Aalto University
School of Science
Bachelor's Programme in Science and Technology

Optimization Opportunities in Reinforcement Learning Fine-Tuning of Large Language Models

Bachelor's Thesis

18. September 2025

Pekko Pesonen

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Pekko Pesonen
Title of thesis:	Optimization Opportunities in Reinforcement Learning Fine-Tuning of Large Language Models
Date:	December 18, 2025
Pages:	28
Major:	Computer Science
Code:	SCI3027
Supervisor:	Professor Lauri Savioja
Instructor:	Doctoral Researcher Zheyue Tan (Department of Computer Science)

Recently, Large Language Models (LLM) have seen success solving complex tasks requiring reasoning capabilities. Reinforcement learning fine-tuning has shown to be capable of training LLMs with the required reasoning capabilities to solve these tasks. RLFT enables models to learn from environmental feedback, but it introduces new system-level optimization challenges not found in traditional LLM learning. These challenges include the management of multiple heterogeneous models, larger context windows, environment management and workload scheduling. This thesis investigates these specific bottlenecks and evaluates the effectiveness of current solutions. Three state-of-the-art RLFT frameworks: HybridFlow, ROLL, and ASearcher, were analyzed, with a focus on their specific strategies for tackling these issues.

The findings of this thesis indicate that recent innovations, such as flexible model placement and asynchronous execution, successfully mitigate key challenges in RLFT. However, it is noted that many of the optimization strategies employed by these frameworks involve trade-offs between training speed, training stability and system complexity. Consequently, the thesis concludes that as model and environment complexity continues to grow, optimizing individual components should not be the end all be all, and future research should focus on holistic, whole-graph optimization strategies to ensure scalable and robust RLFT systems.

Keywords:	Large Language Models, Reinforcement Learning, System Optimization
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Teknistieteellinen kandidaattiohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Pekko Pesonen
Työn nimi:	Optimointimahdollisuudet Suurten Kielimallien Vahvistusoppimisjatkokoulutuksessa
Päiväys:	18. September 2025
Sivumäärä:	28
Pääaine:	Tietotekniikka
Koodi:	SCI3027
Valvoja:	Professori Lauri Savioja
Työn ohjaaja(t):	Tohtoritutkija Zheyue Tan (Department of Computer Science)

Viime aikoina suuret kielimallit (engl. Large Language Model, LLM) ovat menestyneet monimutkaisten päättelyä vaativien tehtävien ratkaisemisessa. Vahvistusoppimisjatkokoulutus (engl. Reinforcement Learning Fine-Tuning, RLFT) on osoittautunut kykeneväksi kouluttamaan kielimalleille näissä tehtävissä tarvittavia päättelykykyjä. RLFT mahdollistaa oppimisen ympäristön palautteesta, mutta tuo mukanaan uusia järjestelmätason optimointihaasteita, joita perinteisessä kielimallien koulutuksessa ei esiinny.

RLFT:n tuomia haasteita ovat useiden heterogeenisten mallien hallinta, suuret konteksti-ikkunat, ympäristöjen hallinta sekä prosessejen aikataulutus. Tämä kandidaatintutkielma tutkii näitä haasteita ja arvioi nykyisten ratkaisujen tehokkuutta. Työ analysoi kolmea nykyaikaista RLFT järjestelmää: HybridFlow, ROLL ja ASearcher. Ja analysoi niiden tapoja ratkaista RLFT:n haasteita.

Työn löydökset osoittavat, että viimeaikaiset innovaatiot, kuten joustava mallien sijoittelu ja asynkroninen suoritus ratkaisevat onnistuneesti RLFT:n keskeisiä haasteita. Huomataan myös, että monet näiden järjestelmien käyttämistä optimointistrategioista sisältävät kompromisseja koulutusnopeuden, järjestelmän monimutkaisuuden ja koulutuksen vakauden välillä. Johtopäätöksenä todetaan, että mallien ja ympäristöjen monimutkaisuuden kasvaessa yksittäisten komponenttien optimoinnin sijaan tutkimuksen tulisi keskittyä mallintamaan järjestelmä yhtenäisenä graafina ja optimoimaan se kokonaisuutena.

Avainsanat:	Suuret kielimallit, Vahvistusoppiminen, Järjestelmoptimointi
Kieli:	Englanti

Contents

1 Introduction

Large language models (LLMs) have become a mainstream tool, used in research, industry and everyday life. These models have shown capabilities such as translation, code generation and problem solving [1]. The transformative potential of LLMs is proven by their rapid widespread adoption across multiple domains, with use cases ranging from coding assistants to automated customer service. Recently, agentic LLMs have gathered increasing interest due to their ability to plan, reason, and execute complicated tasks that previously required a human in the loop.

Scaling LLM training presents challenges in terms of data availability and achievable performance improvements. Traditional training approaches depend on massive corpuses of human-generated text data. Current estimates suggest that LLM training will run out of new human-generated text data on the internet within the next few years [30]. Furthermore, training LLMs on a static dataset does not lead to ever increasing performance in complicated tasks that require reasoning capabilities [14].

Reinforcement Learning Fine-Tuning (RLFT) has emerged as a promising direction to improve the performance of a pre-trained model. Unlike traditional LLM training, which focuses on minimizing prediction errors on a fixed dataset, reinforcement learning is based on an iterative process where the model learns from reward signals given by the environment. This approach is particularly useful for developing agentic capabilities, as it enables models to learn from the outcomes of their actions and improve their behaviour through trial and error.

However, RLFT introduces new optimization challenges that hinder the efficiency and scalability of training systems, if not addressed. Reinforcement learning systems require more complicated data management due to training pipelines containing multiple heterogeneous models and long context windows. Agentic RL also requires an environment for the agent to interact with. The training system must manage these environments and possibly consider the added latency in scheduling tasks.

The goal of this thesis is to investigate the optimization challenges of RLFT and proposed solutions to these challenges. Chapter 2 provides necessary background knowledge on LLMs and reinforcement learning, Chapter 3 highlights the challenge of managing a large model during training. Chapter 4 covers the specific challenges introduced by RLFT. Chapter 5 reviews existing training systems. Finally, conclusions are presented in Chapter 6.

