Moonshot Interview: 24 Game with o1-Paradigm LLM

Yiyan Liao
School of Life Sciences, Peking University

Abstract
This paper explores the application of chain-of-thought (CoT) prompting to enhance the performance of large-scale language models (LLMs) in complex reasoning tasks, specifically through a case study of the 24 Game. We generated training data via a depth-first search (DFS) approach and fine-tuned the Qwen2-0.5B model utilizing CoT outputs, achieving a commendable 76% accuracy in evaluation dataset. Our analysis highlights the effectiveness of CoT in fostering deeper problem-solving capabilities by requiring explicit step-by-step reasoning. We propose that further exploration of CoT applications in complex domains could provide valuable insights into LLM reasoning processes. The codes are publicly available at https://github.com/yiyanliao/Moonshot-Interview-24-Game-with-o1-Paradigm-LLM (please note that the model files are not included due to their size). Key process data and code (including modifications and self-written components) have been provided in the attached zip file.

Contents

1 Introduction

1.1 24 Game

The 24 Game is an arithmetic puzzle where the objective is to manipulate four integers using operations such as addition (+), subtraction (-), multiplication (*), and division (/) to achieve a final result of 24. For example, when we got numbers 3, 3, 8, 8, we can calculate 24 with:

$$8 \div (3 - 8 \div 3) = 24$$

Traditional algorithms, including Monte-Carlo Tree Search (MCTS) and A* search, provide efficient methods for solving the 24 Game. While both are search algorithms, A* is a heuristic search algorithm, whereas MCTS is based on stochastic simulation.

1.2 LLMs & CoT

Although traditional search algorithms have proven effective for the 24 Game, leveraging large language models (LLMs) presents an exciting and innovative approach to this problem. Gandhi et al. (2024)[1] demonstrate that language models can learn search strategies by representing the problem-solving process as a "stream of search" (SoS), thereby enhancing their capability to tackle symbolic reasoning tasks.

LLMs excel at reasoning and performing step-by-step computations when given well-structured prompts, a technique commonly referred to as Chain-of-Thought (CoT) reasoning. With carefully designed prompts, CoT reasoning enables LLMs to decompose complex problems into manageable, sequential steps, closely resembling the way a human thinks through a solution. OpenAI-o1 is OpenAI's latest series of LLMs, optimized for more complex inference capabilities, which is trained with large-scale reinforcement learning to reason using chain of thought[2]. Recently, Violet Xiang et al. (2025) introduced the concept of Meta-CoT, emphasizing the significance of underlying reasoning[3].

In this work, we explore the application of CoT reasoning in LLMs as a novel approach to solving the 24 Game. By carefully fine-tuning Qwen2-0.5B model with designing prompts and facilitating step-by-step thinking throughout the solution generation process, our objective is to evaluate the effectiveness and reliability of LLMs in addressing this combinatorial arithmetic problem.

2 Data Preparation

2.1 Data Generation

We applied the methods outlined by Gandhi et al. (2024)[1] to generate data for fine-tuned training and performance evaluation. By modifying the original script located at `src/countdown_generate.py`, we created a dataset specifically targeting the number 24 using a depth-first search (DFS) strategy. In total, we generated 11,000 data items, stored in a json file, each formatted as follows:

```
{'nums': [33, 32, 68, 27],
 'target': 24,
 'solution': ['33+32=65', '68-27=41', '65-41=24'],
 'search_path':
    "Current State: 24:[33, 32, 68, 27], Operations: []\n
    Exploring Operation: 68-27=41, Resulting Numbers: [33, 32, 41]\n
    Generated Node #0,0: 24:[33, 32, 41] Operation: 68-27=41\n
    ......
    Exploring Operation: 32-8=24, Resulting Numbers: [24]\n
    24,24 equal: Goal Reached\n",
 'rating': 0.9869791666666666,
 'search_type': 'bfs_3',
 'optimal_path':
    "Current State: 24:[33, 32, 68, 27], Operations: []\n
    Exploring Operation: 33+32=65, Resulting Numbers: [68, 27, 65]\n
    Generated Node #2: [68, 27, 65] from Operation: 33+32=65\n
    Current State: 24:[68, 27, 65], Operations: ['33+32=65']\n
    Exploring Operation: 68-27=41, Resulting Numbers: [65, 41]\n
    Generated Node #3: [65, 41] from Operation: 68-27=41\n
    Current State: 24:[65, 41], Operations: ['33+32=65', '68-27=41']\n
    Exploring Operation: 65-41=24, Resulting Numbers: [24]\n
    24,24 equal: Goal Reached\n",
 'heuristic': 'sum_heuristic'}
```

