HW1: LLM Implementation and Finetuning

Hantao Lou
Yuanpei College
Peking University
2200017789
lht_pku@stu.pku.edu.cn

1 Introduction

The code for this project is in https://github.com/htlou/Alignment_hw/tree/main/hw1.

This report focuses on the implementation and fine-tuning of naive LLMs. Specifically, the report contains the following aspects:

- **Tokenization with Byte Pair Encoding (BPE)**: We implemented a Byte Pair Encoding (BPE) tokenizer from scratch and trained it on an example text. This section covers the theoretical foundations of BPE, the implementation details, and an evaluation of the tokenizer' s ability to encode and decode text accurately. Additionally, we compared the output of our custom tokenizer with Hugging Face's GPT-2 tokenizer.
- **Pretraining of a Decoder-Only Transformer (GPT-2)**: We reproduced a GPT-2-like architecture by building its core components, including multi-head attention, positional embeddings, layer normalization, and feedforward networks. Although large-scale pretraining was not performed due to computational limitations, this section provides a detailed explanation of the implementation process, incremental development through Git commits, and key insights into the model' s functionality.
- **Fine-Tuning with Low-Rank Adaptation (LoRA)**: We implemented and experimented with LoRA, a parameter-efficient fine-tuning method that adds low-rank updates to pretrained models. This section describes the mathematical framework of LoRA, its integration into a GPT-2 model, and experimental results on the Alpaca dataset. We analyzed the effects of different LoRA ranks (e.g., 1, 2, 4, 8, 16, 32) on fine-tuning performance, and compared them with original models and full-parameter fine-tuned models.

2 Tokenization

The program output of this section is placed in `bpe/output.log`.

2.1 BPE Algorithm Overview

Byte Pair Encoding (BPE) is an unsupervised algorithm used for subword tokenization. It works by iteratively merging the most frequent pair of tokens (initially individual characters or bytes) to build a vocabulary of subword units. The BPE-based tokenizer training for large language models (LLMs) involves the following steps:

1. Initialize the vocabulary with unique bytes in the data.
2. Iteratively merge the most frequent adjacent token pairs until the desired vocabulary size is reached.

3. Store the merge rules and vocabulary for encoding and decoding text.

2.2 Tokenizer Implementation

We implemented a custom BPE tokenizer and its training process in `bpe/train.py` and `bpe/tokenizer.py`. The implementation consists of:

- `train(data_bytes, vocab_size)`: Trains the tokenizer by creating a vocabulary of size `vocab_size`.
- `encode(text)`: Encodes a string into a sequence of token IDs.
- `decode(ids)`: Decodes a sequence of token IDs back into a string.

The training process involves initializing the vocabulary with byte-level tokens, calculating pair frequencies, merging the most frequent pairs, and updating the tokenization rules. The encoding process applies these rules greedily to compress sequences of tokens, and the decoding process reconstructs the original text using the learned tokens.

2.3 Experiments and Results

2.3.1 Training on Manual Data

We trained the tokenizer using the byte-level data from `manual.txt`, extracted from the Peking University Graduate Handbook (2023 edition), with a vocabulary size of 1024. The first 10 and last 10 tokens in the vocabulary are as follows:

- **First 10 tokens:** `{\n, \ , %, &, (, ), +, ,, -, .}`
- **Last 10 tokens:** Byte sequences representing the Chinese tokens.

The reconstruction of `manual.txt` after encoding and decoding was successful, verifying the functionality of the tokenizer.

2.3.2 Comparison with GPT-2 Tokenizer

We compared our BPE tokenizer with the GPT-2 tokenizer on two sample texts:

1. **English text:** A historical description of Peking University.
2. **Chinese text:** A description of PhD thesis requirements.

The results show the following token lengths:

- **English text**: BPE tokenizer produced 938 tokens, while GPT-2 produced 185 tokens.
- **Chinese text**: BPE tokenizer produced 113 tokens, while GPT-2 produced 306 tokens.

