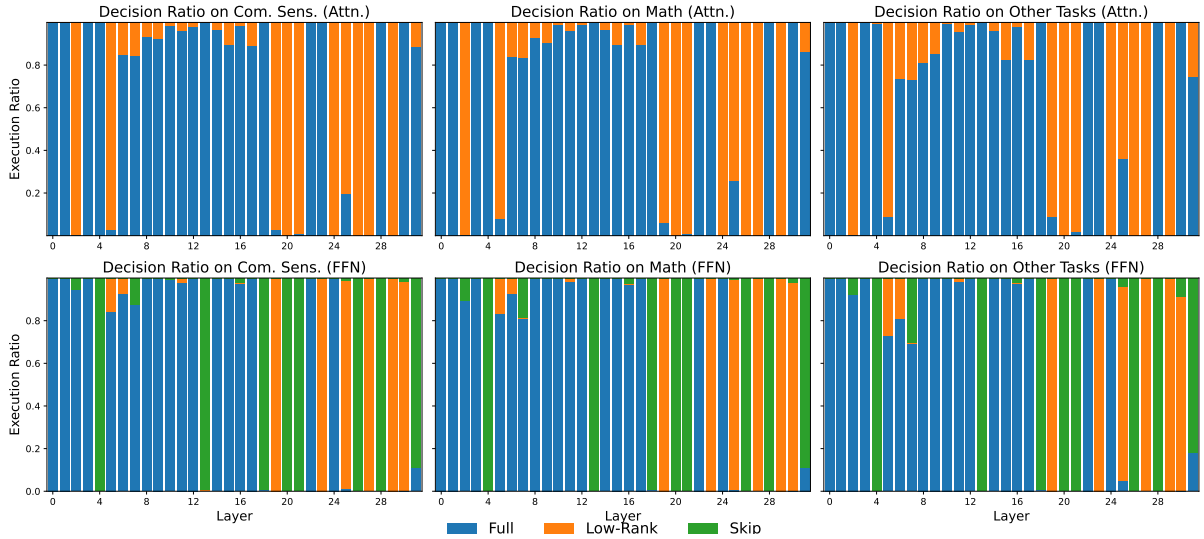


Figure 10: Task decision ratio across layers. (Up): Average decisions on attention layers; (bottom) Average decisions on ffn layers.

tion effects across different settings, we select these factors based on the FLOPs analysis in Appendix B, so that the expected computation matches the desired speedup target. We further sweep several candidate settings and use the main-experiment configuration as the default setting, since it achieves a favorable accuracy–efficiency trade-off. In particular, β_s primarily determines the activation frequency of the skip branch.

E.3 Reproduction of Baselines

Shortened Llama (Kim et al., 2024) [1] prunes less important layers to compress Llama models. We employ a pruning strategy based on Taylor metrics to prune 11 layers.

AdaInfer (Fan et al., 2024) conducts an $early-exit$ strategy to skip redundant layers. We reproduce it by utilizing Gap and Top Prob as input features and adopt an SVM as the classifier.

MoD (Raposo et al., 2024) [2] Replace some layers with a MOD layer that only processes $top-p$ tokens based on predicted scores. We implement dynamic selections on even layers, with the number of k set to 12.5% of the sentence length.

D-LLM (Jiang et al., 2024) [3] skips some unimportant tokens within a layer during inference. We set α in the loss function to 5 to balance the class loss and active loss. We apply the LoRA modules to all trainable modules. We revise the execution to support our FLOPs computation.

[1] https://github.com/Nota-NetsPresso/shortened-llm

[2] https://github.com/astramind-ai/Mixture-of-depths

[3] https://github.com/Jyk-122/D-LLM

F More Experiments

Ablation Study on Initialization of Decision Module. We investigated the impact of three different initialization methods for the decision module on model performance. Specifically, (1) ***uniform*** initializes the decision module using a Kaiming uniform initialization mechanism, ensuring an equal probability of activation for each decision. (2) ***Full-oriented*** aims to bias the model towards executing the full parameters initially. (3) ***Low-rank-oriented*** is designed to prioritize the low-rank path to achieve significant acceleration early in the training process. The results are presented in the corresponding Figure 11, indicating that the ***low-rank-oriented*** can achieve the lowest training loss at the end, which means it can update the low-rank weights more stably and quickly to fit different execution combinations.