Code 1. Example of raw data generated by depth first search (DFS)

'nums': 4 numbers as input. 'target': 24 in this case. 'solution': final solution path. 'search_path': DFS records. 'rating': the longer search_path is, the lower. 'optimal_path': final answer. 'heuristic': heuristic method choice.

2.2 Data Processing

We processed the data using the following steps:

- Duplicate Removal: Initially, we eliminated duplicates based on the nums section, resulting in 10,901 unique data items.

- Filtering for Quality: Next, we filtered the dataset to ensure it contained the necessary Chain of Thought (CoT) leading to the target. Specifically, we removed entries in which the rating was 0.0, resulting in a total of 5,576 valid items.

- Data Splitting: Then, we split the dataset into training (5,476 items) and testing (100 items) subsets using random sampling. This was done to maintain a consistent difficulty distribution, with mean lengths of 4,247 for the training set and 4,326 for the testing set, saved in `train_set.json` and `test_set.json`, respectively. The distributions of these datasets are illustrated in Figure. 1.

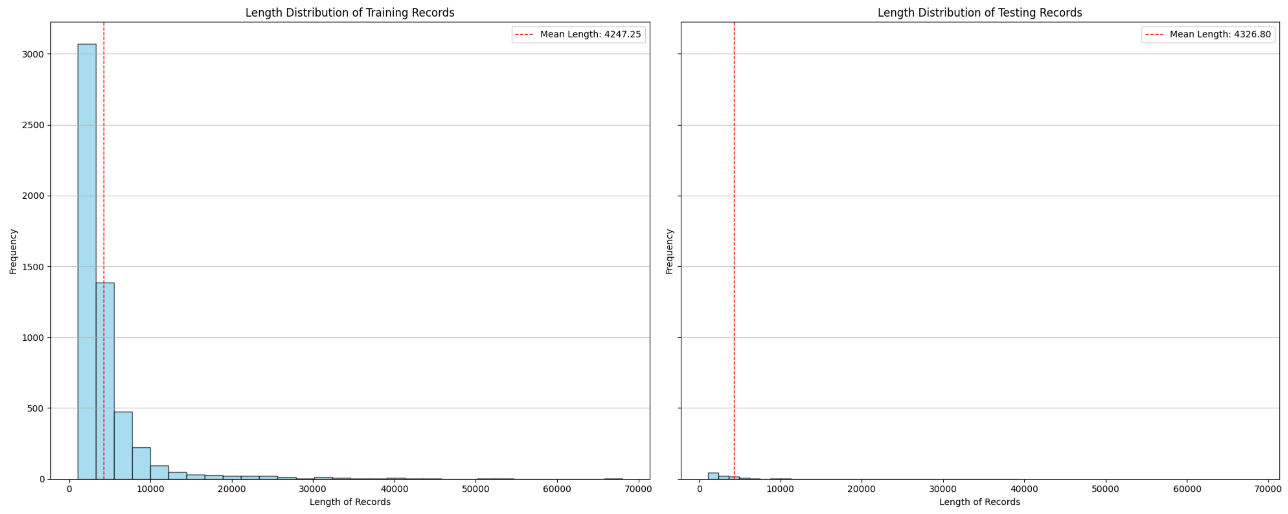

Figure 1: Length distribution of dataset
The distribution shows the variances in data lengths for train and test datasets. Red dashed line show mean of item lengths.