2 Background

The background chapter introduces necessary background information for the rest of the thesis. Section 2.1 gives a brief overview of LLMs and the transformer architecture. Section 2.2 introduces the topic of reinforcement learning. Section 2.3 combines these topics to discuss how reinforcement learning is used to train LLMs.

2.1 Large Language Models

The term large language model (LLM) was coined by the research community to describe a new family of language models with unprecedented scale. The trigger for the rise of LLMs was the release of the transformer architecture [29] by Vaswani et al. in 2017. The transformer architecture eliminated the need for sequential processing of tokens by replacing recurrence with a self-attention mechanism. This change allowed for parallel processing of tokens, which in turn, enabled longer context windows and efficient model training with unprecedented volumes of data. Research has shown that there exists a correlation between model size and performance, that follows predictable scaling laws [11]. Due to this, a substantial amount of resources have been used to scale LLMs to hundreds of billions of parameters.

The model proposed by Vaswani et al. [29] uses an encoder-decoder approach. The encoder is responsible for processing input tokens while the decoder is used to generate output tokens autoregressively, (i.e. all previous tokens are used as input when generating the next token). Currently most state-of-the-art models, such as the GPT family, are based on decoder-only architectures [37].

The encoder and decoder are stacks of identical layers, each layer containing sublayers for self-attention and feed-forward networks. A residual connection followed by layer normalization is used after each sublayer. The output of a sublayer is given by $\text{LayerNorm}(x + \text{Sublayer}(x))$ where $\text{Sublayer}(x)$ is the sublayer function. The key difference between the encoder and decoder is that the attention mechanism in the decoder has an additional mask to prevent tokens from attending to tokens that are after them in the sequence also referred to as causal masking.

The insight behind the attention mechanism is to allow tokens to attend (i.e. share context) with other tokens in the sequence. This is achieved by computing a query vector and a corresponding key-value vector pair. The attention score is computed as a dot product (i.e. similarity) between the query and key vectors. This attention score is used to compute a weighted sum of the value vectors.

For an input sequence with embedding vectors $x_1, x_2, \ldots, x_n$ stacked as rows to form

$X \in \mathbb{R}^{n \times d_{model}}$, the query, key and value matricies are first computed.

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where $W^Q \in \mathbb{R}^{d_{model} \times d_k}, W^K \in \mathbb{R}^{d_{model} \times d_k}, W^V \in \mathbb{R}^{d_{model} \times d_v}$ are the learned weight matrices. Then the attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where d_k is the dimension of the key vectors and $M \in \mathbb{R}^{n \times n}$ is an optional mask matrix. The scaling factor $\frac{1}{\sqrt{d_k}}$ is added to avoid numerical instability with the softmax function.

Due to the softmax function and the weighted averaging in the attention mechanism, a single attention mechanism, also referred to as an attention head, can not effectively capture multiple relationships between tokens at once. Using multiple attention heads in parallel allows the model to attend to multiple different information subspaces at once.

Multi-head attention is computed as:

$$\text{MultiHeadAttention}(X) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

$$\text{where} \quad \text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

where h is the number of attention heads and $W^O \in \mathbb{R}^{hd_{model} \times d_{model}}$ is a projection back to the model dimension.

After an attention layer, each embedding vector in the sequence is separately passed through a feed-forward network (FFN) consisting of two linear transformations with a non-linear activation function in between. Mathematically, the FFN is computed as:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where $\max(0, \cdot)$ is the ReLU (modern models use alternatives such as GeLU) function and W_1, W_2, b_1, b_2 are the learned weight matrices and bias vectors.

While the transformer architecture described above laid the foundation for modern LLMs, a substantial amount of research has been done to optimize it. Algorithmic optimizations such as FlashAttention [2] improve the runtime performance of the model. Architectural changes such as mixture-of-experts [15] improve the scalability of the model. These modifications are crucial for scaling LLMs to ever larger sizes.

2.2 Reinforcement Learning

Reinforcement learning (RL), as defined by Sutton and Barto [26], is a machine learning paradigm focused on finding the optimal behaviour—a mapping from states

to actions—that maximizes a numerical reward signal. Initially, the learning agent does not have any knowledge of its environment or its goal. Learning, in RL, is a trial and error process, unfolding across many iterations of the agent interacting with the environment and receiving rewards. These rewards should be designed to guide the agent towards its goal. Because the agents actions change the state of its environment, potentially altering future rewards, the agent is forced to consider the long-term consequences of its actions.

Sutton and Barto [26] formalize the reinforcement learning problem as a Markov decision process (MDP). The MDP is a model for sequential decision-making, where an agent observes its environment and decides on an action to take. An MDP can formally be defined as a tuple:

$$(\mathcal{S}, \mathcal{A}, P, R, d_0)$$

- $\mathcal{S}$ is the set of states
- $\mathcal{A}$ is the set of actions
- $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is a transition function $P(s'|s, a)$ which gives the probability of state s' when action a is taken in state s.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function $r = R(s, a)$.
- $d_0 \in \Delta(\mathcal{S})$ is the initial state distribution

In an MDP, a T-step trajectory τ (also called episode or rollout) is the sequence of state-action pairs generated by the agent-environment interaction from the start state to the terminal state:

$$\tau = [(s_0, a_0), (s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T)]$$

Each state-action pair (s_t, a_t) represents a decision where the agent chose action a_t while in state s_t. The return of a trajectory $R(\tau)$ represents the accumulated reward over the trajectory. It is usually defined as the discounted sum of rewards:

$$R(\tau) = \sum_{t=1}^{T} \gamma^{t-1} r_t = r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots + \gamma^{T-1} r_T$$

The discount factor $\gamma \in (0, 1]$ determines how much the agent values future rewards relative to immediate.

The agent's behavior is determined by its policy $\pi(a|s)$, a function that maps states to actions. Policies can be deterministic or stochastic depending on whether they map each state to a single action or a distribution over actions. The quality of a policy is measured by the expected return of all trajectories generated by following the policy starting from the initial state distribution d_0.

