MFTCoder: Boosting Code LLMs with Multitask Fine-Tuning

Bingchang Liu
bingchang.lbc@antgroup.com
Ant Group
Hangzhou, China

Chaoyu Chen
chris.ccy@antgroup.com
Ant Group
Hangzhou, China

Zi Gong
gongzi.gz@antgroup.com
Ant Group
Hangzhou, China

Cong Liao
liaocong.lc@antgroup.com
Ant Group
Hangzhou, China

Huan Wang
wh262583@antgroup.com
Ant Group
Hangzhou, China

Zhichao Lei
leizhichao.lzc@antgroup.com
Ant Group
Hangzhou, China

Ming Liang
liangming.liang@antgroup.com
Ant Group
Hangzhou, China

Dajun Chen
chendajun.cdj@antgroup.com
Ant Group
Hangzhou, China

Min Shen
miaozhen.sm@antgroup.com
Ant Group
Hangzhou, China

Hailian Zhou
hailian.zhl@antgroup.com
Ant Group
Hangzhou, China

Wei Jiang
jonny.jw@antgroup.com
Ant Group
Hangzhou, China

Hang Yu*
hyu.hugo@antgroup.com
Ant Group
Hangzhou, China

Jianguo Li*
lijg.zero@antgroup.com
Ant Group
Hangzhou, China

ABSTRACT

Code LLMs have emerged as a specialized research field, with remarkable studies dedicated to enhancing model's coding capabilities through fine-tuning on pre-trained models. Previous fine-tuning approaches were typically tailored to specific downstream tasks or scenarios, which meant separate fine-tuning for each task, requiring extensive training resources and posing challenges in terms of deployment and maintenance. Furthermore, these approaches failed to leverage the inherent interconnectedness among different code-related tasks. To overcome these limitations, we present a multi-task fine-tuning framework, MFTCoder, that enables simultaneous and parallel fine-tuning on multiple tasks. By incorporating various loss functions, we effectively address common challenges in multi-task learning, such as data imbalance, varying difficulty levels, and inconsistent convergence speeds. Extensive experiments have conclusively demonstrated that our multi-task fine-tuning approach outperforms both individual fine-tuning on single tasks and fine-tuning on a mixed ensemble of tasks. Moreover, MFTCoder offers efficient training capabilities, including efficient data tokenization modes and parameter efficient fine-tuning (PEFT) techniques, resulting in significantly improved speed compared to traditional fine-tuning methods. MFTCoder seamlessly integrates with several mainstream open-source LLMs, such as CodeLLama and Qwen. Our MFTCoder fine-tuned CodeFuse-DeepSeek-33B claimed the top spot on the Big Code Models Leaderboard ranked by WinRate as of January 30, 2024. MFTCoder is open-sourced at https://github.com/codefuse-ai/MFTCOder

*Corresponding Authors

KDD '24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-0490-1/24/08
https://doi.org/10.1145/3637528.3671609

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**;
• **Software and its engineering** → *Software notations and tools.*

KEYWORDS

Large Language Model; Code Generation; Multi-task Learning

ACM Reference Format:
Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Wei Jiang, Hang Yu, and Jianguo Li. 2024. MFTCoder: Boosting Code LLMs with Multitask Fine-Tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3637528.3671609

1 INTRODUCTION

The paradigm-shifting emergence of ChatGPT[1], powered by both GPT-3.5 and GPT-4 [46], has set ablaze the landscape of research and development in the realm of large language models (LLMs). This

[1]https://openai.com/blog/chatgpt

breakthrough has further sparked the interest in leveraging LLMs for code understanding and generation, commonly referred to as Code LLMs. By pretraining on extensive code data sources such as the Github public data, these Code LLMs can acquire comprehensive contextual representations that can be applied to various code-related tasks [61].

While the pretraining stage of (Code) LLMs seek to ensure their generalizability to different downstream tasks, the subsequent fine-tuning stage typically only adapt the (Code) LLMs to a specific task or a scenario. However, this approach overlooks two critical challenges. **Firstly, it involves resource-intensive individual finetuning of large LLMs for each task, which hinders efficient deployment in production. Secondly, the interrelated nature of code domain tasks suggests that joint finetuning can enhance performance compared to separate finetuning.** It is therefore imperative to conduct multitask finetuning, enabling simultaneous handling of all tasks while leveraging the strengths of related tasks to enhance performance.

As an illuminating example, suppose we have two related tasks: code completion and code summarization. Code completion involves predicting the next line of code based on a partial code snippet, while code summarization aims to generate a concise human-readable summary of a given code snippet. Traditionally, separate models would be fine-tuned for each task, resulting in resource-intensive duplication. However, code completion and code summarization have inherent connections. Completion of a code snippet relies on understanding the overall functionality and purpose, while generating an accurate summary requires comprehending the structure, dependencies, and intended functionality. By employing multitask learning, a single model can be trained to jointly learn both tasks, leveraging shared knowledge and patterns, leading to improved performance on both tasks. The model understands the contextual dependencies between code elements, aiding in predicting the next snippet and generating informative summaries. Furthermore, multitask learning offers additional benefits beyond individual task performance: the shared representation between tasks helps mitigate overfitting, promote better generalization, and enhance the model's ability to handle data scarcity for specific tasks. If code completion has a larger training dataset than code summarization, the model can leverage the abundance of completion data to enhance performance in summarization, effectively addressing data scarcity challenges. Multitask learning even enables the model to handle unseen but related tasks without specific training data. Overall, multitask learning allows models to jointly learn multiple related tasks, benefiting from shared knowledge, improving performance, enhancing generalization, and handling data scarcity.

Despite the importance of multitask learning for finetuning, only a handful of existing studies have explored this approach in the domain of NLP [1, 5, 49]. These studies incorporate multi-task data and merge it for large-scale model learning, without explicitly separating the tasks. Unfortunately, these studies tend to prioritize tasks with larger sample sizes, disregarding tasks with smaller sample sizes. Furthermore, they fail to ensure equal convergence speed among tasks, leading to over-optimization of some tasks and under-optimization of others.

Our paper concentrates on multitask fine-tuning (MFT) of code LLMs, aiming to balance attention across tasks of different sizes and achieve uniform optimization. We focus on Code LLMs due to the inherent correlations in coding tasks, introducing our method, MFTCoder. We highlight that MFTCoder is easily adaptable to a wide range of related NLP tasks. To boost MFTCoder's efficiency, we utilize Parameter-Efficient Fine-Tuning (PEFT) [24] strategies like LoRA [25] and QLoRA [18]. Our experiments show that multitask models using MFT outshine those fine-tuned separately or with combined data sets. We also prove MFTCoder's effectiveness across a spectrum of baseline pretrained LLMs, like Qwen [7], Baichuan [8], Llama [54], Llama 2 [55], StarCoder [34], and others. Notably, applying MFTCoder to DeepSeek-Coder-33B [23] secured the top spot in terms of WinRate on the Big Code Models Leaderboard [2] as of January 30, 2024. At Antgroup, leveraging models trained with MFTCoder, we have developed CodeFuse, a programming assistant featuring web and IDE plugins. It boasts 12,000 weekly active users, with AI generating nearly 80,000 lines of code weekly. The main contributions of this paper are:

- We introduce MFTCoder, an innovative multitask fine-tuning strategy that simultaneously adapts LLMs to diverse coding tasks. Our approach specifically addresses the issues of data imbalance and inconsistent convergence rates that are common in previous multitask fine-tuning methods.
- We validate MFTCoder on various baseline pretrained models, like Qwen [7], Llama 1/2 [54, 55], StarCoder [34], CodeLLama [50], CodeGeex2 [63], DeepSeek-Coder [23] and Mixtral [28], demonstrating its compatibility with different baseline models.
- Our extensive experiments conclusively demonstrate that our MFT method outperforms traditional single-task fine-tuning and data merging. Specifically, the MFTCoder-fine-tuned CodeFuse-DeepSeek-33B [3] claimed the top spot on the Big Code Models Leaderboard ranked by WinRate as of January 30, 2024.