- ShareGPT Formatting: To fine-tune and test the model, we formatted our data into ShareGPT form as below, recording to `https://github.com/QwenLM/Qwen2.5/blob/main/examples/llama-factory/finetune-zh.md`. Please note that we exclusively use the `'search_path'` section for efficient fine-tuning. We initially attempted to incorporate the `'solution'` and `'optimal_path'` sections, but they did not align with the conclusions drawn from the `'search_path'` section. As a result, we decided to remove them. Processed data was stored in `train_set_sharegpt.jsonl` and `test_set_sharegpt.jsonl` files.

```
{
    "messages": [
        {
            "role": "system",
            "content": "You are a helpful assistant with the ability to solve
            24-point puzzles. Show your reasoning step by step."
        },
        {
            "role": "user",
            "content": "Find a sequence of +,-,/,* which can be applied to the
            numbers 95, 87, 42, 3 to get to 24, where each of the given numbers
            is used exactly once."
        },
        {
            "role": "assistant",
            "content": "Search Path:\n
            Current State: 24:[95, 87, 42, 3], Operations: []\n
            Exploring Operation: 87-42=45, Resulting Numbers: [95, 3, 45]\n
            ......
            Current State: 24:[53, 29], Operations: ['95-42=53', '87/3=29']\n
            Exploring Operation: 53-29=24, Resulting Numbers: [24]\n
            24,24 equal: Goal Reached\n\n\n"
        }
    ]
}
```

Code 2. Example of ShareGPT formatted data

3 Training

3.1 Training Condition

We utilized Google Colab for data preparation, fine-tuning, and evaluation, as illustrated in Table 1, which contains hardware conditions and fine-tuning framework information.

Table 1: Key conditions of training condition

Specification	**Details**
OS	Linux-6.1.85+-x86_64-with-glibc2.35
GPU type	NVIDIA A100-SXM4-40GB
GPU count	1
CUDA Version	12.2
Driver Version	535.104.05
Python Version	3.11.11
Transformers Version	4.46.1
Pytorch Version	2.0.0+cu117

Specification	**Details**
Datasets Version	3.1.0
Tokenizers Version	0.20.3

3.2 Model & Fine-tuning

We selected the Qwen2-0.5B base model as a toy model to conduct these experiments, utilizing LLaMA-Factory as a robust framework for fine-tuning this model, proposed by Yaowei Zheng et al. (2024)[4].

To ensure a smooth fine-tuning process, we needed to generate or modify several files. Specifically, we copied the `dataset_info.json` file from `LLaMA-Factory/data/` to `Qwen2.5/examples/llama-factory/data`, adding the following code:

```
"qwen_train_data": {
  "file_name": "/content/Moonshot_Interview/train_set_sharegpt.jsonl",
  "formatting": "sharegpt",
  "columns": {
    "messages": "messages"
  },
  "tags": {
    "role_tag": "role",
    "content_tag": "content",
    "user_tag": "user",
    "assistant_tag": "assistant",
    "system_tag": "system"
  }
}
```

Code 3. Configure information added to Qwen2.5/examples/llama-factory/data/dataset_info.json

To configure the parameter file, we modified the `Qwen2.5/examples/llama-factory/qwen2-7b-full-sft.yaml` file to create a new `qwen2-0.5b-full-sft.yaml` file in the same directory. With the assistance of ChatGPT, we adjusted several original parameters to achieve better performance in this model. The key components of the final parameter configuration are presented in Table 2.

Table 2: Key parameters selection for fine-tuning (modified from Qwen2.5 github)

Parameter	**Meaning**	**Setting**
model_name_or_path	The model name or path	Qwen/Qwen2-0.5B-Instruct
stage	Training stage	sft
do_train	Training (true) or evaluation (false)	true
finetuning_type	Ways to fine-tune	full
deepspeed	deepspeed file path	/content/Moonshot_Interview/ LLaMA-Factory/examples/ deepspeed/ds_z0_config.json
dataset	Dataset to fine-tune	qwen_train_data
template	Dataset template	qwen
output_dir	Output path	saves/qwen2-0.5b/full/sft
logging_steps	Log output step interval	20

Parameter	**Meaning**	**Setting**
save_steps	Save interval for model breakpoints	500
per_device_train_batch_size	Batch size trained on each device	2
gradient_accumulation_steps	Step number of gradient accumulation	8
learning_rate	Learning rate	1.0e-5
lr_scheduler_type	Learning rate curve	cosine
num_train_epochs	Number of training cycles	3.0
bf16	Whether to use BF16 format	true

After modifying these files, we initiated the fine-tuning of the model by executing `!FORCE_TORCHRUN=1 llamafactory-cli train qwen2-0.5b-full-sft.yaml`. Key indicators are summarized in Table 3, and the training loss curve is depicted in Figure 2.

