Information Processing and Management 63 (2026) 104580

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/ipm

Less is more: Towards green code large language models via unified structural pruning

Guang Yang [a], Yu Zhou [a,*], Xiangyu Zhang [a], Wei Cheng [a], Ke Liu [b], Xiang Chen [c], Terry Yue Zhuo [d], Taolue Chen [e,*]

[a] Nanjing University of Aeronautics and Astronautics, Nanjing City, Jiangsu province, China
[b] National University of Defense Technology, Changsha City, Hunan Province, China
[c] Nantong University, Nantong City, Jiangsu Province, China
[d] Monash University and CSIRO's Data61, Melbourne, Australia
[e] Birkbeck, University of London, London, England

ARTICLE INFO

Keywords:
Large language models
Code intelligence
Structural pruning
Post-training
Code instruction tuning

ABSTRACT

The extensive application of Large Language Models (LLMs) in generative coding tasks has raised concerns due to their high computational demands and energy consumption. Unlike previous structural pruning methods designed for classification models that deal with low-dimensional classification logits, generative Code LLMs produce high-dimensional token logit sequences, making traditional pruning objectives inherently limited. Moreover, existing single-component pruning approaches further constrain the effectiveness when applied to generative Code LLMs. In response, we propose Flab-Pruner, an innovative unified structural pruning method that combines vocabulary, layer, and Feed-Forward Network (FFN) pruning. This approach effectively reduces model parameters while maintaining performance. Additionally, we introduce a customized code instruction data strategy for coding tasks to enhance the performance recovery efficiency of the pruned model. Through extensive evaluations on three state-of-the-art Code LLMs across multiple generative coding tasks, the results demonstrate that Flab-Pruner retains 97% of the original code generation performance on average after pruning 22% of the parameters, and achieves the same or even better performance after post-training. The pruned models exhibit significant improvements in storage, GPU usage, computational efficiency, and environmental impact, while maintaining well robustness. Our research provides a sustainable solution for green software engineering and promotes the efficient deployment of LLMs in real-world generative coding intelligence applications.

1. Introduction

Large Language Models (LLMs) have demonstrated outstanding performance and been deployed across numerous domains (Huang et al., 2023; Yao et al., 2024; Yuan et al., 2023). Software engineering is no exception (Fan et al., 2023; Hou et al., 2023; Kirova et al., 2024; Sallou et al., 2024), with LLMs excelling in tasks like code generation (Gu, 2023), summarization (Nam et al., 2024), and vulnerability detection (Lu et al., 2024). However, the substantial scale and intensive computational requirements of these models

* Corresponding authors.
E-mail addresses: yang.guang@nuaa.edu.cn (G. Yang), zhouyu@nuaa.edu.cn (Y. Zhou), zhangx1angyu@nuaa.edu.cn (X. Zhang), chengweii@nuaa.edu.cn (W. Cheng), liuke23@nudt.edu.cn (K. Liu), xchencs@ntu.edu.cn (X. Chen), terry.zhuo@monash.edu (T.Y. Zhuo), taolue.chen@gmail.com (T. Chen).

https://doi.org/10.1016/j.ipm.2025.104580
Received 24 April 2025; Received in revised form 4 August 2025; Accepted 19 December 2025
Available online 1 January 2026

pose challenges, particularly in resource-constrained environments (Chen et al., 2020; Hort et al., 2023; Liu et al., 2013). Moreover, the energy consumption associated with training and inference leads to high carbon emissions, raising concerns about environmental sustainability (Shi et al., 2024a,b).

To enhance energy efficiency and sustainability, green software engineering has spurred exploration of model pruning, quantization, and knowledge distillation (Liu et al., 2023a; Wang et al., 2024; Wei et al., 2023; Zhu et al., 2023), aimed at reducing the computational and environmental impact of LLMs. Quantization speeds up inference by converting high-precision weights to lower precision. Knowledge distillation transfers knowledge from larger to smaller models, requiring additional computational resources. Among these, model pruning has emerged as a promising strategy, with two main approaches: unstructured pruning that merely zeros out specific weights while maintaining the original parameter count, and structured pruning (Wang et al., 2020b) that effectively reduces model size by removing entire structural components (e.g., neurons, layers) while preserving model integrity and functionality. For instance, unstructured pruning, such as SparseGPT (Frantar and Alistarh, 2023), targets individual weights, achieving sparsity but not significantly reducing hardware demands, thus limiting their use in constrained settings. In contrast, structured pruning, such as ShortGPT (Men et al., 2024), reduces parameter counts and hardware dependencies, enhancing operational efficiency while maintaining model integrity.

Although these pruning methods can be theoretically applied to code generation models, our extensive experiments (cf. RQ1 in Section 5.1) reveal their significant limitations when applied to generative coding tasks. For example, when applying ShortGPT (Men et al., 2024) to the LLAMA model, we observe a complete performance collapse on the HumanEval code generation benchmark. Through careful analysis, we identify two fundamental limitations in existing approaches:

1. **Misaligned Pruning Objective:** Current pruning methods (Kim et al., 2024; Men et al., 2024; Yang et al., 2024b) primarily focus on layer-wise similarity metrics (e.g., angle distance, cosine similarity, and Taylor score between layers), which aim to preserve the model's general language modeling capabilities while overlooking the specific requirements of downstream tasks.
2. **Limited Pruning Scope:** Existing approaches typically adopt single-component pruning strategies (e.g., solely focusing on layer redundancy reduction), failing to leverage the potential synergies that could be achieved through an integrated, multi-granular pruning approach across different model components.
3. **Insufficient Code-specific Post-tuning Method:** Existing pruning methods rely on generic supervised fine-tuning on downstream datasets for performance recovery, without considering domain-specific adaptations for code-related tasks. This generic approach fails to leverage the unique characteristics and requirements of code generation tasks, potentially limiting the effectiveness of the post-pruning recovery process.

We provide a detailed analysis of these limitations and their implications in Section 5.1.

Beyond performance considerations, the reliability and robustness of pruned models in coding tasks raise critical concerns for real-world deployment. These concerns encompass several key aspects: How well do pruned models maintain their robustness against adversarial inputs? Despite the crucial nature of these questions for practical applications, existing research has largely focused on performance metrics while leaving these reliability aspects unexplored. We present a comprehensive analysis of these critical concerns in Section 5.3.

Method. To minimize model parameters while ensuring that pruned models maintain high standards in performance and robustness, First, we define the pruning objectives: since the model generates code through probability distributions over the vocabulary, we use KL divergence to ensure that the pruned model maintains similar token generation probabilities as the original model, directly optimizing for code generation behavior. Then we introduce Flab-Pruner, a unified structural pruning method designed for the combination of three components, i.e., **F**FN Pruning, **La**yer Pruning and Voca**b**ulary Pruning. In particular, the vocabulary pruning component reduces the model's embedding size by eliminating tokens that are absent in the given programming corpus. The FFN pruning component targets specific neurons within the FFN block, reducing the model's size by eliminating certain neurons. The layer pruning component reduces the number of layers in the model by assessing the redundancy between layers. To consider the consistency between pruning objectives and downstream task performance, the above pruning components are all designed to minimize the KL divergence between the pruned model and the original model.

Additionally, we introduce a customized code instruction tuning strategy specifically designed for generative coding tasks. In contrast to directly train the pruned model on the original dataset, we purposefully replace the code generated by the original model into the training dataset by evaluating the performance of the original and pruned models on the training set. Compared to using the original dataset for performance recovery, our approach enhances efficiency by achieving better performance recovery.

Evaluation. We undertake a comprehensive evaluation of Flab-Pruner spanning three widely studied code intelligence tasks, i.e., code generation, CoT generation and code output prediction. The goal of code generation is to convert requirement in natural language into code, bridging the gap between description and execution. CoT generation is about creating thought sequences from prompts, showcasing advanced reasoning. Code output prediction assesses the model's understanding of code and its ability to predict outcomes. By considering these diverse tasks, our evaluation aims to provide a holistic evaluation of Flab-Pruner's capabilities in various code intelligence domains. Our evaluation targets CodeQwen-1.5[1] and its two variants (NxCode[2] and CodeSlerp[3]), which are top-performing 7B-10B models on the BigCode benchmark[4].

[1] https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat
[2] https://huggingface.co/NTQAI/Nxcode-CQ-7B-orpo
[3] https://huggingface.co/dohrisalim/Code-7B-slerp
[4] https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

Besides performance, we conduct efficiency and robustness analyses to provide a comprehensive evaluation. An ablation study is also performed to evaluate the contribution of each component to our pruning methodology, highlighting their synergistic effects.

Findings. By applying Flab-Pruner, we demonstrate the feasibility of pruning Code LLMs while preserving their core capabilities. Our empirical findings indicate that by pruning about 22% of the parameters, the pruned model retains around 97% of the original model's code generation performance on average. After performance recovery, the pruned model achieves comparable or even superior performance to the original model. Significant improvements are observed across various efficiency metrics, including reduced storage requirements, optimized GPU utilization, decreased FLOPs, lower CO_2 emissions, and increased tokens processed per second. Moreover, our structured pruning technique is orthogonal to existing quantization methods, enabling further efficiency gains through their combination. Furthermore, we demonstrate that the pruned model maintains its robustness across different perturbation scenarios, affirming the effectiveness of our pruning methodology.

The main contributions can be summarized as follows.

- We introduce Flab-Pruner, a novel unified structural pruning approach that integrates FFN pruning, layer pruning, and vocabulary pruning to effectively minimize model size while maintaining performance for generative coding tasks.
- We propose a customized training data instruction strategy specifically designed for coding tasks, ensuring that pruned models continue to perform effectively.
- We conduct an extensive evaluation of Flab-Pruner across multiple generative coding tasks in terms of the performance, efficiency, and robustness, demonstrating the practical applicability of our approach in real-world software engineering scenarios.

To our best knowledge, this is one of the first attempts to comprehensively assess the impact of pruning techniques on the performance of LLMs in generative coding tasks. To facilitate the replication, the pruned models[5] and datasets[6] are all made publicly available.

Organization. The rest of the paper is structured as follows. Section 2 provides the background. In Section 3, we detail our proposed method for vocabulary pruning, layer pruning and FFN pruning. Section 4 describes the experiment settings. Section 5 presents the experimental results. Section 6 analyzes threats to validity and Section 7 discusses relevant studies. Finally, we conclude our paper and present directions of future work in Section 8.

2. Background

2.1. Transformer

A standard LLM, such as the Transformer (Vaswani et al., 2017), processes input sequences through several key components, including an embedding layer, multiple self-attention layers, feed-forward networks, and an output layer.

Given an input sequence of token indices $X = (x_1, x_2, \ldots, x_n)$, the LLM first maps these discrete indices into continuous vectors using an embedding layer:

$$E = (e_1, e_2, \ldots, e_n) \quad \text{, where} \quad e_i = W_e[x_i] \tag{1}$$

Here, $W_e \in \mathbb{R}^{|V| \times d}$ is the embedding matrix, $|V|$ is the vocabulary size, and d is the dimension of the embeddings. Each token x_i is mapped to its corresponding embedding vector e_i. The embedded sequence E is then processed through a series of self-attention layers and feed-forward networks. For each layer l, the transformation from input $H^{(l-1)}$ to output $H^{(l)}$ is given by:

$$H^{(l)} = \text{FFN}(\text{SelfAttention}(H^{(l-1)})) + H^{(l-1)} \tag{2}$$

where $H^{(0)} = E$. The SelfAttention$(\cdot)$ function computes contextual representations by attending to various parts of $H^{(l-1)}$, while FFN$(\cdot)$ is a position-wise feed-forward network.

