Fine-tune Llama3 with $(IA)^3$++

Shaoxuan Zheng* Yuntao Zheng† Lingfei He‡

December 8, 2024

1 Introduction

Advancements in fine-tuning techniques have significantly expanded the capabilities of Large Language Models (LLMs) in various natural language processing tasks. Traditional fine-tuning updates all model parameters, which is computationally expensive and prone to catastrophic forgetting. Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA [5] address these issues by fine-tuning only a small subset of parameters, achieving strong performance efficiently. However, even state-of-the-art methods like LoRA still involve updating a considerable number of parameters.

In this work, we aim to further reduce the number of trainable parameters required for effective fine-tuning without compromising model performance. Inspired by the Infused Adapter by Inhibiting and Amplifying Inner Activations $(IA)^3$ [8] and the QK-Norm technique from OLMoE [10], we propose a novel approach, $(IA)^3$++, that combines these methods and introduces an additional trainable vector for the output attention layer. Our method leverages the efficiency of $(IA)^3$ by introducing minimal trainable parameters in the form of tunable vectors that scale the activations within the model. By integrating QK-Norm with RMSNorm, we enhance the stability of the attention mechanism, preventing issues like numerical overflow that can occur during training.

We fine-tune the Llama3.2-1B model using our proposed method on natural language inference datasets, including MMLU and ARC-Challenge. Our experiments demonstrate that $(IA)^3++$ outperforms the original $(IA)^3$ and achieves competitive results compared to LoRA on benchmark datasets, with significantly fewer trainable parameters. By incorporating innovative components like an additional scaling vector and leveraging skip connections, our method strikes a balance between accuracy and parameter efficiency, highlighting its potential for broader application and scalability.

Our code is publicly available at https://github.com/zhengyuntao123/genai_finetune.

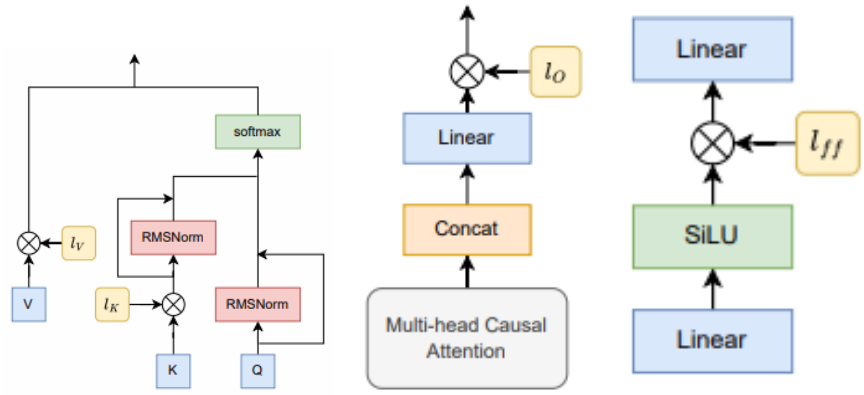

Figure 1: $(IA)^3 + +$ with RMS-Norm and Skip Connection

*shaoxuaz@andrew.cmu.edu
†yuntaozh@andrew.cmu.edu
‡lingfeih@andrew.cmu.edu

2 Dataset and Task

Our task is to fine-tune Llama3.2-1B using LoRA and our proposed method. Then we will evaluate the model's performance before and after fine-tuning on 2 datasets, which are MMLU and ARC-Challenge. Since both datasets are multiple-choice datasets, we will simply use accuracy as our metric.

2.1 MMLU

MMLU(Massive Multitask Language Understanding)[4] is a benchmark for evaluating the broad knowledge and reasoning abilities of language models across a diverse range of tasks. It includes over **57 tasks** spanning *STEM, humanities, social sciences, and other professional fields*, with questions structured in a multiple-choice format. The tasks are designed to test a model's understanding and knowledge in areas that require higher-order reasoning and specialized domain knowledge. This benchmark is commonly used to assess how well large language models generalize across domains and simulate human-like reasoning in complex scenarios.

2.2 ARC-Challenge

ARC (AI2 Reasoning Challenge) [2] is a benchmark designed to evaluate the reasoning and problem-solving capabilities of AI models on grade-school science questions. It comprises over **7,000 questions**, structured in a multiple-choice format, and divided into two subsets: *ARC-Easy* and *ARC-Challenge*. The ARC-Challenge set specifically focuses on questions that require higher-order reasoning, problem-solving skills, and the integration of external knowledge, making it particularly difficult for both traditional machine learning models and large language models. These tasks span diverse science topics, including biology, physics, chemistry, and earth science. The benchmark is widely used to test how well AI models perform in complex reasoning scenarios and their ability to generalize beyond simple fact recall to emulate human-like reasoning processes.

