SPEEDYZERO: MASTERING ATARI WITH LIMITED DATA AND TIME

Anonymous authors
Paper under double-blind review

ABSTRACT

Many recent breakthroughs of deep reinforcement learning (RL) are mainly built upon large-scale distributed training of model-free methods using millions to billions of samples. On the other hand, state-of-the-art model-based RL methods can achieve human-level sample efficiency but often take a much longer overall training time than model-free methods. However, high sample efficiency and fast training time are both important to many real-world applications. We develop SpeedyZero, a distributed RL system built upon a state-of-the-art model-based RL method, EfficientZero, with a dedicated system design for fast distributed computation. We also develop a novel algorithmic technique, Priority Refresh, to stabilize massively parallel model-based training. SpeedyZero maintains on-par sample efficiency compared with EfficientZero while achieving a 17× speedup in wall-clock time, leading to human-level performances on the Atari benchmark within 30 minutes using only 300k samples. In addition, we also present an in-depth analysis on the fundamental challenges in further scaling our system to bring insights to the community.

1 INTRODUCTION

Deep reinforcement learning (RL) has achieved significant successes in the past few years. Prior work has scaled model-free RL training to computing clusters with tens to hundreds of machines, achieving human-level performance or beating human experts on various complex problems (Jaderberg et al., 2019; Baker et al., 2019; Berner et al., 2019; Vinyals et al., 2019). There are two fundamental ideas behind their successes: (1) training with larger batches for faster convergence, as used in the task of hide-and-seek (Baker et al., 2019), DOTA 2 (Berner et al., 2019) and even in many popular PPO projects (Yu et al., 2021; Stooke & Abbeel, 2018), (2) developing systems with high scalability, such as Gorila (Nair et al., 2015), Ape-X (Horgan et al., 2018), IMPALA (Espeholt et al., 2018) and R2D2 (Kapturowski et al., 2018), which can efficiently simulate huge numbers of environments in parallel. Despite the achievements, these model-free-RL applications consume an extremely high volume of samples, which can be infeasible for many real-world scenarios without an efficient simulator accessible.

By contrast, model-based RL methods require substantially fewer samples to train a strong agent. In particular, some recent works have even achieved comparable sample efficiency to humans in complex RL domains like Atari (Ye et al., 2021) or robotic control (Wu et al., 2022). The downside of model-based RL methods is that they often require a long training time (Schrittwieser et al., 2020; Ye et al., 2021). Although people have tried to accelerate simple model-based RL methods in the existing literature (Zhang et al., 2019; Abughalieh & Alawneh, 2019), state-of-the-art sample-efficient model-based RL such as EfficentZero (Ye et al., 2021), which requires complicated model learning and policy planning, are still time-consuming to run.

In this paper, we aim to build a state-of-the-art sample-efficient model-based RL system that trains fast in wall-clock time. To do so, we start with EfficientZero, a state-of-the-art sample efficient model-based RL method, and then accelerate it with massively parallel distributed training. We remark that scaling state-of-the-art model-based RL methods like EfficientZero is non-trivial for two major challenges. First, different from model-free RL, for which massive parallelization can be simply achieved by simultaneously simulating more environments, a single training step in EfficientZero requires substantially more computation steps, including model expansion, back-track search/planning, Q-value backup, and re-analyzing past samples. Therefore, it is non-trivial to effi-

ciently parallelize these components. Second, we empirically notice that when the data producing rate is largely accelerated via parallelization, in order to retain the same sample efficiency, EfficientZero training may suffer from significant instabilities.

We present SpeedyZero, a distributed model-based RL training system, which leverages dedicated system-level optimization to largely reduce computation overhead while inheriting the high sample efficiency from EfficientZero. From the system perspective, SpeedyZero contains three major innovations for a much faster per-iteration training speed, including (1) a non-trivial partition over computation workload to reduce network communications, (2) applying shared memory queues for high-through-put and low-latency intra-process communication, and (3) an optimized data transfer scheduling with reduced CPU-GPU communication and redundant data transmission. Furthermore, from the algorithm perspective, SpeedyZero is equipped with a novel technique, Priority Refresh (P-Refresh), to significantly stabilize model-based training in the case of massive parallelization and larger batch sizes. P-Refresh is a distributed and more aggressive variant of prioritized experience replay (Schaul et al., 2015), which actively re-computes the *accurate* priorities of *all* the samples in the replay buffer and is shown to be critical for the overall success of SpeedyZero.