2 RELATED WORKS

2.1 Code LLMs

Coding capability serves as a critical criterion for evaluating general large language models (LLMs) in code-related tasks. Notable performance on the widely-used HumanEval dataset [12], a benchmark for code generation, has been observed across various models, including LaMDA [53], PaLM [15], PaLM 2 [3], ChatGPT, and GPT-4 [46]. In particular, GPT-4 has set a remarkable record of 67.0% pass@1 score. However, their closed-source nature limits their availability and hinders further collaborative advancements. In contrast, recent open-source LLMs, including LLaMA [54], LLaMA 2 [55], Qwen [7], and Phi-1.5 [35], have demonstrated notable progress in code-related tasks, with commentable scores of 23.7%, 29.9%, 32.3%, and 41.4% respectively. Despite this progress, their performance still lags behind the state-of-the-art closed-source models.

On the other hand, LLMs specifically designed for code-related tasks, often referred to as code LLMs, have also undergone significant developments. Alongside closed-source Code LLMs such as Codex [12], Code-Davinci [12], AlphaCode [36], PaLM-Coder [15], and PanGu-Coder [16], open-source alternatives like including SantaCoder [2], Phi-1.0 [22], CodeGeeX-2 [63], StarCoder [34], Code LLaMA [50] have showcased competitive performance with their

[2] https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
[3] https://huggingface.co/codefuse-ai/CodeFuse-DeepSeek-33B

closed-source counterparts. Notably, CodeLLama-34B-Python [50] obtains a score of 53.7% on HumanEval. Apart from pretraining, another intriguing approach to further enhancing Code LLMs is instruction fine-tuning, as showcased by CodeT5+ [57], Phi-1.0 [22], OctoPack [45], and WizardCoder [43]. By leveraging carefully curated high-quality instruction datasets, these methods exhibit the potential of fine-tuning to enhance code generation capabilities.

2.2 Multitask Learning

Multitask learning (MTL) [10, 17] is a powerful machine learning strategy with a proven track record of boosting model performance and tackling various challenges [17]. By training models on several related tasks, MTL capitalizes on shared knowledge and patterns, which results in better generalization and accuracy. There are two main MTL paradigms: hard parameter sharing [13, 26, 29, 39–41, 62], where models share weights across tasks, and soft parameter sharing [21, 31, 42, 47, 52, 58], featuring task-specific models with distinct weights. Hard parameter sharing is especially suitable for large language models (LLMs) due to their vast parameter counts, enabling them to manage multiple tasks with a shared parameter set. Recent strides in MTL include Google's T5 [49], which experimented with MTL in 2020, and Meta's Muppet [1] in 2021, which introduced multi-task learning between pretraining and fine-tuning, known as pre-fine-tuning (PFT) [1]. PFT was shown to enhance pretrained models' performance on various downstream tasks and expedited fine-tuning, although a minimum of 15 tasks is advised for best performance. Google's subsequent ExT5 [5] increased the task count to 107 and demonstrated that a large task number in pretraining can offset potential task interference, leading to excellent outcomes and outperforming T5. Notably, these studies mainly merged multi-task data for the model to learn without separately addressing tasks, sometimes neglecting issues like data imbalance and varying convergence speeds common in MTL. In our paper, we present MFTCoder, an approach that finetunes LLMs for multiple tasks while effectively handling these issues.

3 APPROACH

In this section, we present MFTCoder [4], our multi-task fine-tuning framework, and its key components design.

3.1 MFTcoder Framework

MFTCoder is designed to flexibly adapt LLMs to various new contexts, optimizing their performance for specific scenarios. The process begins by breaking down a new scenario into smaller, focused tasks that target specific abilities. For example, in the realm of coding LLMs, the broader goal of improving code-related capabilities can be distilled into tasks such as code completion, text-to-code generation, unit test case generation, code repair, code debugging, and cross-language translation. Our hands-on experience shows that MFTCoder can adeptly manage multi-task scales from single tasks to dozens or even hundreds. Compiling fine-tuning datasets is essential for each task, yet data gathering can be challenging for some. To address this, MFTCoder utilizes Self-Instruct [56] techniques and Agents to generate instructional datasets. MFTCoder is designed to fine-tune multiple tasks concurrently, skillfully handling extensive datasets for fine-tuning and guaranteeing efficient training processes. This capability is particularly important in MFT scenarios, which involve a variety of tasks and substantial amounts of data. It offers two effective data tokenization options and employs PEFT techniques to boost training efficiency. In multi-task learning, MFTCoder tackles task imbalances, such as uneven data distribution, diverse task difficulty, and different convergence rates, by introducing or adapting loss functions to balance tasks. Acknowledging the varying strengths and functionalities of different LLMs, MFTCoder enables the choice of appropriate architectures based on the scenario for best results. Adapted to mainstream LLama [54], LLama 2 [55], CodeLLama [50], Qwen [7], Baichuan 1/2 [8], ChatGLM 2 [20], CodeGeeX 2 [63], GPT-NEOX [9], CodeFuse [19], StarCoder [34], DeepSeek [23], Mixtral [27, 28], and others, we continually enhance and broaden compatibility with more models.

Figure 1 depicts the MFTCoder framework. In the following sections, we will delve into the components: dataset construction, tokenization modes, PEFT, and balanced loss functions.

3.2 Dataset Construction and Tokenization

For tasks requiring complex data collection, we use the Self-Instruct technique [56] to create fine-tuning data for code-related tasks in MFTCoder. This means giving tailored prompts to GPT-3.5 or GPT-4 that clearly outline our data generation needs. Additionally, we've integrated the self-instruct method with the Textbook approach [22], to produce the Code Exercises datasets for downstream coding tasks.

We have two implementation strategies: the multi-turn conversation via two agents (i.e., Camel [32]), and the single-turn approach through the ChatGPT API. In the multi-turn setup with Camel, we launch two agents with distinct roles to simulate a conversation that produces theme-based instructional data. For example, in creating Python exercises, one agent acts as the 'teacher' (mirroring ChatGPT's user role) and the other as the 'student' (akin to ChatGPT's assistant role). The teacher asks questions, and the student responds with solutions, generating multiple exercises until we hit task goals or ChatGPT's input length limit. Due to this limit, we can't use large, thematic questions directly but subdivide them into specific Python topics (*e.g.* binary search trees), and run individual Camel sessions for each.

The multi-turn method offers high automation but comes with the expense of running two agents, each making multiple ChatGPT API calls. To cut costs, we've developed a cost-effective single-turn approach detailed in Figure 2. Starting with a base of seed topics (like hundreds of Python concepts), we pair these with set prompt templates to craft structured task prompts. To enhance the variety and accuracy of these prompts, we use Camel to refine them. We then generate task-specific instructions (*e.g.* Python problems on binary search trees) and use ChatGPT for solutions. After compiling and deduplicating the pairs, we create an Python exercise dataset [5].

Tokenization is vital for training LLMs effectively, as it splits texts into manageable units, shaping data use and training efficiency. Traditional SFT (Supervised Fine-tuning) tokenization pads

[4]https://github.com/codefuse-ai/MFTCoder

[5]https://huggingface.co/datasets/codefuse-ai/CodeExercise-Python-27k

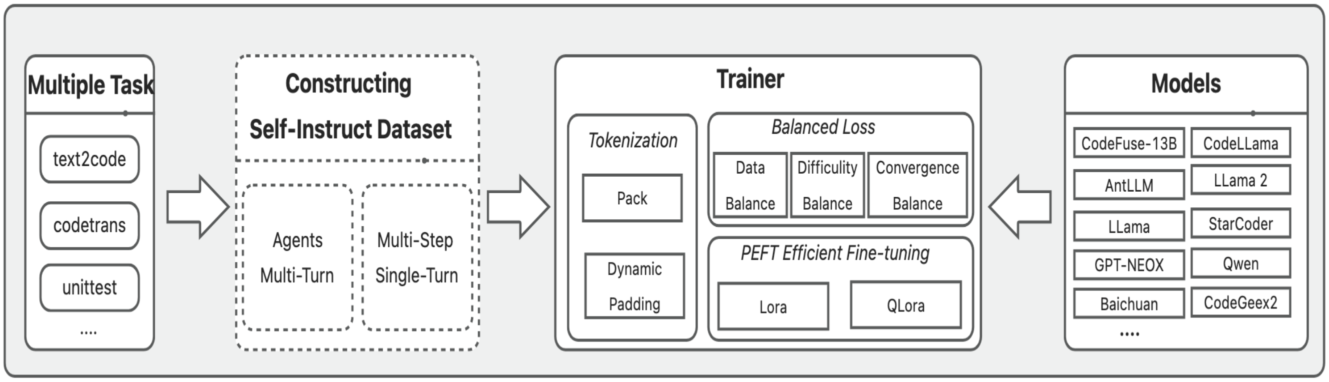


Figure 1: Overview of MFTCoder framework.

