Continual Fine-Tuning of Large Language Models via Program Memory

Anonymous ACL submission

Abstract

Parameter-Efficient Fine-Tuning (PEFT), particularly Low-Rank Adaptation (LoRA), has become a standard approach for adapting Large Language Models (LLMs) under limited compute. However, in continual settings where models are updated sequentially with small datasets, conventional LoRA updates struggle to balance rapid adaptation and knowledge retention. Existing methods typically treat the low-rank space as a homogeneous update region, lacking mechanisms to regulate how short-term updates are consolidated over time. We propose a continual LoRA framework with **Pro**gram memory, inspired by **C**omplementary **L**earning Systems in neuroscience. Our approach, dubbed **ProCL**, organizes LoRA adapters into structured program memory slots that are dynamically retrieved through input-conditioned attention. This enables rapid and localized adaptation, encouraging similar inputs to reuse shared adapter regions while reserving unused capacity for future data. The slots are then combined with the underlying adapter, which maintains a distributed representation that gradually accumulates knowledge across tasks to balance plasticity and stability. Our method operates entirely within the LoRA parameterization and incurs no additional inference cost. Experiments on diverse benchmarks demonstrate improved retention and reduced catastrophic forgetting over other continual LoRA strategies.

1 Introduction

Large language models (LLMs) have achieved strong performance across a wide range of tasks [1, 2]. However, adapting them to evolving real-world scenarios remains challenging, as full fine-tuning is often computationally prohibitive. Parameter-Efficient Fine-Tuning (PEFT), particularly Low-Rank Adaptation (LoRA) [9], addresses this by restricting updates to low-rank subspaces while freezing the backbone, enabling efficient adaptation with minimal additional parameters.

In real-world deployment, model adaptation is rarely a one-time process; LLMs must continually incorporate new knowledge from sequential, often small-scale datasets. For instance, a clinical model fine-tuned for diagnosis may later be updated for treatment recommendation, improving new capabilities while degrading earlier ones. This highlights the stability–plasticity dilemma, where models must learn new tasks without forgetting prior knowledge [16]. Although LoRA constrains updates to low-rank subspaces, sequential fine-tuning can still introduce interference, leading to forgetting and reduced retention [5, 6].

Existing approaches for continual adaptation of LLMs often add adapters or task-specific modules to isolate task-relevant knowledge [18, 21]. While effective at reducing interference, these methods increase architectural complexity, memory usage, and inference latency, making them cumbersome for long task sequences. Alternatively, regularization-based techniques constrain updates to orthogonal subspaces to reduce interference across tasks [22, 25]. While effective at mitigating forgetting, the model's effective plasticity is reduced, limiting its ability to learn new information on later tasks. Recent strategies improve retention by refining low-rank factors with wavelet kernels [6]. However, they still modify the shared low-rank subspace globally, without explicitly controlling how new knowledge is written, partitioned, or consolidated. As a result, LoRA updates remain vulnerable to interference, leading to degraded performance on previously learned tasks. Empirically, prior work has largely focused on simple text classification

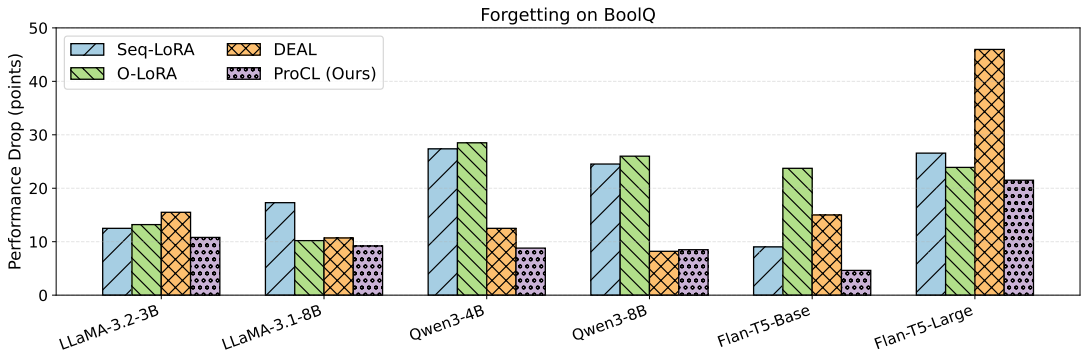

Figure 1: Catastrophic forgetting in QA tasks (lower is better). BoolQ's accuracy drops after fine-tuning on SQuAD and AdversarialQA. While all methods suffer from forgetting across LLM backbones, our approach consistently exhibits the smallest or near-smallest performance degradation.

tasks, which neither reflect the generative capabilities of LLMs nor fully capture catastrophic forgetting in more complex settings. For example, in question answering, we consistently observe substantial performance degradation of prior methods after only a few tasks (see Fig. 1). These limitations raise a key question: *can LoRA be equipped with a principled mechanism for efficient and robust continual learning?*

To address this challenge, we propose a memory-structured adaptation mechanism that organizes how knowledge is written into LoRA adapters during continual learning. Drawing inspiration from Complementary Learning Systems (CLS) in neuroscience [15, 17], we model continual adaptation as the interaction between rapid episodic indexing and slower distributed representations. In our framework, dubbed **ProCL**, low-rank adapters are treated as program memory [12], with the adapter weights partitioned into multiple slots or programs. An input-conditioned attention mechanism dynamically retrieves and composes these programs, allowing similar inputs to activate and update shared adapter regions while routing unrelated inputs to different parts of the parameter space. This helps localize parameter updates, reducing interference between different data regions.

At the same time, the original LoRA weight provides a stable distributed representation that reflects previously learned knowledge. During training, this weight is combined with the program-composed weights to produce the executing adapter, serving as a shared baseline that stabilizes optimization while the programs introduce input-dependent adaptations. These adaptive compositions are periodically integrated into a persistent adapter through a consolidation step, which accumulates routing-based updates over time. During inference, this learned adapter is used directly without program routing. By structuring adaptation in this way, ProCL enables selective plasticity during training while producing a compact, static model for deployment, mitigating catastrophic forgetting without introducing additional inference overhead. Experiments on text classification and question answering continual learning benchmarks demonstrate that this memory-structured adaptation mechanism significantly improves knowledge retention, cross-task generalization, and efficiency compared to existing continual LoRA strategies.

2 Related Work

Continual learning (CL) aims to adapt models to new tasks sequentially without catastrophic forgetting [16]. Applying CL to LLMs is challenging due to their massive parameter spaces, high computational costs, and sensitivity to interference across tasks. Existing approaches for continual LLM fine-tuning can be grouped into two main directions.

Adapter-Modular and Progressive Architectures. Adapter-based and parameter-efficient tuning approaches [8, 9] freeze the backbone and introduce task modules. Progressive or modular extensions, including LoraHub [10] and MOLE [24], dynamically compose or gate multiple LoRA components for cross-task reuse. Although these designs reduce catastrophic forgetting, they increase memory footprint and complexity as tasks accumulate, assuming task identity is

known at inference time.

Regularization-based Methods. Regularization and subspace-based approaches aim to mitigate interference across tasks. Classical methods such as EWC [11] constrain updates based on parameter importance, while more recent techniques focus more on LoRA-specific constraints. For example, O-LoRA [22] enforces orthogonality between low-rank updates, N-LoRA [25] prevents parameter collisions via disjoint subspaces, OLieRA [3] preserves model geometry through Lie group constraints, and DEAL [6] uses wavelet kernels to retain historical knowledge. Despite these advances, these methods lack explicit mechanisms to isolate task-specific updates, and accumulated constraints gradually exhaust the effective parameter space, limiting plasticity over long task sequences. In contrast, inspired by CLS [15, 17], our approach introduces program-memory slots to structure how task knowledge is written and consolidated, enabling selective retention and adaptability.