Ablation Study on Domain Decision In Section D.2 and Figure 10, we observe that the execution pattern across layers is relatively consistent for different tasks. Moreover, for a subset of layers, the decision modules tend to output highly similar decisions for most tokens within the same input. This suggests that, when the majority of tokens already select the same branch, we can approximate token-wise routing by enforcing a unified decision for the entire layer, thereby reducing indexing and batch-

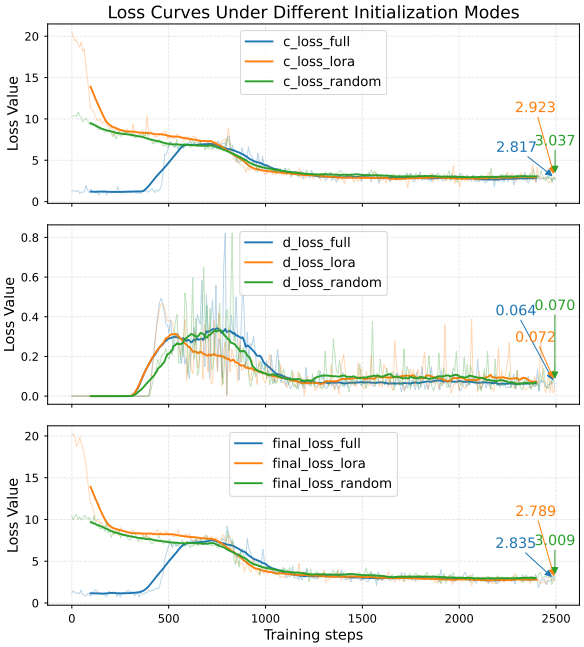


Figure 11: Training loss curve under different initialization methods.

ing overhead.

To study this effect, we introduce a *domain ratio* threshold $\rho \in \{0.6, 0.7, 0.8, 0.9\}$: if more than ρ of the tokens in a layer select the same decision, we force all tokens in that layer to follow this dominant branch. As shown in Table 5, the original adaptive token-wise routing ("Adaptive") achieves the best average accuracy, while the average perplexity improves as the domain ratio decreases (i.e., more aggressive unification), with $\rho = 0.6$ obtaining the lowest PPL. However, the adaptive setting provides a better overall trade-off between accuracy, perplexity, and computational cost, and thus we adopt adaptive token routing as our default configuration.

Domain Ratio	Avg. Acc. ↑	Avg. PPL ↓	Avg. FLOPs ↓
≥ 0.6	44.00	**118.67**	1102.31
≥ 0.7	44.21	122.38	1100.06
≥ 0.8	44.32	139.35	1098.68
≥ 0.9	44.31	144.40	1095.96
Adaptive	**44.58**	170.47	**1050.57**

Table 5: Ablation study on the domain ratio threshold for layer-wise decision unification.

G Throughput Analysis and Practical Overhead

To complement the FLOP-based analysis in Appendix C, we additionally report decoding throughput, measured in tokens per second on an A100 GPU under the standard experimental setting. The results are shown in Table 6, where we compare Deputy with the static LoRA baseline and dynamic baselines including AdaInfer, MoD, and D-LLM.

Table 6: Decoding throughput (tokens/s, higher is better) measured on an A100 GPU.

Method	**Alpaca**	**SAMSum**
LoRA (r=8)	1573.00	4598.91
AdaInfer	1010.78	2277.38
MoD	1290.79	3700.67
D-LLM	587.50	1745.43
Deputy	1010.47	2953.28

As shown in Table 6, Deputy achieves competitive throughput among dynamic methods. On ALPACA, Deputy reaches 1010.47 tokens/s, comparable to AdaInfer (1010.78) and higher than D-LLM (587.50), while remaining below MoD (1290.79) and LoRA (1573.00). On SAMSUM, Deputy achieves 2953.28 tokens/s, outperforming AdaInfer (2277.38) and D-LLM (1745.43), though still below MoD (3700.67) and LoRA (4598.91). These results indicate that Deputy maintains a competitive practical throughput while preserving the accuracy–efficiency trade-off shown in the main paper.