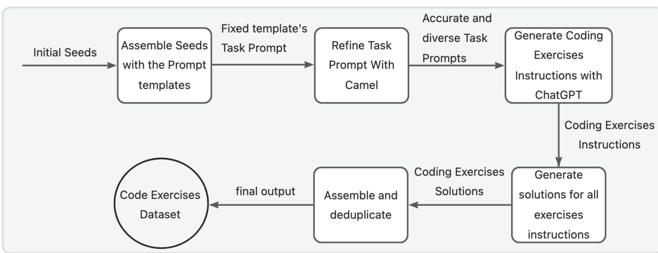


Figure 2: Data Generation Approach for Code Exercises Datasets using Single-turn Conversation Scheme.

all batch samples to the same max length, leading to considerable quantities of meaningless paddings.For instance, using our CodeFuse-13B [19]'s tokenizer across 35 tasks results in an average of 92.22% padding tokens. To reduce this waste, we have implemented dynamic padding and pack modes. Dynamic padding adjusts to the longest sample per micro batch, reducing padding and potentially doubling speed for online tokenization (shown by Figure 5b in Appendix A). Pack mode, similar to Llama 2's approach, packs samples into seq-length windows, minimizing padding below 10% (demonstrated by Figure 5c in Appendix A). This mode increases training speed while preserving effectiveness, suitable for both online and offline scenarios. Additionally, this approach allows samples from the same task positioned earlier in the same window to provide a few-shot capability.

3.3 Parameter Efficient Fine-tuning (PEFT)

Large-scale models with billions of parameters and multi-task learning scenarios with numerous tasks require a significant number of fine-tuning samples. Full fine-tuning of such models on massive datasets presents two main challenges: the demand for vast storage and computational power, and the risk of catastrophic forgetting. To tackle these, MFTCoder utilizes PEFT [24] to achieve efficient, resource-light fine-tuning quickly.

MFTCoder supports two PEFT methods: Lora [25] and QLora [18]. Lora adds an auxiliary branch with a low-rank adaptation where only the expansion matrix $A \in \mathbb{R}^{r\times d}$ and reduction matrices $B \in \mathbb{R}^{d\times r}$ are trained, keeping the original model's weights $W \in \mathbb{R}^{d\times d}$ fixed. The matrix product BA is then added to the original model W, resulting in the newly trained model. QLora builds on Lora by adding 4-bit quantization of $W \in \mathbb{R}^{d\times d}$ and fine-tuning a small number of adapter weights, enabling even larger models to be fine-tuned with limited GPU resources. For instance, MFTCoder can fine-tune a 70B model on a single A100 GPU with 80GB VRAM.

3.4 Multitask Fine-Tuning with Balanced Losses

As a multi-task learning framework, MFTCoder, as described in Section 2, faces a significant challenge of data imbalance, task heterogeneity, and varying convergence speeds. To address these challenges, MFTCoder incorporates a set of loss functions specifically designed to alleviate these imbalances.

To manage data imbalance, we ensure each task's samples are used once per epoch. To prevent bias towards tasks with more data, we've devised a weighted loss strategy. We offer two weighting methods: one by sample count and the other by the count of valid tokens contributing to the loss. The first is simpler, but may not work well for tasks with greatly varying token counts, like simple "yes/no" answers or single-choice questions. The second, based on valid token count, better addresses this variance, as outlined in Equation 1. In Equation 1, N represents the total number of tasks, M_i denotes the number of samples for the i-th task, T_{ij} signifies the count of valid tokens (i.e., tokens involved in loss calculation) for the j-th sample of the i-th task, and t_{ijk} refers to the k-th valid token of the j-th sample for the i-th task.

$$\mathcal{L}(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \frac{\sum_{j=1}^{M_i} \sum_{k=1}^{T_{ij}} -\log(p_\theta(t_{ijk}))}{\sum_{j=1}^{M_i} T_{ij}} \tag{1}$$

To address the issue of task heterogeneity, we drew inspiration from the focal loss approach [37] and incorporated it into MFTCoder. We implemented two different levels of focal loss functions to cater to different granularities. One operates at the sample level, as shown in Equation 2, while the other operates at the task level, as shown in Equation 3.

$$\mathcal{L}_2(\theta) = \min_{\theta} \frac{\sum_{i=1}^{N} \sum_{j=1}^{M_i} -\alpha_i * (1 - P_{ij})^{\gamma} * Q_{ij}}{\sum_{i=1}^{N} M_i}, \tag{2}$$

$$P_{ij} = \frac{1}{T_{ij}} \sum_{k=1}^{T_{ij}} P_{ijk}, \quad Q_{ij} = \frac{1}{T_{ij}} \sum_{k=1}^{T_{ij}} \log(P_{ijk}).$$

$$\mathcal{L}_3(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} -\alpha_i * (1 - P_i)^{\gamma} * Q_i, \tag{3}$$

$$P_i = \frac{1}{M_i} \sum_{j=1}^{M_i} \frac{1}{T_{ij}} \sum_{k=1}^{T_{ij}} P_{ijk}, \quad Q_i = \frac{1}{M_i} \sum_{j=1}^{M_i} \frac{1}{T_{ij}} \sum_{k=1}^{T_{ij}} \log(P_{ijk}).$$

To address the issue of inconsistent convergence speeds, we draw inspiration from the FAMO [38] approach and innovatively apply it to the validation loss to calculate the weight of the training loss for each task. Firstly, we assumed that each task, indexed by i, has its own original loss $\mathcal{L}^i(\theta)$ that can take the form of either Equation 1, 2, or 3. In the t-th iteration, a randomly sampled mini-valid batch is fed into the model for forward and backward propagation after updating parameters. Then we update the weights of each task for $t+1$-th iteration based on the gradients of their corresponding validation losses, aiming to maximize the weight w^i for the task with the slowest convergence speed, as in Equation 4. Here, g_t represents the gradient of the weighted validation loss for all tasks, $c^i(\alpha, g_t)$ denotes the slope (gradient) of the validation loss for the i-th task, θ_t denotes the parameters of the network in the t-th iteration, α is the learning rate, and ϵ is a small constant to prevent division by zero. As a result, at each iteration, we update the task-specific weights based on the gradients of their validation losses. This approach aims to give more importance to tasks with slower convergence speeds, allowing them to have a larger influence on the overall optimization process. By dynamically adjusting the task weights, we create a balanced convergence scenario, where all tasks progress toward their optimal solutions at a similar rate. This mechanism effectively addresses the issue of disparate convergence speeds and enhances the overall stability and performance of the MFTCoder framework.

$$\mathcal{L}_4(\theta) = \max_{g_t} \min_{i} \frac{1}{\alpha} c^i(\alpha, g_t) - \frac{1}{2} \|g_t\|^2, \tag{4}$$

$$g_t = \sum_i w_t^i \nabla \mathcal{L}^i(\theta_t), \quad c^i(\alpha, g_t) = \frac{\mathcal{L}^i(\theta_t) - \mathcal{L}^i(\theta_t - \alpha d_t)}{\mathcal{L}^i(\theta_t) + \epsilon}.$$

Through integrating various loss functions, MFTCoder adeptly tackles the complexities of multitasking, such as data imbalance, task diversity, and uneven convergence rates found in large-scale MTL studies. Its adaptable framework offers a solid response to these challenges, enhancing the efficiency and precision of multi-task models. In particular, the selection of a loss function is dictated by use cases and the set of downstream tasks involved. For example, should we first opt for the cost-efficient Eq. 1 and notice considerable differences in convergence speed across tasks, we might then adopt more resource-intensive but balance-favoring Eq. 4.