Memory in CL. Memory-based methods mitigate forgetting by storing past information, e.g., via experience replay [19] and gradient episodic memory [14, 4, 7]. Another line explores program memory, where reusable weight components are stored and composed across tasks [12, 13]. Related to this, mixture-of-experts (MoE, [20]) models route inputs to specialized experts, but combine *outputs*, whereas program memory methods compose *parameters* to form task-adaptive models. Prior approaches rely on external buffers or modules and are not tailored for parameter-efficient LLM adaptation. In contrast, we integrate program memory directly into LoRA, enabling structured storage, routing, and consolidation within the adapter without external data buffers.

3 Background

LoRA-Based Fine-tuning. LoRA (Low-Rank Adaptation) enhances LLMs by introducing low-rank matrices to parameter updates during fine-tuning. Given a pre-trained weight matrix $W_0 \in \mathbb{R}^{D_1 \times D_2}$, LoRA decomposes the weight update ΔW into two smaller matrices $A \in \mathbb{R}^{D_1 \times R}$ and $B \in \mathbb{R}^{D_2 \times R}$, where $R \ll \min\{D_1, D_2\}$. The update is expressed as:

$$\Delta W = AB^\top \tag{1}$$

This reduces the number of trainable parameters from $D_1 \cdot D_2$ to $D_1 \cdot R + D_2 \cdot R$, enabling efficient adaptation to new tasks with significantly lower computational cost than full fine-tuning. In practice, the adapted linear transformation is applied as $y = x(W_0 + \Delta W)^\top$. Note that the factorization $\Delta W = AB^\top$ is equivalent to the commonly used form BA up to transposition conventions, and does not affect the forward computation. In our method, we operate on a generic low-rank factor (e.g., A or $B^\top$), which we denote as W for simplicity, without loss of generality.

Continual Learning with LoRA. Formally, in the general continual learning setting, consider a sequence of tasks $\mathcal{T}_1, \ldots, \mathcal{T}_t$ with corresponding datasets $\mathcal{D}_1, \ldots, \mathcal{D}_t$. The model is presented with the task datasets sequentially, one at a time. The standard objective is to learn a model that performs well across all seen tasks. At task t, the objective can be expressed as:

$$\min_{\theta_1,\ldots,\theta_t} \frac{1}{t} \sum_{i=1}^{t} \mathcal{L}(\theta_i, \mathcal{D}_i), \tag{2}$$

where θ_i denotes the model parameters after learning task $\mathcal{T}_i$, and $\mathcal{L}$ is the task-specific loss.

In the context of LoRA-based fine-tuning, the model parameters θ_i at task i are decomposed into a fixed pre-trained weight W_0 and low-rank adaptation matrices A_i, B_i for each task. The continual learning objective then becomes:

$$\min_{A_1,B_1,\ldots,A_t,B_t} \frac{1}{t} \sum_{i=1}^{t} \mathcal{L}\big(W_0 + A_i B_i^\top, \mathcal{D}_i\big), \tag{3}$$

Here, the goal is to find low-rank matrices A_i and B_i that allow the model to efficiently adapt to each new task while preserving performance on all previously learned tasks, naturally reflecting the stability–plasticity trade-off central to continual learning. For example, a simple method (Sequential LoRA) shares the same adapters across tasks, i.e., they are initialized from the previously learned matrices: $A_i \leftarrow A_{i-1}$ and $B_i \leftarrow B_{i-1}$, to better accumulate knowledge from past tasks.

4 Method

We propose **ProCL**, a **Pro**gram Memory framework for LoRA inspired by **C**omplementary **L**earning Systems. ProCL addresses catastrophic forgetting in continual fine-tuning by replacing a single monolithic LoRA adapter with a library of small specialist weight matrices, termed programs. The input is compressed into a summary vector and passed through a lightweight encoder to produce task queries, which are compared against learnable program keys via multi-head attention to produce soft routing weights. These weights determine how programs are combined to form the final adapter. As training progresses, different tasks activate different programs, reducing gradient interaction; we show formally that such interaction is bounded and approaches zero as routing specialization sharpens. The final adapter combines a frozen copy of the original adapter with the dynamically routed program blend, preventing overwriting of prior knowledge. Analogous to biological memory consolidation, a moving average step folds the executed adapter back into the persistent weight, gradually integrating new knowledge into a stable representation. The process is depicted in Fig. 2.

4.1 Program Memory as Structured Low-Rank Adapters

Consider a low-rank adapter weight:

$$W \in \mathbb{R}^{R \times D}, \tag{4}$$

where R is the LoRA rank and D the other dimension. Instead of treating W as a monolithic matrix, we partition it into N programs:

$$W = \left[W^{(1)}; \ldots; W^{(N)} \right], \tag{5}$$

where $W^{(n)} \in \mathbb{R}^{r \times D}$, $r = R/N$. Here, $n = 1, \ldots, N$ with N as a hyperparameter that divides R. Each program $W^{(n)}$ serves as a memory slot that stores data-relevant adaptations. We use the terms *program* and *memory slot* interchangeably throughout the paper. Importantly, the adapter weight W remains the only persistent parameter saved by LoRA frameworks. Program memory is not a separate parameter set; it is an execution-time abstraction over the same adapter tensor W, ensuring full compatibility with existing LoRA implementations.

4.2 Input-Conditioned Program Selection

During training for a task, given a training sample: $x \in \mathbb{R}^{T \times D}$, where T and D are the number of tokens and embedding size, respectively, we compute an input representation via mean pooling:

$$z = \frac{1}{T} \sum_{t=1}^{T} x_t. \tag{6}$$

A lightweight task encoder produces task queries:

$$Q = f_{\text{enc}}(z) \in \mathbb{R}^{N \times k_d}. \tag{7}$$

We use the task queries to perform attention over the programs. Attending to slot-based memory resembles the episodic retrieval mechanism in CLS [23]. Since each attention operation selects a program $W^{(n)} \in \mathbb{R}^{r \times D}$, we employ $R/r = N$ parallel attention heads to retrieve N programs, which are then concatenated to form the executed adapter. Each head $h \in [1, N]$ maintains a set of learnable key vectors $K_h = \{K_h^1, \ldots, K_h^N\} \in \mathbb{R}^{N \times k_d}$ associated with the set of programs in Eq. 5. The attention weights for head h over memory slots are computed as:

$$\alpha_h = \text{Softmax}\left(Q[h] K_h^\top \right), \tag{8}$$

where $\alpha_h \in \mathbb{R}^N$. To improve stability during training, we aggregate attention across the batch:

$$\bar{\alpha}_h = \frac{1}{B} \sum_{b=1}^{B} \alpha_h^{(b)}. \tag{9}$$

This yields a batch-level routing distribution that remains input-adaptive across batches while reducing variance from per-sample fluctuations. Since batches in continual fine-tuning predominantly contain samples from the same task, this aggregation preserves task-conditioned routing behavior where different tasks still activate different programs. The effective program for head h is obtained via a soft composition over memory slots:

$$\hat{W}_h = \sum_{n=1}^{N} \bar{\alpha}_{h,n} \cdot \sigma(s_n) \cdot W^{(n)}, \tag{10}$$