We evaluate SpeedyZero on the Atari 100k benchmark (Kaiser et al., 2019), SpeedyZero achieves human-level performance with only 30 minutes of training and 300k samples. Compared with EfficientZero, which requires 8.5 hours of training, SpeedyZero retains a comparable sample efficiency while achieving a $17\times$ speedup in wall-clock time. Ablation studies are also presented to show the effectiveness of each design component and technique in our system. In addition, we also conduct a further study on the effect of batch size, and the results show that when the batch size increases, SpeedyZero may significantly drop. We carefully analyze the underlying bottleneck and hope the insights can benefit the community for future research.

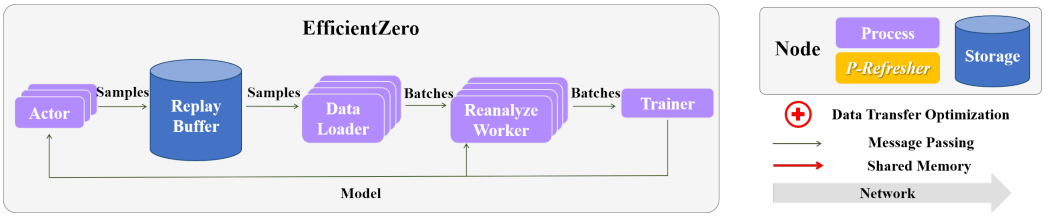
Our main contributions are summarized as follows,

- We develop SpeedyZero, a massively parallel distributed model-based RL training system featuring high sample efficiency and fast training speed. SpeedyZero masters Atari in 30 minutes with 300k samples, achieving $17\times$ speed up and on-par sample efficiency compared with the state-of-the-art EfficientZero algorithm.
- SpeedyZero adopts three system optimization techniques that significantly reduce latency and improve training throughput and further leverages an algorithmic technique, Priority Refresh, to stabilize training with limited training steps and data.
- We present comprehensive ablation studies on our system components and in-depth analysis on the existing bottleneck when further scaling SpeedyZero, leading to practical suggestions and open questions for the community.

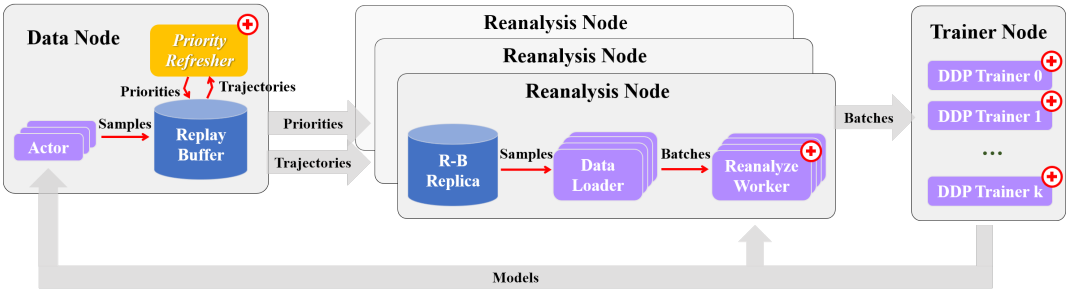
2 Related Work

Distributed Machine Learning With the emergence of larger datasets and larger models, distributed machine learning systems proliferate in industry and in research. Two main branches exist in this field: data parallelism and model parallelism. Data parallelism partitions an enormous dataset into small chunks computationally tractable on single machines (or GPUs) and assigns the chunks to different machines (or GPUs) in the training cluster. Successful frameworks of data parallelism include three generations of parameter servers (Smola & Narayanamurthy, 2010; Dean et al., 2012; Li et al., 2014) and distributed data parallel (Li et al., 2020). On the other hand, model parallelism like Megatron-lm (Shoeybi et al., 2019) handles the problem of training gigantic models with billions of parameters by assigning different layers of the model to different machines. In our case, the model is small enough to fit onto a single GPU, while the sample batch is too large for efficient single GPU training. Therefore, we use distributed data parallel provided by PyTorch (Li et al., 2020) for multi-GPU training in SpeedyZero.