The final hidden state vector $\mathbf{H}_L$ of the last layer is mapped to a vocabulary space using a linear transformation, followed by a softmax function to obtain a probability distribution over the vocabulary: $\mathbf{z} = \mathbf{W_o}\mathbf{H}_L + \mathbf{b}$, where $W_o \in \mathbb{R}^{d \times |V|}$ is the output layer weights and $\mathbf{b}$ is the bias vector. This maps the d-dimensional outputs back to the vocabulary space $|V|$, resulting in a probability distribution over potential output tokens.

The probability of generating the next code token y is computed using the softmax function:

$$P(y) = \text{softmax}(\mathbf{z}) = \frac{\exp(\mathbf{z}_y)}{\sum_{j=1}^{|\mathcal{W}|} \exp(\mathbf{z}_j)} \tag{3}$$

where $\mathbf{z}_y$ is the score for the word y in the vocabulary, and $|\mathcal{W}|$ is the size of the vocabulary.

$$P(Y|X) = \prod_{i=1}^{m} P(y_i|X, y_1, y_2, \cdots, y_{i-1}) \tag{4}$$

The model generates a sequence of code tokens $y_1, y_2, \ldots, y_m$ by sampling from the probability distributions $P(y_1|X), P(y_2|X, y_1), \ldots, P(y_m|X, y_1, \ldots, y_{m-1})$, where each token is conditioned on the previous tokens in the sequence.

[5] https://www.modelscope.cn/profile/FlabPruner

[6] https://huggingface.co/datasets/Flab-Pruner/CodeHarmony

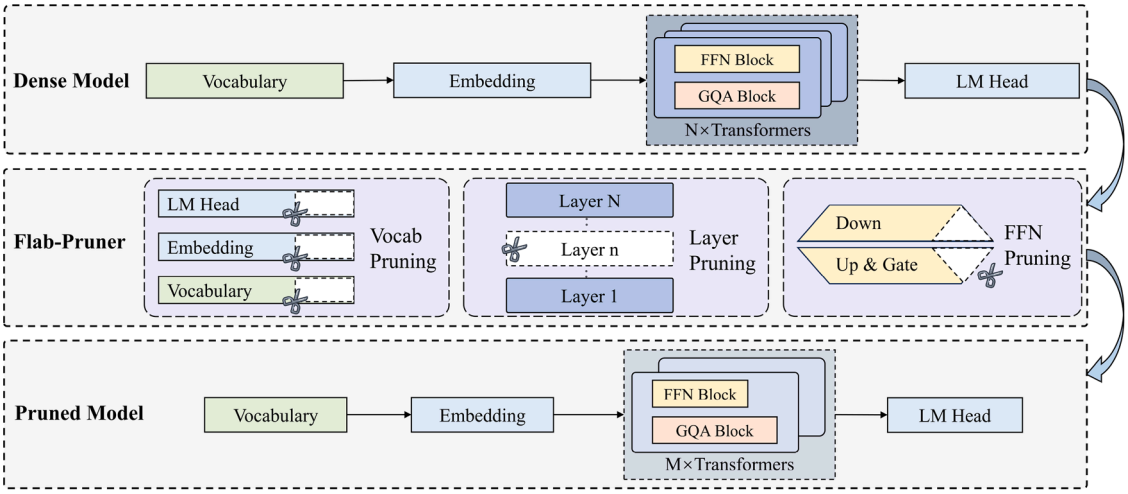


Fig. 1. The framework of Flab-Pruner.

2.2. Model pruning

Model pruning (Liu et al., 2019) is a well-established and efficient technique used for compressing models by reducing redundancy. Pruning methods are typically categorized as unstructured (Mason-Williams and Dahlqvist, 2024; Sun et al., 2023) and structured (Ma et al., 2023). In unstructured pruning, specific parameters, often individual weights or connections, are selectively removed without considering the model's internal structure. Unstructured pruning can be defined as $\boldsymbol{W}_{\text{pruned}} = \boldsymbol{W} \odot M$, where W represents the original weight matrix, M is a binary mask indicating which weights to prune (0 for the pruned weights and 1 for the remaining weights), and $\odot$ denotes element-wise multiplication.

On the other hand, structured pruning offers a more systematic approach that preserves the overall architecture of the model. This method involves eliminating entire neurons, channels, or layers based on predefined criteria, such as the weight magnitudes or the importance of a neuron's contribution to the model's output.

3. Our method

The workflow of Flab-Pruner is shown in Fig. 1. There are three major pruning steps, i.e., vocab pruning, layer pruning and FFN pruning. For a given dense model, we first perform vocab pruning, then layer pruning, and finally FFN pruning.

3.1. Pruning objective

To maintain the pruned model's performance, defining an appropriate pruning objective is crucial. In natural language processing, existing pruning methods typically focus on layer-level similarity. For instance, ShortGPT (Men et al., 2024) employs angle distance between layers, while Laco (Yang et al., 2024b) utilizes cosine similarity to identify and remove redundant layers.

However, for generative coding tasks, we argue that layer-level similarity alone is insufficient. The key challenge lies in ensuring that the pruned model generates code Y' that remains consistent with the original model's output Y. Note that the model generates code through a probability distribution over the vocabulary, obtained by applying a linear transformation followed by a softmax function to the hidden representations $H^{(L)}$ from the last layer. Therefore, preserving the output probability distribution is critical for maintaining generation quality.

Based on this insight, we propose a pruning objective that minimizes the KL divergence between the output probability distributions of the pruned model (P_{pruned}) and the original model (P_{original}). Formally, our objective can be expressed as:

$$\mathcal{L}_{\text{prune}} = \min_{\theta_p} D_{\text{KL}}(P(y|X;\theta_o) \| P(y|X;\theta_p)) \tag{5}$$

where θ_o and θ_p represent the parameters of the original and pruned models respectively, and D_{KL} denotes the Kullback-Leibler divergence between their output probability distributions:

$$D_{\text{KL}}(P(y|X;\theta_o) \| P(y|X;\theta_p)) = \sum_{y \in \mathcal{Y}} P(y|X;\theta_o) \log \frac{P(y|X;\theta_o)}{P(y|X;\theta_p)} \tag{6}$$

This objective ensures that the pruned model preserves the token generation probabilities of the original model, directly optimizing for the model's code generation behavior. By focusing on the output probability distributions that determine code generation, our approach better aligns with the goal of maintaining task-specific performance in generative coding tasks.

3.2. Vocabulary pruning

Algorithm 1 Vocabulary pruning algorithm.

Input:
Original LLM $\mathcal{M}_{\text{ori}}$;
Original Tokenizer $\mathcal{T}_{\text{ori}}$;
Code DataSet $\mathcal{D}$;
Output:
Pruned LLM $\mathcal{M}_{\text{pru}}$;
Pruned Tokenizer $\mathcal{T}_{\text{pru}}$;

$\mathcal{S} \leftarrow \text{CollectTokens}(\mathcal{D}, \mathcal{T}_{\text{ori}})$
$\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{T}_{\text{ori}}[\text{“special tokens”}]$
$\mathcal{T}_{\text{pru}} \leftarrow \text{PruneTokenizer}(\mathcal{T}_{\text{ori}}, \mathcal{S})$ $\mathcal{M}_{\text{pru}} \leftarrow \text{PruneModel}(\mathcal{M}_{\text{ori}}, \mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{pru}}, \mathcal{S})$
return $\mathcal{M}_{\text{pru}}, \mathcal{T}_{\text{pru}}$
Function `CollectTokens` $\mathcal{D}, \mathcal{T}_{\text{ori}}$
$\mathcal{S} \leftarrow \emptyset$
for $code \in \mathcal{D}$ **do**
for $token \in \mathcal{T}_{\text{ori}}.tokenize(code)$ **do**
$\mathcal{S} \leftarrow \mathcal{S} \cup \{token\}$
return $\mathcal{S}$
Function `PruneTokenizer` $\mathcal{T}_{\text{ori}}, \mathcal{S}$
$Vocab \leftarrow \emptyset$
$Merges \leftarrow \emptyset$
$id \leftarrow 0$
for $token \in \mathcal{S}$ **do**
$Vocab[token] \leftarrow id$
$id \leftarrow id + 1$
for each $m \in \mathcal{T}_{\text{ori}}[\text{“merges”}]$ **do**
if $\forall m[0], m[1], m[0] + m[1] \in \mathcal{S}$ **then**
$Merges \leftarrow Merges \cup \{m\}$
$\mathcal{T}_{\text{pru}}[\text{“vocab”}] \leftarrow Vocab$
$\mathcal{T}_{\text{pru}}[\text{“merges”}] \leftarrow Merges$
return $\mathcal{T}_{\text{pru}}$
Function `PruneModel` $\mathcal{M}_{\text{ori}}, \mathcal{T}_{\text{ori}}, \mathcal{T}_{\text{pru}}, \mathcal{S}$
$Embed \leftarrow \emptyset$
$LM \leftarrow \emptyset$ **for** $token \in \mathcal{S}$
$Embed[\mathcal{T}_{\text{pru}}[\text{“vocab”}][token]] \leftarrow \mathcal{M}_{\text{ori}}[\text{“embed tokens”}][\mathcal{T}_{\text{ori}}[\text{“vocab”}][token]]$
$LM[\mathcal{T}_{\text{pru}}[\text{“vocab”}][token]] \leftarrow \mathcal{M}_{\text{ori}}[\text{“lm head”}][\mathcal{T}_{\text{ori}}[\text{“vocab”}][token]]$
$\mathcal{M}_{\text{pru}}[\text{“embed tokens”}] \leftarrow Embed$
$\mathcal{M}_{\text{pru}}[\text{“lm head”}] \leftarrow LM$
return $\mathcal{M}_{\text{pru}}$

Vocabulary pruning, typically applied in tasks like text classification (Chen et al., 2019) and machine translation (Nair et al., 2023), involves removing these seldom-used tokens, thus reducing the overall model size without significantly affecting its capabilities.

In the context of Code LLMs, many models have expanded their vocabularies to accommodate a wide range of programming languages. This expansion often leads to a large vocabulary size as the models strive to cover various syntaxes and language-specific terminologies. However, in practice, developers usually work with a limited set of programming languages, which means that a significant portion of the vocabulary is rarely used. This observation leads to the hypothesis that the absence of these infrequent tokens might not substantially impact the model's ability to capture the nuances of any particular language effectively (Yang et al., 2022).

Formalization. Formally, given a vocabulary V and a usage statistic U, the pruned vocabulary V' is defined as:

$$V' = \{v \in V \mid U(v) > \tau\} \tag{7}$$

where τ is a threshold value representing the minimum frequency a token must have to remain in the vocabulary. Consequently, the embedding matrix $W_e \in \mathbb{R}^{|V| \times d}$ is reduced to $W'_e \in \mathbb{R}^{|V'| \times d}$, removing unnecessary token embeddings and reducing model complexity. Meanwhile, the output layer weights $W_o \in \mathbb{R}^{d \times |V|}$ is also reduced to $W'_o \in \mathbb{R}^{d \times} |V'|$.

Algorithms. In the realm of LLMs, the byte-level BPE algorithm (Wang et al., 2020a) is widely adopted for tokenization, as seen in models such as LLAMA (Touvron et al., 2023) and QWen (Bai et al., 2023). The BPE consists of two key elements: the vocabulary,

which is a set of tokens that the model identifies to interpret data, and merge rules, which guide the combination of token pairs into complex structures.

