A Framework for Training Large Language Models for Code Generation via Proximal Policy Optimization

Chi Zhang
ByteDance Inc.
Shanghai, China
zhangchi.usc1992@bytedance.com

Guangming Sheng
The University of Hong Kong
Hong Kong, China
gmsheng@connect.hku.hk

Siyao Liu
ByteDance Inc.
Beijing, China
liusiyao.sine@bytedance.com

Jiahao Li
ByteDance Inc.
Shanghai, China
lijiahao.plus@bytedance.com

Ziyuan Feng
ByteDance Inc.
Beijing, China
fengziyuan@bytedance.com

Zherui Liu
ByteDance Inc.
Beijing, China
liuzherui@bytedance.com

Xin Liu
ByteDance Inc.
Seattle, USA
liuxin.ai@bytedance.com

Xiaoying Jia
ByteDance Inc.
Beijing, China
jiaxiaoying@bytedance.com

Yanghua Peng
ByteDance Inc.
Seattle, USA
pengyanghua.yanghua@bytedance.com

Haibin Lin
ByteDance Inc.
Seattle, USA
haibin.lin@bytedance.com

Chuan Wu
The University of Hong Kong
Hong Kong, China
cwu@cs.hku.hk

Abstract

Proximal Policy Optimization (PPO) has proven to be effective in training large language models (LLMs) for code generation. In this work, we propose a framework for training LLMs for code generation that tackles two challenges: 1) insufficient GPU memory caused by increasing model sizes. ii) requirement to execute untrusted generated code for reward modeling. Standard PPO for code generation involves 4 LLMs including actor, critic, reference model and reward model. We propose to use Ray to provide an abstraction between models and GPU resources. This abstraction allows different models to be placed onto the same or different GPUs with a few lines of code. By placing different models onto different GPUs, our framework avoids insufficient GPU memory issues by utilizing existing LLM training infrastructure such as Pytorch FSDP, DeepSpeed, Megatron-LM, etc. To support executing model-generated code for reward modeling, we developed a sandbox service that supports isolation, parallel execution, and dataset management via Function as a service (FaaS). As a demonstration, we train the StarCoder2 3b model on the APPS dataset using our framework. Our experimental results demonstrate 40% pass@1 on the test set of the introductory split of the APPS dataset.

1 Introduction

Following [1, 9, 14], we utilize Proximal Policy Optimization (PPO) [12] to train LLMs for code generation. PPO-based RL training system for code generation typically consists of three LLMs: an *actor*, a *critic*, and a *reference policy* network. PPO algorithm proceeds in iterations, each with three stages: (1) response *generation* using the actor model with a batch of prompts; the prompts are the code questions and the responses are the generated code. (2) *preparation* of training data by scoring the generated code responses via sandbox, computing the values and KL divergence through a single forward pass of the critic, reference policy, and computing the advantage function via Generalized Advantage Estimation [11]; (3) *learning* from the data by updating actor and critic through PPO loss [12] using AdamW optimizer [6].

Building a scalable infrastructure to train LLMs for code generation is challenging because: i) as the model sizes grow, colocating all the models inside all the processes may easily lead to GPU Out-of-Memory. ii) executing generated code may be insecure, slow, and possibly timed out. Existing works [5, 14] train PPO for code generation by using data parallelism [10] with all the models colocating in all the processes. Also, the generated code is executed locally without proper isolation and parallelism for speedup.

In this work, we propose a framework for training LLMs for code generation via PPO that tackles the aforementioned challenges. Specifically, our contributions are:

- Our framework adopts Ray to provide an abstraction between models and their placement onto GPUs. This allows placing different models onto different GPUs for training and inference to avoid Out-of-Memory.
- We utilize Pytorch FSDP [10] for model training and vLLM [4] for fast rollout.
- We utilize FaaS (Function as a service) to implement our sandbox that provides isolation and auto-scaling.

2 Training Infrastructures

2.1 Ray-Based RL training infrastructure

RL can be considered as a computation flow, whereas LLM computation can be treated as a black box and a function approximator for various RL algorithms. To effectively manage the diverse computational demands of different LLMs, we introduce a novel Ray-based [8] RL training infrastructure tailored for LLM applications.

2.1.1 *WorkerGroup.*

We introduce the concept of a *WorkerGroup*, which manages a list of workers, each being a Ray remote process that holds a GPU. The WorkerGroup encapsulates LLM training, inference, and generation into primitive APIs that can be invoked by the Ray driver program. It also provides virtualization that enables all workers to initialize their distributed parallel groups and to be executed under the Single Program Multiple Data (SPMD) paradigm, which is widely adopted in existing LLM training and serving frameworks. We integrate PyTorch FSDP [10] and vLLM [4] in the WorkerGroup for LLM training and generation in code generation tasks. We will further integrate other LLM frameworks such as Megatron-LM [13] and VeScale [3] to support larger model scales.