4 EVALUATION

In this section, we will conduct multiple sets of experiments using MFTCoder to validate the effectiveness and superiority of the MFT method. Specifically, we aim to address the following three research questions:

RQ1: **Does the MFT model, achieved by multi-task fine-tuning using MFT methodology, outperform the SFT-S models, where each task is fine-tuned individually?**

RQ2: **Does the MFT model outperform the SFT-Mixed model, where multiple tasks are combined and fine-tuned together as one?**

RQ3: **In terms of generalization to unseen tasks, does the MFT model outperform the SFT-Mixed model?**

Next, we will commence by presenting the experimental setup. Subsequently, we will showcase and delve into the experimental results. Finally, we will culminate by summarizing and addressing the research questions raised in this section.

4.1 Evaluation Setup

To address the three research questions, we have chosen to center our investigations around code, selecting five code-related tasks. The rationale behind selecting these particular code-related tasks stems from their high interconnectivity, which fosters a stronger synergy among various coding tasks. Additionally, our year-long commitment to utilizing large language models in the coding domain has provided us with improved access to training data in this field. We built their respective fine-tuning datasets as outlined in Table 1 This table outlines each task's enhancement goals and sample counts. For example, the CODECOMPLETION-TASK aims to boost code completion skills with 192,547 samples, while the CODETRANS-TASK seeks to improve code translation with 307,585 samples. In particular, some of these data were generated using the method described in Section 3.2, but not all. For example, a portion of the UNITTEST data was extracted from open-source code. In the case of the data generated using the method described in Section 3.2, before using it for training, we apply filtering to prevent potential data leakage or contamination. Specifically, for models designed for competitive benchmarking, we assess the BLEU score by comparing docstrings from the HumanEval [12]'s methods requiring completion with each generated sample. Any samples with a BLEU score above 0.5 are excluded from consideration. Accordingly, we trained 7 models, including individual SFT-S-* models for each task, a collective SFT-MIXED model with data from all 5 tasks, and an MFT-5TASKS model using the MFT method. SFT-MIXED and MFT-5TASKS models use identical datasets, but differ in task delineation during training. SFT-MIXED blends all downstream task data into a single task for fine-tuning, without distinguishing between tasks. Conversely, MFT-5TASKS treats each downstream task as separate, calculating a distinct loss for each and deriving the total task loss by weighting these losses.

In our experiment, every model shared the same configuration except for the training data, all based on CodeLlama-13B-Python [50]. We trained each using 16 A100 GPUs with 80GB VRAM, a micro batch size of 8, and a global batch size of 128. We opted for the Adam optimizer with a starting learning rate of $2e-4$, and a minimum learning rate of $1e-5$. Fine-tuning was conducted with MFTCoder's

Table 1: Various experimental models and their corresponding training data.

Experimental Model	Task	Desired Ability	#Samples before/after packing
SFT-S-CodeCompletion	code-completion	Code Completion	192,547 / 18,811
SFT-S-Text2Code	text2code	Text2Code Generation	94,086 / 14,399
SFT-S-CodeComment	code-comment	Comment Generation	645,711 / 134,775
SFT-S-CodeTrans	code-trans	Code Translation	307,585 / 71,573
SFT-S-UnitTest	unit-test	Unit test generation	390,393 / 77,681
SFT-Mixed	Mix of 5 tasks	All of the above	1,630,322 / 317,239
MFT-5Tasks	The above 5 tasks	All of the above	1,630,322 / 317,239

QLora-INT4 mode, maintaining a consistent fine-tuning parameter ratio of 2.52%, and uniform positioning and initialization of trainable parameters. All models used the Data-Balance Loss (i.e., Equation 1) and pack tokenization to minimize computational demands, given resource and time constraints. With a single task, this loss function defaults to the standard loss used in typical GPT pre-training. We determined each model's convergence using an early-stopping strategy, halting training when the validation loss didn't improve for two successive epochs, ensuring optimal convergence for each.

4.2 Evaluation Datasets

In this paper, we utilized publicly available and representative code assessment benchmarks for comparative evaluation. These benchmarks include:

- **HumanEval:** [12] A widely used Python code completion evaluation dataset, meticulously curated by researchers at OpenAI.
- **HumanEval-X:** [63] An expansion of HumanEval, it includes translations into different programming languages for evaluating multi-language code completion.
- **DS-1000:** [30] Designed to test a model's data science capabilities in Python, this benchmark covers key libraries like Numpy, Pandas, TensorFlow, Pytorch, Scipy, Sklearn, and Matplotlib.
- **MBPP:** [6] Featuring 1000 Python problems sourced via crowdsourcing, it gauges basic Python proficiency. For this study, we chose 500 problems (ID 11-510) from MBPP to test text-to-code generation, specifically coding from problem descriptions.
- **CodeFuseEval:** [19] Builds on HumanEval and HumanEval-X, adding Chinese code completion (with Chinese docstrings), code translation, and unit test case generation, known as CodeFuseEval-CN, CodeFuseEval-CodeTrans, and CodeFuseEval-UnitTest.

Throughout these evaluation datasets, we employed **pass@1** [12] as the evaluation metric in this paper.

4.3 Evaluation Results

In this section, we present the results of seven trained models. The individual SFT-S-* models will be evaluated for their designated tasks; for example, we will only assess the SFT-S-CodeCompletion model's code completion performance. Meanwhile, we'll test the SFT-Mixed and MFT-5Tasks models across all tasks, comparing their results with the respective SFT-S-* models. Our testing will cover code completion, text-to-code generation, code comment generation, code translation, and unit test case generation.

Table 2: Pass@1 performance on HumanEval (Code Completion) and MBPP (Text-to-Code Generation). We utilized the greedy decoding strategy with zero-shot. The values of CodeLlama-Python-base are taken from [50].

Model	Size	Humaneval pass@1	MBPP pass@1	Average
CodeLlama-Python-base [50]	13B	43.3%	49.0%	46.15%
SFT-S-CodeCompletion	13B	59.76%	NA	NA
SFT-S-Text2Code	13B	NA	54.2%	NA
SFT-Mixed	13B	57.93%	53.6%	55.765%
MFT-5Tasks	13B	**60.37%**	**56.0%**	**58.185%**

Table 3: Comparison of pass@1 Metric Performance on the multilingual HumanEval-X (zero-shot, greedy-decoding)

Model	Java	C++	JavaScript	Go	Average
CodeLlama-13B-Py-base	43.3%	41.46%	34.76%	38.41%	29.27%
SFT-S-CodeCompletion	50.0%	39.02%	47.56%	40.23%	44.20%
SFT-Mixed	56.1%	**48.17%**	**56.10%**	37.80%	49.54%
MFT-5Tasks	**57.32%**	46.34%	54.27%	**45.12%**	**50.76%**

4.3.1 Code Completion. For code completion, we used the HumanEval [12] and HumanEval-X [63] datasets to measure performance. We assessed three models: SFT-S-CodeCompletion, SFT-Mixed, and MFT-5Tasks. The models' scores on HumanEval are shown in Table 2 (Column III), with the MFT-5Tasks model topping the rest by 2.44% over SFT-Mixed, which used mixed task data. Notably, SFT-Mixed lagged behind SFT-S-CodeCompletion, trained solely for code completion. We also evaluated on the multilingual HumanEval-X dataset, detailed in Table 3. The MFT-5Tasks model led in Java and Go, while SFT-Mixed took the lead in C++ and JavaScript. Overall, the MFT-5Tasks model outshined its counterparts, averaging 1.22% better than SFT-Mixed. **In other words., in terms of code completion tasks, models trained using the MFT method outperform both individually fine-tuned models and models fine-tuned after combining multiple tasks.**

4.3.2 Text-to-Code Generation. To test code generation from descriptions, we used the MBPP [6] with the pass@1 metric, designed for evaluating Python program synthesis from natural language. We assessed the SFT-S-Text2Code, SFT-Mixed, and MFT-5Tasks models on MBPP, with results in Table 2 (Column IV) showing MFT-5Tasks outperformed others, beating SFT-Mixed by 2.4%. Similarly, mixed-task fine-tuning underperformed compared to task-specific tuning. **Overall, for text-to-code tasks, MFT-trained models surpass those fine-tuned individually or on mixed tasks.**

4.3.3 Code Comment Generation. The goal of code comment generation is to enable models to add comments, including line and interface comments, to code without altering it, enhancing readability and usability. We evaluated this using 500 MBPP test questions (id 11-510). The SFT-S-CodeComment, SFT-Mixed, and MFT-5Tasks models generated comments for each, with GPT-4 determining the best results based on established comment quality criteria, marking indecisive outcomes as UNKNOWN. Table 4 shows that the MFT-5Tasks model led with 38.8%, outdoing SFT-Mixed by 7.4% and SFT-S-CodeComment by 10.8%, while GPT-4 found 1.8% of

Table 4: Performance Comparison of Three Models on Code Commenting Task. GPT-4 Determines the Best Performing Model for Each Question. This Table Presents the Proportion of Questions Where Each Model Performs the Best. In particular, 1.8% of the evaluation cases were indeterminate for GPT-4 to determine the best-performing model.