Algorithm 1 provides a detailed pseudo-code description of the vocabulary pruning process, structured into modular steps for enhanced clarity and maintainability. The process begins with the collection of all unique tokens from a given code dataset using the original tokenizer, as implemented in the 'CollectTokens' function (Lines 6–11). This set of tokens, $\mathcal{S}$, also includes essential special tokens (Line 3). The algorithm then moves to prune the tokenizer via the 'PruneTokenizer' function, where a streamlined vocabulary and updated merge rules are created based only on the tokens present in the collected set (Lines 12–24). Subsequently, the 'PruneModel' function handles the pruning of the language model itself, retaining only the embeddings and language model output layers corresponding to the preserved tokens (Lines 25–33). The final output, consisting of a reduced and optimized version of the original model and tokenizer, ensures the preservation of core functional capabilities while eliminating redundant elements (Lines 4–5).

Analysis. The computational expense associated with vocabulary pruning is primarily attributed to the tokenization process, which can vary from a few seconds to several minutes contingent upon the size of the corpus.

3.3. Layer pruning

Layer pruning involves the removal of entire layers to reduce its depth and computation, which is theoretically justified by the observed redundancy (Dalvi et al., 2020) within deep neural networks and the notable similarity (Gromov et al., 2024) across layers. In NLP, studies (Chen et al., 2024; Kim et al., 2024; Men et al., 2024; Song et al., 2024) reveal that the performance of LLMs remains largely intact even after the removal of many layers, indicating an underlying redundancy that pruning can exploit.

Algorithm 2 Layer pruning algorithm.

Input:
Original LLM $\mathcal{M}_{\text{ori}}$;
Code Generation Datasets $\mathcal{D}$;
Number of layers to prune k;
Output:
Pruned LLM $\mathcal{M}_{\text{pru}}$;

1: $\mathcal{D}_{\text{correct}} \leftarrow \text{FilterCorrectSamples}(\mathcal{D}, \mathcal{M}_{\text{ori}})$
2: $P_{\text{original}} \leftarrow \mathcal{M}_{\text{ori}}(\mathcal{D}_{\text{correct}})$
3: $\mathcal{M}_{\text{pru}} \leftarrow \mathcal{M}_{\text{ori}}$
4: **While** number of layers removed $< k$ **do**
5: $best\_layer, best\_score \leftarrow \text{FindBestLayerToPrune}(\mathcal{M}_{\text{pru}}, H^{(L)}_{\text{original}})$
6: $\mathcal{M}_{\text{pru}} \leftarrow \text{RemoveLayer}(\mathcal{M}_{\text{pru}}, best\_layer)$
7: **return** $\mathcal{M}_{\text{pru}}$
8: **Function** `FilterCorrectSamples` $\mathcal{D}, \mathcal{M}_{\text{ori}}$
9: $\mathcal{D}_{\text{correct}} \leftarrow \emptyset$
10: **for** $sample \in \mathcal{D}$ **do**
11: **if** isCorrectlyGenerated$(sample, \mathcal{M}_{\text{ori}})$ **then**
12: $\mathcal{D}_{\text{correct}} \leftarrow \mathcal{D}_{\text{correct}} \cup \{sample\}$
13: **return** $\mathcal{D}_{\text{correct}}$
14: **Function** `FindBestLayerToPrune` $\mathcal{M}_{\text{pru}}, P_{\text{original}}$
15: $best\_score \leftarrow \infty$
16: $best\_layer \leftarrow$ None
17: **for** each layer l in $\mathcal{M}_{\text{pru}}$ **do**
18: $\mathcal{M}_{\text{temp}} \leftarrow \text{RemoveLayer}(\mathcal{M}_{\text{pru}}, l)$
19: $P_{\text{pruned}} \leftarrow \mathcal{M}_{\text{temp}}(\mathcal{D}_{\text{correct}})$
20: $score \leftarrow \text{kl\_divergence}(P_{\text{pruned}}, P_{\text{original}})$
21: **if** score $<$ best_score **then**
22: $best\_score \leftarrow score$
23: $best\_layer \leftarrow l$
24: **return** $best\_layer, best\_score$

Algorithms. Our layer pruning approach diverges from existing methods that typically rely on layer similarity or language modeling capabilities as the primary criterion for pruning. We assert that the primary purpose of pruning should be to ensure that the model continues to deliver satisfactory performance, which we discussed in Section 3.1.

Furthermore, existing methods (Gromov et al., 2024; Ma et al., 2023; Men et al., 2024; Razzhigaev et al., 2024; Song et al., 2024) primarily focus on one-shot removal strategies, which determine the redundancy of each layer in a single computation pass and select the top_k layers or a contiguous block of k layers for removal. These methods are straightforward but may not account for the dynamic

changes in the model's state following the removal of layers. Beyond one-shot removal used in existing studies, our approach deploys an iterative process to meticulously evaluate each layer's contribution to the semantic representation of the final model output.

Algorithm 2 provides the pseudo-code to describe the detailed layer pruning process. The algorithm is designed to iteratively prune the layers of a LLM to reduce its complexity while maintaining performance. It starts by filtering the code generation dataset to identify samples for which the model correctly generates code, as detailed in the 'FilterCorrectSamples' function (Lines 8–13). The main pruning loop (Lines 4–6) continues until the specified number of layers, k, have been pruned. In each iteration, the 'FindBestLayerToPrune' function is used to determine the most dispensable layer by calculating the KL divergence between the probability distribution of the original and pruned models(Lines 14–24). Once identified, this layer is permanently removed from the model using 'RemoveLayer'. The process returns a pruned version of the original model, $\mathcal{M}_{\text{pru}}$.

Analysis. The computational cost associated with our layer pruning algorithm primarily stems from the iterative evaluation of layer removal and the subsequent similarity calculations between the pruned and original models' final layer representations.

3.4. FFN Pruning

FFN pruning is based on the observation that not all neurons in transformer models are equally vital (Zhu et al., 2023). Each layer in Transformer is composed of a GQA module and an FFN module, the GQA's inherent characteristics necessitate a specific ratio between key_value_heads and $attention\_heads$. Pruning any can lead to a significant degradation in model performance. Given this constraint, our focus shifts to the intermediate size of the FFN, which represents the expansion of the model's representational capacity and presents a more flexible target for pruning.

Algorithm. To streamline the FFN pruning process and minimize its computational overhead, we have devised a set of four heuristic rules to determine which neurons within the FFN to eliminate:

- **Top-K Neurons.** Retain the first K neurons, which are hypothesized to be the most influential.
- **Bottom-K Neurons.** Retain the last K neurons, assuming they may carry critical information not present in the initial neurons.
- **Middle-K Neurons.** Retain the middle K neurons, which might be less prone to noise and more stable in their contribution.
- **Random Sampling.** Randomly select K neurons to be retained, introducing an element of stochasticity to the pruning process.

Following these heuristics, we generate a structured pruning mask that guides the pruning process. Finally, we still choose the best rule through computing the similarity.

Analysis. Our approach to FFN pruning leverages heuristic rules, which significantly reduces the computational cost. The time required for this process is merely a matter of minutes, making it a highly efficient method for model optimization.

3.5. Performance recovery

After pruning some model parameters, it is crucial to implement effective strategies to recover the model's performance. For performance recovery, we fine-tune the pruned model on the training split of our CodeHarmony dataset (detailed in Section 4.3), which contains 15,800 samples for code generation and CoT tasks and 100,706 samples for output prediction tasks. The following pseudocode describes our proposed Performance Recovery Algorithm.

Algorithm 3 Performance recovery algorithm.

Input:
Original LLM $\mathcal{M}_{\text{ori}}$;
Pruned LLM $\mathcal{M}_{\text{pru}}$;
Training Dataset with Test Case Set $\mathcal{D} = [x, y, T]$;
Output:
Recovered LLM $\mathcal{M}_{\text{rec}}$;

for sample $s \in \mathcal{D}$ **do**
  $original\_code \leftarrow \mathcal{M}_{\text{ori}}.\text{generate}(s.x)$
  if TestPass(original_code, $s.T$) **then**
    $s.y \leftarrow original\_code$
    $\mathcal{D}$.updates(s)
$\mathcal{M}_{\text{rec}} \leftarrow \text{Train}(\mathcal{M}_{\text{pru}}, \mathcal{D})$
return $\mathcal{M}_{\text{rec}}$
Function `TestPass` $code, T$
  for test case $t \in T$ **do**
    if not isPass(code, t) **then**
      return False
return True

Algorithm 3 presents the pseudocode for the Performance Recovery process. The algorithm aims to improve the performance of the pruned model by leveraging the code generation capabilities of the original model. The process begins by iterating through the

provided training dataset D, which consists of samples including input data, expected outputs, and associated test cases (Lines 2–5). For each sample, the algorithm uses the original model $\mathcal{M}_{\text{ori}}$ to generate code based on the input data. If the generated code passes the corresponding test cases, the expected output in the dataset is replaced with this generated code.

By replacing outputs with semantically correct code generated by the original model, the overall quality of the dataset is consistently high, ensuring the training data is always reliable. Furthermore, the pruned model is able to fit the training data more efficiently, reducing loss more quickly and effectively during training. This enhances the convergence speed and helps the pruned model achieve better performance faster.

Finally, the pruned model $\mathcal{M}_{\text{pru}}$ is retrained on the updated dataset. This training aims to equip the pruned model with the necessary knowledge to handle previously challenging scenarios. The outcome is a recovered version of the pruned model, denoted as $\mathcal{M}_{\text{rec}}$.

Specifically, we refer to our proposed performance recovery method as PT. For comparison, the standard fine-tuning (FT) method involves fine-tuning the pruned model on the original CodeHarmony dataset (detailed in Section 4.3), which contains 15,800 samples for code generation and CoT tasks and 100,706 samples for output prediction tasks.

We employ the LoRA technique (Hu et al., 2021) for refined and efficient post-training of the pruned model. This method is the most efficient alternative to full-parameter fine-tuning (Liu et al., 2023b; Weyssow et al., 2023; Zhuo et al., 2024b), as it updates less than 5% of the model parameters and notably does not increase the total number of parameters. Post-training typically takes a few hours, contingent upon the size of the dataset.

4. Experiment setup

To assess the effectiveness of Flab-Pruner, we design the following three research questions (RQs):

RQ1 (Performance Comparison) This RQ is designed to evaluate the performance of Flab-Pruner compared to dense models and other structured pruning methods. Furthermore, we investigate the impact of three components proposed in Flab-Pruner and conducts a hyperparameter analysis.

RQ2 (Efficiency Comparison) This RQ is designed to examine the resource deployment efficiency of Flab-Pruner compared to dense models.

RQ3 (Robustness Analysis) This RQ is designed to assess the robustness of pruned models compared to dense models, particularly under various prompt perturbations.

4.1. Subject models

In our research, we select three state-of-the-art CodeLLMs, i.e., CodeQwen-1.5 and its two variants (NxCode and CodeSlerp), which are top-performing 7B-parameter models on the HumanEval leaderboard. These models are chosen for their strong performance and all three models share the same transformer-based architecture, making them ideal candidates for our comparative study of pruning techniques.