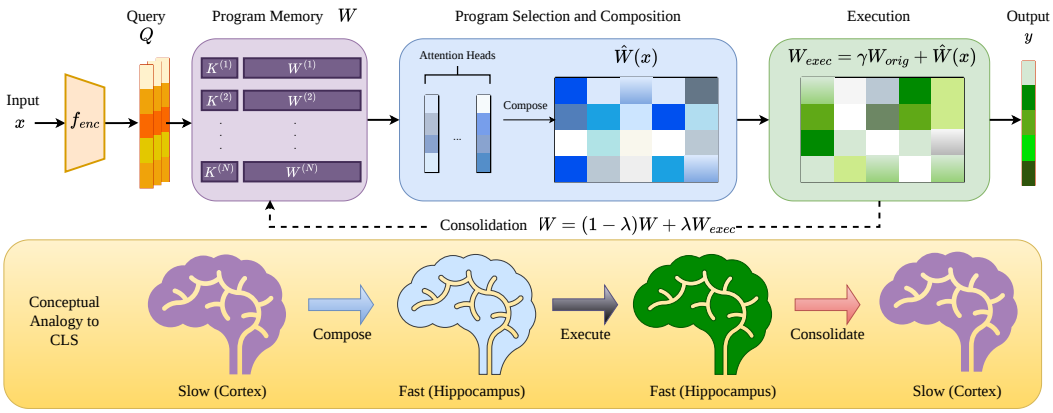

Figure 2: Overview of ProCL. Given an input x, the model queries relevant parameter slots in the Program Memory W. Through attention, selected programs are composed into an input-conditioned weight $\hat{W}(x)$, which is combined with the base weight for execution to produce the output y. The resulting execution weight is then distilled back into the memory via consolidation. This process is conceptually inspired by Complementary Learning Systems (CLS).

where s_n is a learnable scaling parameter and $\sigma(\cdot)$ ensures stable gating. We emphasize that this operation performs a *soft selection* over programs, allowing multiple memory slots to contribute to the composed adapter. Then, all head-specific programs are row-stacked:

$$\hat{W} = \left[\hat{W}_1; \ldots; \hat{W}_N\right]. \tag{11}$$

Theoretically, our proposed mechanism reduces cross-task gradient interaction by routing gradient updates through input-conditioned programs. The theoretical result is summarized below:

Remark 1: Program Routing Bounds Cross-Task Gradient Interaction. Let $J(t,t') := |\langle G^t, G^{t'}\rangle_F|$ denote the magnitude of cross-task gradient interaction. ProCL satisfies $J_{\text{ProCL}}(t,t') \leq (\max_h \beta_h)\, S(t,t')$, where $\beta_h \in [0,1]$ is the routing-induced gating coefficient and $S(t,t') := \sum_h |\langle \tilde{g}_{h,t}, \tilde{g}_{h,t'}\rangle_F|$. Under routing specialization (A1), $\max_h \beta_h \to 0$ as task-specific routing distributions become disjoint, driving $J_{\text{ProCL}}(t,t') \to 0$.

Proof. See Appendix A.1. □

4.3 Stable–Adaptive Weight Composition and Consolidation

The composed programs are aggregated with the original adapter to form the executing adapter:

$$W_{\text{exec}} = \gamma W_{\text{orig}} + \hat{W}(x), \tag{12}$$

where W_{orig} denotes the initial LoRA adapter weight before task-specific training, $\hat{W}(x)$ is the program-composed weight produced by the input-conditioned routing mechanism (Eq. 11), and γ is a learnable scaling parameter controlling the contribution of the original adapter. γ is initialized per-layer as $\gamma_0 = \sigma(-\log \text{RMS}(W))$, where $\text{RMS}(W)$ denotes the root-mean-square of the layer weights. This norm-adaptive scheme assigns smaller γ to large-norm layers and larger γ to small-norm layers, balancing their contributions, constrained to $(0,1)$ via the sigmoid function σ. The executing adapter W_{exec} is used in the same manner as standard LoRA to produce the output y. In this formulation, W_{orig} acts as a frozen stability anchor that preserves previously acquired knowledge, while $\hat{W}$ provides fast, input-specific adaptation through the selected programs. The mixing coefficient γ controls this stability–plasticity trade-off, with smaller values emphasizing adaptation and larger values emphasizing retention. During training, gradients flow only through the program-composed adapter $\hat{W}$, while W_{orig} remains fixed. This allows ProCL to learn localized updates relative to a stable baseline, reducing destructive interference with prior knowledge. However, dynamic execution alone introduces routing overhead and does not permanently store newly acquired knowledge. We therefore introduce a consolidation step:

$$W \leftarrow (1-\lambda)W + \lambda W_{\text{exec}}, \tag{13}$$

where $\lambda \in [0, 1]$ controls the consolidation rate. This update compresses input-conditioned program compositions into the persistent weight W. Specifically, $W_{\text{exec}} = \gamma W_{\text{orig}} + \hat{W}(x)$ captures fast, input-dependent adaptations, while W accumulates their average effect over time. This mirrors CLS theory: routing enables rapid, context-specific updates (fast memory), and consolidation distills them into a stable parametric representation (slow memory). As a result, W evolves via both gradient updates and consolidation, acting as a compact summary of program-memory-based learning.

Parameter Scope: At the beginning of each continual round, the routing-only parameters $(f_{\text{enc}}, K_h, s_n, \gamma)$ are reinitialized and trained only to guide consolidation; they are discarded after each round, while only the consolidated adapter W is carried forward and used at inference.

Remark 2: Consolidation Approximates the Expected Executing Adapter. Let $\Delta(x) := \hat{W}(x)$ denote the routed program composition, and define

$$W^* := \gamma W_{\text{orig}} + \mathbb{E}_{x \sim \mathcal{D}}[\Delta(x)].$$

Under late-training stationarity of the routing map and bounded $\Delta(x)$, the consolidation update drives W toward W^* in expectation within each task.

Proof. See Appendix A.2. □

Efficient Inference. At inference time, we discard the routing mechanism and directly use the consolidated adapter W (i.e., $W_{\text{exec}} = W$). The consolidation step integrates the executing adapters encountered during training into a single persistent parameter, allowing W to approximate the average effect of input-conditioned program composition (Remark 2). This eliminates the need for program selection and storage for W_{orig} at test time, while preserving the benefits of program-based learning.

5 Experimental Results

5.1 Experimental Setup

Benchmarks. The benchmark consists of two continual learning settings. First, we construct a *question answering* (QA) setting with three QA datasets (BoolQ→SQuAD→AdversarialQA) presented sequentially, each sampled to 5,000 training and 1,000 test examples. They are ordered to maximise the distribution shift, increasing difficulty. Since each task requires the model to generate an answer string matched against a ground-truth response, we report the mean per-CL-round accuracy, averaged first across tasks within each continual learning round and then across all rounds. Second, following [6], we adopt a *text classification* (TC) setting with three curricula of increasing length: 3-task, 4-task, and 15-task. For this setting, we report Average Accuracy (AA), the mean classification accuracy across tasks per continual learning round averaged over all rounds, and ROUGE-1 (R-1), the unigram F1 overlap between model outputs and ground-truth labels, computed in the same manner.

Baselines. We evaluate ProCL against modern LoRA CL baselines with: Sequential LoRA (Seq-LoRA), which initialises adapters from the previously learned matrices; O-LoRA [22], which enforces orthogonality between task adapters; and DEAL [6], the current state-of-the-art continual learning method for LoRA. In QA tasks, we also include classical continual learning methods adapted to LoRA parameters, including EWC [11] and Replay [19]. For TC tasks, prior work has shown that modern CL methods consistently outperform classical approaches [6]; therefore, we omit the classical baselines to reduce computational cost.