The differences arise due to:

- **Training Dataset**: Our Custom BPE is trained on a piece of text contained mainly Chinese, so it much more optimized in Chinese context compared to English context.
- **Granularity:** GPT-2 uses larger subword units optimized for general text, leading to fewer tokens.
- **Encoding strategy:** Custom BPE starts with byte-level granularity, producing finer splits in text.

2.4 Answers to Questions

1. Unicode functions: `ord()` converts a character to Unicode; `chr()` converts Unicode to a character. Example:
   - ” 北”: Unicode = 21271; ” 大”: Unicode = 22823.

- Unicode 22823, 27169, 22411 correspond to " 大", " 模", " 型".

2. **Vocabulary size trade-off:** Larger vocabularies capture more semantic meaning and reduce token count, improving efficiency for common words. However, it requires more memory and can lead to overfitting. Smaller vocabularies generalize better and reduce overfitting risks, but they increase the token count, making the model slower and less efficient for long texts.
3. **LLM limitations on string manipulation:** Limited by the tokenizer, LMs cannot recognize the symbolic relation between tokens, especially those tokens aggregated by other tokens. Also, LMs are trained primarily for probabilistic text generation, not deterministic operations. Tasks like string reversal require explicit algorithmic reasoning, which is outside the model's primary training objective.
4. **Reason of Non-English performance:** Scarcity of non-English text in training data, and LMs need to generalize from English, which is harder than direct learning.
5. **Arithmetic issues:** Limited by the tokenizer, LMs cannot recognize numbers in a symbolic perspective. For example, for number `678`, it might be tokenized into `6`, `78` and this challenges the LMs to recognize that number `6` is on the hundred digit.
6. **Python coding difficulties:** Limited exposure to precise programming tasks in training datasets. Also, functions and data structures are broken by the tokenization process.
7. **Behavior with `<|endoftext|>`:** This token is treated as a stop signal during generation.
8. **SolidGoldMagikarp crash:** This token might be built in the tokenizer, but there's scarce training data on this token, so strange behaviors take place.
9. **YAML vs JSON:** In YAML files, the data is presented directly, and the brackets in JSON files make LLMs harder to recognize the structure and content of the file.
10. **LLM is not end-to-end:** Intermediate processing steps like tokenization break the end-to-end paradigm.

3 LLM Implementation

3.1 Code Implementation

See https://github.com/htlou/Alignment_hw/tree/main/hw1/pretrain for the commit history and the source code. We provide a brief record of commit history here in Figure 1, 2, 3, 4, 5.

3.2 Experiment Report

This section documents the process of implementing and validating the pretraining framework, as well as the step-by-step improvements applied to enhance the training efficiency and stability. Each subsection corresponds to the major milestones achieved during the experiment.

3.2.1 Initial Framework Setup (Section 1)

The first step was to set up the GPT-2 framework. To ensure compatibility with Hugging Face's GPT-2 model parameters, all parameter names were aligned with their counterparts in the Hugging Face implementation. This alignment facilitated seamless loading of pretrained parameters for validation purposes. Also, for a clear code architecture, we modified the code files and separated modeling_gpt2, train, and dataloader.

After building the architecture, the forward function was implemented, followed by a utility function, `from_pretrained`, which allowed loading Hugging Face's pre-trained GPT-2 weights for validation. Validation confirmed no issues with the architecture.

Next, a small dataset, *Shakespeare*, was used to test training functionality. A basic dataloader was written to load the dataset, and the training setup included a loss function, an

optimizer, and parameter initialization as described in relevant papers. The training loop was successfully executed, providing the foundational training framework.

3.2.2 Training Optimization (Section 2)

To improve training efficiency, multiple optimizations were introduced:

- **TF32 Precision:** The precision was adjusted to TF32, leveraging the GPU's tensor core capabilities.
- **Model Compilation:** The training pipeline was compiled using `model.compile()`, optimizing backend execution.
- **Flash Attention:** Flash attention techniques were integrated to accelerate attention computation.
- **Vocabulary Size Adjustment:** The vocabulary size was adjusted to the nearest power of 2 for improved computational efficiency.