2.3 Foundation Model

Due to the limit of GPU memory, we choose to use Llama3.2-1B as our foundation model. The pre-existing open source code can be found here.

3 Related Work

Fine-tuning all parameters of a pre-trained model is a straightforward adaptation approach but faces drawbacks such as catastrophic forgetting and high computational costs due to the large parameter count. Parameter-Efficient Fine-Tuning (PEFT) methods address these issues by fine-tuning a small subset of parameters while keeping the rest fixed, achieving comparable performance with reduced computational overhead.

Early PEFT methods, inspired by In-Context Learning (ICL) [1], enable task-specific learning without extensive parameter updates. Techniques like Prefix Tuning [7], Prompt Tuning [6], and P-Tuning [9] add prompts or prefixes to input sequences while keeping model parameters frozen. Despite their effectiveness, these methods can suffer from performance issues, particularly with smaller models.

Low-rank adaptation methods such as LoRA [5] significantly reduce trainable parameters by introducing trainable low-rank matrices integrated with pre-trained weights. LoRA achieves high performance on downstream tasks with only 0.1% of parameters being trainable. AdaLoRA [12] further improves efficiency by adaptively allocating parameter budgets based on weight matrix importance.

Activation-based methods like $(IA)^3$ [8] use trainable vectors to scale model activations in attention and feed-forward layers, enabling competitive performance with only 0.01% of parameters being trainable. The T-Few recipe extends $(IA)^3$ with multi-task fine-tuning, surpassing In-Context Learning in accuracy and efficiency, especially with limited labeled data.

However, $(IA)^3$ introduces potential numerical instability by scaling the key matrix K with trainable vectors in the attention mechanism. Inspired by QK-Norm in OLMoE [10], which normalizes queries and keys for stable attention scores, we integrate RMSNorm to enhance training stability. Building on these ideas, we propose a novel PEFT approach that combines the strengths of previous methods with an additional trainable vector for the output attention layer.

4 Approach

Our inspiration mainly originates from OLMoE [10], LNLoRA[11], and $(IA)^3$ [8], and . From OLMoE, we adopt the idea of QK-Norm idea that adding layer normalization before softmax can enhance the stability of attention mechanism. In LNLoRA, combining layer normalization with PEFT method is proved to be more efficient. In addition, $(IA)^3$ is a novel PEFT method that scales activations with learned vectors, which achieves high performance on novel tasks while updating minimal parameters around 0.01% of total parameters. We propose our own method $(IA)^3$++ based on it with several advanced techniques.

4.1 QK-Norm

OLMoE introduces QK-Norm, which can prevent large logits in the following attention operation that may lead to overflow and make the training unstable, by adding a layer normalization after the query and key projections. In our $(IA)^3$++, we use RMSNorm instead of standard layer normalization. For input $\mathbf{x} \in R^d$, we define the parametric RMSNorm as:

$$\mathbf{y} = \frac{\mathbf{x}}{\sqrt{\sum_{i=1}^{D} x_i^2}} * \alpha, where\ \alpha \in R^d$$

4.2 Infused Adapter by Inhibiting and Amplifying Inner Activations

As shown in Figure 2, $(IA)^3$ introduces three extra column vectors, l_v, l_k and l_{ff} for each fine-tune task, where $l_v \in R^{d_v}$, $l_k \in R^{d_k}$ and $l_{\text{ff}} \in R^{d_{\text{ff}}}$.
Mathematically, $X_{attn} = \text{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$ and $X_{\text{ff}} = \sigma\left(l_{\text{ff}} \odot \gamma(W_1 x)\right) W_2$. Each of the introduced tunable vectors is initialized to be all 1's, so when they are added at the beginning, they won't affect training.

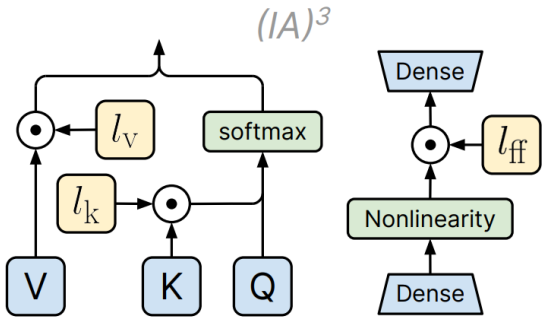

Figure 2: $(IA)^3$

4.3 $(IA)^3$++

We propose our own method $(IA)^3$++ by incorporating the ideas from QK-Norm and $(IA)^3$, and we add an extra vector $l_o \in R^{d_o}$ for the output attention layer. Mathematically, after we have the X_{attn} from above, the output of our approach is $l_o \odot (W_o X_{attn})$, and to stabilize training, we use QK-Norm on Q and K. We also add skip connection and set the initial scale of QK-Norm, α to 0 in order to keep the same output as original model at the beginning of the fine-tuning.