4.2. Downstream tasks

In our research, we focus on the generation tasks in code intelligence. We delve into the following three key tasks, each with its unique set of challenges and methodologies:

- **Code Generation.** This task (Chen et al., 2021) involves the generation of code snippets from a given natural language description and signature, utilizing a zero-shot learning strategy.
- **CoT Generation.** This task (Yang et al., 2024a) involves the generation of a chain of thought leading to the solution, given a natural language description and signature. It is executed through a one-shot learning approach, where the model learns from a single example.
- **Output Prediction.** This task (Gu et al., 2024a) focuses on predicting the output of a code snippet without execution, employing a two-shot learning approach. The model must deduce the expected output based on the code's logic, given the code and its corresponding input.

4.3. Datasets

For the code generation task, we focus on function-level code generation, primarily using the widely-used HumanEval[7] and OpenEval[8] for broader evaluation. For the CoT generation task, we employ the HumanEval-CoT and OpenEval-CoT datasets, developed by Yang et al. (2024a). For the output prediction task, we select the CRUX dataset[9], introduced by Meta (Gu et al., 2024b).

[7] https://huggingface.co/datasets/openai/openai_humaneval
[8] https://huggingface.co/datasets/NTUYG/openeval
[9] https://huggingface.co/datasets/cruxeval-org/cruxeval

Table 1
Statistical information of our selected tasks and corresponding datasets, existing datasets are limited in number of samples and have no training data.

Task	Dataset	Train	Valid	Test
Code Genaration (Zero-Shot)	HumanEval	-	-	164
	OpenEval	-	-	178
	CodeHarmony	15,800	200	153
CoT Genaration (One-Shot)	HumanEval	-	-	164
	OpenEval	-	-	178
	CodeHarmony	15,800	200	153
Output Prediciton (Two-Shot)	Crux-O	-	-	800
	CodeHarmony	100,706	-	452

Moreover, to address the limitations of existing datasets in terms of scale, we introduce CodeHarmony, a comprehensive benchmark dataset for code intelligence evaluation. The construction of CodeHarmony follows a systematic three-step process:

1. **Data Collection:** We aggregate function-level Python code from well-established open-source repositories, primarily the Evol dataset (Luo et al., 2024) and the OSS dataset (Wei et al., 2024). Using carefully designed regular expressions, we extract syntactically valid and self-contained functions to ensure code quality.
2. **Test Case Generation:** To validate the semantic correctness of the collected code, we implement a hybrid verification approach. We leverage state-of-the-art language models (GPT-4o and Gemini) to automatically generate diverse test cases. Specifically, we follow Beau and Crabbé (2024) to construct three test cases for each function to ensure the code's correctness.
3. **Chain-of-Thought Integration:** Inspired by recent advances in multi-agent alignment techniques (Yang et al., 2024a), we augment the dataset with CoT, which provides intermediate reasoning steps, enhancing the dataset's utility for evaluating models' reasoning capabilities in code understanding and generation tasks.

The resulting dataset comprises a diverse collection of code samples with corresponding test cases and CoT. Table 1 presents detailed statistics of CodeHarmony, demonstrating its comprehensive coverage and suitability for evaluating various aspects of code intelligence models.

4.4. Evaluation metrics

To assess the efficacy of the models in these tasks, we have adopted the following evaluation metrics.

- **Pass@1** quantifies the percentage of generated code snippets that successfully pass the associated test cases (Chen et al., 2021).
- **BLEU** evaluates lexical similarity overlap between texts using n-gram comparison (Papineni et al., 2002). We use BLEU-4 to assess the quality of the generated CoTs.
- **Exact Match (EM)** metric evaluates the precision by determining if the generated output exactly matches the expected output.

4.5. Implementation details.

For FFN pruning, we remove the 256 neurons of each layer. For layer pruning, we remove the 4 layers of the model. For vocabulary pruning, we reduce the vocabulary size from 92,416 to 17,176. Based on the above settings, we reduce the model's parameter size from 7.3 billion to 5.7 billion, a reduction of 22%.

For the model inference, we use greedy decoding method to calculate the Pass@1, BLEU and EM scores. For the remaining hyperparameters, we have meticulously tuned the following settings: a learning rate of 5e-4, a LoRA rank of 64, and a LoRA alpha of 32. Our experiments, using PyTorch and Transformers, were run on a system with an Intel Xeon Silver 4210 CPU and a GeForce RTX 3090 GPU. The model pruning and post-training took approximately 6 hours to complete.

5. Experiment result

5.1. RQ1: Performance comparison

One of the primary objectives of this study is to evaluate the effectiveness of Flab-Pruner in reducing model size while preserving the performance. To this end, we design a series of experiments to compare the performance of Flab-Pruner against other structured pruning methods as well as the original dense models. We consider five representative pruning methods, including ShortGPT, UIDL, Linearity, SLEB and LLM-pruner as the baselines. Each of these methods has a unique focus and explores different aspects of model pruning. For each baseline, we control the pruning ratio to be 20% (slightly lower than our method's 22%) to ensure a fair comparison across all methods.

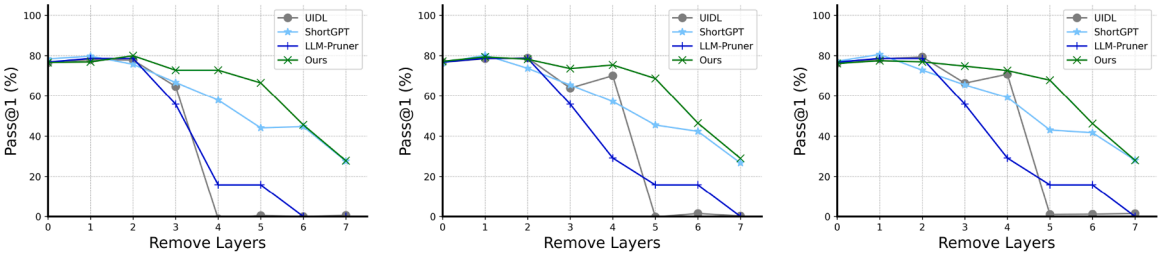
Table 2
Performance comparison of Flab-Pruner and other baselines, where HE represents HumanEval, OE represents OpenEval, and CH represents CodeHarmony. The best results (compared with other pruning methods) are highlighted in bold. The best results (compared with the dense model) are highlighted in gray. FT means traditional fine-tuning, and PT means post-training we proposed.

Model	Method	Code Generation(Pass@1)				CoT Generation(BLEU-4)				Output Prediciton(EM)		
		HE	OE	CH	Avg.	HE-CoT	OE-CoT	CH-CoT	Avg.	Crux-O	CH-O	Avg.
CodeQwen	Dense	77.44	42.13	60.78	60.12	33.95	41.14	23.81	32.97	37.13	77.43	57.28
	ShortGPT	42.68	20.79	41.83	35.10	15.92	19.02	11.41	15.45	27.63	59.29	43.46
	UIDL	0	0	0	0	0	0	0	0	0	0	0
	Linearity	0	0	0.65	0.22	0	0	0	0	0	0	0
	SLEB	20.73	17.42	49.67	29.27	1.72	5.58	2.92	3.41	18.75	42.70	30.73
	LLM-pruner	15.85	10.67	37.25	21.25	3.88	8.33	3.99	5.40	16.50	25.44	20.97
	Flab-Pruner	**75.00**	**37.64**	**64.05**	**58.90**	**31.42**	**35.88**	**20.80**	**29.37**	**31.75**	**70.58**	**51.17**
	Flab-Pruner w FT	76.22	36.52	66.63	59.79	32.17	34.65	25.80	30.87	40.38	77.10	58.74
	Flab-Pruner w PT	78.05	39.89	67.97	61.97	34.37	34.67	26.60	31.88	40.63	77.43	59.03
NxCode	Dense	77.44	41.57	62.09	60.37	32.87	39.90	24.15	32.31	37.00	77.43	57.22
	ShortGPT	40.85	22.47	38.56	33.96	19.03	20.00	12.53	17.19	28.75	59.51	44.13
	UIDL	0	0	0	0	0	0	0	0	0	0	0
	Linearity	0	0	0.65	0.22	0	0	0	0	0	0	0
	SLEB	20.12	17.98	47.71	28.60	2.42	5.88	3.37	3.89	18.38	41.15	29.77
	LLM-pruner	17.07	9.55	36.60	21.07	3.31	8.96	4.13	5.47	15.00	23.23	19.12
	Flab-Pruner	**74.39**	**35.96**	**64.05**	**58.13**	**31.03**	**35.31**	**21.76**	**29.37**	**32.63**	**70.58**	**51.61**
	Flab-Pruner w FT	75.51	38.20	62.36	58.69	30.86	34.88	24.14	29.96	40.13	77.10	58.62
	Flab-Pruner w PT	81.71	39.33	64.05	61.70	32.96	37.40	24.71	31.69	40.63	77.21	58.92
CodeSlerp	Dense	77.44	43.26	61.44	60.71	32.18	40.49	24.51	32.39	37.25	76.77	57.01
	ShortGPT	41.46	21.35	39.61	34.14	18.03	20.00	12.53	16.85	29.38	60.00	44.69
	UIDL	0	0	0	0	0	0	0	0	0	0	0
	Linearity	0	0	0.65	0.22	0	0	0	0	0	0	0
	SLEB	21.34	18.54	48.37	29.42	2.12	5.22	3.12	3.49	18.44	41.25	29.85
	LLM-pruner	16.89	11.35	38.60	22.28	3.65	9.25	5.11	6.00	15.25	24.25	19.75
	Flab-Pruner	**75.00**	**35.96**	**64.05**	**58.34**	**31.48**	**35.96**	**21.83**	**29.76**	**32.25**	**70.35**	**51.30**
	Flab-Pruner w FT	73.17	37.20	65.36	58.58	31.66	34.03	25.81	30.50	37.50	75.60	56.55
	Flab-Pruner w PT	78.66	37.64	66.01	60.77	32.27	37.17	25.96	31.80	37.63	75.66	56.65

- **ShortGPT** (Men et al., 2024) prunes layers guided by the *cosine similarity* between layer representations.
- **UIDL** (Gromov et al., 2024) prunes layers by evaluating the *angular distance* between layer representations.
- **Linearity** (Razzhigaev et al., 2024) prunes layers by evaluating the *linear relationship* of layer representations.
- **SLEB** (Song et al., 2024) prunes layers that have the least impact on the model by calculating the *perplexity metric*.
- **LLM-pruner** (Ma et al., 2023) prunes layers that have the least impact on the model by utilizing gradients derived from Taylor's formula.

(1) Compared with single component pruning methods: Our empirical results, shown in Table 2, demonstrate the performance of Flab-Pruner across various tasks. Without post-training, Flab-Pruner achieves the highest performance in the code generation task. Taking HumanEval as an example, the Pass@1 of Flab-Pruner on the CodeQwen model is 75.00%, while the best Pass@1 of other baseline methods is 42.68%, compared to the relative performance improvement of 75.73%. In the CoT generation task, Flab-Pruner also performs better, with a BLEU-4 of 31.42% on the CodeQwen model, while the best BLEU-4 of other baseline methods is 15.92%, compared to the relative performance improvement of 97.98%. For the code output prediction task, Flab-Pruner also outperforms other methods. Taking Crux-O as an example, the EM of Flab-Pruner on the CodeQwen model is 31.75%, while the best EM of other baseline methods is 27.63%, compared to the relative performance improvement of 14.93%.