3.2.3 Training Stabilization and Multi-GPU Utilization (Section 3 and 4)

To enhance training stability, the following strategies were implemented:

- **Clip Loss:** Gradient clipping was added to stabilize training and mitigate exploding gradients.
- **Learning Rate Scheduler:** A scheduler was introduced to adjust the learning rate dynamically, reducing it over the course of training.

For scaling the training process, multi-GPU support was implemented using the `torch.distributed.data_parallel` (DDP) library. Each GPU was assigned a unique process identifier, ensuring distributed training across GPUs. The dataloader was modified to distribute data evenly across GPUs, with each GPU loading a distinct subset of the dataset.

To evaluate training progress, *HellaSwag* was used for intermediate validation. This step provided insight into the model's performance after specific training milestones.

3.2.4 Reflections and Learning Notes

Each commit in the implementation process was a critical step towards understanding GPT-2's architecture, training mechanisms, and optimization strategies. The gradual enhancements reflected in the commit history demonstrated incremental improvements in both training speed and stability. Detailed notes were maintained for each step, emphasizing the purpose and impact of the changes made.

For detailed commit history and source code, see https://github.com/htlou/Alignment_hw/tree/main/hw1/pretrain (or pretrain/ in submission file). The figures (1, 2, 3, 4, 5) illustrate the chronological progression of code implementation.