$$X_{\text{attn}} = \text{softmax}\left(\frac{(\text{Norm}(Q) + Q)\,(\text{Norm}(l_k \odot K^T) + l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

$$\text{Out} = l_o \odot (W_o X_{\text{attn}})$$

$$X_{\text{ff}} = \sigma\left(l_{\text{ff}} \odot \gamma(W_1 x)\right) W_2$$

The amount of trainable parameters introduced are $L \times (d_v + d_k + d_o + d_{\text{ff}} + 2 \times d_{rms})$ for L-layer decoder-only transformers. During inference, all of the newly introduced column vectors can be merged with projection matrices to speed up inference. Our contribution combines two state-of-the-art works and brings novelty by adding l_o column vector.
Besides our proposed method, we will also fine-tune the pre-trained model using LoRA, and compare our results with it and the raw pre-trained model.

5 Experiments

In this section, we describe the experiments we have conducted to evaluate the effectiveness of our proposed approach in comparison to LoRA, a baseline pre-trained model, and the original $(IA)^3$ without QK-Norm enhancement and a scaling vector in output projection. We fine-tuned the Llama3.2-1B model on the MMLU and ARC-Challenge datasets, aiming to explore the accuracy and parameter efficiency of our QK-Norm-enhanced $(IA)^3$ method. Our objective is to determine whether our method can match or surpass LoRA and the original $(IA)^3$ in terms of accuracy while maintaining a lower number of trainable parameters. All training and experiments were performed using a single NVIDIA 4090 GPU. The results of our experiments are summarized in Table 1 and 2.

Through our experiments, we aim to demonstrate the following:

- **Accuracy**: We will compare the overall and domain-specific accuracies of our method to LoRA, the baseline model, and the original $(IA)^3$ to assess the effectiveness of our proposed method.
- **Parameter Efficiency**: We expect our method to require fewer trainable parameters than LoRA, as shown in the skeleton table, without sacrificing performance.

As shown in Table 1, our ablation study reveals that adding l_o to $(IA)^3++$ improves performance on MMLU compared to the version without it, highlighting the significance of this component.

Our model also demonstrates superior performance compared to $(IA)^3$ on both the MMLU and ARC-Challenge benchmarks. While achieving comparable average results to LoRA on MMLU, it surpasses LoRA on the ARC-Challenge dataset. Notably, the trainable parameter count for our model is on par with $(IA)^3$ and significantly smaller than that of LoRA, highlighting its efficiency and effectiveness.

Models	MMLU					ARC-Challenge	
	Overall (acc)	Humanities	Other	Social Sciences	STEM	acc	acc_norm
Base Model	0.3430	0.3220	0.3624	0.3780	0.3213	0.3055	0.3507
LoRA	0.4070	0.3853	0.4557	0.4547	0.3448	0.3345	0.3618
$(IA)^3$	0.3848	0.3586	0.4271	0.4166	0.3514	0.3387	0.3763
Ours w/o skip-conn	0.2299	0.2417	0.2398	0.2177	0.2144	0.2150	0.2560
Ours w/o l_o	0.3862	0.3569	0.4294	0.4212	0.3533	0.3447	0.3857
Ours	0.3925	0.3639	0.4380	0.4296	0.3539	0.3447	0.3831

Table 1: Comparison of different models on MMLU and ARC-Challenge datasets. Metrics include accuracy (acc) for both MMLU and ARC-Challenge, and normalized accuracy (acc_norm) for ARC-Challenge. For details on the fine-tuning hyperparameters and prompt template used, refer to Appendix A.

In addition to performance metrics, we also recorded timing information for our experiments. On the MMLU dataset, evaluation times were 01:48 for LoRA, 01:19 for $(IA)^3$, and 08:06 for our method, while on ARC-Challenge, the evaluation times were 00:10, 00:06, and 00:27, respectively. For convergence, LoRA required 2:34:24 on MMLU and 01:23 on ARC-Challenge, $(IA)^3$ needed 9:46:09 and 00:04:38, and our method converged in 3:30:17 and 00:05:04. These results highlight the trade-offs between evaluation speed and convergence time across the methods.