Table 3: Key indicators of fine-tuning result

Indicator	**Value**
total_flos	3.008388464954573e+16
train_loss	0.029482580479476 36
train_runtime	1516.3411
train_samples_per_second	9.75
train_steps_per_second	0.609

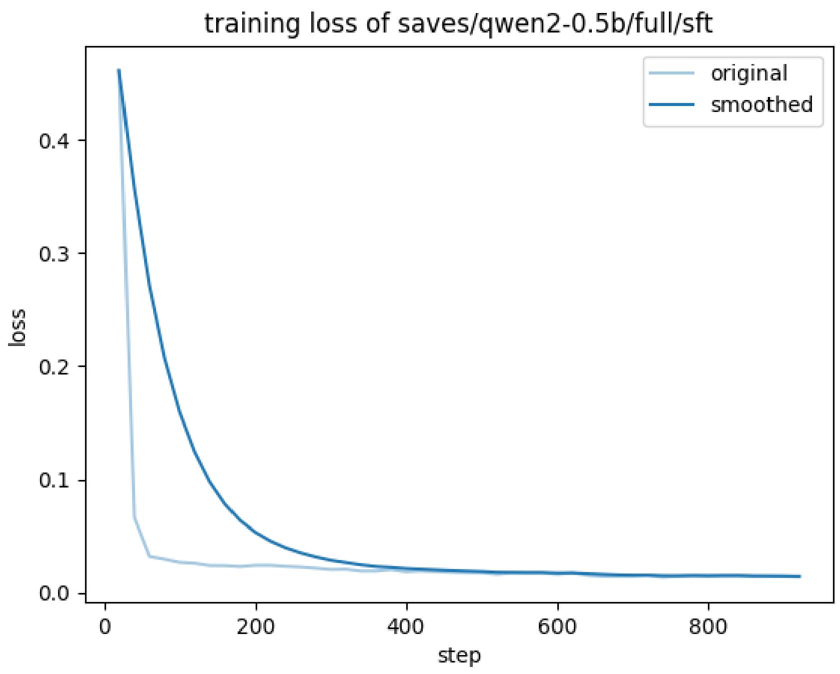

Figure 2: Training loss curve of model fine-tuning

The fine-tuned model is stored in `/content/Moonshot_Interview/Qwen2.5/examples/llama-factory/saves/qwen2-0.5b/full/sft`, which can be used to evaluate its performance on the test dataset, as demonstrated below.

4 Evaluation

4.1 Evaluation Process

We modified the original script from the Qwen2.5 GitHub repository to carry out the evaluation. The prompt format we employed is identical to that of the training dataset, with the exception of the exclusion of the assistant section. The hyperparameters used in this section are presented in Table 4.

Table 4: Hyperparameters used in evaluation

Hyperparameter	**Value**
max_new_tokens	10,000
temperature	0.5
top_p	0.95

Here we need to further explain our selection of hyperparameters. Due to the length of the CoT data, we needed to increase max_new_tokens to generate complete answers. However, considering our computational limitations, we could only set it to 10,000, which may still result in some truncated outputs. Nonetheless, as we will demonstrate in the next section, despite these truncated outputs being deemed incorrect, we were still able to achieve high overall accuracy. We opted for a temperature setting of 0.5, which is lower than the recommended balanced selection of 0.7 by ChatGPT. This choice was made because we found that lower temperatures are essential for ensuring the reliability of numeric computations during CoT.

We saved our output data into the `test_results.jsonl` file, which contains example data as shown in Code 4.

```
{"id": 1,
 "user_input":
    "Find a sequence of +,-,/,* which can be applied to the numbers 54,
    20, 67, 77 to get to 24, where each of the given numbers is used exactly once.",
 "model_output":
    "Search Path:\n
    Current State: 24:[54, 20, 67, 77], Operations: []\n
    Exploring Operation: 77-54=23, Resulting Numbers: [20, 67, 23]\n
    Generated Node #0,0: 24:[20, 67, 23] Operation: 77-54=23\n
    Moving to Node #0,0\n
    Current State: 24:[20, 67, 23], Operations: ['77-54=23']\n
    Exploring Operation: 67-20=47, Resulting Numbers: [23, 47]\n
    Generated Node #0,0,0: 24:[23, 47] Operation: 67-20=47\n
    Moving to Node #0,0,0\n
    Current State: 24:[23, 47], Operations: ['77-54=23', '67-20=47']\n
    Exploring Operation: 47-23=24, Resulting Numbers: [24]\n
    24,24 equal: Goal Reached\n\n\n"}
```