The probability of trajectory τ occuring when following policy π is given by:

$$P(\tau|\pi) = d_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

The expected return of a policy over all trajectories is given by:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] = \sum_{\tau} P(\tau|\pi)R(\tau)$$

Therefore, the problem of finding the optimal policy π^*—a behaviour that maximizes the agent's expected return—can be expressed as:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

Classic RL methods work well when the state and action spaces are small enough that the agent can visit most state-action pairs sufficiently many times during training. They also often assume that we know the transition dynamics of the environment or can accurately approximate them. However, often in practice problems have large state and action spaces where these assumptions do not hold. Deep reinforcement learning addresses these challenges by using functions to approximate the values of states and actions.

The state-value function $V^\pi(s)$ gives the expected return of executing the policy π starting from state s. Similarly, the action-value function $Q^\pi(s, a)$ gives the expected return of taking action a in state s continuing to follow policy π:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s], \quad Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s, a_0 = a]$$

The advantage function $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ gives a numerical measure of how much better (or worse) action a is compared to the average action of the policy in state s.

The large state and action spaces in deep RL make computing exact values for these functions impossible. Therefore, deep RL uses functions such as neural networks to approximate them.

$$V^\pi(s) \approx V_\phi(s), \quad Q^\pi(s, a) \approx Q_\phi(s, a)$$

where ϕ represents the learnable parameters of these function approximators.

Policy gradient methods [27] [12] are a widely adopted class of methods for training policies in deep RL. In policy gradient methods, the policy is approximated by its own function approximator. During training, the policy parameters θ are directly adjusted in the direction that increases the expected return. Crucially, the policy gradient theorem states that we can compute an estimate for the gradient of the expected return using trajectories sampled from the current policy:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t\right]$$

$\nabla_\theta \log \pi_\theta(a_t|s_t)$ represents the direction that the parameters should be changed to increase the probability of taking action a_t in state s_t, and $\hat{A}_t$ is some estimate of the advantage function such as generalized advantage estimation (GAE) [20]. Intuitively, changing the parameters θ in the direction of the gradient increases the probability of taking actions that are better than average. The same logic also applies to decreasing the likelihood of actions that have negative advantage.

2.3 Reinforcement Learning for Large Language Models

In the context of LLMs, neither of the assumptions of classical RL hold. The state space (possible sequences of tokens) and action space (model vocabulary) are extremely large. Furthermore, the transition dynamics of how each token influences subsequent states is not known. These factors necessitate RLFT to use deep RL methods.

The RLFT training loop involves iterating through three stages [31] [23]: generation, inference, and training. In the generation stage, the actor model generates responses for a batch of prompts. In agentic RL, the actor model interacts with its environment during response generation. The output of the generation stage is called a rollout, which consists of responses (rollout samples) generated during the stage.

In the inference stage, the critic, reference and reward models evalute the rollout samples. The critic model estimates the value, for computing the advantage, of each token in the in the sample. The reference model computes KL penalties, which prevent large deviations in the actor model. Lastly, the reward model scores the quality of generated samples.

The training stage combines the outputs of the three models in the inference stage with the rollout samples from the generation stage to compute the final training objective. The training objective is used to update the model parameters of the actor. Depending on the algorithm, the parameters of the critic model are also updated during the training stage. The updated parameters are then synchronized across all devices for another iteration of the training loop.

Two popular policy gradient methods in RLFT are Proximal Policy Optimization (PPO) [19] and Group Relative Policy Optimization [21]. PPO improves upon earlier policy gradient methods by using a clipped surrogate objective that discourages large deviations in the policy. Moreover, PPO improves sample efficiency by using the same sample for multiple policy updates. GRPO is a variant of PPO, which eliminates the need for a separate value function (critic model) by computing advantages relative to a group of rollout samples. This significantly reduces computational complexity of the training loop and shows comparable training performance to other policy gradient methods. Figure 1 shows a comparison of PPO and GRPO.

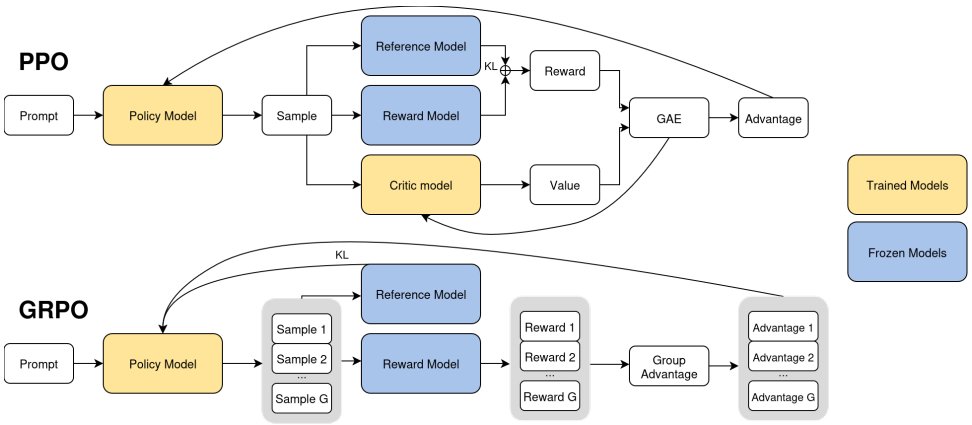


Figure 1: Dataflow graphs of PPO and GRPO training loops. Adapted from [21].

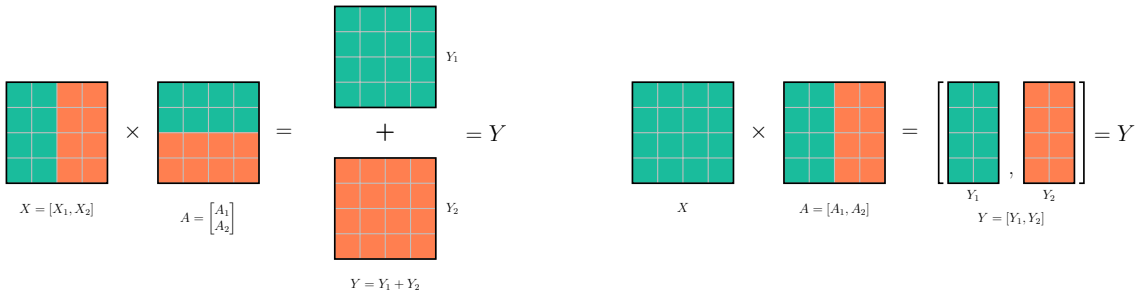
3 Challenges in Scaling LLM Training

Scaling LLM training is not as simple as increasing the computational resources. In addition to compute, training systems require massive amounts of memory which no single device can provide. These memory constraints require training of large models to be distributed across multiple devices, which leads to other challenges, such as communication bottlenecks, decreasing training efficiency (i.e. compute utilization). A central challenge of scaling LLM training is architecting the training pipeline to maximize compute utilization while simultaneously attending to these constraints. This chapter discusses the parallelism strategies and model placement techniques required to scale LLM training.