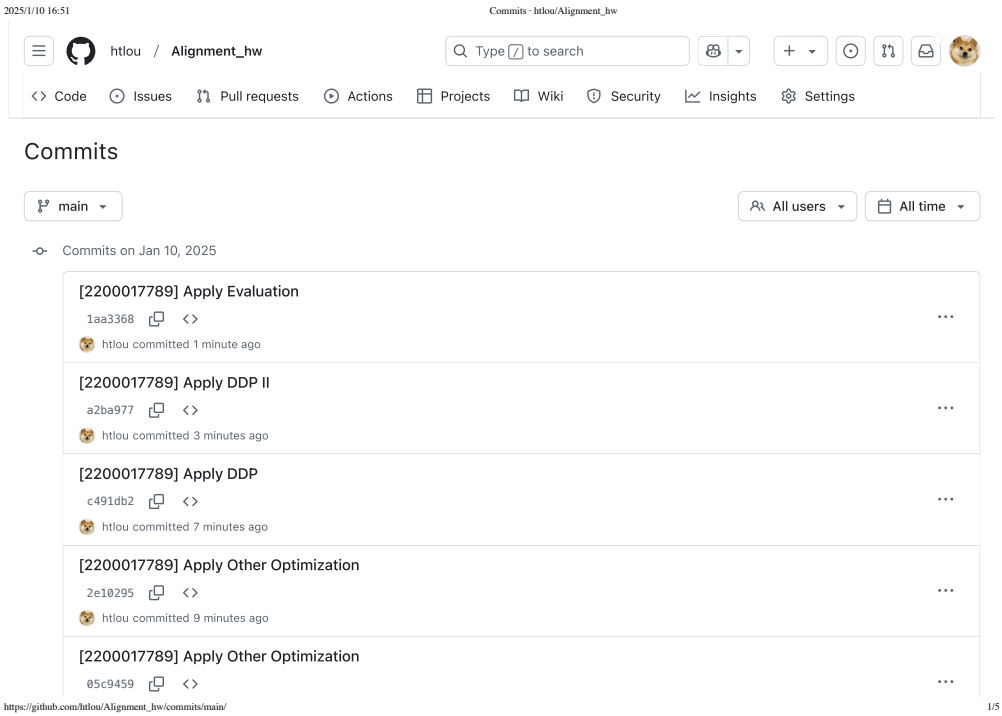


Figure 1: **Commit History of GPT-2 Implementation (Page 1).**

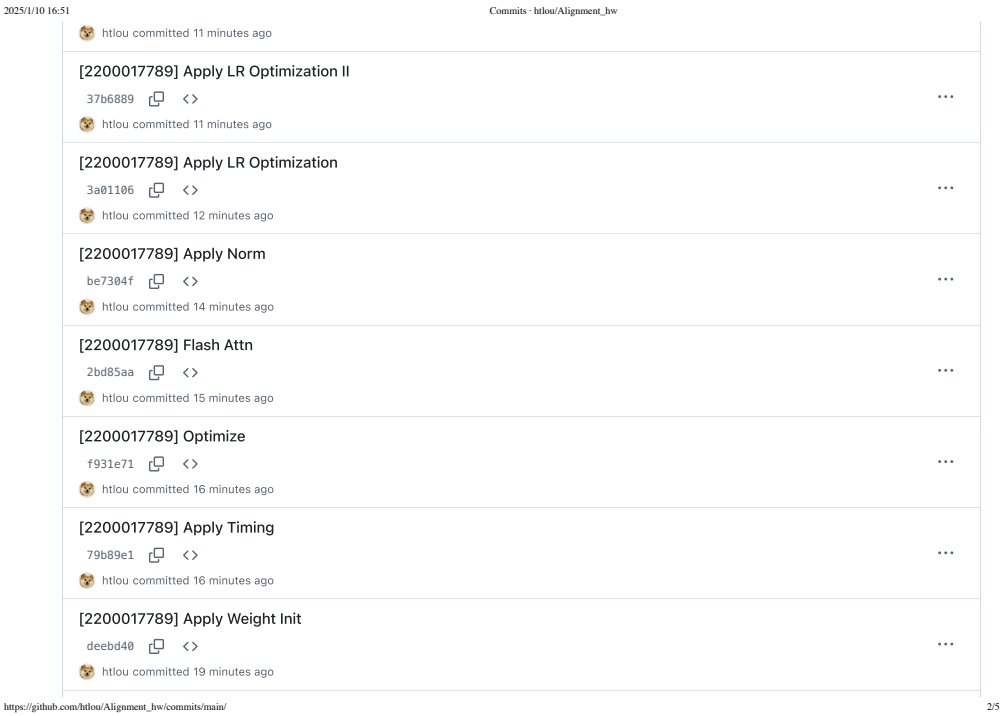


Figure 2: **Commit History of GPT-2 Implementation (Page 2).**

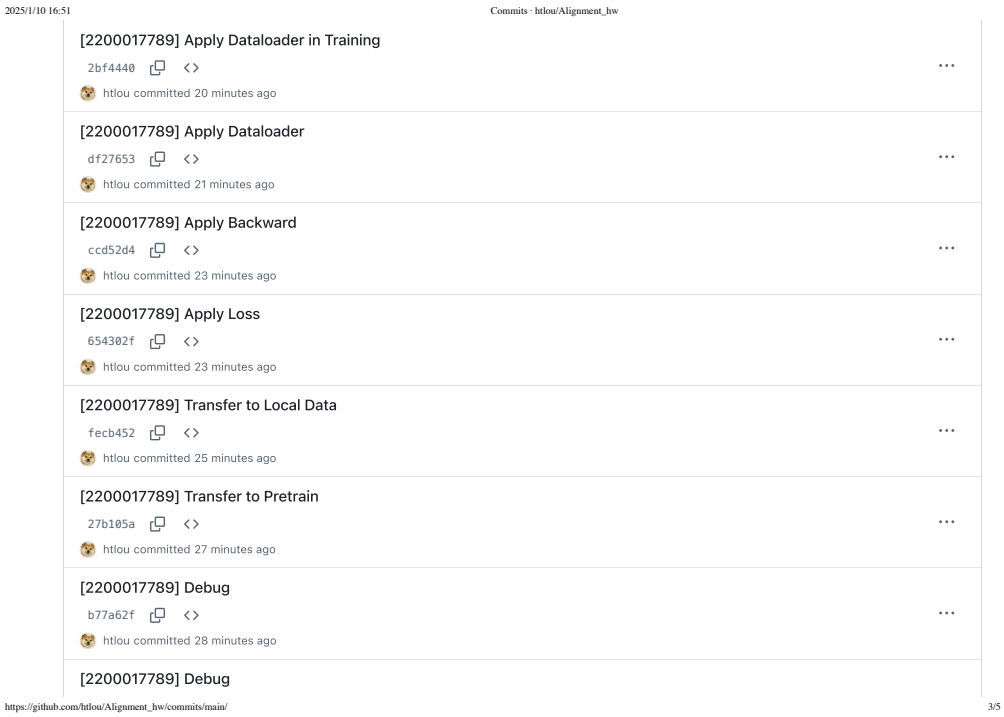


Figure 3: **Commit History of GPT-2 Implementation (Page 3).**

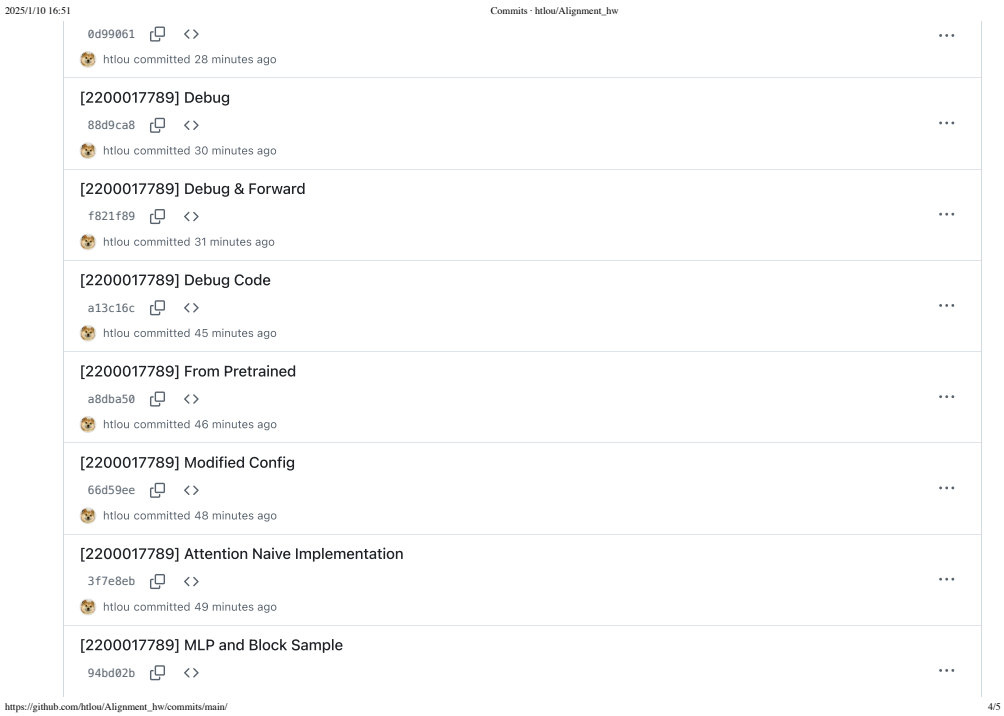


Figure 4: **Commit History of GPT-2 Implementation (Page 4).**

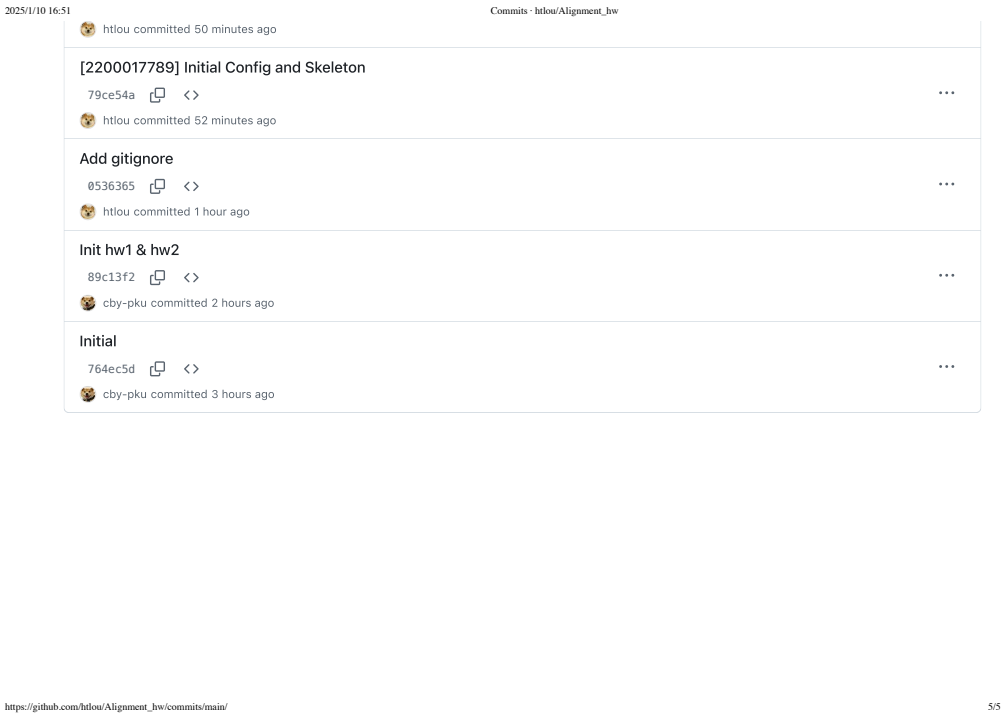


Figure 5: **Commit History of GPT-2 Implementation (Page 5).**

4 LoRA Fine-tuning

4.1 LoRA Implementation and Explanation

This subsection explains the complete implementation of the `lora.py` script. The implementation focuses on adding LoRA weights, freezing original parameters, and enabling fine-tuning through LoRA-specific parameters. Below, we detail the key components and their contributions to the final design.

4.1.1 LoRALinear Class

The `LoRALinear` class modifies a standard linear layer by introducing two low-rank weight matrices, `lora_right_weight` and `lora_left_weight`. These matrices are learned during fine-tuning, while the original weights are frozen.

Key implementation steps:

- **Parameter Initialization**: The low-rank weight matrices are initialized following the standard practice in the LoRA paper: `lora_right_weight` uses Kaiming initialization, while `lora_left_weight` is initialized to zero. This ensures a stable starting point for learning.