Code 4. Example of evaluation output data

4.2 Accuracy Analysis

We manually checked the correctness of the evaluation output for several reasons. First, some items contained numeric computation errors that regular expressions cannot identify. Additionally, the reliability of external LLMs for this specific task has not been tested, so we wanted to avoid the risk of using them. Finally, since we are primarily concerned with the final steps of the CoT and the dataset is relatively small, we were able to perform the manual checks quickly.

Finally, we found that 76 out of 100 outputs were correct, resulting in an accuracy of 76%. Consequently, we further explored the reasons for the incorrect outputs, which included truncated responses, numeric computation errors, and numeric memory errors. The percentages of each type are illustrated in Figure 3.

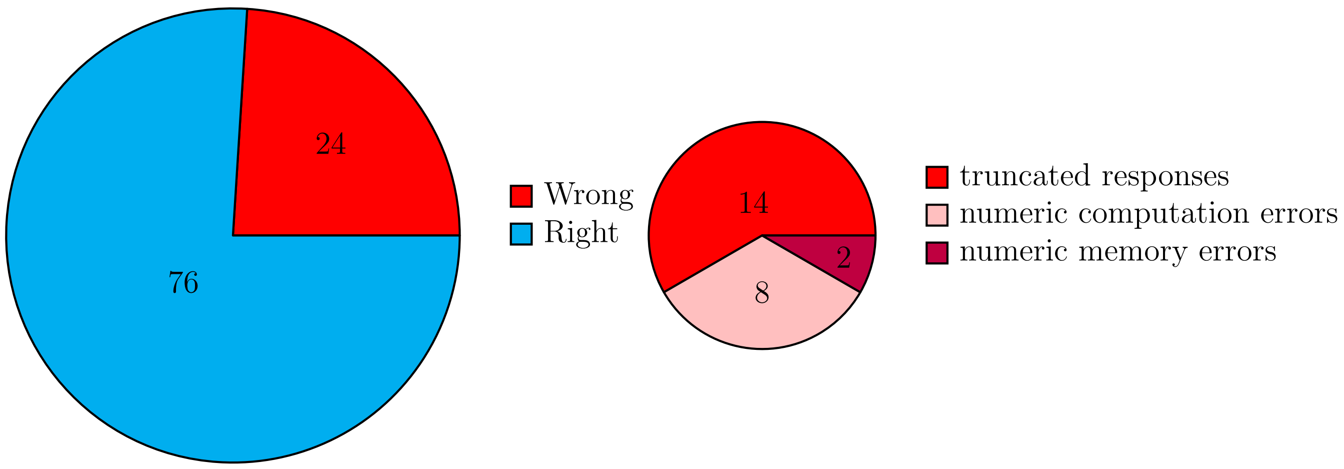

Figure 3: Distribution of evaluation of output data.
Left: Percentage of correct and incorrect outputs. Right: Detailed distribution of incorrect scenarios.