3.1 Data Parallelism

Data parallelism is the most common parallelism strategy when scaling LLM training. It splits the training data into multiple batches which are processed in parallel on different devices. After processing a batch, the gradients from each device are aggregated to update the model parameters. Data parallelism is widely used due to its simplicity and high scalability. The main drawback of naive data parallelism is that each data parallel process requires a full copy of the model parameters.

For small models, it is possible to use data parallelism on a single machine with multiple GPUs. For larger models, distributed data parallelism (DDP) is required. A simple approach to DDP is the parameter server architecture [13], which divides nodes into workers and servers. Worker nodes pull the model parameters from the parameter server, process a batch of input data, and push updates back to the parameter server. However, this creates a communication bottleneck when multiple workers try to synchronize parameters simultaneously, limiting the scalability of the parameter server architecture.



(a) Row Parallel Matrix Multiplication **(b) Column Parallel Matrix Multiplication**

Figure 2: Parallelism strategies for matrix multiplication. (a) Input and weight matrices partitioned, results summed. (b) Only weight matrix partitioned, results concatenated. Adapted from [24].

Modern approaches to data parallelism such as ZeRO [17], forgo the need to duplicate model parameters for each data parallel process by partitioning model parameters across processes. This partitioning shows only a 1.5x increase in communication overhead while providing memory savings relative to the degree of data parallelism [17].

3.2 Tensor Parallelism

Tensor parallelism, as decribed in the original Megatron-LM [24] paper, is an intra-layer partitioning, where the tensors within layers are partitioned across multiple devices along a specific dimension. In transformer architectures, the attention and feed-forward layers contain large matrix multiplications that can be parallelized using tensor parallelism. Figure 2 shows two strategies for parallelizing matrix multiplication across multiple devices. Tensor parallelism distributes memory consumption and computational workload across multiple devices, reducing the per-device memory requirements. However, tensor parallelism introduces communication overhead as the intermediate tensors inside a layer must be synchronized using all-reduce operations at least twice per layer. These frequent synchronization operations necessitate high-bandwidth communication between tensor parallel devices e.g. NVLink or NVSwitch.

3.3 Pipeline Parallelism

Pipeline parallelism partitions the model across layers, placing different layers on different devices. As with tensor parallelism, the goal of pipeline parallelism is to enable efficient training of large models that can not fit into the memory of a single device. In pipeline parallelism, the activations of one pipeline stage are passed as the input to the next stage. Unlike tensor parallelism, this does not require frequent communication between

devices, but it does introduce sequential dependency between pipeline stages. The naive approach would be to process the entire input in every stage before passing it to the next stage. This creates pipeline bubbles, where devices are idle while waiting for inputs. To mitigate this, training data is divided into smaller micro-batches that are processed in a pipelined fashion. This approach reduces device idle time by overlapping computation across stages, but creates imbalances in memory consumption between pipeline stages, as earlier stages must store activations for multiple micro-batches. Multiple pipeline scheduling strategies have been proposed e.g. GPipe [8] and PipeDream [7] that increase the efficiency of pipeline parallelism by reducing pipeline bubbles and balancing memory consumption between pipeline stages.

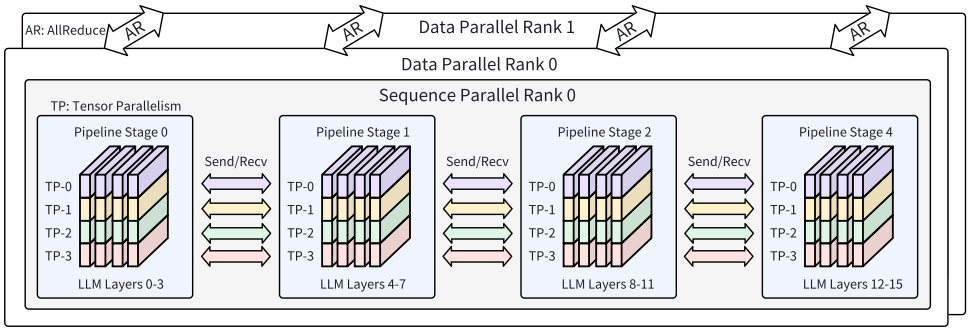


Figure 3: 3D parallel pipeline configuration with data, tensor and pipeline parallelism. Reprinted from [4]

3.4 3D Parallelism & Auto Parallelism

The combination of data, tensor and pipeline parallelism is known as 3D parallelism. Figure 3 shows a typical 3D parallel pipeline configuration, where tensor parallelism is applied within nodes, pipeline parallelism is applied across multiple nodes, and data parallelism is applied across multiple identical pipelines. In modern 3D parallelism setups, FSDP can be used as the implementation for the data parallelism dimension to further improve memory efficiency. Combining parallelism strategies creates a vast space of configuration choices, where each configuration choice affects not only training throughput but also memory consumption, communication overhead and training stability. Furthermore, the optimal configuration may not be transferrable between models different model architectures and hardware setups. Handcrafting optimal parallelsim strategies is time-consuming and requires extensive expertise.

Auto parallelism systems aim to automate the process of finding the optimal parallelism strategy. Frameworks such as Alpa [39] and FlexFlow [9] use search algorithms to find optimal configurations. FlexFlow simulates different parallelism strategies and predicts their runtime performance. Alpa takes a hierarchical approach by separating intra-layer

(tensor) parallelism from inter-layer (pipeline) parallelism and uses a compiler to evaluate configurations within this reduced search space.