- **Freezing Original Parameters:** The original weight and bias parameters (`self.weight` and `self.bias`) are frozen by setting `requires_grad=False`, ensuring that only LoRA parameters are updated during fine-tuning.
- **Forward Pass:** The forward function adds a low-rank residual computed using LoRA weights:

  $$\text{output} = X \cdot W^T + \text{bias} + \text{scaling} \cdot ((X \cdot W_{\text{right}}) \cdot W_{\text{left}})$$

  Here, W_{right} and W_{left} represent the LoRA-specific weights.

4.1.2 Converting Layers to LoRA

The `convert_linear_layer_to_lora` function identifies and replaces eligible linear layers with `LoRALinear` layers.

Steps:

1. **Module Identification:** Linear layers and `transformers.pytorch_utils.Conv1D` modules matching the specified `part_module_name` are targeted for replacement.
2. **Replacement with LoRA:** Each eligible module is replaced by a corresponding `LoRALinear` instance, retaining the same device and data type as the original module.

4.1.3 Optimizing Only LoRA Parameters

The `only_optimize_lora_parameters` function freezes all parameters except the LoRA weights (`lora_right_weight` and `lora_left_weight`).

Implementation:

- Set `requires_grad=False` for all model parameters by default.
- Unfreeze LoRA-specific weights by setting `requires_grad=True`.

This function ensures efficient fine-tuning with minimal computational overhead, as only a small subset of parameters is optimized.

4.1.4 Saving and Loading LoRA Parameters

The `get_lora_state_dict` function extracts and saves LoRA-specific parameters into a state dictionary. The saved state can later be used for reloading LoRA weights during inference or transfer learning.

Implementation:

- Iterate through the model's modules to identify `LoRALinear` instances.
- Save `lora_right_weight` and `lora_left_weight` into a dictionary, with keys formatted as `name.lora_right_weight` and `name.lora_left_weight`.

4.2 Experiment Results

The experiment results are as follows:

4.2.1 LoRA rank0, *i.e.*, full-paramter fine-tuning (Figure 6)

At LoRA rank 0, the model undergoes full-parameter fine-tuning, which serves as the baseline for comparison. The initial train and evaluation losses are high, indicating underfitting. The training loss decreases rapidly with large fluctuations as fine-tuning begins, but the evaluation loss remains relatively stagnant. This suggests that full-parameter fine-tuning may not be optimal for small-scale tasks or limited datasets due to overfitting.