(2) Compared with other pruning objectives: As shown in Fig. 2, we compare the effect of pruning different number of layers on the model performance and find that the more layers are pruned, the more the model performance decreases. Furthermore, we conduct an in-depth analysis of our proposed pruning objectives in Layer Pruning. We conduct comprehensive comparisons between our KL divergence-based approach and several alternative pruning objectives. When examining the impact of the progressively increasing number of pruned layers, we observe that with minimal pruning (1-2 layers), all methods maintain performance comparable to the original model. However, as more layers are removed, significant differences emerge between pruning objectives. The angular distance-based method (UIDL) deteriorates most dramatically, with pass@1 metrics dropping from above 60% to 0% when pruning beyond a certain threshold. Both Taylor formula-based gradients (LLM-Pruner) and cosine similarity-based methods (ShortGPT) show substantial performance degradation as pruning increases, while our KL divergence approach maintains higher performance with a more gradual decline. When pruning ratios become extremely high, all methods eventually experience severe performance drops,



(a) Results on CodeQwen model (b) Results on Nxcode model (c) Results on CodeSlerp model

Fig. 2. Performance comparison of different layer pruning models with different number of layers pruned..

Table 3
Ablation Comparison of Flab-Pruner with different pruning components.

Model	Method	Code Generation(Pass@1)			CoT Generation(BLEU-4)			Output Prediciton(EM)	
		HE	OE	CH	HE-CoT	OE-CoT	CH-CoT	Crux-O	CH-O
CodeQwen	Dense	77.44	42.13	60.78	33.95	41.14	23.81	37.13	77.43
	+ Vocab Pruning	77.44	42.70	62.09	33.85	41.44	22.21	36.50	75.66
	+ FFN (Top-K)	74.39	37.64	64.05	31.42	35.88	20.80	31.75	70.58
	Bottom-K	71.34	35.39	62.09	31.51	37.13	21.54	31.37	70.35
	Middle-K	67.68	35.96	62.09	30.16	37.06	21.23	31.00	71.02
	FIM	75.00	35.96	64.05	31.10	36.96	21.83	31.87	71.90
	Random	67.68	34.83	62.09	31.96	35.66	22.51	32.50	70.35
NxCode	Dense	77.44	41.57	62.09	32.87	39.90	24.15	37.00	77.43
	+ Vocab Pruning	76.22	44.38	62.09	34.16	40.27	24.00	36.38	75.44
	+ FFN (Top-K)	74.39	35.96	64.05	31.03	35.31	21.76	32.63	70.58
	Bottom-K	71.34	34.83	62.09	31.03	35.99	22.33	31.87	72.35
	Middle-K	66.46	37.08	60.78	29.93	34.29	22.19	31.75	70.58
	FIM	74.39	36.52	64.71	31.22	35.67	22.05	32.75	71.02
	Random	62.80	35.96	58.17	32.44	35.73	21.47	33.00	71.90
CodeSlerp	Dense	77.44	43.26	61.44	32.18	40.49	24.51	37.25	76.77
	+ Vocab Pruning	74.39	41.57	61.44	34.42	40.31	24.76	36.88	75.00
	+ FFN (Top-K)	75.00	35.96	64.05	31.48	35.96	21.83	32.25	70.35
	Bottom-K	71.34	35.96	61.44	31.43	36.32	22.58	32.13	71.24
	Middle-K	67.07	37.64	62.09	31.10	36.05	22.71	31.37	71.24
	FIM	75.61	36.52	64.71	31.65	36.22	22.10	32.50	70.80
	Random	70.73	35.96	61.44	30.48	36.47	22.45	32.38	70.35

though our approach consistently degrades more gracefully. These results demonstrate that KL divergence captures layer importance for code intelligence tasks more effectively than both gradient-based and representation similarity-based methods.

These performance differences are primarily attributed to the inherent limitations in the layer selection and importance assessment strategies of other methods. SLEB's reliance on perplexity metrics may not fully correspond to code functionality accuracy. ShortGPT, UIDL and Linearity's isolated layer evaluations fail to capture the collective impact of layers on overall model performance. Additionally, LLM-Pruner's dependence on back propagation for gradient computation introduces unnecessary computational overhead for code intelligence tasks. This suggests that our proposed layer pruning strategy can better capture the importance of the each layer, thus alleviating the problem of inconsistent pruning objectives mentioned in Section 1 and better guiding model pruning.

(3) Strengths in our proposed performance recovery strategy: Compared to traditional fine-tuning method, our proposed strategy achieves better performance recovery. As shown in Table 2, after post-training, Flab-Pruner achieves the best performance across all tasks in three models. Taking HumanEval as an example, the Pass@1 of Flab-Pruner on the CodeQwen model is 78.05%, while the best Pass@1 of traditional fine-tuning method is 76.22%, compared to the relative performance improvement of 2.40%.

(4) Ablation Study: To further understand the individual contributions of different components within our approach, we conduct an ablation study to evaluate the impact of each component on the overall performance of the pruned models. Table 3 demonstrates that vocabulary pruning contributes minimally to performance changes in the pruned models, as evidenced by our ablation analysis. The FFN pruning strategy, on the other hand, shows a slight decrease in performance compared to the dense model.

Furthermore, we conduct a comparative analysis between our heuristic rule-based methods and a neuron importance evaluation approach using Fisher Information Matrix (FIM Ma et al. (2023)), which is shown in Table 3. The FIM-based approach achieves similar performance compared to our Top-K method but with a much higher computational cost. Our proposed method requires approximately one minute while the FIM-based approach requires approximately 30 minutes. In addition, the FIM-based approach requires additional forward and backward passes through the network for each batch of data.

Table 4
Efficiency analysis of Flab-Pruner.

Value	BF16			FP8			INT4		
	Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT
GPU	13.55	10.72	10.72	7.51	5.52	5.52	4.57	3.02	3.02
Token/s	30	38	38	34	44	44	37	47	47
CO_2	2.14	1.84	1.84	2.02	1.67	1.67	1.94	1.57	1.57
FLOPs	7.04T	5.64T	5.64T	3.52T	2.82T	2.82T	1.76T	1.41T	1.41T

Based on these results, we validate our design choice and demonstrate that, while contribution-based method may offer theoretical advantages, our heuristic rule-based approach provides comparable performance with significantly less computational requirements.

Summary of RQ1 A comprehensive evaluation of Flab-Pruner's performance across three code intelligence tasks indicate that Flab-Pruner can retain 97% of the original code generation performance on average after pruning 22% of parameters and achieves the same or even better performance after post-training.

5.2. RQ2: Efficiency analysis

In this RQ, we first compare the efficiency of the pruned model to the original model, focusing on key metrics such as GPU usage, speed, CO2 emissions, and FLOPs. These metrics are crucial for practical deployment considerations in software engineering (Shi et al., 2024a; Wei et al., 2023), where GPU utilization and token processing speed are calculated by gpu_poor[10], CO_2 emissions are calculated using codecarbon[11], and FLOPs are calculated using the Calflops[12].

Considering that our pruning method can be combined with state-of-the-art quantization methods to further improve efficiency, we evaluate the efficiency of Flab-Pruner with different precision, including BF16, FP8, and INT4 (we implement INT4 by AutoAWQ[13]) in Table 4.

The "GPU Usage" column reflects GPU memory consumption during model inference, with the dense model requiring 13.55 G and Flab-Pruner, both with and without post-training, operating within a margin of 10.72G. This suggests that the original model, operating at FP16 precision, can only be deployed on GPUs with more than 12G of memory, while the pruned model can run on GPUs with less than 12G of memory. Furthermore, the GPU usage of Flab-Pruner at INT4 precision is only 3.02G, indicating that it can run on GPUs with 4G of memory. The "Token/s" metric indicates the throughput of the models, i.e., the number of tokens processed per second, directly related to the speed of inference. Here, the dense model processes 30 tokens per second, whereas Flab-Pruner manages a throughput of 38 tokens per second, suggesting a more efficient use of computational resources. The "Emissions" column provides insight into the environmental impact of the models, with the dense model emitting 2.14 g of evaluating the HumanEval, while Flab-Pruner emits 1.84 g of CO_2. The "FLOPs" (Floating Point Operations per Second) column provides insight into the computational intensity of the models. The dense model demands a significant 7.04 trillion FLOPs for inference, whereas Flab-Pruner operates at a reduced computational intensity of 5.64 trillion FLOPs.

In addition to model efficiency, we also compare the performance of the models at different precisions in Table 5. Flab-Pruner strikes a balance between lowered resource use and sustained performance. The results show that the performance of the models at the INT4 precision is slightly lower than that at the BF16 precision, with minimal impact on the model's performance.

End-to-End Efficiency Analysis: To provide a comprehensive view of the efficiency gains across the entire model lifecycle, we analyze the one-off computational cost of pruning and fine-tuning versus the recurring benefits during inference. This analysis addresses the complete efficiency landscape from model compression to deployment.

As shown in Table 6, the entire pruning and recovery process takes approximately 6 hours on RTX 3090. While this represents a high initial investment, our analysis shows that these costs are quickly amortized through inference efficiency gains.

For the computational efficiency analysis, we need to consider the one-off cost of pruning versus the recurring savings during inference. In terms of FLOPs, each inference with the dense model requires 7.04T FLOPs, while the pruned model requires only 5.64T FLOPs, saving 1.4T FLOPs (19.9%) per inference. Considering the dense model's approximate throughput of 7.04T FLOPs during our experiments, the 6-hour pruning and fine-tuning process consumes approximately $6 \times 3600 \times 7.04 = 152,064$T FLOPs (we assume that the frequency of calculation is 1 time/second). To determine the break-even point, we calculate:

$$\text{Break-even Point} = \frac{\text{One-time Cost (FLOPs)}}{\text{Per-inference Savings (FLOPs)}} = \frac{152,064T}{1.4T} \approx 108,617 \text{ inference runs} \tag{8}$$

Namely, the computational investment is redeemed after approximately 108,617 inference runs.

For real-world deployments where models are typically invoked millions of times especially in production environments or as part of developer tools and code assistants, this initial cost is readily justified. For example, a deployment serving 10,000 inference

[10] https://rahulschand.github.io/gpu_poor/
[11] https://github.com/mlco2/codecarbon
[12] https://pypi.org/project/calflops/
[13] https://github.com/casper-hansen/AutoAWQ

Table 5
Performance analysis of Flab-Pruner in different precision.

Model	Precision	Code Generation(Pass@1)			CoT Generation(BLEU-4)			Output Prediciton(EM)	
		HE	OE	CH	HE-CoT	OE-CoT	CH-CoT	Crux-O	CH-O
CodeQwen	FP8 Dense	76.83	41.57	61.44	33.51	40.40	23.78	37.25	78.10
	FP8 Flab-Pruner	75.61	38.76	63.40	30.35	36.11	21.15	31.00	70.80
	FP8 w PT	79.27	39.33	66.01	34.20	36.21	26.01	41.38	77.43
	INT4 Dense	75.00	42.13	60.78	23.88	23.12	14.78	36.88	76.33
	INT4 Flab-Pruner	71.95	38.20	60.78	27.87	31.60	19.12	31.25	69.25
	INT4 w PT	77.44	38.08	66.67	32.80	34.02	24.94	40.50	75.22
NxCode	FP8 Dense	76.22	41.01	61.44	33.01	39.60	24.48	37.50	78.10
	FP8 Flab-Pruner	75.61	40.45	61.44	30.06	35.8	20.77	31.75	70.80
	FP8 w PT	80.49	38.20	63.40	33.78	36.95	25.46	40.63	77.21
	INT4 Dense	75.00	42.70	61.44	30.67	37.02	23.26	36.63	76.11
	INT4 Flab-Pruner	69.51	38.76	62.09	29.68	34.75	19.73	31.50	69.69
	INT4 w PT	74.39	38.20	62.75	31.88	34.94	24.10	40.63	75.22
CodeSlerp	FP8 Dense	77.44	41.57	61.44	33.14	39.16	24.32	36.88	77.88
	FP8 Flab-Pruner	75.61	38.76	63.40	30.93	36.35	21.50	31.25	70.58
	FP8 w PT	81.71	41.01	64.05	32.87	35.92	25.46	37.75	76.33
	INT4 Dense	76.83	43.82	60.78	28.24	34.37	18.52	37.3	76.11
	INT4 Flab-Pruner	67.07	37.08	60.13	26.88	32.23	19.67	30.25	69.25
	INT4 w PT	78.66	39.89	67.97	32.91	36.90	25.91	38.13	75.00

Table 6
Comprehensive efficiency analysis across different stages of Flab-Pruner deployment in One-time Cost and Inference Efficiency.