4 Challenges Introduced by Reinforcement Learning Fine-Tuning

In addition to the challenges common to all large language model training, reinforcement learning fine-tuning introduces its own set of challenges. Agentic RL, where models interact with dynamic environments, adds yet another layer of difficulty to the training process. This chapter discusses the specific challenges introduced by RLFT and agentic RL training. In sections 4.1 and 4.2 we discuss the added complexities in data management relevant to all RLFT systems, specifically managing model weights and the context windows. Then we proceed to the challenges of managing environments in section 4.3 and workload scheduling in section 4.4, which are more specific to agentic RL training.

4.1 Model Management

While supervised training typically requires managing only a single model, reinforcement learning introduces more system complexity by requiring the training system to manage multiple heterogeneous models [25]. PPO-based RL training [19] typically involves four LLMs [40]: an actor model that is being trained, a critic model for value estimation, a reference model for KL penalties, and a reward model for evaluating generated responses. These models differ in their sizes, memory requirements, and computational needs. This heterogeneity, combined with differing data dependencies and computational patterns, makes model placement a complex optimization challenge.

The actor model, requires both forward passes for response generation and backward passes for gradient computation. The critic model evaluates the quality of the actor's outputs by estimating the value function. In PPO the critic model is also being trained, along with the actor. Other algorithms, such as GRPO [21], eliminate the critic completely. The reference model is a frozen copy of the actor that matches it in size but requires less computation, performing only forward passes during the inference stage. Lastly, the reward model evaluates the responses generated by the actor.

Placement of these models across available devices presents several options, each with different trade-offs. One approach is to place models on separate sets of GPUs, which enables parallel execution when no data dependencies exist between the models. However, this strategy leads to GPU idle time during stages when only a some of the models are

active, resulting in idle time [23]. Another approach is to colocate multiple models on the same GPUs, which enables memory sharing and reduces data transfers but requires sequential execution that could otherwise run in parallel. Models that are not used in the current stage can also be offloaded to CPU memory [18], which is slower but more abundant than GPU memory.

4.2 Context Management

In RLFT, models tend to generate progressively longer responses over the course of training [3]. While this increase in context length does initially provide improvements in reasoning capabilities and model performance, it introduces system-level complications that ultimately constrain the scalability of the training process [28]. As models generate longer responses, the memory required for training grows, leading to out-of-memory errors or context truncation. For agentic reinforcement learning, involving multi-turn interactions, the problem becomes more severe as each episode accumulates context across multiple interactions between the agent and its environment.

A case study by Tan et al. [28] demonstrates the practical consequences of these limitations. They report training a 4-billion parameter model in a Tic-Tac-Toe environment with a maximum context window of 8,192 tokens. In the beginning, the length of responses generated by the model grew steadily. At training iteration 13, the context length of a single episode reached the 8,192 token limit. This caused context truncation, which degraded the quality of reasoning chains and resulted in severe performance degradation (see Figure 4).

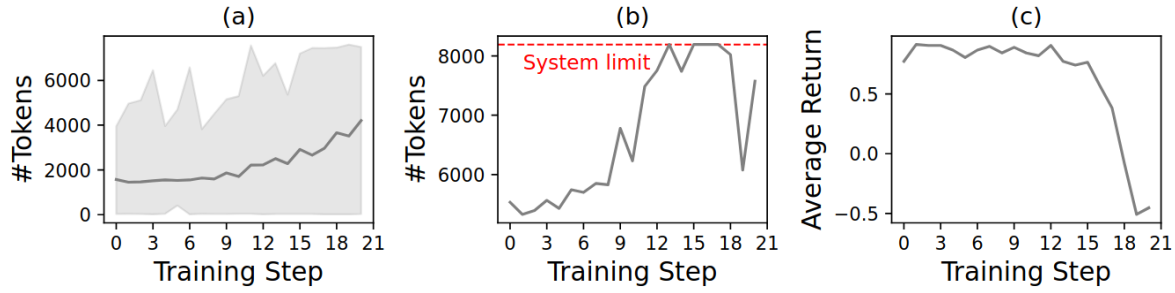


Figure 4: Impact of context truncation when training a 4B-parameter LLM on Tic-Tac-Toe. (a) Turn-level context length increases steadily. (b) Episode-level context length reaches system limit. (c) Sharp decrease in performance caused by context truncation. Reprinted from [28].

Furthermore, Tan et al. [28] observe, that longer context lengths introduce communication bottlenecks in distributed training settings. Longer contexts require transferring more data between training stages. The size of these data transfers scales linearly with context

length. In a test conducted by them, a model with 32,000 token context length required transmitting roughly 1 TB of intermediate data per iteration, taking over 20 minutes to transfer when using 25 Gbps network connection and consuming about a quarter of the total training time.

4.3 Managing Environments

The core difference between conventional RL and agentic RL lies in the complexity of the environments. In agentic RL, environments allow the actor to explore, observe, and modify their state through multi-turn interactions. Environments for coding (debug-gym [35], SWE-bench [10]), web browsing (WebArena [41], WebShop [34]), GUI operating (OSWorld [33]), and video games (AgentGym [32]) represent full software stacks including code execution environments, web browser interfaces, operating system interfaces, and game engines. These environments allow for tasks that require long interactions, by the agent, to solve.

Depending on the environment in question, different challenges may arise, such as the environments having varying latencies, computational requirements, and dependencies. Many of the currently available RL training environments are designed to be self-managed [36] such that they do not depend on anything external. WebArena [41], for example, contains mock copies of multiple websites such as a webshop, Reddit, a map, and an English Wikipedia. Self managed environments allow the training system to control all aspects of the environment, but they also require the training system to construct and manage them, adding more complexity. Environments may also be externally managed, such as using real websites and APIs. Using externally managed environments removes the burden of constructing and managing the environment, but it introduces other challenges such as higher latency and unreliability and costs.

4.4 Workload Scheduling

The generation stage accounts for the majority, around 70% [6], of the total training time in RLFT systems. Because training pipelines process data synchronously in batches, the duration of the training stage is dictated by the longest response in the batch, creating a bottleneck known as the straggler effect [40]. This synchronization requirement leads to significant GPU underutilization, as devices that finish generating shorter responses remain idle until the longest sequence is complete, see Figure 6 (a). Furthermore, this issue is exacerbated by the model's tendency to generate progressively longer responses over the course of training (Section 4.2). Rollout scheduling aims to mitigate this issue by dynamically scheduling rollout tasks when resources become available.