Distributed Deep Reinforcement Learning There have been many successful attempts to scale out model-based deep RL methods with distributed training. Assuming known environment models, AlphaGo (Silver et al., 2016) achieves super-human performance in the game of Go by distributing the rollout process in the Monte-Carlo Tree Search (MCTS) over hundreds of machines. Its successor MuZero (Schrittwieser et al., 2020) achieves astonishing results in the game of Go with thousands of TPUs. (Zhang et al., 2019) proposes an asynchronized framework for paralleling model-based



(a) EfficientZero System Architecture



(b) SpeedyZero System Architecture

Figure 1: **System Architecture Comparison between EfficientZero and SpeedyZero:** EfficientZero finishes all computation on a single machine. In comparison, SpeedyZero partitions the workflow into data collection (Data Node), batch reanalysis (Reanalysis Node), and training (Trainer node) and distributes the three stages to different machines. SpeedyZero has three main novelties in its architecture (Sec. 4.2): (1) modular design with non-trivial workload partition, (2) efficient on-node communication with shared memory, (3) data transfer optimizations that reduce CPU-GPU communication and overlap data transfer with computation. It also implements our proposed Priority Refresh (Sec. 4.3) in priority refresher of the data node for more stable training.

RL training using the idea of parameter servers. However, their method is based on a simple model-based RL method, which is easy to parallelize but has relatively low sample efficiency. In the domain of model predictive control, many works (Abughalieh & Alawneh, 2019) study parallelism for improved speed, but focus on simple settings that are far less complicated than games like Atari. There are also many efforts in scaling out model-free deep RL methods, including Gorila (Nair et al., 2015), Ape-X (Horgan et al., 2018), IMPALA (Espeholt et al., 2018) and R2D2 (Kapturowski et al., 2018). These prior works typically focus on data-rich settings, requiring millions to billions of samples and hours to days of training. (Stooke & Abbeel, 2018) studies accelerated training of model-free methods such as PPO and A2C on a single machine. Our focus is on speeding up the training of model-based RL methods while maintaining high sample efficiency, which could bridge the training speed gap between model-based and model-free methods.

Besides system design optimizations, many prior works also adopt algorithmic improvements for better performance in distributed training of deep RL agents. IMPALA (Espeholt et al., 2018) introduces V-trace to correct the policy lag between the actors and learners. R2D2 (Kapturowski et al., 2018) proposes 'burn-in' steps to deal with the parameter lag in the recurrent neural networks. Many works also adopt prioritized experience replay (PER) in distributed training settings (Horgan et al., 2018; Kapturowski et al., 2018; Schrittwieser et al., 2020). Different from previous PER methods, our proposed prioritized sampling method, Priority Refresh, is able to stabilize value function training in the case of limited training data and training steps.

3 PRELIMINARY

Reinforcement Learning. A partially observable Markov decision process (POMDP) is defined by $M = \langle \mathcal{S}, \mathcal{A}, T, U, \Omega, \mathcal{O} \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is the set of possible actions, T is a transition function over next states given the actions at current states, and $U : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. Ω is the set of observations of the agent and $\mathcal{O}$ maps states to probability distributions over observations. We use $o_{\leq t}$ to denote the history of observations at timestep t. The objective of the agent is to find a policy π that maximizes the expected discounted return $\mathbb{E}[\sum_t \gamma^t u_t | a_t \sim \pi(\cdot|o_{\leq t})]$ where u_t is the reward at step t.

MuZero. MuZero (Schrittwieser et al., 2020) is a model-based RL method based on the Monte-Carlo Tree Search (MCTS) algorithm. MuZero learns the environment dynamics and performs MCTS over the learned environment model to find a better policy. More specifically, MuZero models the environment with a representation function h, a dynamics function g, and a prediction function f. To find a high-quality policy given a history of observations $o_{\leq t}$, MuZero first encodes the observation history by $s_t^0 = h(o_{\leq t})$, which is used as the latent state at the root of the tree. To perform MCTS, MuZero runs N simulation steps. In the k-th simulation step, a leaf node s' and an unexplored action a' on the leaf node are chosen by employing the UCT rule (Kocsis & Szepesvári, 2006; Rosin, 2011). Then a node expansion step comes by computing the next latent state s_t^{k+1} and reward r_t^{k+1} by the dynamics function g: $s_t^{k+1}, r_t^{k+1} = g(s', a')$, as well as the policy and value at s_t^{k+1} by the prediction function f: $v_t^{k+1}, p_t^{k+1} = f(s_t^{k+1})$. At the end of each simulation step, the value v_t^{k+1} is back-propagated along the tree path to update the Q values. The MCTS process is computationally expensive since it requires extensive CPU operations to do tree search as well as GPU resources for model inference.