Implementation details. We experiment with three LLM families spanning a range of scales: Llama3 (3B, 8B), Qwen3 (4B, 8B), and Flan-T5 (Base, Large) (see Appendix Table 4). Qwen is omitted for TC because the codebase does not support it in this setting. All baselines were re-run under our unified training and evaluation pipeline based on DEAL's official codebase, with the same LoRA rank and common hyperparameters across methods to ensure fair comparison. The experiments were conducted on either a single NVIDIA V100 GPU with 40GB memory or a single NVIDIA H100 GPU with 80GB memory. Results are averaged over 3 runs. Statistical ties are declared using Cohen's $d < 0.5$ (small effect size). Full implementation details, including training configurations, hyperparameters, and datasets, are provided in Appendix B.1. For ProCL, if not

LLM	Seq-LoRA	EWC	Replay	O-LoRA	DEAL	ProCL
Llama-3.2-3B	67.7 ± 0.9	68.1 ± 1.2	67.6 ± 1.1	68.1 ± 0.3	66.4 ± 1.6	**69.8 ± 0.6**
Llama-3.1-8B	64.5 ± 1.4	65.3 ± 1.1	62.8 ± 1.3	**72.4 ± 0.4**	**72.1 ± 2.1**	**73.2 ± 0.9**
Qwen3-4B	52.5 ± 1.9	54.5 ± 0.6	54.8 ± 1.9	56.6 ± 0.2	68.2 ± 0.7	**72.5 ± 0.2**
Qwen3-8B	59.1 ± 1.7	58.7 ± 0.8	58.8 ± 1.5	58.6 ± 1.2	**72.9 ± 1.5**	**72.8 ± 2.4**
Flan-T5-Base	61.3 ± 2.2	60.9 ± 1.1	60.6 ± 1.3	54.2 ± 0.7	61.4 ± 1.9	**62.8 ± 1.0**
Flan-T5-Large	**62.0 ± 1.7**	62.6 ± 0.9	**62.8 ± 1.2**	61.6 ± 1.0	55.9 ± 8.7	**63.4 ± 1.7**
Average	61.2 ± 1.6	61.7 ± 1.0	61.2 ± 1.4	61.9 ± 1.6	66.2 ± 2.8	**69.1 ± 1.1**

Table 1: Final average accuracy across continual learning rounds on QA benchmarks. Bold denotes the best method, or those within a small effect size of it (Cohen's $d < 0.5$).

stated otherwise, the default hyperparameters are $N = 4$, $k_d = 16$, and $\lambda = 0.9$ (see Sec. 5.4).

5.2 Benchmarking Results

QA Benchmarks. Table 1 demonstrates that ProCL achieves the best overall performance with an average accuracy of 69.1, outperforming all baselines by a clear margin (e.g., +2.9 over DEAL and +7.9 over Seq-LoRA). While DEAL is competitive on some larger models, its performance is inconsistent across architectures. In contrast, ProCL delivers stable gains across all models, with particularly strong improvements in more challenging settings such as Qwen3-4B. These results highlight the robustness and effectiveness of ProCL in balancing stability and plasticity.

Text Classification Benchmarks. Table 2 shows that ProCL consistently achieves the best or second-best results across curricula, model scales, and both AA and R-1 metrics. Its advantage is more evident on longer task sequences: on the 15-task curriculum, ProCL improves AA over the strongest baseline, DEAL, by approximately 0.9–1.3% across models. The gains are also clear in the 4-task setting, where ProCL outperforms DEAL by 2.7% on Flan-T5-Large (81.0 vs. 78.3) and 1.0% on Flan-T5-Base (68.2 vs. 67.2). Even in the shorter 3-task setting, ProCL maintains a consistent advantage, suggesting stronger early-stage retention. Across architectures, Flan-T5 generally achieves higher absolute accuracy than LLaMA, likely because its encoder-decoder design is better aligned with classification-style objectives; nevertheless, ProCL remains effective on both families.

5.3 Ablation Study

In Table 3, we analyze the contribution of each component of ProCL across multiple backbone models on the QA benchmark. Removing the original adapter weight W_{orig} leads to substantial performance degradation on LLaMA-3.2-3B (69.8 → 65.9) and Qwen3-4B (72.5 → 67.1), highlighting the importance of maintaining a stable reference representation during continual adaptation. On Flan-T5-Base, disabling consolidation causes the largest drop (62.8 → 60.1), indicating that transferring the dynamically composed execution weights into a persistent adapter is particularly important for encoder–decoder architectures. Replacing learned routing with random or uniform routing also consistently degrades performance across models, confirming that input-conditioned program selection is essential for reducing interference and improving retention.

5.4 Model Analysis

Hyperparameter Sensitivity. Appendix Figure 3 (left) reports the sensitivity of ProCL to the number of programs (N), key dimension (k_d), and consolidation strength (λ) on the QA benchmark using LLaMA-3.2-3B as the representative model. The default configuration, $N = 4$, $k_d = 16$, and $\lambda = 0.9$, achieves the best performance of 69.78 ± 0.60. Using only a single program significantly reduces performance to 62.87 ± 3.18, while increasing the memory size to $N = 16$ also degrades accuracy to 66.29 ± 1.70, suggesting that moderate program capacity is preferable. Similarly, $k_d = 16$ outperforms both smaller and larger key dimensions, indicating a balance between expressiveness and stability. For consolidation, large but non-extreme values of λ remain consistently effective, whereas full replacement updates ($\lambda = 1$) noticeably reduce performance to 67.09 ± 0.84. Overall, ProCL is relatively robust to variations in k_d and λ, with the best results

Method	LLM	3-Task (TC)		4-Task (Standard)		15-Task (Large)		Avg
		AA	R-1	AA	R-1	AA	R-1	(AA)
Seq-LoRA		60.8 ± 0.2	62.1 ± 0.5	55.3 ± 0.3	57.0 ± 0.6	49.6 ± 0.2	52.4 ± 0.4	55.2 ± 0.3
O-LoRA	LLaMA-3.1	59.7 ± 0.5	61.0 ± 0.3	53.4 ± 0.4	55.2 ± 0.7	48.9 ± 0.2	**52.8 ± 0.5**	54.0 ± 0.4
DEAL	8B	61.5 ± 0.3	**62.9 ± 0.6**	56.4 ± 0.2	58.2 ± 0.4	50.8 ± 0.5	52.6 ± 0.3	56.2 ± 0.3
ProCL		**62.6 ± 0.2**	**62.5 ± 0.7**	**57.9 ± 0.3**	**59.6 ± 0.5**	**52.1 ± 0.1**	**53.7 ± 0.4**	**57.5 ± 0.2**
Seq-LoRA		56.1 ± 0.6	57.8 ± 0.4	50.4 ± 0.2	52.9 ± 0.5	45.2 ± 0.3	48.0 ± 0.6	50.6 ± 0.3
O-LoRA	LLaMA-3.2	54.8 ± 0.5	56.5 ± 0.3	48.3 ± 0.4	50.6 ± 0.2	44.1 ± 0.7	48.5 ± 0.4	49.1 ± 0.5
DEAL	3B	56.9 ± 0.2	**58.6 ± 0.5**	51.3 ± 0.6	53.8 ± 0.3	46.0 ± 0.1	48.2 ± 0.6	51.4 ± 0.2
ProCL		**58.0 ± 0.3**	58.2 ± 0.6	**52.9 ± 0.2**	**55.1 ± 0.5**	**47.3 ± 0.3**	**49.4 ± 0.4**	**52.7 ± 0.2**
Seq-LoRA		86.4 ± 0.2	88.1 ± 0.3	77.2 ± 0.7	81.2 ± 0.4	72.6 ± 0.2	77.8 ± 0.6	78.7 ± 0.3
O-LoRA	Flan-T5	85.0 ± 0.6	86.9 ± 0.5	71.0 ± 0.7	73.1 ± 0.6	70.6 ± 0.4	**80.1 ± 0.5**	75.5 ± 0.6
DEAL	Large	87.5 ± 0.2	**89.2 ± 0.4**	78.3 ± 0.5	82.3 ± 0.3	73.7 ± 0.1	78.9 ± 0.5	79.8 ± 0.2
ProCL		**88.3 ± 0.3**	**88.9 ± 0.6**	**81.0 ± 0.4**	**85.7 ± 0.5**	**74.6 ± 0.2**	**80.0 ± 0.4**	**81.3 ± 0.3**
Seq-LoRA		75.7 ± 0.3	77.5 ± 0.5	66.8 ± 0.6	76.3 ± 0.4	55.6 ± 0.2	63.0 ± 0.3	66.0 ± 0.4
O-LoRA	Flan-T5	75.8 ± 0.5	77.6 ± 0.2	66.4 ± 0.3	76.0 ± 0.6	53.4 ± 0.4	62.0 ± 0.2	65.2 ± 0.3
DEAL	Base	77.0 ± 0.2	78.7 ± 0.6	67.2 ± 0.3	76.8 ± 0.2	56.2 ± 0.5	63.4 ± 0.3	66.8 ± 0.3
ProCL		**78.1 ± 0.4**	**79.8 ± 0.5**	**68.2 ± 0.3**	**77.8 ± 0.2**	**57.3 ± 0.6**	**64.4 ± 0.4**	**67.9 ± 0.4**

Table 2: Final average accuracy across continual learning rounds on text classification. Results are reported as mean ± standard deviation over 3 runs. Avg (AA) denotes the mean accuracy across the three curricula. Bold denotes the best method, or all methods within a small effect size of it (Cohen's $d < 0.5$).