5 Review of existing Reinforcement Learning Fine-Tuning Frameworks

This chapter reviews three state-of-the-art RLFT frameworks that propose solutions to the optimization challenges discussed in chapters 3 and 4. First we introduce HybridFlow, which introduces a new hybrid programming model to make model management (Section 4.1) in RLFT systems easier. Next we introduce ROLL, which builds upon the innovations in HybridFlow and introduces new optimizations to the generation stage to mitigate workload scheduling bottlenecks (Section 4.4); and new tools for managing environments (Section 4.3). Finally we introduce ASearcher, which implements a fully asynchronous RL system to efficiently train models with extremely long trajectories.

5.1 HybridFlow

HybridFlow, better known for its open-source implementation *verl*, presented by Sheng et al. [23], aims to tackle the challenge the complex model management (Section 4.1). RLFT systems preceding HybridFlow, such as NeMo-Aligner [22] use a multi-controller architecture to manage the different models required in RLFT. Multi-controller architectures have very little communication overhead, thus being highly efficient and scalable. However, due to nodes depending directly on other nodes, the multi-controller architecture is inflexible to modifications. HybridFlow introduces a hybrid programming model, which uses a single-controller approach to express the diverese dataflows in RLFT systems, while still using a multi-controller approach to execute intra-node distributed computations.

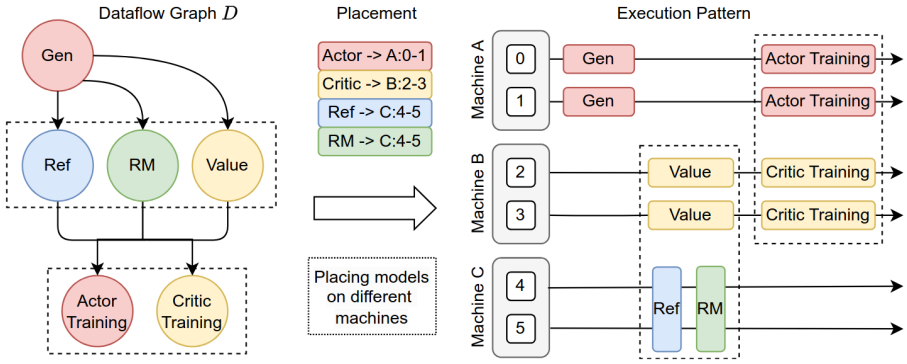


Figure 5: Example dataflow in HybridFlow. Executions in dashed boxes are parallel. Reference and reward models are colocated and executed sequentially. Reprinted from [23]

Different models are expressed as workers in HybridFlow. The goal of the worker is

to decouple intra-node distributed computation from inter-node data dependencies. The base class `3DParallelWorker` automatically handles 3D parallelism strategies, and classes for different models inherit from this base class to implement model specific operations, such as `ActorWorker`, `CriticWorker`, `ReferenceWorker`, and `RewardWorker`. For example, the `ActorWorker` implements `generate_sequences` for generating responses, while the `RewardWorker` implements `compute_reward` for evaluating responses. HybridFlow also supports modern parallelism strategies such as PyTorch FSDP [38] and ZeRO [16] through the `FSDPWorker` and `ZeROWorker` base classes respectively. The worker abstraction allows the single controller to conveniently manage data dependencies between models, without the need to know the specific parallelism strategies used to execute the models.

The heterogeneous models in RLFT systems employ different data parallelism strategies, which necessitates efficient data resharding between models. HybridFlow introduces a new unified transfer protocol to address this challenge. Operations in each model class are associated with a transfer protocol. A transfer protocol consists of a collect function and a distribute function. The collect function gathers output data of the function, and the distribute function distributes input data to the function. When the controller initiates an operation on a model, the function uses the gather function to immediately return all output data as data futures. Using these data futures, the controller can initiate subsequent operations on other models without blocking. The operations are executed asynchronously, as outputs become available. Crucially, the actual data transfers between models are handled directly between GPUs, not through the controller.

HybridFlow provisions resources using the `ResourcePool` class. The `ResourcePool` provides an abstraction layer over the physical devices. When a model class is given a `ResourcePool` instance, the model is automatically distributed across the device set using the parallelism strategy defined by the model. If multiple models share the same `ResourcePool` instance, they are colocated on the same set of GPUs and executed sequentially. Conversely, if they are assigned different `ResourcePool` instances, they are placed on separate GPU sets and can be executed in parallel, see figure 5.

5.2 ROLL

ROLL (Reinforcement Learning Optimization for Large-Scale Learning) [31] is a library developed by Alibaba. It is designed to support multiple RLFT algorithms: RL from human feedback (RLHF), RL with verifiable rewards (RLVR), multi-turn agentic RL; and multiple use-cases from efficient large-scale training to algorithm research in compute constrained environments. ROLL builds upon the concepts introduced by HybridFlow [23] and introduces a new environment worker and rollout scheduler with sample-level

control.

To mitigate the straggler effect (Section 4.4), ROLL introduces a new `RolloutScheduler` that can control the lifecycle of individual generation processes. This combined with asynchronous reward computation that computes rewards for complete responses immediately, allows the generation stage to be optimized with dynamic sampling. Dynamic sampling is the strategy of filtering out samples that do not produce meaningful gradients (i.e. accuracy score is either 1 or 0). With the sample-level control of the `RolloutScheduler`, new prompt samples can be added to the generation stage based on available resources. After a threshold number of satisfactory samples have been generated, remaining generation processes can be terminated to free up resources.