The most common incorrect scenario is truncated responses, which may be mitigated when computing resources are sufficient. The primary operation causing numeric computation errors is division, potentially resulting from the ability of remainder division or approximate division of base model. Numeric memory errors can arise from high temperatures, which we have attempted to reduce, as mentioned earlier. We provide examples for each incorrect scenario in Code 5.

```
Truncated responses:
{"id": 5,
 "user_input":
     "Find a sequence of +,-,/,* which can be applied to the numbers 3, 6, 18, 10 to
     get to 24, where each of the given numbers is used exactly once.",
 "model_output":
     "Search Path:\n
     Current State: 24:[3, 6, 18, 10], Operations: []\n
     Exploring Operation: 3*6=18, Resulting Numbers: [18, 10, 18]\n
     Generated Node #0,0: 24:[18, 10, 18] Operation: 3*6=18\n
     ......
     Moving to Node #0,0,28\nCurrent State: 24:["}

Numeric computation errors:
{"id": 25,
 "user_input":
     "Find a sequence of +,-,/,* which can be applied to the numbers 74, 26, 10, 20
     to get to 24, where each of the given numbers is used exactly once.",
 "model_output":
     "Search Path:\n
     Current State: 24:[74, 26, 10, 20], Operations: []\n
     Exploring Operation: 74-26=48, Resulting Numbers: [10, 20, 48]\n
     ......
     Current State: 24:[20, 4], Operations: ['74-26=48', '48/10=4']\n
     Exploring Operation: 20+4=24, Resulting Numbers: [24]\n
     24,24 equal: Goal Reached\n\n\n"}

Numeric memory errors:
{"id": 82,
 "user_input":
     "Find a sequence of +,-,/,* which can be applied to the numbers 16, 26, 4, 48 to
     get to 24, where each of the given numbers is used exactly once.",
 "model_output":
     "Search Path:\n
     Current State: 24:[16, 26, 4, 48], Operations: []\n
     Exploring Operation: 48/4=12, Resulting Numbers: [16, 26, 12]\n
     Generated Node #0,0: 24:[16, 26, 12] Operation: 48/4=12\n
     ......
     Current State: 24:[16, 8], Operations: ['48/4=12', '26-18=8']\n
     Exploring Operation: 16+8=24, Resulting Numbers: [24]\n
     24,24 equal: Goal Reached\n\n\n"}
```

Code 5. Example of incorrect scenarios

5 Conclusions

Chain-of-thought (CoT) is an advanced prompting technique designed to enhance the performance of large-scale language models (LLMs) in complex reasoning tasks. Complex problems, especially those involving mathematics, can be challenging for models to solve accurately. This includes tasks like arithmetic reasoning, commonsense reasoning, and symbolic reasoning. CoT improves the capabilities of large models by requiring them to explicitly articulate a step-by-step reasoning process before arriving at a final answer. This simple yet effective method fosters deeper reasoning.

Applying chain-of-thought (CoT) to questions solvable by traditional search algorithms presents a novel method for evaluating CoT's effectiveness in addressing problems with multiple complex steps. In this context, we fine-tuned the base model Qwen2-0.5B to solve the 24 Game using CoT outputs. By generating training data through a depth-first search (DFS) approach, we ensured an ample supply of data for effective training, ultimately achieving a 76% accuracy in our evaluations. Following this, we conducted a careful analysis of the potential reasons behind the incorrect outputs, which provides pathways for further exploration and optimization.

The success of this approach underlines the power of CoT in enhancing problem-solving capabilities, especially in games like the 24 Game that require not just sequential reasoning but also the ability to explore various combinations and pathways to reach a solution. As we move forward, exploring further applications of CoT in other complex domains, such as strategic game playing or multi-step mathematical problem-solving, could yield significant insights into the model's reasoning processes.

This work has some limitations. Firstly, due to constraints in computational resources, we were unable to fully evaluate the performance capabilities of the fine-tuned model. Secondly, we did not establish an efficient method for assessing the correctness of the outputs, a limitation that could potentially be addressed through the inclusion of more structured data.

6 Acknowledgement

I would like to express my heartfelt gratitude to the interviewer from Moonshot for giving me the opportunity to implement this project. I am also deeply thankful to Dr. Tong Yang for recommending me for the Moonshot internship. Additionally, I would like to extend my appreciation to my best friend, Junting Zhou, for his invaluable advice and support in answering my questions throughout this project. Finally, I would like to thank ChatGPT for providing guidance in the coding process and for assisting me in writing this report.

References

[1] Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (sos): Learning to search in language. *arXiv preprint arXiv:2404.03683*, 2024. Submitted on 1 Apr 2024.

[2] OpenAI. Openai o1 system card, 2024. December 5, 2024.

[3] Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, Nathan Lile, Dakota Mahan, Louis Castricato, Jan-Philipp Franken, Nick Haber, and Chelsea Finn. Towards system 2 reasoning in llms: Learning how

to think with meta chain-of-thought. *arXiv preprint arXiv:2501.04682*, 2025. Submitted on 8 Jan 2025.

[4] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024. Submitted on 20 Mar 2024, last revised 27 Jun 2024 (this version, v4).