Configuration	LLaMA3-3B	Qwen3-4B	T5-Base
w/o W_{orig}	65.9 ± 0.3	67.1 ± 0.4	61.6 ± 0.8
w/o Cons.	68.2 ± 1.0	69.4 ± 0.8	60.1 ± 1.3
Random α	67.5 ± 1.4	68.6 ± 1.2	61.8 ± 1.2
Uniform α	69.0 ± 1.6	70.1 ± 1.3	60.4 ± 0.9
Full	**69.8 ± 0.6**	**72.5 ± 0.2**	**62.8 ± 1.0**

Table 3: ProCL ablation on the QA benchmark across different backbone models.

achieved using balanced hyperparameters.

On the Forgetting Issues. We measure the forgetting on BoolQ as the average accuracy drop after subsequent tasks: $\frac{1}{2}\big[(\text{Acc}(\text{BoolQ} \mid \text{round } 1) - \text{Acc}(\text{BoolQ} \mid \text{round } 2)) + (\text{Acc}(\text{BoolQ} \mid \text{round } 1) - \text{Acc}(\text{BoolQ} \mid \text{round } 3))\big]$, capturing both immediate and cumulative forgetting. As shown in Fig. 1, existing CL methods exhibit noticeable forgetting, with accuracy drops typically ranging from moderate to severe across models (e.g., DEAL reaches 45.97 on Flan-T5-Large, while Seq-LoRA and O-LoRA often exceed 20 points on Qwen models). While DEAL achieves strong final accuracy on some architectures, its forgetting remains highly variable, reflecting a trade-off between plasticity and stability. In contrast, ProCL consistently reduces forgetting, achieving the lowest or near-lowest drop across all models (e.g., 4.63 on Flan-T5-Base and below 10 on Qwen3-4B/8B). Together with final accuracy, this demonstrates that ProCL more effectively balances knowledge retention and adaptation compared to existing approaches.

Efficiency. Appendix Fig. 3 (right) benchmarks ProCL against the second-best baseline DEAL across multiple LLM backbones on a single GPU, averaging over sequence lengths (128, 512) and batch sizes $(4, 16)$ using 800 samples per model. With routing enabled during training, ProCL still runs ∼1.7–2.0× faster than DEAL and achieves a ∼6–8× speedup at inference by removing routing entirely.

6 Conclusion

We proposed ProCL, a memory-structured continual LoRA framework that organizes adaptation through program memory and consolidation. By enabling input-conditioned adapter composition during training and a compact static adapter at inference, ProCL achieves a favorable balance between plasticity and stability without additional deployment cost. Across diverse LLM backbones and tasks, our method consistently reduces catastrophic forgetting and improves retention over existing continual LoRA approaches. These results suggest that structuring the low-rank space with program-level memory is a promising direction for scalable continual adaptation of LLMs.

Limitations

ProCL has two key limitations. First, its effectiveness relies on the emergence of routing specialization. When tasks are highly similar or share overlapping features, the routing may collapse to a few dominant programs, reducing the intended separation and limiting gains in mitigating interference. Second, ProCL assumes a fixed set of programs; as the number or diversity of tasks grows, this fixed capacity may become a bottleneck, forcing unrelated tasks to share programs and degrading long-term retention. Future work could improve routing robustness by encouraging program diversity (e.g., via entropy regularization) to prevent collapse under similar tasks. Additionally, extending ProCL with dynamic program expansion or pruning could alleviate fixed-capacity limitations as task diversity grows.

Ethical Consideration

ProCL improves continual fine-tuning by reducing catastrophic forgetting, which can enhance the efficiency and adaptability of deployed language models in evolving environments. This may benefit applications requiring incremental updates, such as personalized assistants or domain adaptation, by reducing retraining costs and improving knowledge retention. However, improved retention may also preserve outdated or biased information if not carefully managed. Additionally, more efficient continual learning could lower the barrier to deploying large models, potentially amplifying misuse. Future work should consider mechanisms for controlled forgetting, bias mitigation, and responsible deployment.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[3] Kefan Cao and Shuaicheng Wu. 2025. Orthogonal low-rank adaptation in lie groups for continual learning of large language models. *arXiv preprint arXiv:2509.06100*.

[4] Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K Dokania, Philip HS Torr, and Marc'Aurelio Ranzato. 2019. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*.

[5] Xuanwen Ding, Jie Zhou, Liang Dou, Qin Chen, Yuanbin Wu, Arlene Chen, and Liang He. 2024. Boosting large language models with continual learning for aspect-based sentiment analysis. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4367–4377.

[6] Xiao Han, ZIMO ZHAO, Wanyu Wang, Maolin Wang, Zitao Liu, Yi Chang, and Xiangyu Zhao. 2025. Data efficient adaptation in large language models via continuous low-rank fine-tuning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[7] Thanh Duc Hoang, Duy-Hung Nguyen, Bao-Sinh Nguyen, Huy Hoang Nguyen, Hung Le, and 1 others. 2023. Universal graph continual learning. *Transactions on Machine Learning Research*.

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

[9] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

[10] Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. 2024. Lorahub: Efficient cross-task generalization via dynamic lora composition. In *First Conference on Language Modeling*.

[11] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, and 1 others. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

[12] Hung Le, Truyen Tran, and Svetha Venkatesh. 2019. Neural stored-program memory. In *International Conference on Learning Representations*.

[13] Hung Le and Svetha Venkatesh. 2022. Neurocoder: General-purpose computation using stored neural programs. In *International Conference on Machine Learning*, pages 12204–12221. PMLR.

[14] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient episodic memory for continual learning. *Advances in neural information processing systems*, 30.

[15] James L McClelland, Bruce L McNaughton, and Randall C O'Reilly. 1995. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419.

[16] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

[17] Kenneth A Norman and Randall C O'Reilly. 2003. Modeling hippocampal and neocortical contributions to recognition memory: a complementary-learning-systems approach. *Psychological review*, 110(4):611.

[18] Pengjie Ren, Chengshun Shi, Shiguang Wu, Mengqi Zhang, Zhaochun Ren, Maarten de Rijke, Zhumin Chen, and Jiahuan Pei. 2024. Melora: Mini-ensemble low-rank adapters for parameter-efficient fine-tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3052–3064.

[19] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. 2019. Experience replay for continual learning. *Advances in neural information processing systems*, 32.

[20] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*.

[21] Yimin Tian, Bolin Zhang, Zhiying Tu, and Dianhui Chu. 2025. Adapters selector: Cross-domains and multi-tasks lora modules integration usage method. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 593–605.

[22] Xiao Wang, Tianze Chen, Qiming Ge, Han Xia, Rong Bao, Rui Zheng, Qi Zhang, Tao Gui, and Xuan-Jing Huang. 2023. Orthogonal subspace learning for language model continual learning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10658–10671.

[23] James CR Whittington, William Dorrell, Timothy EJ Behrens, Surya Ganguli, and Mohamady El-Gaby. 2025. A tale of two algorithms: Structured slots explain prefrontal sequence memory and are unified with hippocampal cognitive maps. *Neuron*, 113(2):321–333.