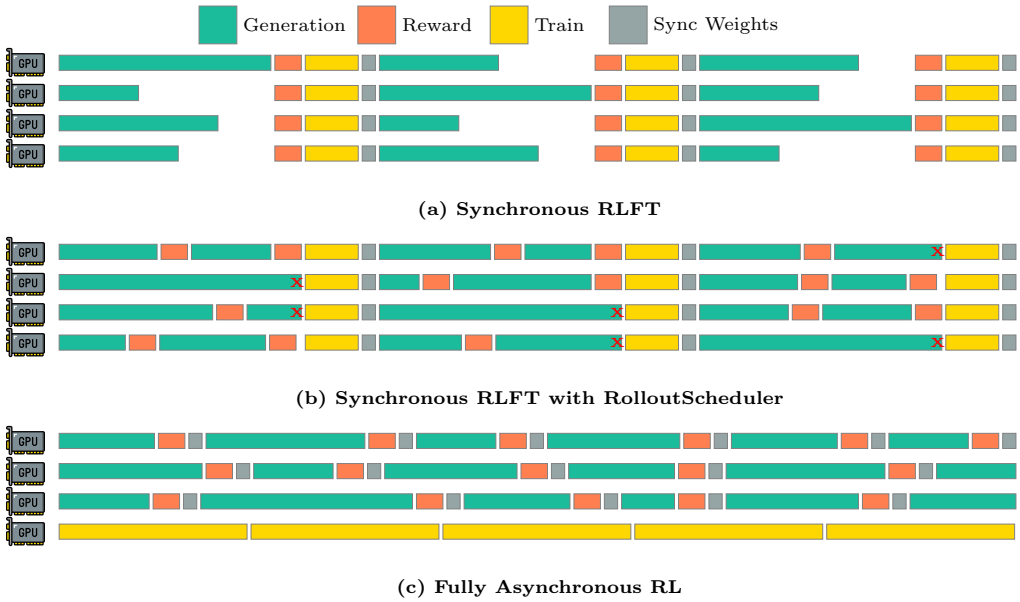


Figure 6: Execution diagrams of (a) Synchronous RLFT where training is bottlenecked by the longest response in a batch, (b) RLFT using asynchronous rewards and `RolloutScheduler` to add and abort samples during generation stage, and (c) fully asynchronous RLFT.

ROLL addresses the challenge of managing environments (Section 4.3) by introducing a dedicated `EnvironmentWorker` that is responsible for managing the environments with which the actor interacts. As with rewards, ROLL handles environment interactions asynchronously, such that the actor can process other prompts while waiting for a response from the environment. Furthermore, The reward worker in ROLL supports multiple different types of rewards such as mathematical verification, code execution, and LLM-as-a-Judge. The sample-level control of the `RolloutScheduler` allows generated responses to be routed to appropriate reward workers based on the task type.

5.3 ASearcher

ASearcher [5] aims to train AI agents to solve complex, knowledge-intensive search tasks. Gao et al. [5] give an example of such task: "As of December 31, 2024, what were the numbers of gold, silver, and bronze medals won by China in the 2012 London Olympics" for which there exists multiple conflicting answers online. To solve this task the agent must be able to identify that some athletes were disqualified for doping. Complex search tasks, such as this, demand extremely long trajectories, as the required information has to be gathered from multiple sources.

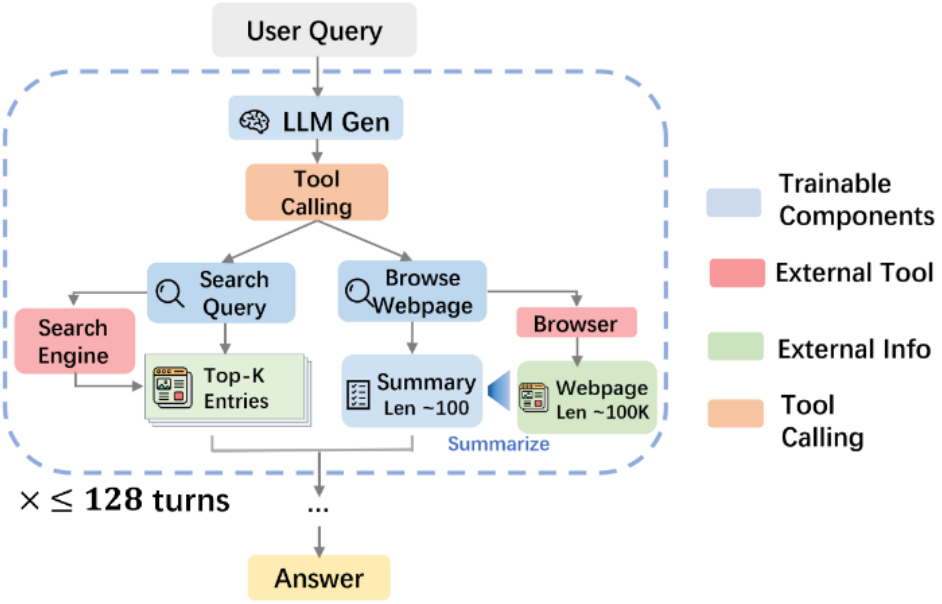


Figure 7: The agent-environment interaction loop in ASearcher. The actor alternates between querying and browsing web pages. Webpages are summarized to manage context length. Reprinted from [5].

To address the efficiency bottleneck caused by long tailed responses (Section 4.4) that leave other devices idle, existing online RL training systems typically impose a turn limit (e.g. $\leq$ 10 turns) during the generation stage [5]. ASearcher implements a fully asynchronous RL system, which completely decouples the generation stage from model updates in the training stage, see figure 6 (c). Asynchronous RL prevents long rollouts from blocking the training loop and allows for extremely long search trajectories, extending to hundreds of turns, without sacrificing training efficiency. Similar to ROLL [31], ASearcher also implements dynamic sampling to filter out samples that would not contribute meaningful gradients to the training process.

While fully asynchronous approaches like ASearcher effectively reduce device idle time, they introduce a new problem: policy staleness. Policy staleness occurs because long trajectories are generated using older weights than shorter trajectories, possibly leading

to training instability and decreases in model accuracy [6]. The staleness problem has left some researchers hesitant to adopt fully asynchronous approaches despite their superior efficiency.

ASearcher gives the actor model two tools to interact with: a search engine and a web browser. As shown in Figure 7, the generation stage begins with a sampled prompt question. The actor searches for information in an iterative cycle where the it decides whether to issue a search query or browse a specific webpage. To manage the context window of the actor from being cluttered by irrelevant data from the webpages and address the context truncation issue (Section 4.2), ASearcher uses an intermediate summarization model. Instead of feeding raw webpage content into the context of the actor, the content is passed through a summarization model that compresses the information into approximately 100 tokens. This allows the actor to gather information from numerous sources without exhausting its context window. Asearcher allows for the summarization model to be trained along with the actor model.