Model	Best identified by GPT-4
SFT-S-CodeComment	28%
SFT-Mixed	31.4%
MFT-5Tasks	**38.8%**

Table 5: Pass@1 Metric Performance on the codefuseEVAL-CodeTranslation [19] (zero-shot, greedy-decoding)

Model	Py2Java	Py2C++	Java2Py	C++2Py	Java2C++	C++2Java	Avg.
SFT-S-CodeTrans	59.52%	57.40%	**70.73%**	62.20%	**67.07%**	62.80%	63.29%
SFT-Mixed	80.16%	71.20%	67.68%	**72.56%**	65.85%	82.31%	73.29%
MFT-5Tasks	**82.16%**	**77.20%**	65.85%	70.73%	64.64%	**84.76%**	**74.22%**

cases indeterminable. **Overall, MFT-trained models show the strongest performance in code comment generation.**

4.3.4 Code Translation. The aim of code translation is to convert code from a source language to a target language with accuracy and precision, maintaining identical functionality. We used the code translation subset of codefuseEval [6] [19] for bidirectional translation among Java, Python, and C++. Test cases check the accuracy and functional equivalence of the translated code, with successful runs indicated by the pass@1 metric. Results in Table 5 show MFT-5Tasks leading in Python-to-Java, Python-to-C++, and C++-to-Java. SFT-Mixed is best for C++-to-Python, while SFT-S-CodeTrans tops Java-to-Python and Java-to-C++ translations. On average, MFT-5Tasks outperforms SFT-Mixed by 0.93% and SFT-S-CodeTrans by 10.9%. **Overall, this task further confirms that MFT-trained models surpass the other two training approaches.**

4.3.5 Unit Test Case Generation. Our goal is to create unit test cases, training a model to generate tests for a given code snippet to confirm its correctness. We're using codefuseEVAL-UnitTest [19] for our tests, evaluated by the pass@1 metric, signifying success if generated test cases are all passed by the sample program. We tested three models' test generation for Python, Java, and JavaScript, with results in Table 6 showing MFT-5Tasks as the standout in Python, leading SFT-Mixed by 5.73% and SFT-S-UnitTest by 10.19%. For JavaScript, MFT-5Tasks also leads, boasting a 7.93% lead. However, in Java, while MFT-5Tasks surpasses SFT-S-UnitTest by 5.37%, it slightly trails SFT-Mixed by 5.44%. Overall, MFT-5Tasks consistently outperforms, averaging 2.74% better than SFT-Mixed and 7.83% better than SFT-S-UnitTest. **Training with MFT method proves superior to mixed-task fine-tuning and excels over models individually tuned for the UNIT-TEST task.**

4.3.6 Generalization on an Unseen Task. We assessed our models' performance on training data tasks to address RQ1 and RQ2. Additionally, we aimed to answer RQ3: *Do models trained with the Multi-Task Fine-Tuning (MFT) method generalize better to new tasks compared to models trained with an SFT strategy that uses a mix of datasets?* To explore this, we used the Text-to-SQL generation task as our benchmark, which was not part of the training for our seven models. This code-related yet distinct task from the previous five adds a different challenge. We adopted two metrics for evaluation: the BLEU score for textual similarity and logical accuracy for SQL correctness and equivalence. We evaluated over five text-to-SQL datasets—WikiSQL [64], Spider [60], CSpider [44], CoSQL [59], and BirdSQL [33]—sampling 200 examples each. Results presented in Table 7 show that the MFT-5Tasks consistently outperformed SFT-Mixed models, with BLEU scores averaging 2.78 times higher and logical accuracy 2.18 times better overall—up to 4.67 times more accurate on WikiSQL. For a specific example, please refer to Table 12 in the Appendix. **Numerically, MFT-5Tasks exhibits superior performance compared to SFT-Mixed, presenting stronger generalization of MFT-trained models on the Text-to-SQL task, which is an unseen task during training.**

[6]https://github.com/codefuse-ai/codefuse-evaluation

Table 6: Pass@1 Metric Performance on the codefuseEVAL-TestcaseGeneration [19] (zero-shot, greedy-decoding)

Trained Model	Python	Java	JavaScript	Average
SFT-S-UnitTest	33.76%	32.43%	41.46%	35.88%
SFT-Mixed	38.22%	**43.24%**	41.46%	40.97%
MFT-5Tasks	**43.95%**	37.8%	**49.39%**	**43.71%**

Table 7: Comparison of generalization capabilities between MFT-5Tasks and SFT-Mixed on the Text-to-SQL task. The evaluation metrics include SQL logical accuracy and BLEU score.

Model	WIKISQL	SPIDER	CSPIDER	COSQL	BiRDSQL	AVG
Logical Accuracy						
SFT-Mixed	1.5%	2.0%	7.0%	6.5%	5.5%	4.5%
MFT-5Tasks	7.0% (4.67x)	4.5% (2.25x)	16.5% (2.36x)	10.5% (1.62x)	10.5% (1.91x)	9.8% (2.18x)
BLEU						
SFT-Mixed	0.032	0.047	0.025	0.081	0.026	0.042
MFT-5Tasks	0.138	0.138	0.116	0.119	0.074	0.117

Table 8: Resource and time costs of 7 evaluation models.

Model	GPUs	Early Stopped @Epoch	Time (minutes)
MFT-5Tasks	16	6.93	3686
SFT-Mixed	16	6.93	3704
SFT-S-CodeCompletion	16	4.11	188
SFT-S-Text2Code	16	6.25	217
SFT-S-CodeTrans	16	4.92	188
SFT-S-CodeComment	16	3.99	1248
SFT-S-UnitTest	16	6.93	1246

4.4 Evaluation Summary

We chose five code-related downstream tasks and trained a total of seven models: individual SFT-S-* models for each task, an SFT-Mixed model using combined task data, and an MFT-5Tasks model employing the multi-task fine-tuning technique. We assessed each model's specific performance and their generalizability on novel tasks, comparing the MFT approach with the mixed SFT method.

Besides, we compare the resource and time costs of training seven models in Table 8. Notably, the MFT-5Tasks and SFT-Mixed models have similar training times, as shown in the 4th column, due to identical sample sizes and early stopping points—with MFT being marginally faster. In total, the key findings are:

i **MFT-trained models outshined those individually fine-tuned, affirming RQ1 positively.**
ii **MFT-trained models also surpassed those fine-tuned on a mixed set of tasks, positively addressing RQ2.**
iii **MFT models demonstrated superior generalization on unseen tasks compared to mixed-data SFT models.**

5 APPLICATION

Considering the outstanding performance of the MFT training method, we have leveraged our MFTCoder [4] to fine-tune the existing representative open-source LLM models. *e.g.* QWen [7], Baichuan [8], CodeGeex2 [63], Llama [54], LLama2 [55], CodeLLama [50], StarCoder [34], DeepSeek [23], and Mistral [27, 28].