[24] Xun Wu, Shaohan Huang, and Furu Wei. 2024. Mixture of lora experts. In *The Twelfth International Conference on Learning Representations*.

[25] Shuo Yang, Kun-Peng Ning, Yu-Yang Liu, Jia-Yu Yao, Yong-Hong Tian, Yi-Bing Song, and Li Yuan. 2025. Is parameter collision hindering continual learning in llms? In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 4243–4259.

Appendix

A Theoretical Results

For notational simplicity, we describe the theoretical properties of our program memory framework in terms of a general weight matrix W. In practice, W corresponds to the adapter weights, which are combined with the pretrained LLM weights to produce the final output. We omit this detail for brevity, as it does not affect the nature of the theoretical analysis. We also assume $\gamma = 1$ for notation simplicity (the result extends trivially by absorbing γ into W_{orig}), without affecting the validity of the proof.

A.1 Program Routing Reduces Cross-Task Interaction

Setup and Assumptions

Consider a continual learning sequence of tasks $\{1, \ldots, T\}$. For each task t, let $x_t \in \mathbb{R}^{B\times D}$ be the batched input, y_t^* the target, and $\mathcal{L}_t$ the task loss. Recall that $\hat{W} = [\hat{W}_1; \ldots; \hat{W}_N]$ is formed by concatenating N head outputs, where head h produces:

$$\hat{W}_h = \sum_{n=1}^{N} \bar{\alpha}_{h,n}\, \sigma(s_n)\, W^{(n)} \tag{14}$$

where $\hat{W}_h \in \mathbb{R}^{r\times D}$, $r = R/N$ and $\bar{\alpha}_{h,n} = \frac{1}{B}\sum_{b=1}^{B} \mathrm{Softmax}(Q[h]K_h^\top)_{b,n}$ is the batch-averaged attention weight of head h over program n. The executing weight is $W_{\text{exec}} = W_{\text{orig}} + \hat{W}$, with W_{orig} frozen throughout training.

For simplicity, the output is $y = xW_{\text{exec}}^\top$, so the gradient of the loss with respect to W_{exec} is:

$$\frac{\partial \mathcal{L}_t}{\partial W_{\text{exec}}} = (x_t W_{\text{exec}}^\top - y_t^*)^\top x_t \tag{15}$$

Since W_{orig} is frozen, we define the residual target:

$$r_t \triangleq y_t^* - x_t W_{\text{orig}}^\top \tag{16}$$

which isolates the novel task signal not yet explained by prior knowledge in W_{orig}. The gradient then simplifies to:

$$\frac{\partial \mathcal{L}_t}{\partial W_{\text{exec}}} = (x_t \hat{W}^\top - r_t)^\top x_t \tag{17}$$

Since $W^{(n)} \in \mathbb{R}^{r\times D}$ contributes to every head h via $\bar{\alpha}_{h,n}$, and $\hat{W}_h$ occupies the h-th row block of $\hat{W}$, the gradient with respect to program $W^{(n)}$ accumulates across all heads:

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial W^{(n)}} &= \sum_{h=1}^{N} \frac{\partial \mathcal{L}_t}{\partial \hat{W}_h} \cdot \frac{\partial \hat{W}_h}{\partial W^{(n)}} \\ &= \sigma(s_n) \sum_{h=1}^{N} \bar{\alpha}_{h,n} \cdot \left[(x_t \hat{W}^\top - r_t)^\top x_t\right]_h\end{aligned} \tag{18}$$

where $[\cdot]_h$ denotes the h-th row block of size $r \times D$. Program n thus receives a *head-weighted aggregation* of gradient signals, gated jointly by $\sigma(s_n)$ and the per-head routing weights $\{\bar{\alpha}_{h,n}\}_{h=1}^{N}$.

We analyze **ProCL** that optimizes programs $\{W^{(n)}\}$ via Eq. (18), where W_{orig} anchors gradients to the residual r_t, and $\{\bar{\alpha}_{h,n}\}$ selectively gates updates per program per head. We make the following assumption:

A1. (Routing specialization) As training proceeds, attention sharpens so each task t concentrates routing mass on a subset $\mathcal{S}_t \subseteq \{1, \ldots, N\}$, with $\mathcal{S}_t \cap \mathcal{S}_{t'} \to \emptyset$ for tasks with distinct input statistics.

Proposition A.1 (Program Routing Bounds Cross-Task Gradient Interaction)**.** *Let $\mathcal{J}(t, t')$ denote the cross-task gradient interaction magnitude between tasks t and $t' < t$:*

$$\mathcal{J}(t, t') \triangleq \left| \left\langle \frac{\partial \mathcal{L}_t}{\partial W_{\text{exec}}}, \frac{\partial \mathcal{L}_{t'}}{\partial W_{\text{exec}}} \right\rangle_F \right| \geq 0, \tag{19}$$

*and let $\mathcal{S}(t, t') \triangleq \sum_h |\langle \tilde{g}_{h,t}, \tilde{g}_{h,t'}\rangle_F|$ aggregate per-head interaction magnitudes. Then **ProCL** satisfies*

$$\mathcal{J}_{\text{ProCL}}(t, t') \;\leq\; \big(\max_h \beta_h\big)\, \mathcal{S}(t, t'). \tag{20}$$

*Under Assumption **A1**, $\max_h \beta_h \to 0$ as routing specialization increases, and consequently $\mathcal{J}_{\text{ProCL}}(t, t') \to 0$.*

Proof. Let $G^t = \frac{\partial \mathcal{L}_t}{\partial W_{\text{exec}}}$ denote the effective adapter gradient of task t. Since $\hat{W} = [\hat{W}_1, \ldots, \hat{W}_N]$ is formed by concatenating the outputs of N heads along the row dimension, the gradient naturally decomposes into disjoint row blocks. Let $\tilde{g}_{h,t}$ denote the gradient block corresponding to head h, embedded in the full $R \times D$ space. Because different heads occupy

non-overlapping row blocks, their supports are disjoint, and thus for $h \neq h'$,

$$\langle \tilde{g}_{h,t}, \tilde{g}_{h',t'} \rangle_F = 0.$$

Therefore, the full inner product decomposes as

$$\langle G^t, G^{t'} \rangle_F = \sum_h \langle \tilde{g}_{h,t}, \tilde{g}_{h,t'} \rangle_F.$$

Under ProCL, gradients are routed through programs as in Eq. 18, yielding

$$\langle G^t, G^{t'} \rangle_F^{\text{ProCL}} = \sum_h \beta_h \langle \tilde{g}_{h,t}, \tilde{g}_{h,t'} \rangle_F,$$

where

$$\beta_h = \sum_n \sigma(s_{n,t})\sigma(s_{n,t'})\bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'}.$$

Here, because s_n is a task-specific parameter (Sec. 4.3), we write $s_{n,t}$ to denote the value learned during task t. We first bound β_h. Define

$$u_n = \sigma(s_{n,t})\bar{\alpha}_{h,n,t}, \qquad v_n = \sigma(s_{n,t'})\bar{\alpha}_{h,n,t'}.$$

Then $\beta_h = \sum_n u_n v_n$. By the Cauchy–Schwarz inequality,

$$\beta_h \leq \|u\|_2 \|v\|_2.$$

Since $0 < \sigma(\cdot) < 1$ and routing weights form a probability distribution,

$$\sum_n \bar{\alpha}_{h,n,t} = 1, \qquad 0 \leq \bar{\alpha}_{h,n,t} \leq 1,$$

we obtain

$$\begin{aligned} \|u\|_2^2 &= \sum_n \sigma(s_{n,t})^2 \bar{\alpha}_{h,n,t}^2 \\ &\leq \sum_n \bar{\alpha}_{h,n,t}^2 \\ &\leq \sum_n \bar{\alpha}_{h,n,t} \\ &= 1. \end{aligned} \tag{21}$$