6 Conclusion

This thesis investigated the current optimization challenges in reinforcement learning fine-tuning systems, specifically looking at the challenges of heterogeneous model management, context management, environment management, and rollout scheduling. We also reviewed three state-of-the-art RLFT frameworks that each tackle these challenges in different ways. HybridFlow shows that complex model management can be made reasonably simple through a combination of novel abstractions and unified communication protocols. For workload scheduling, ROLL and ASearcher show that the straggler effect can be mitigated through asynchronous execution and dynamic sampling strategies. Though, adoption of asynchronous RLFT is still limited due to the yet unsolved problem of policy staleness. Furthermore, ROLL addresses environment management with dedicated environment workers, while ASearcher tackles context management constraints in long-horizon tasks by using a intermediate summarization model.

Existing RLFT frameworks offer successful solutions to the challenges discussed in this thesis, but these solutions will eventually encounter new bottlenecks as large language models and training systems continue to grow in size and complexity, necessitating the development of further optimizations. A possible direction to unlock further optimizations is the development of whole-graph optimized RLFT systems. All current RLFT systems are compositions of individually optimized black boxes. These fragmented systems lack the ability for holistic end-to-end optimization. To achieve such optimizations, the entire RLFT pipeline with all of its models, environments and

dataflows should be treated as a single computation graph rather than a composition of individually optimized components.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

[2] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL `https://arxiv.org/abs/2205.14135`.

[3] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan

Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[4] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, Xipeng Qiu, Dahua Lin, Yonggang Wen, Xin Jin, Tianwei Zhang and Peng Sun. Efficient training of large language models on distributed infrastructures: A survey, 2024. URL https://arxiv.org/abs/2407.20018.

[5] Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu and Yi Wu. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous rl, 2025. URL https://arxiv.org/abs/2508.07976.

[6] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng and Wei Wang. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training, 2025. URL https://arxiv.org/abs/2509.21009.

[7] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018. URL https://arxiv.org/abs/1806.03377.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019. URL https://arxiv.org/abs/1811.06965.

[9] Zhihao Jia, Matei Zaharia and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018. URL https://arxiv.org/abs/1807.05358.

[10] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[11] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

[12] Matthias Lehmann. The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations, 2024. URL https://arxiv.org/abs/2401.13662.

[13] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[14] Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, Yingying Zhang, Fei Yin, Jiahua Dong, Zhiwei Li, Bao-Long Bi, Ling-Rui Mei, Junfeng Fang, Xiao Liang, Zhijiang Guo, Le Song and Cheng-Lin Liu. From system 1 to system 2: A survey of reasoning large language models, 2025. URL https://arxiv.org/abs/2502.17419.

[15] Siyuan Mu and Sen Lin. A comprehensive survey of mixture-of-experts: Algorithms, theory, and applications, 2025. URL https://arxiv.org/abs/2503.07137.

[16] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. URL https://arxiv.org/abs/1910.02054.

[17] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. URL https://arxiv.org/abs/1910.02054.

[18] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. URL https://arxiv.org/abs/2101.06840.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[20] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. URL https://arxiv.org/abs/1506.02438.

[21] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

[22] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, Markel Sanz Ausin, Ashwath Aithal and Oleksii Kuchaiev. Nemo-aligner: Scalable toolkit for efficient model alignment, 2024. URL https://arxiv.org/abs/2405.01481.

[23] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297. ACM, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

[24] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

[25] Pedro F. Silvestre and Peter Pietzuch. Systems opportunities for llm fine-tuning using reinforcement learning. *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 90–99, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715389. doi: 10.1145/3721146.3721944. URL https://doi.org/10.1145/3721146.3721944.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, second edition, 2018. URL https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf.

[27] Richard S Sutton, David McAllester, Satinder Singh and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, S. Solla, T. Leen and K. Müller, editors, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

[28] Zheyue Tan, Mustapha Abdullahi, Tuo Shi, Huining Yuan, Zelai Xu, Chao Yu, Boxun Li and Bo Zhao. Earl: Efficient agentic reinforcement learning systems for large language models, 2025. URL https://arxiv.org/abs/2510.05943.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

[30] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim and Marius Hobbhahn. Will we run out of data? limits of llm scaling based on human-generated data, 2024. URL https://arxiv.org/abs/2211.04325.

[31] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang and Bo Zheng. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library, 2025. URL https://arxiv.org/abs/2506.06122.

[32] Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, Songyang Gao, Lu Chen, Rui Zheng, Yicheng Zou, Tao Gui, Qi Zhang, Xipeng Qiu, Xuanjing Huang, Zuxuan Wu and Yu-Gang Jiang. Agentgym: Evolving large language model-based agents across diverse environments, 2024. URL https://arxiv.org/abs/2406.04151.

[33] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

[34] Shunyu Yao, Howard Chen, John Yang and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023. URL https://arxiv.org/abs/2207.01206.

[35] Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni and Marc-Alexandre Côté. debug-gym: A text-based environment for interactive debugging, 2025. URL https://arxiv.org/abs/2503.21557.

[36] Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, Yifan Zhou, Yang Chen, Chen Zhang, Yutao Fan, Zihu Wang, Songtao Huang, Francisco Piedrahita-Velez, Yue Liao, Hongru Wang, Mengyue Yang, Heng Ji, Jun Wang, Shuicheng Yan, Philip Torr and Lei Bai. The landscape of agentic reinforcement learning for llms: A survey, 2025. URL https://arxiv.org/abs/2509.02547.

[37] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie and Ji-Rong Wen. A survey of large language models, 2025. URL https://arxiv.org/abs/2303.18223.

[38] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL https://arxiv.org/abs/2304.11277.

[39] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022. URL https://arxiv.org/abs/2201.12023.

[40] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu and Xin Jin. Optimizing rlhf training for large language models with stage fusion, 2025. URL https://arxiv.org/abs/2409.13221.

[41] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.