Models	Trainable Params	All Params	Trainable (%)
$(IA)^3$	147,456	1,235,961,856	0.0119
LoRA	11,272,192	1,247,086,592	0.9039
Ours	172,048	1,236,027,440	0.0139

Table 2: Comparison of trainable parameters for $(IA)^3$, LoRA, and Ours. The table shows the number of trainable parameters, total parameters, and the percentage of trainable parameters.

6 Code Overview

Load Model and Tokenizer from Hugging Face, shown in Figure 3
Load MMLU dataset and Construct fine-tune Prompt, shown in Figure 4
Load Arc Challenge dataset and Construct fine-tune Prompt, shown in Figure 5
Fine-tune Model with our own method, shown in Figure 6
Our Foundation Model with RMS-Norm and Skip Connection based on Llama3.2-1B, shown in Figure 7 8

7 Timeline

Table 3 summarizes the time spent on key activities in this project, including researching the Hugging Face community and adapting to its library formats for seamless integration. Significant effort was devoted to implementing our $(IA)^3$++ method by modifying existing code and developing new components. Time was also allocated to conducting baseline experiments, validating our approach, and compiling the insights into this document.

Activity	**Hours Spent**
Reading papers and dataset websites	25
Reading code documentation and adapting to library formats	15
Conducting experiments with existing methods	10
Modifying existing code to implement $(IA)^3$++	20
Running experiments	30
Writing this document	9

Table 3: Hours Spent on Various Research Activities

8 Research Log

Our research journey involved several twists and turns, with challenges that required persistent effort and innovative thinking to overcome. Below, we outline the key phases of the project, the difficulties we encountered, and how we addressed them.

8.1 Initial Challenges with Hugging Face Integration

At the start of the project, we spent significant amount of time researching the Hugging Face community and adapting our implementation to its ecosystem. This involved navigating steps such as account registration, understanding library formats, and aligning our fine-tuning scripts with the framework. While this delayed experimental work, it was essential for ensuring seamless integration and reducing compatibility issues, providing a strong foundation for tackling the core challenges of our fine-tuning method.

8.2 Struggles with Initial Experiments

When we began experimenting with our proposed approach, we encountered a major challenge: the model's performance was unexpectedly poor due to RMSNorm disrupting outputs at the start of fine-tuning. Unlike LayerNorm in methods like LN-LoRA, which integrates with minimal structural impact, RMSNorm altered output scaling, forcing the model to relearn from scratch. Additionally, QK-Norm, effective in pretraining, proved less suitable for fine-tuning. These challenges highlighted the need for a more robust and stable method.

8.3 Breakthrough with Skip Connections

In response to these challenges, we revisited foundational ideas in deep learning and found inspiration in skip connections, first introduced in ResNet [3]. Skip connections were designed to address the vanishing gradient problem by allowing information to bypass certain layers, enabling the network to learn identity mappings.

We adapted this concept to our fine-tuning method by:

1. Initializing the scale parameter of RMSNorm to zero, ensuring that it did not alter the model's outputs at the start of training.
2. Introducing a skip connection, which allowed the model to gradually integrate the scaling effects during fine-tuning.

This adjustment transformed the training process. With skip connections, our method achieved improved performance, outperforming the original $(IA)^3$ approach. Compared to LoRA, our method performed better on some datasets while showing slightly inferior results on others.

9 Conclusion

9.1 Finding

In this work, we propose a novel Parameter-Efficient Fine-Tuning (PEFT) method inspired by the Infused Adapter by Inhibiting and Amplifying Inner Activations ($(IA)^3$) and QK-Norm techniques, while incorporating a new trainable vector for the output projection layer. Our method addresses the limitations of existing PEFT approaches, such as numerical instability and high trainable parameter requirements, by stabilizing attention mechanisms using parametric RMSNorm and enhancing training with skip connections.

9.2 Future Work

- **Extending to Larger Models**: In this work, we focus on Llama3.2-1B due to computational constraints, future work could explore scaling our method to larger models like Llama-7B or 13B to evaluate its applicability to larger models.
- **Generalization Across Tasks**: Although we tested on MMLU and ARC-Challenge, additional benchmarks—such as QA datasets, summarization tasks, and conversational AI—could further validate the generalizability of our approach.
- **Real-World Applications**: Testing our method in practical deployment scenarios (e.g., real-time applications or low-resource settings) would provide insights into its scalability and feasibility for industry use.

By addressing critical limitations in existing PEFT methods, our approach demonstrates significant improvements in efficiency and stability without compromising performance. With continued exploration and refinement, our method holds the potential to become a versatile and effective tool for fine-tuning large language models across diverse tasks and domains.

References

[1] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

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

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[5] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[7] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[8] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022.

[9] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 5:208–215, 2024.

[10] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, et al. Olmoe: Open mixture-of-experts language models. *arXiv preprint arXiv:2409.02060*, 2024.