MFTCoder harnesses Lora and QLora to significantly reduce the training parameter count. When coupled with dual quantization for compression, it greatly lowers GPU memory demand. We set trainable parameters in MFTCoder to just 0.1-5% of the total, finding performance plateaus beyond this range. In practice, under 5% is typically enough to match full-scale fine-tuning results. For multitasking, we fine-tune models on 3-7 tasks, using Lora for models under 20B and QLora for those above. Post-tuning, we assess them on code completion and text-to-code generation, specifically on HumanEval [12] and MBPP [6], as detailed in Table 9 Columns III and IV. Our analysis shows an average 6.26-13.78% improvement in MFT fine-tuning over base models, with gains on HumanEval outstripping those on MBPP, as shown in column 5. We also tested the code completion prowess of our MFTCoder-tuned models on HumanEval-X [63], a multilingual benchmark, with findings in Table 10. Remarkably, our fine-tuned CodeFuse-DeepSeek-33B model scored an average pass@1 rate of 67.07% across Python, Java, C++, JavaScript, and Go.

Table 9 includes results from fine-tuned open-source models like OctoPack [45] and WizardCoder-Python [43], alongside top closed-source models such as Claude 2 [4] and GPT-4 [46], on the HumanEval and MBPP benchmarks. Notably, **our fine-tuned CodeFuse-DeepSeek-33B tops the Big Code Models Leaderboard [2] by WinRate, outperforming all open-source contenders in Table 9**. This ranking is a definitive gauge of open-source models' coding prowess, rigorously evaluated via the HumanEval [12] and MultiPL-E [11] benchmarks.

As an example, we benchmarked our CodeFuse-CodeLlama-34B model across various datasets — HumanEval-X [63], MBPP [6], DS-1000 [30], and codefuseEval [19] — comparing its performance with GPT-3.5 and GPT-4, as illustrated in Figure 3. It excels beyond GPT-4 on CodeFuseEval-UnitTest and HumanEval, ties in code translation, but lags in Chinese code completion (CodeFuseEval-CN), multi-language completion (HumanEval-X), data science tasks (DS-1000), and text-to-code (MBPP). Yet, it meets or beats GPT-3.5 across all tests. We also evaluated the effect of fine-tuning models with MFTCoder and code-related data on NLP task performance, as shown in Figure 4. Using CodeFuse-QWen-14B as an example, we measured it against the original QWen-14B model and Alibaba Cloud's officially fine-tuned QWen-14B-chat. CodeFuse-QWen-14B not only retained its NLP strengths but also showed slight improvements in language, reasoning, and comprehension skills over the other two models. On the other hand, it experienced a small drop in its examination ability relative to the QWen-14B base model, with QWen-14B-chat exhibiting a similar trend.

At Antgroup, leveraging models trained with MFTCoder, we developed CodeFuse, a programming assistant featuring web and IDE plugins. It boasts 12k weekly active users, with AI generating nearly 80k lines of code weekly.

Table 9: Pass@1 Performance on HumanEval and MBPP Post-MFTCoder Fine-Tuning. The *CodeFuse-*-MFT* models were assessed using greedy decoding and a zero-shot strategy, whereas metrics for other models were sourced from their published papers/reports, or open-source project homepages.

Model	Size	Humaneval pass@1	MBPP pass@1	Average
Open-source base models				
QWen-base [7]	14B	32.3%	40.8%	36.55%
Llama-base [54]	65B	23.7%	37.7%	30.7%
Llama2-base [55]	70B	29.9%	45.0%	37.45%
StarCoder-base [34]	15B	33.6%	52.7%	43.15%
CodeGeex2-base [63]	6B	35.9%	42.4%	39.15%
CodeLlama-Python-base [50]	13B	43.3%	49.0%	46.15%
CodeLlama-Python-base [50]	34B	53.7%	56.2%	54.95%
DeepSeek-Coder-base	33B	56.1%	66.0%	61.05%
MFT fine-tuned models				
CodeFuse-QWen [7]	14B	48.78%	43.8%	46.29% (+9.74%)
CodeFuse-Llama	65B	34.76%	41.8%	38.28% (+7.58)
CodeFuse-Llama2	70B	40.85%	40.8%	40.83% (+3.38%)
CodeFuse-StarCoder [8]	15B	54.90%	49.60%	52.25% (+9.10%)
CodeFuse-CodeGeex2	6B	45.12%	46.2%	45.66% (+6.51%)
CodeFuse-CodeLlama	13B	60.37%	56.0%	58.19% (+12.04%)
CodeFuse-CodeLLama [9]	34B	74.4%	61.0%	67.70% (+12.75%)
CodeFuse-DeepSeek [3]	33B	78.65%	**71.0%**	**74.83%** (+13.78%)
Open-source fine-tuned models				
QWen-chat [7]	14B	43.9%	46.4%	45.15%
PHI-1 [22]	1.3B	50.6%	55.5%	53.05%
OctoCoder [45]	15B	46.2%	NA	NA
WizardCoder [43]	15B	57.3%	51.8%	54.55%
Phind-CodeLlama-v2 [48]	34B	71.95%	NA	NA
WizardCoder-Python [43]	34B	73.2%	61.2%	67.2%
DeepSeek-Coder-Instruct	33B	**79.3%**	70.0%	74.65%
Closed-source models				
PanGu-Coder2 [51]	15B	61.2%	NA	NA
Unnatural CodeLlama [50]	34B	62.2%	61.2%	61.7%
Claude2 [4]	NA	71.2%	NA	NA
GPT-3.5 [46]	175B	48.1%	52.2%	50.15%
GPT-4 (zero-shot) [46]	NA	67.00%	NA	NA

6 DISCUSSION, OUTLOOK AND CONCLUSION

While the MFT training method has been shown to outperform the SFT approach based on task data mixing in our tests, its success hinges on the strategy used to divide tasks. MFT isn't universally applicable; for instance, our attempts to split tasks by difficulty level and train with MFT didn't surpass the SFT method. Similarly, segmenting code completion by programming language for MFT training didn't beat SFT. Our practical insights suggest that MFT is best for tasks with distinct core abilities, but not for those with similar goals. Future research will aim to refine task delineation guidelines (*i.e.* task relationship learning [17]) for MFT training.

[7]https://huggingface.co/codefuse-ai/CodeFuse-QWen-14B
[8]https://huggingface.co/codefuse-ai/CodeFuse-StarCoder-15B
[9]https://huggingface.co/codefuse-ai/CodeFuse-CodeLlama-34B

Table 10: Pass@1 performance on HumanEval-X [63] after fine-tuning with MFTCoder across diverse open-source models. The metric values marked with an asterisk (*) were obtained from the models' corresponding papers, technical reports, or open-source project homepages, while the remaining metric values were evaluated using a combination of greedy decoding and zero-shot strategy.

Model	Size	Python	Java	C++	JS	Go	AVG.
QWen	14B	32.3%*	35.37%	30.49%	32.93%	21.34%	30.49%
CodeFuse-QWen	14B	48.78%	41.46%	38.41%	46.34%	26.83%	40.36%
Llama	65B	23.7%*	29.26%	20.73%	23.78%	18.9%	23.27%
CodeFuse-Llama	65B	34.76%	37.2%	29.88%	32.93%	23.78%	31.71%
Llama2	70B	29.9%*	39.02%	31.10%	35.98%	23.78%	31.96%
CodeFuse-Llama2	70B	40.85%	35.98%	32.32%	38.41%	27.44%	35.00%
StarCoder	15B	33.6%*	34.15%	25.61%	22.56%	22.56%	29.48%
CodeFuse-StarCoder	15B	54.9%	47.56	46.34%	48.17%	37.20%	46.83%
CodeGeex2	6B	35.9%*	30.8%*	29.3%*	32.2%*	22.5%*	30.14%
CodeFuse-CodeGeex2	6B	45.12%	45.73%	37.2%	37.2%	28.05%	38.66%
CodeLlama-Python	13B	43.3%*	41.46%	34.76%	38.41%	29.27%	37.44%
CodeFuse-CodeLlama	13B	60.37%	57.32%	46.34%	54.27%	45.12%	52.68%
CodeLlama-Python	34B	53.7%*	45.73%	42.68%	45.73%	31.71%	43.91%
CodeFuse-CodeLLama	34B	74.4%	61.6%	54.3%	61.0%	50.6%	60.38%
DeepSeek-Coder	33B	56.1%	51.9%	58.4%	55.3%		
CodeFuse-DeepSeek	33B	78.65%	67.68%	65.85%	67.07%	56.10%	67.07%

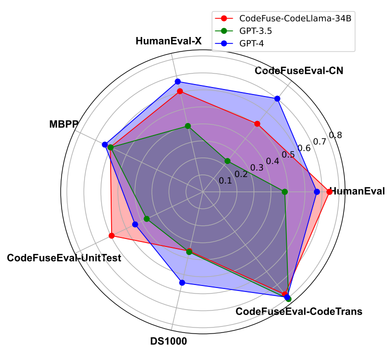


Figure 3: Radar Chart of CodeFuse-CodeLlama-34B Model on code benchmarks compared to GPT-3.5 and GPT-4.