Stage	Time (hours)	GPU Memory (GB)	FLOPs (T)
Dense	-	13.55	7.04
+ Vocabulary Pruning	0.02	12.15	6.80
+ Layer Pruning	0.3	11.10	5.98
+ FFN Pruning	0.04	10.72	5.64
+ Performance Recovery	5.5	10.72	5.64
Flab-Pruner	5.86	10.72	5.64

Table 7
Robustness Comparison of Flab-Pruner and dense model in the presence of RECODE and EvoEval perturbations.

Perturbed	Method	CodeQwen			NxCode			CodeSlerp		
		Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT
ReCode	Format	85.98	78.66	82.32	82.32	76.83	80.49	85.37	75.61	81.10
	Func_Name	75.61	75.00	77.44	75.00	71.95	76.83	76.22	74.39	79.88
	NatGen	86.59	77.44	81.10	85.98	76.83	79.88	85.37	77.44	79.27
	NlaugEnter	64.63	60.98	65.85	64.02	59.76	66.46	64.02	59.76	64.63
EvoEval	Tool_Use	54.00	54.00	56.00	54.00	53.00	58.00	55.00	53.00	56.00
	Combine	27.00	23.00	18.00	25.00	24.00	19.00	27.00	23.00	23.00
	Subtle	62.00	61.00	62.00	60.00	62.00	64.00	60.00	59.00	65.00
	Creative	35.00	25.00	33.00	34.00	24.00	36.00	36.00	24.00	35.00
	Difficult	37.00	31.00	30.00	35.00	28.00	30.00	35.00	27.00	31.00

requests daily would redeem the initial pruning investment in approximately 11 days, after which all efficiency gains represent net computational savings.

Summary of RQ2 The efficiency comparison underscores the practical advantages of Flab-Pruner in resource-constrained settings, providing a sustainable solution for deploying powerful code intelligence models with reduced computational overhead. When considering the full model lifecycle, the initial pruning costs are quickly amortized, resulting in significant efficiency gains for real-world deployments.

5.3. RQ3: Robustness analysis

To thoroughly evaluate the robustness of the pruned models, we design a comprehensive experimental framework that builds upon the pioneering ReCode work of Wang et al. (2023). Leveraging the code generation task as a critical benchmark, we aim to

Table 8
Performance comparison on BigCodeBench (Pass@11%).

Model	Dense	Flab-Pruner	
		w/o PT	w/ PT
CodeQwen	39.60	36.20	39.80
NxCode	39.60	36.10	40.00
CodeSlerp	39.80	37.10	39.50

assess model performance under various perturbed conditions. Four token-level perturbation methods are constructed in HumanEval: format, func_name, natgen, and nlaugenter. Each of these perturbation methods is specifically tailored to probe the model's resilience to specific types of noise, offering a comprehensive perspective on the models' ability to withstand real-world disturbances.

- **Format.** This perturbation method introduces noise in the code format, such as insert the newline or replace space indent with tab.
- **Func_name.** This perturbation method alters function names in the code, including applying character-level or word-level natural text transformations on component words.
- **Natgen.** This perturbation method introduces code syntax noise, such as inserting the deadcode or swaping operand.
- **Nlaugenter.** This perturbation method introduces natural language noise in the docstrings, such as applying SynonymSubstitution or BackTranslation.

The empirical results, as shown in Table 7, indicate that the pruned models exhibit a slight decrease in performance under partially token-level perturbed conditions compared to the dense model. The performance of the three pruned models under the four perturbation methods is slightly lower than that of the dense model, with a maximum decrease of less than 10%. However, after post-training, the performance of the pruned models is even better than the dense model under certain perturbations. Taking the CodeQwen model as an example, after post-training, the performance of Flab-Pruner under the Func_Name perturbation is 77.44%, while the performance of the dense model is 75.61%. This indicates that our post-training strategy can enhance the robustness of the model.

Furthermore, we also conduct experiments using the EvoEval dataset (Xia et al., 2024a). This dataset includes a variety of semantic-altering operations:

- **Tool_use.** This perturbation method introduces a new problem with a main issue and additional helper functions that can assist in solving it.
- **Combine.** This perturbation method merges two distinct problems by integrating concepts from both.
- **Subtle.** This perturbation method makes minor adjustments to the original problem, such as inverting or substituting a requirement.
- **Creative.** This perturbation method develops a more imaginative problem by incorporating stories or unique narratives.
- **Difficult.** This perturbation method increases complexity by adding extra constraints, replacing common requirements with less common ones, or adding more reasoning steps.

We find that the Flab-Pruner performs similarly in Tool_Use and Subtle. After post-training, the performance of the pruned models is even better than the dense model under Tool_Use and Subtle. However, in Combine and Difficult, Flab-Pruner shows slightly lower performance than the dense model. Overall, our experimental results indicate that Flab-Pruner performs well in terms of robustness, maintaining or even enhancing model performance under certain disturbances. We attribute this primarily to the post-training phase.

Summary of RQ3 The results reveal that the pruned models show a slight decrease in robustness compared to the dense model in some cases. This suggests that future research needs to focus on robustness in addition to maintaining model performance.

6. Discussion

6.1. Evaluation on bigcodebench

To ensure a thorough evaluation, we conduct additional experiments on BigCodeBench (Zhuo et al., 2024a), a more diverse and challenging benchmark designed to evaluate code models across various dimensions of programming tasks.

BigCodeBench consists of 1140 function-level tasks specifically designed to challenge language models in following instructions and composing multiple function calls from 139 different libraries as tools. Unlike HumanEval and OpenEval, BigCodeBench provides a more rigorous evaluation framework with comprehensive test coverage, complex user-oriented instructions, open-ended problem solving and tool composition.

As shown in Table 8, our findings on BigCodeBench align with our main results:

- Without PT, the pruned models maintain approximately 91-93% of the dense models' performance, demonstrating reasonable retention of capabilities despite parameter reduction.
- With PT, the pruned models achieve performance comparable to or slightly better than the dense models, with CodeQwen and NxCode showing small improvements of 0.2-0.4% in Pass@1 scores, while CodeSlerp experiences a negligible decrease of 0.3%.

Table 9
Generalizability of Flab-Pruner on other programming languages in HumanEval-XL.

Language	CodeQwen			NxCode			CodeSlerp		
	Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT	Dense	Flab-Pruner	w PT
Kotlin	56.25	28.75	42.50	53.75	32.50	40.00	53.75	27.50	40.00
Ruby	46.25	22.50	33.75	46.25	23.75	36.25	46.25	22.50	36.25
Swift	48.75	31.25	35.00	48.75	32.50	35.00	45.00	27.50	35.00

Table 10
Impact of Pass@1 on different pruning strategy combinations on the CodeQwen model (w/o PT).

Pruning Strategy	Model Parameters	HumanEval	OpenEval	CodeHarmony
None (Dense)	7.25B	77.44	42.13	60.78
Vocab only	6.63B	77.44	42.70	62.09
Layer only	6.44B	75.00	37.64	64.05
FFN only	7.16B	76.54	39.80	66.00
Vocab + Layer	5.82B	77.20	42.13	64.92
Vocab + FFN	6.54B	74.39	37.64	64.05
Layer + FFN	6.35B	76.24	36.52	66.83
All three (Flab-Pruner)	5.73B	76.22	36.52	66.63

- The models' ability to compose function calls from different libraries remains intact after pruning, indicating that Flab-Pruner preserves the models' semantic understanding and tool utilization capabilities.

The consistent performance patterns across this more challenging benchmark corroborate our findings from HumanEval and OpenEval, demonstrating that Flab-Pruner maintains its effectiveness even on tasks requiring complex reasoning and tool composition. It also demonstrates the generalizability of our approach across different task complexities and problem-solving scenarios.

6.2. Evaluation on other programming languages

To assess the generalizability of Flab-Pruner beyond Python, we extend our evaluation to include other popular programming languages: Kotlin, Ruby, and Swift. These languages represent diverse paradigms and syntactical structures, providing a robust test for Flab-Pruner. We conduct experiments using the HumanEval-XL (Peng et al., 2024) benchmark, which offers code generation tasks across multiple languages, similar to the primary evaluation setup for Python. Table 9 presents the performance of the dense (original) models, the pruned models without performance recovery (Flab-Pruner), and the pruned models with performance recovery (Flab-Pruner w PT) across CodeQwen, NxCode and CodeSlerp on Kotlin, Ruby and Swift.

Due to the fact that Flab-Pruner utilizes the CodeHarmony dataset as the basis for evaluating model component importance during the pruning phase, the performance of the models pruned by Flab-Pruner without post-training (columns labeled 'Flab-Pruner') could only maintain between 50–65% of their original performance on these new languages. This consistent drop across Kotlin, Ruby, and Swift highlights that the initial pruning, guided by a Python-centric dataset, indeed affects performance on unseen language distributions.

In contrast, the models subject to post-training (columns labeled 'w PT'), despite being trained solely on the Python dataset, demonstrate remarkable abilities to generalize and recover performance on other programming languages. They consistently restore the pruned model's performance to 75–80% of the original dense model's capability. While this recovery does not reach the same level as observed on Python, we posit that expanding the training dataset to include samples from these programming languages would further enhance the performance of the pruned models. This demonstrates the broad applicability of Flab-Pruner's pruning and recovery mechanisms across diverse code generation tasks, even with limited cross-language fine-tuning.

6.3. Impact between strategies

To address the question of how our three pruning strategies (vocabulary pruning, layer pruning and FFN pruning) interact with each other, we conduct a detailed ablation analysis of their individual and combined effects, which

is performed on the CodeQwen model. Table 10 presents the results on the HumanEval benchmark and the parameter reduction achieved in each configuration. Furthermore, we also report the model parameters of the dense model and the pruned model in each configuration.

Our findings reveal several key insights:

- **Individual Impact**: Vocabulary pruning demonstrates the least performance degradation, maintaining full performance on HumanEval (77.44%) and even slightly improving on OpenEval (42.70% vs. 42.13%). In contrast, layer pruning shows the most

Fig. 3. Comparison between Top-K (index-based) and FIM (importance-based) pruning methods across different layers of the CodeQwen model. Red circles indicate neurons retained by the Top-K method, while green crosses show neurons retained by the FIM method. The heatmap represents neuron importance values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

significant impact on HumanEval (dropping to 75.00%) and OpenEval (dropping to 37.64%). Interestingly, all three individual strategies actually improve performance on CodeHarmony, with FFN pruning providing the largest gain (66.00% vs. 60.78%).