We also note that, although Deputy reduces FLOPs, dynamic token-wise routing introduces additional system overhead in practice. Input-dependent branch decisions lead to irregular GPU execution, such as control-flow divergence and extra gather/scatter operations, so FLOP reduction does not always translate into proportional wall-clock speedup. This is a common challenge for dynamic inference methods. Further system-level optimizations, such as token grouping or branch-wise sharding, are promising directions for future work.

H Preliminary

H.1 Token-wise Redundancy

During the inference process of Large Language Models (LLMs), not all tokens carry equal importance. Function words with less semantic content (such as articles) and punctuation marks often exhibit a certain degree of redundancy. Studies suggest that pruning these tokens has a negligible impact on the model's final performance (Jiang et al., 2024).

Furthermore, the importance of the same token varies significantly across different layers of the LLM. As shown in Figure 12, we calculated the cosine similarity between the input and output representations of various tokens across different layers of the model. It can be observed that for the first two layers and the final layer, the similarity scores for all tokens are relatively low. This indicates that these layers are responsible for actively processing and transforming token features, which is critical for maintaining model performance.

Conversely, in the deeper layers (particularly the middle-to-late layers), the cosine similarity for tokens tends to increase universally. High similarity implies minimal changes in representation, suggesting that these layers are highly redundant for most tokens. Consequently, tokens in these layers can be considered of low importance, allowing for computational skipping. Notably, for certain specific tokens (such as the period shown in the figure), the similarity remains high across almost all layers, implying that computations for such tokens can be safely skipped throughout the network.

In summary, to accelerate model inference and reduce unnecessary computational overhead, it is feasible to dynamically select important tokens for execution at each layer while skipping less significant ones.

H.2 KV Cache

The KV cache stores key (K) and value (V) representations from previous self-attention computations, enabling efficient reuse during inference. The attention score is computed using Attention $(Q, K^{(t)}_{\text{cache}}, V^{(t)}_{\text{cache}})$, where Q is the query for the current token. At each time step t, the cache is updated as:

$$K^{(t)}_{\text{cache}} = [K^{(t-1)}_{\text{cache}}; K^t],\ V^{(t)}_{\text{cache}} = [V^{(t-1)}_{\text{cache}}; V^t] \ , \quad (11)$$

where K^t and V^t are the key and value of the current token.

However, existing conditional computation methods face two critical challenges when handling KV cache: (1) During batch processing, the **inconsistent number of processed tokens** across sequences within the same batch, which is caused by varying decisions from the decision module, significantly hinders efficient batch processing and KV cache storage. (2) The KV cache of initially **skipped tokens is not preserved,** which prevents subsequent tokens from attending to them. This limitation proves particularly detrimental for tokens with long-range dependencies, ultimately compromising model performance (Luo et al., 2025).

H.3 Low-Rank Adaptation

Low-Rank Adaption (LoRA) is a popular Parameters-Efficient Fine-Tuning (PEFT) method to fine-tune the LLMs. It introduces two low-rank matrices, $\mathbf{A} \in \mathbb{R}^{r\times m}$ and $\mathbf{B} \in \mathbb{R}^{n\times r}$ to approximate the update matrix $\Delta\mathbf{W} \in \mathbb{R}^{m\times n}$ for the weight matrix of each module $\mathbf{W}$. In this approach, the original weight matrix $\mathbf{W}$ remains frozen, while only $\Delta\mathbf{W}$, represented by the product $\mathbf{AB}$, is updated. The forward computation can be expressed as:

$$f(\mathbf{x}) = (\mathbf{W} + \Delta\mathbf{W})\mathbf{x} = \mathbf{W}\mathbf{x} + \mathbf{B}\mathbf{A}\mathbf{x} \ . \quad (12)$$

Given that the rank r is shared across layers and is typically much smaller than the dimension d, the computational cost is significantly reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(2dr)$. This optimization can reduce the trainable parameters during the learning process. Typically, the matrix $\mathbf{A}$ is initialized by a Gaussian distribution with a small standard deviation, and $\mathbf{B}$ is initialized as a zero matrix. Hence, at the beginning of fine-tuning, the model behaves identically to the pre-trained model.