With our WorkerGroup design, users can simply set the data and tensor parallel size to initialize the LLMs using FSDP and vLLM. Furthermore, the primitive APIs for constructing RL computation flow can be easily implemented by reusing the computation scripts from existing LLM systems. For example, the computation involved in `update_actor` and `generate` function of `ActorWorkerGroup` class (for actor model in Fig. 1) is similar to the pre-training scripts in PyTorch FSDP and generation script in vLLM. With such code reuse, each component of the RL training system, including rollout, actor, critic, and reference policy, is implemented as a `WorkerGroup` with different parallelism strategies and several primitive APIs.

2.1.2 *ResourcePool.*

A Ray cluster involves a head node and a bunch of worker nodes. The head node is typically a CPU machine responsible for resource scheduling and managing computation tasks within the cluster. Each worker node represents a GPU device dedicated to computation. We provide a `ResourcePool` class that virtualizes a set of GPU devices. When applying a `ResourcePool` instance to a WorkerGroup, the corresponding LLM and its distributed computation will be mapped to the allocated GPUs. LLMs share the same GPU set when using the same `ResourcePool` instance, but are allocated to different GPU sets when assigning different `ResourcePool`. We assume no overlap between different `ResourcePool` instances.

2.1.3 *Driver Program for PPO algorithm.*

The primitive APIs of each model enable streamlined development of various RL algorithms. A user can implement an RL algorithm in a few lines of code as a single process *driver program* to run on the head node of a Ray cluster. During execution, the driver program utilizes Remote Process Calls (RPC) to coordinate the execution order following the RL algorithm, involving a sequence of primitive API calls to invoke distributed computation of models. As shown in Figure 1, PPO can be implemented in just 9 lines by invoking model operations including `compute_values` and `generate_sequences`, which are executed within their corresponding `WorkerGroup` on multiple GPUs. The register with `3D_PROTO` and `DP_PROTO` denotes that the corresponding model operation is executed under 3D parallelism or data parallelism respectively, instructing the driver program to dispatch the input data and collect the output data of these functions.

```python
class ActorWorkerGroup(WorkerGroup):
    # Examples of distributed computation
    @register(transfer_mode=3D_PROTO)
    def generate(self, prompts: DataProto):
        ...
    @register(transfer_mode=DP_PROTO)
    def update_actor(self, prompts: DataProto):
        ...
# Allocate machines for a ResourcePool
resource_pool = ResourcePool([n_gpus_per_machine] * n_machines)
# Map the worker with allocated devices and init models
actor = ActorWorkerGroup(config, resource_pool)
     ... # Omit other models initialization.
# An example of PPO
for prompts in dataloader:
     # Generate responses
     batch = actor.generate_sequences (prompts)
     # Prepare experience
     batch = actor.compute_log_prob (batch)
     batch = critic.compute_values(batch)
     batch = reference.compute_log_prob(batch)
     batch = reward_function(batch) # Executed in the sandbox
     batch = compute_advantages(batch)
     # Update actor and critic
     critic_metrics = critic.update_critic(batch)
     actor_metrics = actor_hybrid_engine.update_actor(batch)
```

Figure 1: Illustration of the WorkerGroup, ResourcePool, and the PPO algorithm implementation.

2.1.4 *Benefits.*

With the above design, we decouple the LLM computation from the data dependencies among different LLMs in various RL algorithms. This decoupling enables algorithm researchers to focus solely on exploring novel RL computation flow and corresponding numerical computation, without addressing the LLM distributed computation. Moreover, it also enables system researchers to optimize the performance of each RL system component independently. The WorkerGroup abstraction provides a framework-agnostic interface, allowing seamless integration of recent advancements in LLM training and serving frameworks, by directly incorporating their existing codebases to implement the RL components.

2.2 Sandbox

Since generated code may contain malicious contents, it is mandatory to execute the generated code inside a secure environment. Also, running generated code with a large batch sequentially may be slow. Thus, parallelism is also desired to speed up the reward computation.

2.2.1 *Isolation.*

Our implementation uses a standard Docker runtime container to run each generated code. The container is not privileged, thus running any code inside the container can't impact the host machine. Compared with the gVisitor container runtime used by OpenAI [1], our solution can't test code that requires kernel. However, our solution is more general as most cloud providers don't support containers with privileged access.

2.2.2 *Parallelism.*

In order to ensure the execution of generated code doesn't interfere with each other, the maximum number of parallelism in each contain is 1. This also ensures the CPU and memory resources for each data. We also perform necessary process cleanup and file system restoration after each run. The auto-scaling capability is provided

2.2.3 *Dataset Management.*

Since the test cases may be large for different problems, it is undesirable to send the test cases to the sandbox server with problems each time. Instead, we give each problem a unique ID and store the test cases in the sandbox server. We only send the ID and generated code to the sandbox server.