- **Pairwise Combinations**: The combination of vocabulary and layer pruning shows a synergistic effect, with minimal impact on HumanEval (77.20%) and no degradation on OpenEval (42.13%). However, when FFN pruning is combined with either vocabulary or layer pruning, we observe more significant performance drops on HumanEval and OpenEval. For CodeHarmony, all pairwise combinations yield substantial improvements, with layer + FFN pruning showing the highest gain (66.83%).
- **Complementary Effects**: When all three strategies are combined (Flab-Pruner), we observe a balanced trade-off. The performance on HumanEval (76.22%) and OpenEval (36.52%) shows moderate decreases, while CodeHarmony performance improves substantially (66.63%). This suggests that the strategies have task-dependent interactions, with some benchmark tasks being more sensitive to specific types of pruning than others.

These findings indicate that the three pruning strategies target different aspects of the model's parameter space and have complementary effects. While each strategy contributes to parameter reduction, their careful combination, followed by effective performance recovery, is essential for maintaining or even enhancing model capabilities across diverse coding tasks.

6.4. FFN Pruning heuristics

In Section 3.4, we introduce heuristic rules for FFN pruning (Top-K, Bottom-K, Middle-K, and Random Sampling) which are primarily based on neuron index order. To provide a clearer justification for these index-based pruning strategies, we conduct a comparative analysis between our heuristic approach and a neuron importance evaluation approach using Fisher Information Matrix (FIM).

Fig. 3 illustrates the relationship between neuron importance (measured by FIM) and neuron index position across five representative layers (layers 1, 7, 14, 21 and 28) of the CodeQwen model. Our analysis reveals several key insights:

First, we observe a moderate overlap between neurons selected by our index-based Top-K method and those selected by the FIM-based importance method. Specifically, the overlap percentages are 41.4% for layer 1, 50.8% for layer 7, 42.6% for layer 14, 43.0% for layer 21, and 46.1% for layer 28. This suggests that while neuron index position does not perfectly correlate with functional importance as determined by FIM, there exists a substantial relationship across different layers of the model. The visualization shows that important neurons (identified by FIM) are not randomly distributed but tend to cluster in certain regions, with many appearing in the initial portion of the index range. This pattern helps explain why our Top-K heuristic, which retains neurons from the beginning of the index range, achieves competitive performance despite its simplicity.

Furthermore, as shown in Table 3, the performance difference between our Top-K method and the FIM-based approach is minimal across all three models, with differences typically less than 1% on evaluation metrics. However, the computational cost differs dramatically, where our Top-K method requires only about one minute of processing time compared to approximately 30 minutes for the FIM approach, representing a 30x reduction in computational overhead.

These findings validate our design choice of using simple heuristic rules for FFN pruning. While contribution-based methods such as FIM may offer theoretical advantages in identifying the most important neurons with greater precision, our rule-based approach

provides comparable performance with significantly reduced computational requirements. This efficiency is particularly valuable in the context of large language models, where computational resources are often limited.

6.5. Threats to validity

Threats to Internal Validity. The first internal threat involves the potential for implementation errors in Flab-Pruner. We counteract the possibility of errors in Flab-Pruner with comprehensive code reviews and trusted libraries like PyTorch and Transformers. The second internal threat pertains to the accuracy of the implemented baselines. To mitigate this risk, we reproduced all baselines using their shared scripts. Furthermore, to ensure a fair evaluation and uphold the integrity of the model architectures, we only focused on **structured pruning** methods in our comparative analysis.

Threats to External Validity. Our primary external validity concern is the representativeness of the datasets used. We selected high-quality datasets to reflect the domain accurately, focusing on Python due to its widespread support in code LLMs. We plan to expand to other languages and levels in future work to improve the generalizability of our results. Furthermore, we considered three state-of-the-art Code LLMs in our study, which may limit the generalizability of our findings to other models. We will expand the assessment to include additional models to improve the external validity of the results.

Threats to Construct Validity. The main challenge in construct validity is the choice of metrics for automated evaluation. We included diverse metrics such as Pass@1, BLEU and EM to comprehensively assess model performance, providing different perspectives on their capabilities.

7. Related work

7.1. Code intelligence

The success of models like BERT (Devlin et al., 2019) in NLP has inspired the creation of pre-trained models for code processing. Models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CuBERT (Kanade et al., 2020) are designed for generating informative code embeddings vital for tasks like software defect detection (Yang et al., 2021). Building on the success of GPT (Floridi and Chiriatti, 2020) in NLP, models like CodeGPT (Lu et al., 2021), trained on datasets like CodeSearchNet (Husain et al., 2019), have shown promise in coding tasks. Recent advancements in deep learning have led to models like CodeGeeX (Zheng et al., 2023) and CodeLlama (Roziere et al., 2023), excelling in complex programming challenges and demonstrating superior performance.

Code Generation. The code generation task, where code is produced from natural language, is a hot topic (Li et al., 2022; Svyatkovskiy et al., 2020; Yang et al., 2023; Zhang et al., 2024), with models trained for specific goals like next-token prediction or the “filling in the middle” technique for contextual code completion, as seen in InCoder (Fried et al., 2023) and StarCoder (Li et al., 2023b; Lozhkov et al., 2024).

The field has seen the rise of diverse models, each with unique training approaches and strengths. Notable examples include WizardCoder (Luo et al., 2024), OpencodeInterpreter (Zheng et al., 2024), and Magicoder (Wei et al., 2024), all aimed at improving precision and efficiency in code generation challenges.

CoT Generation. CoT generation is about crafting logical steps in natural language to reach a code solution, improving output reliability through clear reasoning. Researchers (Zhuo, 2024) have adapted this for code intelligence, with approaches like Jiang’s self-planning (Jiang et al., 2023) and Li’s structured CoT (Li et al., 2023a) to tackle complex coding challenges.

Code Output Prediciton. Predicting code output from inputs is a tough test of a language model’s comprehension skills (Jain et al., 2024). Gu et al. (2024a) found top models did well in HumanEval but not in output prediction using CRUXEval.

7.2. Model compression

Model compression reduces model size, boosts transformer efficiency, and maintains performance through techniques like knowledge refinement, quantization, and pruning.

Knowledge distillation. Knowledge distillation trains a compact student model to emulate a larger teacher model. In software engineering, Compressor (Shi et al., 2022) employs task-specific distillation to enhance transformer efficiency with neural architecture search. Yang et al. (2024a) introduced COTTON to boost lightweight models by transferring reasoning skills from larger models using rules and agent alignment.

Quantization. Quantization trims neural network precision to optimize memory and efficiency. Wei et al. (2023) examined quantized models in code generation, noting performance trade-offs. Xiao et al. (2023) introduced SmoothQuant for weight and activation quantization. Gptq (Frantar et al., 2022) utilizes second-order info for quantization, and Qlora (Dettmers et al., 2024) backpropagates through a 4-bit model into Low Rank Adapters. Quantization can enhance efficiency but may affect accuracy.

Pruning. Pruning boosts model efficiency by creating sparser or smaller models. Unstructured pruning zeros parameters for sparsity, like SparseGPT (Frantar and Alistarh, 2023), treating it as a sparse regression issue, but risking irregular structures. Structured pruning removes components based on criteria, with tools like LLM-Pruner (Ma et al., 2023) using gradients to cut less critical parts, ShearedLLaMA (Xia et al., 2024b) applying targeted pruning and dynamic loading, and ShortGPT (Men et al., 2024) removing whole layers.

Considering that existing pruning methods may compromise model performance, we propose Flab-Pruner, a comprehensive pruning approach that maintains model efficiency and performance in code intelligence tasks. Moreover, Flab-Pruner utilizes the

LoRA technique for effective post-training, guaranteeing that the pruned models attain performance levels akin to the original dense models.

8. Conclusion and future work

The development and evaluation of Flab-Pruner underscore the viability of structural pruning for LLMs in the generative coding tasks. Our approach has demonstrated that it is possible to significantly reduce the computational footprint of LLMs without compromising their core capabilities. By pruning 22% of parameters, Flab-Pruner retains 97% of the original code generation performance on average, which further achieves the same or even better performance after post-training. The pruned models also exhibit enhanced efficiency in GPU usage, Flops, CO2 emissions, and token processing speed, aligning with the goals of green software engineering. Moreover, the comprehensive evaluation, including robustness analysis, assures that the pruned models maintain a high standard of performance.

In future work, we shall develop more efficient model pruning methods to reduce the computational overhead of our current approaches. Potential directions include cherry data selection techniques (Li et al., 2024) to select a smaller dataset for evaluation and transfer-based methods that leverage importance patterns from previously pruned models to new architectures.

Additionally, we plan to explore more Code LLMs and evaluate the effectiveness of Flab-Pruner on a broader range of generative coding tasks. We will release more pruned models on the open source platform to facilitate the deployment of pruned models in real-world software engineering scenarios. Moreover, we plan to expand Flab-Pruner to support a broader range of other fields, such as natural language processing, computer vision, and audio processing.

CRediT authorship contribution statement

Guang Yang: Writing – original draft, Software, Methodology, Data curation; **Yu Zhou:** Validation, Funding acquisition, Conceptualization; **Xiangyu Zhang:** Validation, Data curation; **Wei Cheng:** Validation, Software, Data curation; **Ke Liu:** Writing – review & editing, Validation; **Xiang Chen:** Writing – review & editing, Validation; **Terry Yue Zhuo:** Writing – review & editing, Validation; **Taolue Chen:** Writing – review & editing, Validation, Methodology.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (NSFC, No. 61972197 and No. 62372232), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Short-term Visiting Program of Nanjing University of Aeronautics and Astronautics for Ph.D. Students Abroad (No. 240501DF16). T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03).

References

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F. et al. (2023). Qwen technical report. arXiv:2309.16609.

Beau, N., & Crabbé, B. (2024). Codeinsight: A curated dataset of practical coding solutions from stack overflow. In *Findings of the association for computational linguistics ACL 2024* (pp. 5935–5947).

Chen, C., Zhang, P., Zhang, H., Dai, J., Yi, Y., Zhang, H., & Zhang, Y. (2020). Deep learning on computational-resource-limited platforms: A survey. *Mobile Information Systems*, *2020*(1), 8454327.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Oliveira, P. H. P. D., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. et al. (2021). Evaluating large language models trained on code. arXiv:2107.03374.

Chen, W., Su, Y., Shen, Y., Chen, Z., Yan, X., & Wang, W. (2019). How large a vocabulary does text classification need? a variational approach to vocabulary selection. In *Proceedings of NAACL-HLT* (pp. 3487–3497).

Chen, X., Hu, Y., & Zhang, J. (2024). Compressing large language models by streamlining the unimportant layer. arXiv:2403.19135.

Dalvi, F., Sajjad, H., Durrani, N., & Belinkov, Y. (2020). Analyzing redundancy in pretrained transformer models. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)* (pp. 4908–4926).

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2024). Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, *36*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. 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)* (pp. 4171–4186).

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International conference on software engineering: Future of software engineering (ICSE-foSE)* (pp. 31–53). IEEE.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536–1547).

Floridi, L., & Chiriatti, M. (2020). Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, *30*(4), 681–694.