Thus $\|u\|_2 \leq 1$ and similarly $\|v\|_2 \leq 1$, implying

$$0 \leq \beta_h \leq 1.$$

Applying the triangle inequality to the routed inner-product decomposition,

$$\begin{aligned} \mathcal{J}_{\text{ProCL}}(t,t') &= \Big| \sum_h \beta_h \langle \tilde{g}_{h,t}, \tilde{g}_{h,t'} \rangle_F \Big| \\ &\leq \sum_h \beta_h \big| \langle \tilde{g}_{h,t}, \tilde{g}_{h,t'} \rangle_F \big|. \end{aligned} \tag{22}$$

Bounding each β_h by $\max_{h'} \beta_{h'}$,

$$\begin{aligned} \mathcal{J}_{\text{ProCL}}(t,t') &\leq \big(\max_h \beta_h\big) \sum_h \big| \langle \tilde{g}_{h,t}, \tilde{g}_{h,t'} \rangle_F \big| \\ &= \big(\max_h \beta_h\big) \mathcal{S}(t,t'). \end{aligned} \tag{23}$$

Finally, under Assumption A1 (routing specialization), for each task t and head h, the routing distribution $\bar{\alpha}_{h,\cdot,t}$ concentrates on a subset $\mathcal{S}_t \subseteq \{1, \ldots, N\}$ such that

$$\sum_{n \in \mathcal{S}_t} \bar{\alpha}_{h,n,t} \to 1, \qquad \sum_{n \notin \mathcal{S}_t} \bar{\alpha}_{h,n,t} \to 0,$$

and for tasks with distinct input statistics, these subsets become asymptotically disjoint, i.e.,

$$\mathcal{S}_t \cap \mathcal{S}_{t'} = \emptyset.$$

Since $0 < \sigma(\cdot) \leq 1$, we upper bound

$$\beta_h \leq \sum_n \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} = \langle \bar{\alpha}_{h,\cdot,t}, \bar{\alpha}_{h,\cdot,t'} \rangle.$$

We now show that this inner product vanishes. Decompose over $\mathcal{S}_t$:

$$\sum_n \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} = \sum_{n \in \mathcal{S}_t} \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} + \sum_{n \notin \mathcal{S}_t} \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'}.$$

For the first term, since $\mathcal{S}_t \cap \mathcal{S}_{t'} = \emptyset$, all indices in $\mathcal{S}_t$ lie outside $\mathcal{S}_{t'}$, and thus

$$\sum_{n \in \mathcal{S}_t} \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} \leq \Big(\sum_{n \in \mathcal{S}_t} \bar{\alpha}_{h,n,t} \Big) \cdot \max_{n \in \mathcal{S}_t} \bar{\alpha}_{h,n,t'} \;\to\; 0,$$

since $\bar{\alpha}_{h,n,t'} \to 0$ for $n \notin \mathcal{S}_{t'}$.

For the second term,

$$\sum_{n \notin \mathcal{S}_t} \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} \leq \sum_{n \notin \mathcal{S}_t} \bar{\alpha}_{h,n,t} \;\to\; 0.$$

Combining both bounds yields

$$\begin{aligned} &\sum_n \bar{\alpha}_{h,n,t}\bar{\alpha}_{h,n,t'} \to 0, \\ &\text{and hence} \qquad \beta_h \to 0 \qquad \text{for every } h. \end{aligned} \tag{24}$$

Therefore $\max_h \beta_h \to 0$. Since $\mathcal{S}(t,t')$ is a finite sum of finite-dimensional gradient inner products, the bound $\mathcal{J}_{\text{ProCL}}(t,t') \leq (\max_h \beta_h)\, \mathcal{S}(t,t')$ gives

$$\mathcal{J}_{\text{ProCL}}(t,t') \to 0.$$

□

A.2 Theoretical Justification of Consolidation and Efficient Inference

We formalize the consolidation mechanism within a single task, under a late-training regime where the program memory and routing behavior are approximately stable. The adapter is structured as

$$W = [W^{(1)}; \ldots ; W^{(N)}],$$

and the routed program composition is

$$\Delta(x) := \hat{W}(x) = \sum_{n=1}^{N} \alpha_n(x)\, \sigma(s_n)\, W^{(n)}, \quad (25)$$

where $\alpha_n(x)$ are routing weights with $\sum_n \alpha_n(x) = 1$. The executing adapter is

$$W_{\text{exec}}(x) = \gamma W_{\text{orig}} + \Delta(x),$$

and consolidation updates

$$W^{(t+1)} = (1-\lambda)W^{(t)} + \lambda W_{\text{exec}}(x_t), \qquad x_t \sim \mathcal{D},$$

where $\lambda \in (0, 1)$ is the consolidation rate.

Setup and Assumptions. Within the analysis window, we make the following assumptions:

A2. (Late-training stationarity) The map $x \mapsto \Delta(x)$ is approximately stationary: the slot drift induced by consolidation is small relative to the routing-induced averaging over the analysis window. We treat $\Delta(x)$ as a fixed function of x for the purposes of the recursion analysis.

A3. (Sampling) The samples $x_t \sim \mathcal{D}$ are drawn i.i.d., so that $\mathbb{E}[\Delta(x_t) \mid W^{(t)}] = \mathbb{E}_{x\sim\mathcal{D}}[\Delta(x)]$.

A4. (Boundedness) $\mathbb{E}\|\Delta(x)\|_F < \infty$, ensuring W^* is well-defined and $\mathbb{E}[\Delta(x)]$ exists as a finite matrix.

Scope. Algorithm 1 resets $W_{\text{orig}} \leftarrow W$ at task boundaries. Proposition A.2 therefore applies within a single task, where W_{orig} and γ are held fixed. Across tasks, the recursion restarts from a new anchor.

Proposition A.2 (Consolidation Approximates Expected Routed Adapter)**.** *Define*

$$W^* := \gamma W_{\text{orig}} + \mathbb{E}_{x\sim\mathcal{D}}[\Delta(x)].$$

Under Assumptions A2–A4, $\mathbb{E}[W^{(t)}] \to W^*$ *as* $t \to \infty$.

Proof. Under A2, $\Delta(x)$ is a linear combination of fixed program slots:

$$\Delta(x) = \sum_{n=1}^{N} \alpha_n(x)\, \sigma(s_n)\, W^{(n)}.$$

By linearity of expectation, justified by A4,

$$\mathbb{E}[\Delta(x)] = \sum_{n=1}^{N} \mathbb{E}[\alpha_n(x)]\, \sigma(s_n)\, W^{(n)}.$$

Define $E^{(t)} := W^{(t)} - W^*$. From the consolidation update,

$$\begin{aligned} E^{(t+1)} &= (1-\lambda)W^{(t)} + \lambda\big(\gamma W_{\text{orig}} + \Delta(x_t)\big) - W^* \\ &= (1-\lambda)W^{(t)} - (1-\lambda)\gamma W_{\text{orig}} \\ &\quad + \lambda\Delta(x_t) - \mathbb{E}[\Delta(x)] \\ &= (1-\lambda)\big(W^{(t)} - \gamma W_{\text{orig}} - \mathbb{E}[\Delta(x)]\big) \\ &\quad + \lambda\big(\Delta(x_t) - \mathbb{E}[\Delta(x)]\big) \\ &= (1-\lambda)E^{(t)} + \lambda\xi_t, \end{aligned}$$

where $\xi_t := \Delta(x_t) - \mathbb{E}[\Delta(x)]$.