In our task generalization tests, MFT-trained models yielded outputs closer to reference answers and with less verbosity, whereas SFT-trained models with mixed tasks produced more Chain-of-Thought details. Each has its merits, with MFT preferred in contexts like IDE plugins and SFT favored in web assistants. It's not a matter of one method being superior; it depends on the use case. We're currently investigating the causes of these performance variations. Regarding scalability, we have fine-tuned MFTCoder on 32 downstream tasks in practice. Theoretically, this is not its upper limit. We are planning to scale up to a larger number of tasks to more objectively measure its scalability.

MFT, as a multi-task learning method, grapples with the issue of varied task convergence rates during training. For instance, our experiments showed code completion tasks converge quicker than

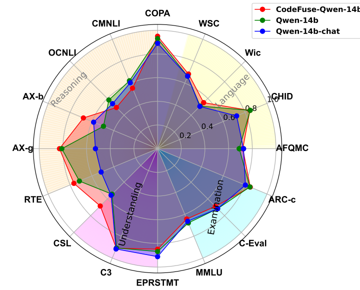


Figure 4: Performance comparison of CodeFuse-QWen-14B fine-tuned with MFTCoder and code-related data, QWen-14B base model, and officially fine-tuned model QWen-14B-chat on NLP evaluation datasets. Detailed data can be found in Table 11.

unit test-case generation tasks (Figure 6 in Appendix B). This discrepancy complicates finding a checkpoint that excels across all tasks, often leading to under-convergence or overfitting. To tackle this, we trialed solutions like FAMO [38] for multi-task learning balance. However, FAMO demands dual back-propagation each iteration—using both a training and a validation mini-batch—roughly doubling training time. Additionally, it lengthens the required epochs for convergence without much control over convergence speed. This significant increase in cost didn't equate to substantial benefits. Consequently, we're in the process of creating a more efficient, adaptive approach for multi-task optimization balance.

Furthermore, even after balancing the convergence speeds of multiple tasks, where the same set of parameters is updated, it is still challenging to fundamentally eliminate the inherent conflicts in weight updates across different tasks. To address this issue, we are currently exploring the utilization of MoE [14] to achieve MFT.

Regarding the construction process of the instruction dataset, the current automated workflow lacks a verification stage, which compromises the quality of the automatically generated data. To address this, we are integrating a check process into the workflow. Specifically, we instruct GPT to output both solutions and test cases, and we execute the combined solution-and-test-case code to verify that all tests pass.

This paper presents MFTCoder, a framework designed for multi-task fine-tuning that effectively tackles issues such as data imbalance, task disparity, and uneven convergence rates by employing diverse loss functions. Our experiments show it surpasses task-specific fine-tuning and mixed-task ensemble fine-tuning in performance. MFTCoder also promotes efficient training, optimizes data usage, and includes PEFT. It offers an effective method for creating high-quality instructional datasets. MFTCoder-fine-tuned CodeFuse- DeepSeek-33B claimed the top spot on the Big Code Models Leaderboard ranked by WinRate as of January 30, 2024.

ACKNOWLEDGMENTS

We are grateful to the CodeFuse project team at Ant Group, as well as to the software and hardware infrastructure teams at Ant Group for their support throughout the completion of this work.

REFERENCES

[1] Armen Aghajanyan, Anchit Gupta, Akshat Shrivastava, Xilun Chen, Luke Zettlemoyer, and Sonal Gupta. 2021. Muppet: Massive Multi-task Representations with Pre-Finetuning. arXiv:2101.11038 [cs.CL]

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, et al. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE]

[3] Rohan Anil, Andrew M. Dai, Orhan Firat, et al. 2023. PaLM 2 Technical Report. arXiv:2305.10403 [cs.CL]

[4] Anthropic. 2023. *Model Card and Evaluations for Claude Models*. https://www-files.anthropic.com/production/images/Model-Card-Claude-2.pdf

[5] Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q. Tran, Dara Bahri, Jianmo Ni, Jai Gupta, Kai Hui, Sebastian Ruder, and Donald Metzler. 2022. ExT5: Towards Extreme Multi-Task Scaling for Transfer Learning. arXiv:2111.10952 [cs.CL]

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[7] Jinze Bai, Shuai Bai, Yunfei Chu, et al. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[8] Baichuan. 2023. Baichuan 2: Open Large-scale Language Models. *arXiv preprint arXiv:2309.10305* (2023). https://arxiv.org/abs/2309.10305

[9] Sid Black, Stella Biderman, Eric Hallahan, et al. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. arXiv:2204.06745 [cs.CL]

[10] Rich Caruana. 1997. Multitask learning. *Machine learning* 28 (1997), 41–75.

[11] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. arXiv:2208.08227 [cs.LG]

[12] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[13] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. 2018. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International conference on machine learning*. PMLR, 794–803.

[14] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. 2022. Towards Understanding Mixture of Experts in Deep Learning. arXiv:2208.02813 [cs.LG]

[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[16] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, et al. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. arXiv:2207.11280 [cs.LG]

[17] Michael Crawshaw. 2020. Multi-task learning with deep neural networks: A survey. *arXiv preprint arXiv:2009.09796* (2020).

[18] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314 [cs.LG]

[19] Peng Di, Jianguo Li, Hang Yu, et al. 2023. CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model. arXiv:2310.06266 [cs.SE]

[20] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 320–335.

[21] Long Duong, Trevor Cohn, Steven Bird, and Paul Cook. 2015. Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th international joint conference on natural language processing (volume 2: short papers)*. 845–850.

[22] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644* (2023).

[23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE]

[24] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. arXiv:1902.00751 [cs.LG]

[25] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

[26] Sébastien Jean, Orhan Firat, and Melvin Johnson. 2019. Adaptive scheduling for multi-task learning. *arXiv preprint arXiv:1909.06434* (2019).

[27] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, et al. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]

[28] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, et al. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG]

[29] Alex Kendall, Yarin Gal, and Roberto Cipolla. 2018. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7482–7491.

[30] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *ArXiv* abs/2211.11501 (2022).

[31] Hae Beom Lee, Eunho Yang, and Sung Ju Hwang. 2018. Deep asymmetric multi-task feature learning. In *International Conference on Machine Learning*. PMLR, 2956–2964.

[32] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Scale Language Model Society. arXiv:2303.17760 [cs.AI]

[33] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv:2305.03111 [cs.CL]

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[35] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks Are All You Need II: **phi-1.5** technical report. *arXiv preprint arXiv:2309.05463* (2023).

[36] Yujia Li, David Choi, Junyoung Chung, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (dec 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[37] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV]

[38] Bo Liu, Yihao Feng, Peter Stone, and Qiang Liu. 2023. FAMO: Fast Adaptive Multitask Optimization. arXiv:2306.03792 [cs.LG]

[39] Shikun Liu, Edward Johns, and Andrew J Davison. 2019. End-to-end multi-task learning with attention. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 1871–1880.

[40] Shengchao Liu, Yingyu Liang, and Anthony Gitter. 2019. Loss-balanced task weighting to reduce negative transfer in multi-task learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 9977–9978.

[41] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. 2019. Multi-Task Deep Neural Networks for Natural Language Understanding. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4487–4496.