Frantar, E., & Alistarh, D. (2023). Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning* (pp. 10323–10337). PMLR.
Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022). Optq: Accurate quantization for generative pre-trained transformers. In *The eleventh international conference on learning representations*.
Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., & Lewis, M. (2023). Incoder: A generative model for code infilling and synthesis. In *The eleventh international conference on learning representations*.
Gromov, A., Tirumala, K., Shapourian, H., Glorioso, P., & Roberts, D. A. (2024). The unreasonable ineffectiveness of the deeper layers. arXiv:2403.17887.
Gu, A., Roziere, B., Leather, H. J., Solar-Lezama, A., Synnaeve, G., & Wang, S. (2024a). Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first international conference on machine learning*.
Gu, A., Roziere, B., Leather, H. J., Solar-Lezama, A., Synnaeve, G., & Wang, S. (2024b). Cruxeval: A benchmark for code reasoning, understanding and execution. In *Iclr 2024 workshop on navigating and addressing data problems for foundation models*.
Gu, Q. (2023). Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 2201–2203).
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S. et al. (2021). GraphcodeBERT: Pre-training code representations with data flow. In *International conference on learning representations*.
Hort, M., Grishina, A., & Moonen, L. (2023). An exploratory literature study on sharing and energy use of language models for source code. In *2023 ACM/IEEE International symposium on empirical software engineering and measurement (ESEM)* (pp. 1–12). IEEE.
Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2023). Large language models for software engineering: A systematic literature review. arXiv:2308.10620.
Hu, E. J., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W. et al. (2021). LoRA: Low-rank adaptation of large language models. In *International conference on learning representations*.
Huang, H., Zheng, O., Wang, D., Yin, J., Wang, Z., Ding, S., Yin, H., Xu, C., Yang, R., Zheng, Q. et al. (2023). ChatGPT for shaping the future of dentistry: The potential of multi-modal large language model. *International Journal of Oral Science, 15*(1), 29.
Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. arXiv:1909.09436.
Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Koushik, S., & Stoica, I. (2024). Livecodebench: Holistic and contamination free evaluation of large language models for code. arXiv:2403.07974.
Jiang, X., Dong, Y., Wang, L., Zheng, F., Shang, Q., Li, G., Jin, Z., & Jiao, W. (2023). Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.
Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *International conference on machine learning* (pp. 5110–5121). PMLR.
Kim, B.-K., Kim, G., Kim, T.-H., Castells, T., Choi, S., Shin, J., & Song, H.-K. (2024). Shortened llama: A simple depth pruning for large language models. arXiv:2402.02834.
Kirova, V. D., Ku, C. S., Laracy, J. R., & Marlowe, T. J. (2024). Software engineering education must adapt and evolve for an llm environment. In *Proceedings of the 55th ACM technical symposium on computer science education v. 1* (pp. 666–672).
Li, J., Li, G., Li, Y., & Jin, Z. (2023a). Structured chain-of-thought prompting for code generation. 2305.
Li, M., Zhang, Y., Li, Z., Chen, J., Chen, L., Cheng, N., Wang, J., Zhou, T., & Xiao, J. (2024). From quantity to quality: Boosting LLM performance with self-guided data selection for instruction tuning. In *Proceedings of the 2024 conference of the north american chapter of the association for computational linguistics: Human language technologies (volume 1: Long papers)* (pp. 7595–7628).
Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J. et al. (2023b). Starcoder: may the source be with you! arXiv:2305.06161.
Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A. et al. (2022). Competition-level code generation with alphacode. *Science*, *378*(6624), 1092–1097.
Liu, F., Shu, P., Jin, H., Ding, L., Yu, J., Niu, D., & Li, B. (2013). Gearing resource-poor mobile devices with powerful clouds: Architectures, challenges, and applications. *IEEE Wireless communications*, *20*(3), 14–22.
Liu, J., Li, L., Xiang, T., Wang, B., & Qian, Y. (2023a). Tcra-llm: Token compression retrieval augmented large language model for inference cost reduction. In *Findings of the association for computational linguistics: EMNLP 2023* (pp. 9796–9810).
Liu, J., Sha, C., & Peng, X. (2023b). An empirical study of parameter-efficient fine-tuning methods for pre-trained code models. In *2023 38th IEEE/ACM international conference on automated software engineering (ASE)* (pp. 397–408). IEEE.
Liu, Z., Sun, M., Zhou, T., Huang, G., & Darrell, T. (2019). Rethinking the value of network pruning. In *International conference on learning representations*.
Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y. et al. (2024). Starcoder 2 and the stack v2: The next generation. arXiv:2402.19173.
Lu, G., Ju, X., Chen, X., Pei, W., & Cai, Z. (2024). Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, *212*, 112031.
Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D. et al. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth conference on neural information processing systems datasets and benchmarks track (round 1)*.
Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., & Jiang, D. (2024). Wizardcoder: Empowering code large language models with evol-instruct. In *The twelfth international conference on learning representations*.
Ma, X., Fang, G., & Wang, X. (2023). Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, *36*, 21702–21720.
Mason-Williams, G., & Dahlqvist, F. (2024). What makes a good prune? maximal unstructured pruning for maximal cosine similarity. In *The twelfth international conference on learning representations*.
Men, X., Xu, M., Zhang, Q., Wang, B., Lin, H., Lu, Y., Han, X., & Chen, W. (2024). Shortgpt: Layers in large language models are more redundant than you expect. arXiv:2403.03853.
Nair, S., Yang, E., Lawrie, D., Mayfield, J., & Oard, D. W. (2023). Blade: Combining vocabulary pruning and intermediate pretraining for scaleable neural clir. In *Proceedings of the 46th international ACM SIGIR conference on research and development in information retrieval* (pp. 1219–1229).
Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024). Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th international conference on software engineering* (pp. 1–13).
Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the association for computational linguistics* (pp. 311–318).
Peng, Q., Chai, Y., & Li, X. (2024). Humaneval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 joint international conference on computational linguistics, language resources and evaluation (LREC-COLING 2024)* (pp. 8383–8394).
Razzhigaev, A., Mikhalchuk, M., Goncharova, E., Gerasimenko, N., Oseledets, I., Dimitrov, D., & Kuznetsov, A. (2024). Your transformer is secretly linear. arXiv:2405.12250.
Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J. et al. (2023). Code llama: Open foundation models for code. arXiv:2308.12950.
Sallou, J., Durieux, T., & Panichella, A. (2024). Breaking the silence: The threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th international conference on software engineering: New ideas and emerging results* (pp. 102–106).
Shi, J., Yang, Z., Kang, H. J., Xu, B., He, J., & Lo, D. (2024a). Greening large language models of code. In *Proceedings of the 46th international conference on software engineering: Software engineering in society* (pp. 142–153).
Shi, J., Yang, Z., & Lo, D. (2024b). Efficient and green large language models for software engineering: Vision and the road ahead. arXiv:2404.04566.

Shi, J., Yang, Z., Xu, B., Kang, H. J., & Lo, D. (2022). Compressing pre-trained models of code into 3 mb. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering* (pp. 1–12).
Song, J., Oh, K., Kim, T., Kim, H., Kim, Y., & Kim, J.-J. (2024). Sleb: Streamlining llms through redundancy verification and elimination of transformer blocks. arXiv:2402.09025.
Sun, M., Liu, Z., Bair, A., & Kolter, J. Z. (2023). A simple and effective pruning approach for large language models. In *Workshop on efficient systems for foundation models@ ICML2023*.
Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 1433–1443).
Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. et al. (2023). Llama: Open and efficient foundation language models. arXiv:2302.13971.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.
Wang, C., Cho, K., & Gu, J. (2020a). Neural machine translation with byte-level subwords. In *Proceedings of the AAAI conference on artificial intelligence* (pp. 9154–9160). (*34*).
Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., Kumar, V., Tan, S., Ray, B., Bhatia, P. et al. (2023). Recode: Robustness evaluation of code generation models. In *The 61st annual meeting of the association for computational linguistics*.
Wang, W., Chen, W., Luo, Y., Long, Y., Lin, Z., Zhang, L., Lin, B., Cai, D., & He, X. (2024). Model compression and efficient inference for large language models: A survey. arXiv:2402.09748.
Wang, Z., Wohlwend, J., & Lei, T. (2020b). Structured pruning of large language models. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)* (pp. 6151–6162).
Wei, X., Gonugondla, S. K., Wang, S., Ahmad, W., Ray, B., Qian, H., Li, X., Kumar, V., Wang, Z., Tian, Y. et al. (2023). Towards greener yet powerful code generation via quantization: An empirical study. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 224–236).
Wei, Y., Wang, Z., Liu, J., Ding, Y., & Zhang, L. (2024). Magicoder: Empowering code generation with oss-instruct. In *Forty-first international conference on machine learning*.
Weyssow, M., Zhou, X., Kim, K., Lo, D., & Sahraoui, H. (2023). Exploring parameter-efficient fine-tuning techniques for code generation with large language models. arXiv:2308.10462.
Xia, C. S., Deng, Y., & Zhang, L. (2024a). Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via LLM.
Xia, M., Gao, T., Zeng, Z., & Chen, D. (2024b). Sheared LLaMA: Accelerating language model pre-training via structured pruning. In *The twelfth international conference on learning representations*.
Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., & Han, S. (2023). Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning* (pp. 38087–38099). PMLR.
Yang, G., Zhou, Y., Chen, X., Zhang, X., Xu, Y., Han, T., & Chen, T. (2023). A syntax-guided multi-task learning approach for turducken-style code generation. *Empirical Software Engineering*, *28*(6), 141.
Yang, G., Zhou, Y., Chen, X., Zhang, X., Zhuo, T. Y., & Chen, T. (2024a). Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*.
Yang, G., Zhou, Y., Yu, C., & Chen, X. (2021). Deepscc: Source code classification based on fine-tuned roberta. arXiv:2110.00914.
Yang, Y., Cao, Z., & Zhao, H. (2024b). Laco: Large language model pruning via layer collapse. arXiv:2402.11187.
Yang, Z., Cui, Y., & Chen, Z. (2022). Textpruner: A model pruning toolkit for pre-trained language models. In *Proceedings of the 60th annual meeting of the association for computational linguistics: System demonstrations* (pp. 35–43).
Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., & Zhang, Y. (2024). A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, (p. 100211).
Yuan, L., Chen, Y., Cui, G., Gao, H., Zou, F., Cheng, X., Ji, H., Liu, Z., & Sun, M. (2023). Revisiting out-of-distribution robustness in NLP: benchmarks, analysis, and LLMs evaluations. *Advances in Neural Information Processing Systems*, *36*, 58478–58507.
Zhang, X., Zhou, Y., Yang, G., Han, T., & Chen, T. (2024). Context-aware code generation with synchronous bidirectional decoder. *Journal of Systems and Software*, *214*, 112066.
Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y. et al. (2023). Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining* (pp. 5673–5684).
Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., & Yue, X. (2024). Opencodeinterpreter: Integrating code generation with execution and refinement. arXiv:2402.14658.
Zhu, X., Li, J., Liu, Y., Ma, C., & Wang, W. (2023). A survey on model compression for large language models. arXiv:2308.07633.
Zhuo, T. Y. (2024). Ice-score: Instructing large language models to evaluate code. In *Findings of the association for computational linguistics: EACL 2024* (pp. 2232–2242).
Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I. et al. (2024a). Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. arXiv:2406.15877.
Zhuo, T. Y., Zebaze, A., Suppattarachai, N., von Werra, L., de Vries, H., Liu, Q., & Muennighoff, N. (2024b). Astraios: Parameter-efficient instruction tuning code large language models. arXiv:2401.00788.