Under A3, $\mathbb{E}[\xi_t \mid W^{(t)}] = 0$. Taking expectation,

$$\mathbb{E}[E^{(t+1)}] = (1-\lambda)\, \mathbb{E}[E^{(t)}].$$

Unrolling the recursion gives

$$\mathbb{E}[E^{(t)}] = (1-\lambda)^t\, E^{(0)} \to 0 \quad \text{as } t \to \infty,$$

since $\lambda \in (0, 1)$ implies $|1-\lambda| < 1$. Therefore $\mathbb{E}[W^{(t)}] \to W^*$. □

Remark on the late-training regime. Proposition A.2 is a within-task, late-training result, applying once the program memory and routing are approximately stable — a condition that is natural as training converges. Under this regime, consolidation performs online stochastic approximation of W^*, with each program slot $W^{(n)}$ contributing proportionally to its expected routing frequency $\mathbb{E}[\alpha_n(x)]$ and scaling factor $\sigma(s_n)$: frequently activated and highly scaled programs dominate the consolidated adapter. Outside this regime, the target W_t^* becomes time-varying; consolidation then tracks a slowly drifting mean, with λ governing the trade-off between tracking speed and variance.

Algorithm 1 ProCL Training

Require: Continual datasets $\{\mathcal{D}_1, \ldots, \mathcal{D}_T\}$, initial LoRA adapter $W \in \mathbb{R}^{R \times D}$, number of programs N, consolidation rate λ

Ensure: Final consolidated adapter W

Partition W into programs $\{W^{(1)}, \ldots, W^{(N)}\}$, where $W^{(n)} \in \mathbb{R}^{r \times D}$ and $r = R/N$
$W_{\text{orig}} \leftarrow W$
for task $t = 1$ **to** T **do**
  for mini-batch $(\mathbf{x}, \mathbf{y}^*) \sim \mathcal{D}_t$ **do**
    for sample $b = 1$ **to** B **do**
      $\mathbf{z}^{(b)} \leftarrow \frac{1}{T} \sum_{\ell=1}^{T} \mathbf{x}_\ell^{(b)}$
      $\mathbf{Q}^{(b)} \leftarrow f_{\text{enc}}(\mathbf{z}^{(b)}) \in \mathbb{R}^{N \times k_d}$
    end for
    for head $h = 1$ **to** N **do**
      for sample $b = 1$ **to** B **do**
        $\boldsymbol{\alpha}_h^{(b)} \leftarrow \text{Softmax}\big(\mathbf{Q}^{(b)}[h] K_h^\top\big)$
      end for
      $\bar{\boldsymbol{\alpha}}_h \leftarrow \frac{1}{B} \sum_{b=1}^{B} \boldsymbol{\alpha}_h^{(b)}$
      $\widehat{W}_h \leftarrow \sum_{n=1}^{N} \bar{\alpha}_{h,n} \sigma(s_n) W^{(n)}$
    end for
    $\widehat{W} \leftarrow [\widehat{W}_1 \parallel \cdots \parallel \widehat{W}_N]$
    $W_{\text{exec}} \leftarrow \gamma W_{\text{orig}} + \widehat{W}$
    Compute predictions y using W_{exec}
    Compute loss $\mathcal{L}(\mathbf{y}, \mathbf{y}^*)$
    Update trainable parameters via backpropagation ▷ W_{orig} is frozen
    $W \leftarrow (1 - \lambda) W + \lambda W_{\text{exec}}$ ▷ Consolidation
  end for
  $W_{\text{orig}} \leftarrow W$ ▷ Freeze consolidated adapter for next task
end for
return W

B Method Details

B.1 Training Configurations and Hyperparameters

Unless otherwise stated, we use LoRA-based continual fine-tuning, where the adapter from round $t-1$ initializes the adapter in round t. For text classification (TC) continual experiments, we follow prior work exactly[1] (Creative Commons Attribution-NonCommercial 4.0 International license), using a constant learning rate with no warmup and sequence-length limits of 512/50 (source/target). For QA tasks, we use the same base learning rate and epoch count with task-specific generation lengths. The default hyperparameters for all baselines are reported in Table 5. A single continual learning run typically takes 2–24 hours on a V100 and 1–10 hours on an H100, depending on model size and task sequence.

We implemented the experiments in Python using PyTorch, Hugging Face Transformers, Datasets, PEFT, and Tokenizers. Tokenization was performed with the corresponding pretrained-model tokenizer loaded via AutoTokenizer; inputs were truncated/padded to a maximum source length of 512 and targets/generation to length 50. LoRA adapters were implemented with PEFT. For evaluation, we used exact match/accuracy and ROUGE-1/ROUGE-L, with ROUGE computed using rouge_score/the repository ROUGE implementation with stemming enabled where applicable. Classification F1 metrics were computed using scikit-learn. Package versions are listed in the released repository requirement files.

The paper is written with the support of AI tools to improve writing and fix grammar.

[1] https://github.com/zzm-black/DEAL-Continuous-Low-Rank-Fine-Tuning

Datasets. For TC continual learning, we use up to 15 datasets: `yelp`, `amazon`, `MNLI`, `CB`, `COPA`, `QQP`, `RTE`, `IMDB`, `SST-2`, `dbpedia`, `agnews`, `yahoo`, `MultiRC`, `BoolQA`, and `WiC`. For QA continual learning, we use up to 3 datasets: `google/boolq`, `squad`, and `adversarial_qa`.

Model	Size	Hugging Face Link
LLaMA-3.2	3B	https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct
LLaMA-3.1	8B	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
Qwen3	4B	https://huggingface.co/Qwen/Qwen3-4B
Qwen3	8B	https://huggingface.co/Qwen/Qwen3-8B
Flan-T5	Base	https://huggingface.co/google/flan-t5-base
Flan-T5	Large	https://huggingface.co/google/flan-t5-large

Table 4: LLM used in our experiments. All models are publicly available on Hugging Face.

Hyperparameter	Value
Backbone models	Llama-3 / Flan-T5 / Qwen3
PEFT method	LoRA (continual adapter initialization)
LoRA rank r	32 (UIE), 16 (QA)
LoRA scaling α	32
LoRA dropout	0.1
Learning rate	1×10^{-5}
Train batch size (per device)	4 (TC), 16 (QA)
Gradient accumulation steps	2 (TC), 1 (QA)
Number of epochs	1 per task (except for TC's MNLI: 2)
Input / output length	512 / 50 (TC), QA decoding up to 32 new tokens

Table 5: Core training hyperparameters used across experiments.

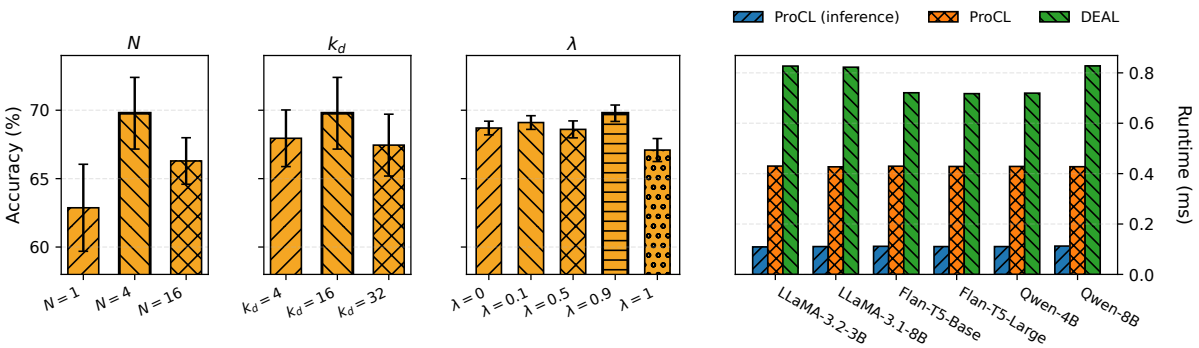

Figure 3: Left: Hyperparameter sensitivity of ProCL on QA benchmark. Right: Average runtime (ms per iteration) comparison of ProCL and DEAL across LLMs