[11] Zheng Yuan, Jie Zhang, and Shiguang Shan. Fulllora-at: Efficiently boosting the robustness of pretrained vision transformers. *arXiv preprint arXiv:2401.01752*, 2024.

[12] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*, 2023.

A Fine-tuning Hyperparameters

The hyperparameters used for fine-tuning the model are detailed below. These settings were applied using the `SFTTrainer` with the following configurations:

Hyperparameter	Value
max_sequence_length	5020
learning_rate	3e-4
lr_scheduler_type	Linear
per_device_train_batch_size	3
gradient_accumulation_steps	16
num_train_epochs	20 (ARC-Challenge), 3 (MMLU)
fp16	True
logging_steps	1
Optimizer	AdamW (8-bit version)
weight_decay	0.01
warmup_steps	10
seed	0

Table 4: Fine-tuning hyperparameters used in the training process.

These hyperparameters were selected to ensure efficient and stable training while utilizing mixed-precision where supported. Sequence packing was employed to optimize memory utilization and training speed.

Prompt Template

MMLU:

> Davis decided to kill Adams. He set out for Adams's house. Before he got there he saw Brooks, who resembled Adams. Thinking that Brooks was Adams, Davis shot at Brooks. The shot missed Brooks but wounded Case, who was some distance away. Davis had not seen Case. In a prosecution under a statute that proscribes any attempt to commit murder, the district attorney should indicate that the intended victim(s) was/were
>
> A. Adams only.
>
> B. Brooks only.
>
> C. Case only.
>
> D. Adams and Brooks.
>
> Answer: B

ARC-Challenge:

> George wants to warm his hands quickly by rubbing them. Which skin surface will produce the most heat?
> Answer: dry palms

For ARC-Challenge, we don't use the multiple-choice template like what we did on MMLU, the reason is the way we evaluate ARC-Challenge is by appending each of the four choices after the question, and select the one with the largest log-likelihood.

```python
base_model="yuntaozh/custom_llama_residual"
max_seq_length=5020

model = AutoModelForCausalLM.from_pretrained(base_model, trust_remote_code=True)
tokenizer = AutoTokenizer.from_pretrained(base_model, trust_remote_code=True)
# 不加这两句话 trainer.train()可能报错
if tokenizer.pad_token_id is None:
    tokenizer.pad_token_id = tokenizer.eos_token_id
if model.config.pad_token_id is None:
    model.config.pad_token_id = model.config.eos_token_id

print(model)
```