[42] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and Philip S Yu. 2017. Learning multiple tasks with multilinear relationship networks. *Advances in neural information processing systems* 30 (2017).

[43] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[44] Qingkai Min, Yuefeng Shi, and Yue Zhang. 2019. A Pilot Study for Chinese SQL Semantic Parsing. arXiv:1909.13293 [cs.CL]

[45] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction Tuning Code Large Language Models. arXiv:2308.07124 [cs.CL]

[46] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[47] Lucas Pascal, Pietro Michiardi, Xavier Bost, Benoit Huet, and Maria A Zuluaga. 2021. Maximum roaming multi-task learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 9331–9341.

[48] Phind. 2023. *Phind-CodeLlama-34B-v2*. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2

[49] Colin Raffel, Noam Shazeer, Adam Roberts, et al. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[50] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[51] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936* (2023).

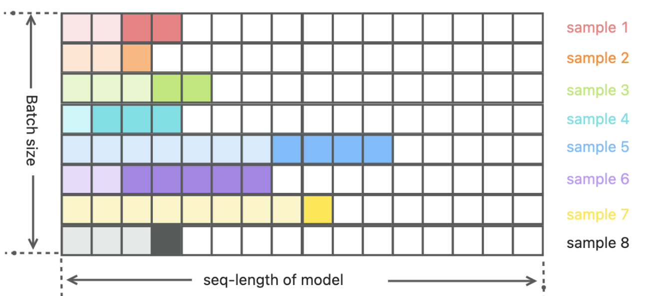
[52] Ximeng Sun, Rameswar Panda, Rogerio Feris, and Kate Saenko. 2020. Adashare: Learning what to share for efficient deep multi-task learning. *Advances in Neural Information Processing Systems* 33 (2020), 8728–8740.

[53] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).

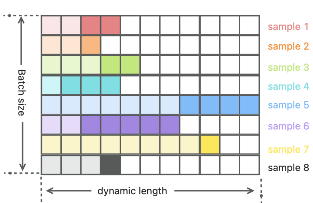
[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv*

preprint arXiv:2302.13971 (2023).
[55] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[56] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560* (2022).
[57] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs.CL]
[58] Yongxin Yang and Timothy Hospedales. 2017. Trace Norm Regularised Deep Multi-Task Learning. In *5th International Conference on Learning Representations.*
[59] Tao Yu, Rui Zhang, He Yang Er, et al. 2019. CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases. arXiv:1909.05378 [cs.CL]
[60] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL]
[61] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv:2311.07989 [cs.CL]
[62] Xiangyun Zhao, Haoxiang Li, Xiaohui Shen, Xiaodan Liang, and Ying Wu. 2018. A modulation module for multi-task learning with applications in image retrieval. In *Proceedings of the European Conference on Computer Vision (ECCV).* 401–416.
[63] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. In *KDD.*
[64] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).

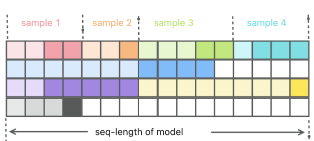
A TOKENIZATION MODES



(a) Normal SFT Mode



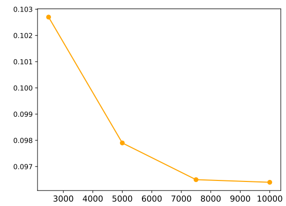
(b) Dynamic Padding Mode



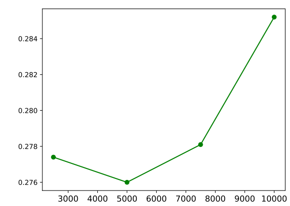
(c) Pack SFT Mode

Figure 5: Illustration of the differences in sample organization within a batch between normal SFT, dynamic padding, and Pack SFT tokenization modes. The light-colored squares in the figure represent the Prompt section of the samples, while the dark-colored squares represent the Label section (participating in loss calculation). The blank squares represent the padding section.

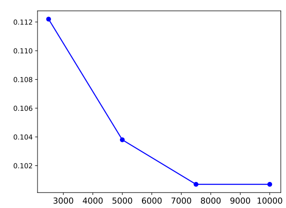
B VALIDATION LOSS CONVERGENCE SPEEDS



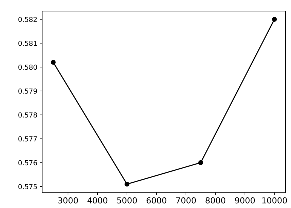
(a) CodeComment Task



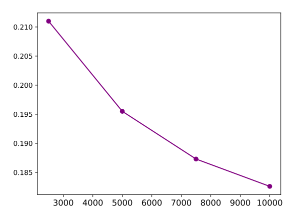
(b) CodeCompletion Task



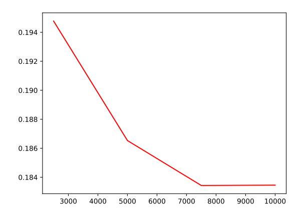
(c) CodeTranslation Task



(d) Text2Code Task



(e) UnitTest task



(f) Overall of 5 tasks

Figure 6: Validation Loss Convergence Speeds: A Comparative Analysis of 5 Code-related Downstream Tasks and Overall Training Progress using the data-balanced loss function during training MFT-5Tasks. Each task has varied convergence rates during training, *e.g.* code completion tasks converge quicker than unit test-case generation tasks.

Table 11: Comparisons of the performances of CodeFuse-QWen-14B, QWen-14B, and QWen-14b-chat on several NLP evaluation datasets. QWen-14B is a base model trained by Alibaba Cloud, QWen-14B-chat is a model fine-tuned by themselves on top of the QWen-14B base model, and CodeFuse-QWen-14B is a model fine-tuned by us using MFTCoder and code data.

		QWen-14B	QWen-14B-chat	CodeFuse-QWen-14B
Language	AFQMC	69.00%	72.6%	71.99%
	CHID	84.7%	72.3%	84.42%
	Wic	50.9%	50.5%	55.02%
	WSC	66.3%	66.3%	68.27%
	Average	67.73%	65.43%	**69.93%**
Reasoning	COPA	93%	89%	95.00%
	CMNLI	62.1%	60.3%	55.33%
	OCNLI	58.2%	53.8%	49.22%
	AX-b	49.5%	58.5%	67.84%
	AX-g	80.9%	52.5%	82.87%
	RTE	71.5%	51.6%	76.53%
	Average	69.20%	60.95%	**71.13%**
Understanding	CSL	54.4%	55.6%	68.50%
	C3	90.8%	91.7%	91.01%
	EPRSTMT	86.9%	91.2%	84.92%
	Average	77.37%	79.50%	**81.48%**
Examination	AX-b	67.9%	66.4%	64.27%
	AX-g	71.7%	71.7%	68.98%
	RTE	84.4%	80.3%	84.75%
	Average	**74.67%**	72.80%	72.67%

Table 12: Sample Text-to-SQL Test Cases from the Spider [60] dataset and Example Predictions by the SFT-Mixed Model and MFT-5Tasks Model. MFT-5Tasks produces cleaner results, while SFT-Mixed provides more explanations, which may be preferred in certain scenarios. Moreover, MFT-5Tasks demonstrates better performance in terms of logical accuracy, achieving an overall accuracy that is 2.18 times higher than SFT-Mixed model,

<table>
<tr><td>PROMPT:</td><td><|role_start|>system<|role_end|>You are a professional SQL engineer and you are writing SQL queries for data query tasks.
<|role_start|>human<|role_end|>What are the names of the heads who are born outside the California state?
schema:head(head_ID|name|born_state|age
<|role_start|>bot<|role_end|></td></tr>
<tr><td>LABEL:</td><td>SELECT name FROM head WHERE born_state != 'California'</td></tr>
<tr><td>SFT-Mixed:</td><td>You can use the SQL SELECT statement to retrieve the names of the heads who are born outside the California state. Here is a sample query:

"'sql
SELECT name
FROM head
WHERE born_state <> 'California';
"'

This query selects the name column from the head table where the born_state column is not equal to 'California'.</td></tr>
<tr><td>MFT-5Tasks:</td><td>SELECT name
FROM head
WHERE born_state <> 'California';</td></tr>
</table>