H.4 LLMs Architecture

In natural language processing tasks, a sequence of words is first transformed into a sequence of tokens, denoted as $X = (x^1, x^2, \ldots, x^n)$ via a tokenization process. These tokens are then processed in an autoregressive manner by the LLM, to predict the subsequent token x^{n+1}. Formally, the inference procedure of an LLM can be expressed as:

$$x^{n+1} = f_{\text{head}} \circ f_L \circ f_{L-1} \circ \cdots \circ f_1 \circ f_{\text{embed}}(X) \ , \quad (13)$$

where L represents the total number of transformer layers, f_{embed} denotes the word embedding function, and f_{head} corresponds to the classification head.

Transformer Layer. Each transformer layer in a decoder-only LLM comprises a Multi-Head self-Attention (MHA) mechanism followed by a Feed-Forward Network (FFN). The computation for the l-th transformer layer can be defined as:

$$h^n_l = \text{MHA}(x^n_l) + x^n_l \ , \quad (14)$$

$$x^n_{l+1} = \text{FFN}(h^n_l) + h^n_l \ , \quad (15)$$

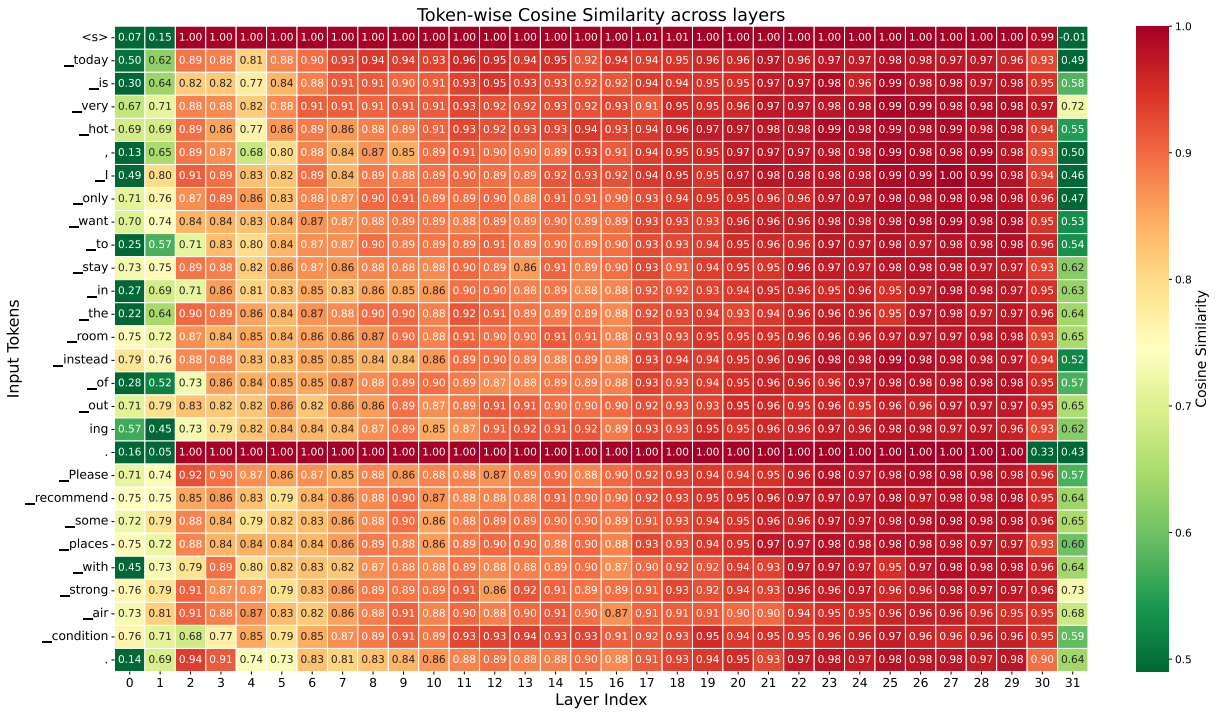


Figure 12: Token similarity across layers.

where x_l^n represents the embedding of the n-th token at the l-th transformer layer, and h_l^n denotes the output of the MHA mechanism.