Figure 3: Load Model and Tokenizer

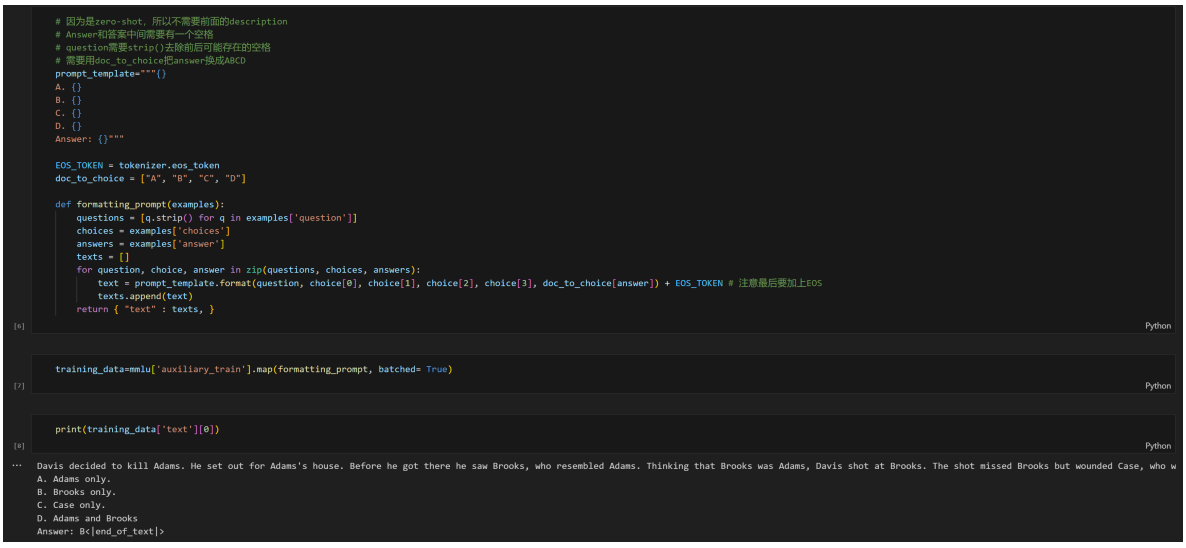

Figure 4: Load MMLU and Construct Prompt

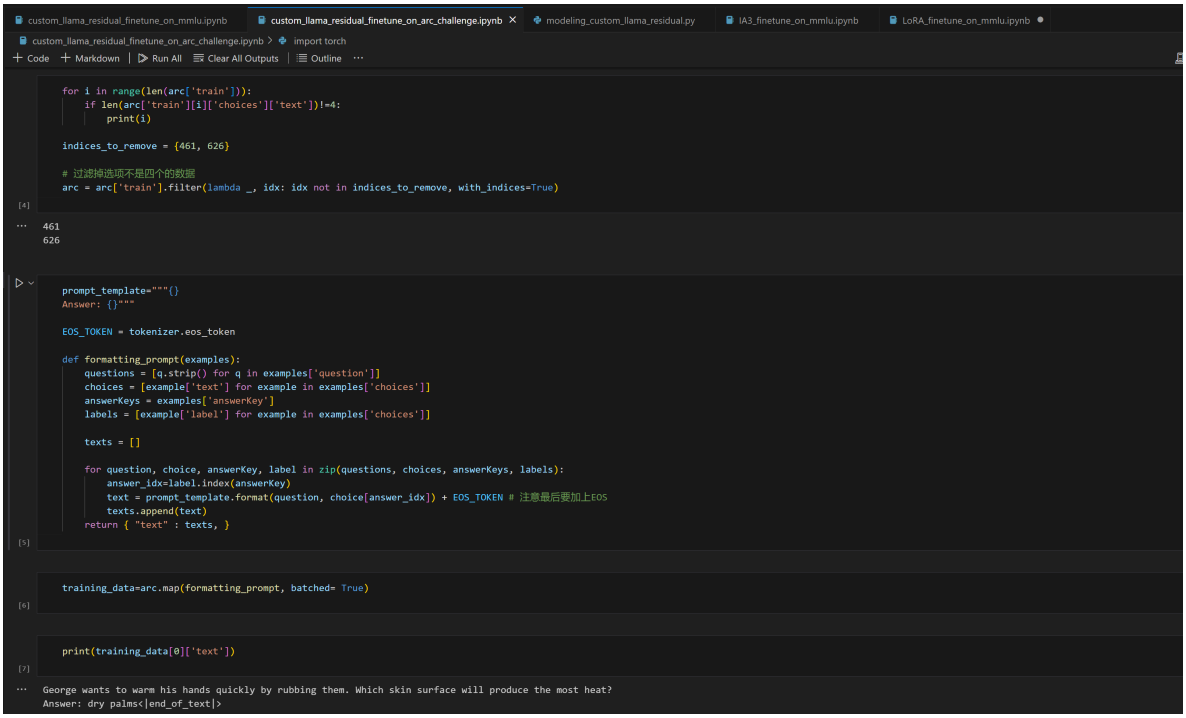

Figure 5: Load Arc Challenge and Construct Prompt

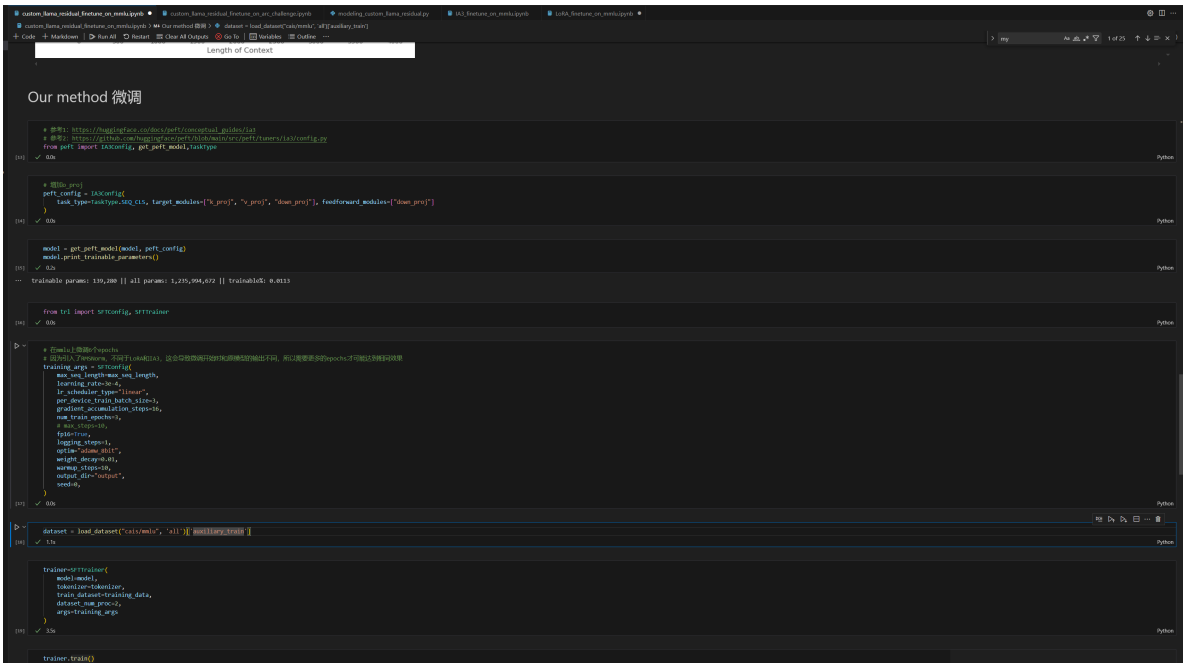

Figure 6: Fine-tune Model Using Our Method

```python
from transformers import LlamaForCausalLM, LlamaPreTrainedModel, LlamaModel
import torch.nn as nn
import torch
from typing import Type, Optional, Tuple
from .configuration_custom_llama_residual import MyLlamaConfig

def apply_to_all_named_modules(module: nn.Module, fn, parent_name: str = ""):
    '''Recursively applies a function to all named modules in a PyTorch module.'''
    # Recurse through children with their instance names
    for name, child in module.named_children():
        # Construct the full name path for the current module
        full_name = parent_name + ("." if parent_name else "") + name
        # Apply the function to the current module
        fn(full_name, module, name, child)
        # Recurse into the child module
        apply_to_all_named_modules(child, fn, full_name)

def print_model_layers(model: nn.Module):
    '''Recursively prints the variable names of all layers in a PyTorch model and their type.'''
    apply_to_all_named_modules(
        model,
        lambda full_name, module, name, child: print(f"{full_name}: {child.__class__.__name__}")
    )

def replace_module_by_class_and_name(module: Type[nn.Module],
                                     target_class: str,
                                     target_name: str,
                                     replacement_class: Type[nn.Module],
                                     other_init_args: Tuple = ()):
    '''
    替换类名为target_class, 实例名target_name为的模块
    '''

    # Lambda function used to replace the target module with the replacement module
    def replace_module_by_class_and_name_fn(full_name, module, name, child):
        # print(f"{full_name}: {child.__class__.__name__}")
        # If the current module is of the target class, replace it
        if name == target_name and child.__class__.__name__ == target_class:
            print("Replacing: ", target_class, replacement_class)
            # 用原本的attention层初始化
            setattr(module, name, replacement_class(child, *other_init_args))

    # Recursively apply the replacement function to all named modules
    apply_to_all_named_modules(