3 Experiment

In this section, we provide a detailed demo that uses our framework to train an LLM for code generation.

3.1 Data

3.1.1 Dataset Selection. We use the APPS dataset [2] that contains 10k Python coding problems. Each problem has a question statement, a starter code, several solutions, and several test cases. The code problems are categorized into *introductory*, *interview*, and *competition*. We only focus on *introductory* part in our demo that contains 2.64k training data and 1k testing data.

3.1.2 Data Preprocessing. We propose 3 steps to perform data preprocessing: i) PPO requires test cases to judge whether the generated code is correct or not. Thus, we remove all the data without test cases. ii) We run all the solutions against the test cases to verify the correctness. We removed all the solutions that can't pass all the test cases. We remove data if all the test cases fail. iii) Due to the computation budget, we only keep the data where the question is less than 1024 tokens. After preprocessing, there remains 2639 training data and 717 testing data.

3.2 Evaluation

We evaluate the trained model using the test set of APPS introductory split. For each problem, we prompt the model with the question and obtain the response via greedy generation. We send the response and test case to the sandbox and receive the percentage of the passed tests. We report the percentage of the passed tests as pass@1 score.

3.3 Results

3.3.1 Pretraining and SFT. We use StarCoder2 3b [7] as our pretrained model. We first perform evaluation on StarCoder2 3b. The pass@1 score is only less than 1%. This is mainly because the pretrained model cannot follow the instructions and produces C++ code even though the instructions ask it to write solutions in Python. We perform Supervised Finetuning (SFT) on the training set for 3 epochs and achieve 34% pass@1 test score.

3.3.2 Reinforcement Learning. We use FSDP zero2 [10] to implement actor and rollout so that actor and rollout can fully share the weights. We use FSDP zero3 to implement critic training, reference policy and reward model. Our implementation runs on a single node with 8 A800 80G GPUs. We summarize the hyper-parameters used in our training in Table 1. We show the training curve in Figure 2. We observe that the training score increases from 0.34 to 0.75 and the testing score increases from 0.34 to 0.40. This indicates that our framework can successfully train an LLM using PPO for code generation.

Table 1: Hyper-parameters of our experiments

Hyper-parameters	Value
Total epochs	20
PPO epochs	1
Global batch size	1024
PPO mini-batch size	256
PPO clip ratio	0.2
Optimizer	AdamW [6]
Max prompt length	1024
Max response length	512
Gradient clipping	1.0
Sampling temperature	1.0
Discount factor	1.0
GAE Lambda	1.0

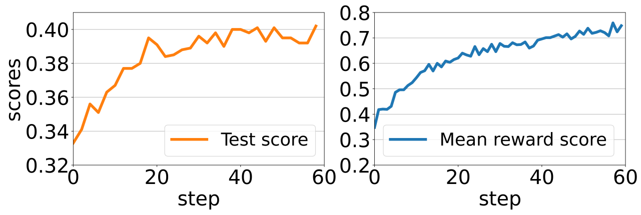

Figure 2: Training score and test score

4 Conclusion

In this work, we introduced a novel framework for training large language models (LLMs) for code generation using Proximal Policy Optimization (PPO). Our approach effectively tackles the challenges of insufficient GPU memory in training multiple LLMs and the secure execution of untrusted generated code. By integrating Ray, our framework enables precise model placement on GPUs to prevent memory overload. Furthermore, it utilizes existing LLM infrastructures like PyTorch FSDP and vLLM to accelerate the RL training process. To ensure the secure and scalable execution of generated code, we have implemented a robust sandbox environment using Function as a Service (FaaS). We evaluate the effectiveness of our framework by training the StarCoder2 3B model on the APPS dataset, achieving a 40% pass@1 on the test set of the introductory split. Our work presents a step towards scalable and efficient training of LLMs for code generation, enabling researchers to focus on developing high-performing models and RL algorithms without being hindered by infrastructure constraints.

Acknowledgments

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204423 and C7004-22G(CRF).

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[2] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob

Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *CoRR* abs/2105.09938 (2021). arXiv:2105.09938 https://arxiv.org/abs/2105.09938

[3] ByteDance Inc. 2024. *VeScale: A PyTorch Native LLM Training Framework.* https://github.com/volcengine/veScale

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[5] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG]

[6] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017). arXiv:1711.05101 http://arxiv.org/abs/1711.05101

[7] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[9] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[11] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2018. High-Dimensional Continuous Control Using Generalized Advantage Estimation. arXiv:1506.02438 [cs.LG]

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

[13] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[14] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based Code Generation using Deep Reinforcement Learning. arXiv:2301.13816 [cs.LG]