Self-Attention Mechanism. The Self-Attention (SA) mechanism enables each token in a sequence to selectively attend to other tokens, thereby capturing rich contextual dependencies. Each token generates a query (Q), a key (K), and a value (V) vector from $\mathbf{W}_Q, \mathbf{W}_K$, and$\mathbf{W}_V$. The attention scores are calculated by the scaled dot-product formulation:

$$\mathrm{SA}(X) = \mathrm{Softmax}\left(\frac{x_{l,i}^n \circ \mathbf{W}_Q \circ (X \circ \mathbf{W}_K)^T}{\sqrt{d}} + M\right) \circ X \circ \mathbf{W}_V \ , \quad (16)$$

where $X = (x_{l,i}^1, x_{l,i}^2, \ldots, x_{l,i}^n)$ represents the input feature sequence, d is the dimensionality of each head, M is the causal attention mask, and $(\cdot)^T$ denotes the transpose operation.

To capture features from multiple representation subspaces, the multi-head self-attention (MHA) mechanism aggregates the outputs computed over h heads:

$$\mathrm{MHA}(x_l^n) = \mathrm{Concat}\left(\mathrm{SA}(x_{l,1}^n), \ldots, \mathrm{SA}(x_{l,h}^n)\right) \circ \mathbf{W}_O \ , \quad (17)$$

where $\mathbf{W}_O$ refers to the output matrix for the MHA. The MHA mechanism essentially concatenates the outputs of self-attention operations applied to the token embeddings divided into h heads, $\{x_{l,i}^n\}_{i=1}^h$.

Feed-Forward Network. The Feed-Forward Network (FFN) consists of multiple linear layers interspersed with activation functions following the MHA layer. It is formally defined as:

$$\mathrm{FFN}(x_{l,i}^n) = \mathbf{W}_{\text{down}} \circ \left(\mathrm{act}(\mathbf{W}_{\text{gate}} \circ x_{l,i}^n) * (\mathbf{W}_{\text{up}} \circ x_{l,i}^n)\right) \ , \quad (18)$$

where $\mathbf{W}_{\text{down}}$, $\mathbf{W}_{\text{gate}}$, and $\mathbf{W}_{\text{up}}$ are learnable parameter matrices, and $\mathrm{act}(\cdot)$ represents an activation function.

I LLM versions

We provide the Hugging Face link of LLMs used in the experiment: Llama 2-7B [4], Llama 3-8B [5], Llama 3.2-1B [6], Llama 1-13B [7].

J Software and hardware configuration

Our experiments were conducted on a high-performance computing server equipped with an Intel® Xeon® Platinum 8358P processor (128 cores at 2.60 GHz), 1 TB of RAM for memory-intensive operations, and NVIDIA® A100 Tensor

[4] https://huggingface.co/meta-llama/Llama-2-7b

[5] https://huggingface.co/meta-llama/Meta-Llama-3.1-8B

[6] https://huggingface.co/meta-llama/Llama-3.2-1B

[7] https://huggingface.co/yahma/llama-13b-hf

Core GPUs with 80 GB HBM2 memory. The software stack comprised PyTorch 2.1.2 with CUDA 12.3 acceleration, complemented by the Transformers library (v4.41.0) and PEFT (v0.11.1) for parameter-efficient fine-tuning, and the thop library [8] for FLOPs measurement, all running on an Ubuntu 20.04 LTS operating system.

[8] https://github.com/Lyken17/pytorch-OpCounter