        module,
        replace_module_by_class_and_name_fn,
    )

class MyRMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super(MyRMSNorm, self).__init__()
        self.eps = eps
        self.scale = nn.Parameter(torch.zeros(dim)) # 把scale初始化为0

    def forward(self, x):
        rms = x.pow(2).mean(-1, keepdim=True).sqrt()
        x_normed = x / (rms + self.eps)
        return self.scale * x_normed

class MyLinear(nn.Linear):
    # 之所以要继承nn.Linear是因为tuners\ia3\model.py中的_create_new_module定死了只能接受target_base_layer为torch.nn.Linear
    # 设置为(1,1)和out_features是为了(几乎)不增加参数量，因为虽然我们继承了nn.Linear，但根本不会去用其中的参数
    def __init__(self, old_linear: nn.Linear, out_features):
        super().__init__(1,1,bias=False)
        self.linear = old_linear
        self.rms_norm=MyRMSNorm(out_features, eps=1e-6)

    def forward(self, x):
        x = self.linear(x)
        return self.rms_norm(x)+x

class CustomLlamaModel(LlamaModel):
    config_class = MyLlamaConfig

    def __init__(self, config):
        super().__init__(config)
        # Replace 'q_proj' and 'k_proj' layers with 'MyLinear'
        replace_module_by_class_and_name(self.layers, 'Linear', 'q_proj', MyLinear, (2048,))
        replace_module_by_class_and_name(self.layers, 'Linear', 'k_proj', MyLinear, (512,))
        # Initialize weights and apply final processing
        self.post_init()

    def apply_custom_modifications(self):
        def replace_module_by_class_and_name(module: nn.Module,
                                             target_class: str,
                                             target_name: str,
                                             replacement_class: Type[nn.Module],
                                             other_init_args: Tuple = ()):
            def replace_module_by_class_and_name_fn(full_name, module, name, child):
                if name == target_name and child.__class__.__name__ == target_class:
                    setattr(module, name, replacement_class(child, *other_init_args))
            apply_to_all_named_modules(module, replace_module_by_class_and_name_fn)

class CustomLlamaForCausalLM(LlamaForCausalLM):
    config_class = MyLlamaConfig

    def __init__(self, config):
        super().__init__(config)
        self.model = CustomLlamaModel(config)
        self.post_init()

    def save_checkpoint(self, dir):
        # to bypass the code line 2291 in transformers.trainer
        pass
```