4.2.2 LoRA rank 1-32 (Figures 7)

With LoRA, the model does not experience overfit, and the evaluation loss decreases along with train loss. However, the performance of different LoRA ranks seems to be similar, with the loss curve looks almost the same.

4.3 Case Analysis

Detailed outputs are provided in `finetune/results` for further examination.

From qualitative observations, LoRA fine-tuned models consistently produce more creative and detailed outputs compared to the original and full-parameter fine-tuned models. They

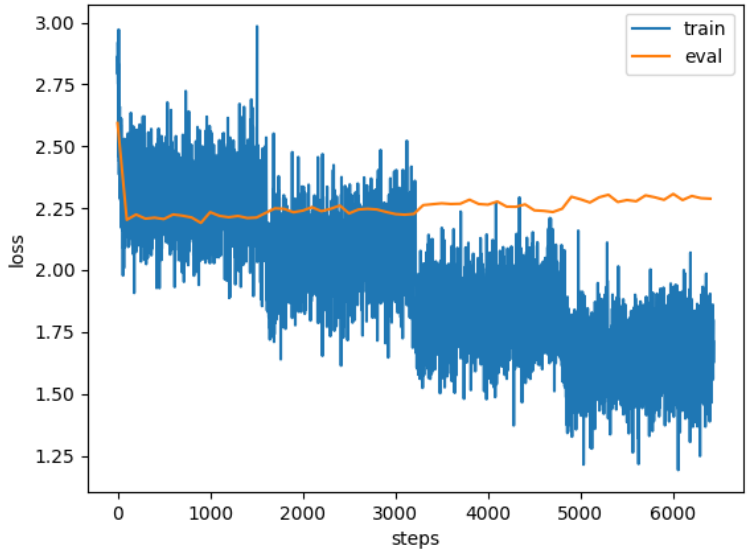
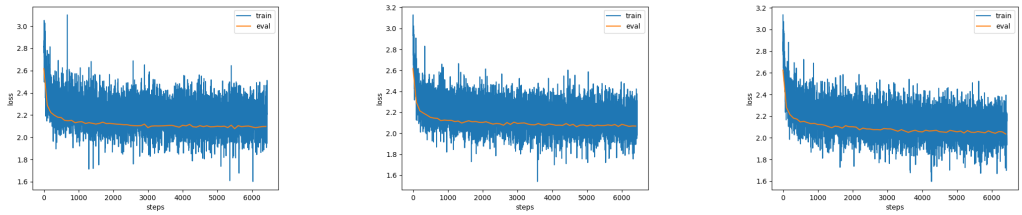


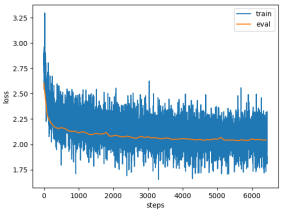
Figure 6: Learning curve of full-parameter fine-tuning. The overfitting phenomenon is very obvious.



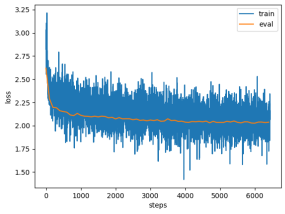
(a) 1 Rank

(b) 2 Rank

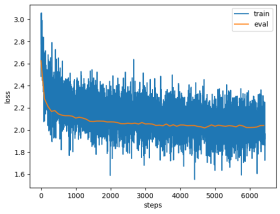
(c) 4 Rank



(d) 8 Rank



(e) 16 Rank



(f) 32 Rank

Figure 7: Learning curves for LoRA fine-tuning experiments with varying steps. Each subplot shows the train and evaluation loss over steps.

demonstrate a tendency to generate longer, more nuanced responses, often exceeding the maximum token limits imposed during generation. This suggests that LoRA fine-tuning enables the model to better utilize its latent capacity for text generation, potentially due to the focused optimization of task-relevant parameters.

4.4 Conclusion

The results indicate that LoRA fine-tuning efficiently minimizes train and eval losses with minimal parameter updates.