Figure 7: Our Foundation Model with RMS-Norm and Skip Connection

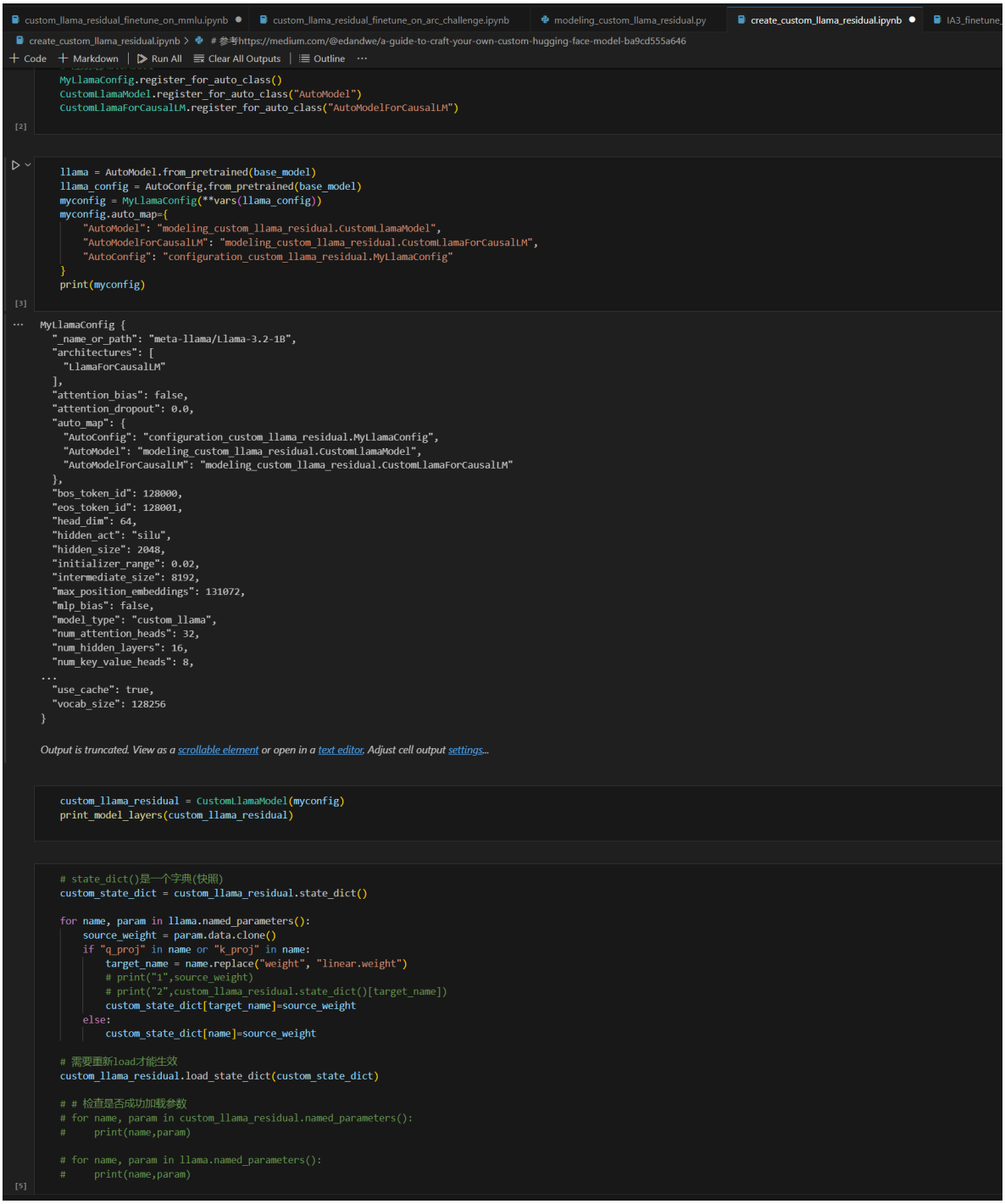

Figure 8: Our Foundation Model with RMS-Norm and Skip Connection