MuZero interacts with the environment by searching a policy using MCTS over the learned environment model. The trajectories of data are then stored in the replay buffer. During training, a batch of observation histories are sampled and MuZero rolls out the environment model on the batch of observation histories $\{o_{\leq t}\}$ along the actions $\{a_{t \ldots t+K-1}\}$ at the following K steps, and predicts a batch of rewards $\{r_t^{1 \ldots K-1}\}$, policies $\{\pi_t^{0 \ldots K-1}\}$ and values $\{v_t^{0 \ldots K-1}\}$. To learn the models, the following loss is minimized,

$$\sum_k \mathcal{L}(u_{t+k}, r_t^k) + \lambda_1 \mathcal{L}(\pi_{t+k}, p_t^k) + \lambda_2 \mathcal{L}(z_{t+k}, v_t^k)$$

where u_{t+k} is the environment reward, π_{t+k} is the target policy obtained through MCTS over a target model, $z_{t+k} = \sum_{i=0}^{n-1} \gamma^i u_{t+k+i} + \gamma^n v'_{t+k+n}$ is the discounted n-step return, v'_t is the value computed by the target model.

To improve the sample efficiency, MuZero Reanalyze algorithm (Schrittwieser et al., 2020) regenerates the policy and value of a training batch when the batch is sampled from the replay buffer. Compared with MuZero, MuZero Reanalyze uses significantly fewer samples while still achieving strong results. It is worth noting that the reanalysis step over the sampled batch is computationally expensive since it involves an additional MCTS procedure.

EfficientZero. EfficientZero (Ye et al., 2021) is a sample-efficient visual RL algorithm built on top of MuZero Reanalyze algorithm, which re-computes the target policies via MCTS when a training batch is sampled from the replay buffer. EfficientZero further proposes several augmentations in visual RL tasks, including using self-supervised consistency loss to provide more training signals to the environment model, predicting the value prefix instead of the reward to deal with aleatoric uncertainty and off-policy correction for the n-step return. The workflow of EfficientZero is shown in Fig. 1a, where the Reanalyze workers continuously generate the training batch. EfficientZero suffers from the same computation expense issue as MuZero Reanalyze due to the reliance on reanalyzing batches. Our method SpeedyZero inherits the sample efficiency optimizations from EfficientZero and boosts its training speed by 17×.

4 SPEEDYZERO: A FAST AND EFFICIENT MODEL-BASED RL SYSTEM

4.1 OVERVIEW

The ultimate goal of SpeedyZero is to speed up the training of EfficientZero-based RL agents while maintaining on-par sample efficiency. We achieve this through efforts on both the system side and algorithm side in SpeedyZero, as shown in Fig. 1. The system optimizations in SpeedyZero help us reduce the time needed for each training step. The algorithm optimizations reduce the number of training steps needed while maintaining the stability of the training process. We will discuss the system optimizations in Sec. 4.2 and the algorithm optimizations in Sec. 4.3 respectively.

4.2 SPEEDYZERO SYSTEM DESIGN

As shown in Fig. 1, SpeedyZero partitions the workflow of EfficientZero into three stages: data collection, batch generation, and training. The three stages are distributed to the data node, the

reanalysis node, and the trainer node respectively. SpeedyZero features the following three system design novelties for higher training step throughput and lower latency on critical modules.

Modular Design and Non-Trivial Workload Partition: A naive partition of workload to multiple machines entails massive network data transfer, inducing high latency and low throughput. SpeedyZero follows a modular design, in which we partition the workflow into three major stages so that the data transfer across stages is reduced and we assign one type of node for each stage. The machines (nodes) in SpeedyZero work together asynchronously and we can easily add more machines for a single type of node if it becomes the bottleneck. Our partition strategy helps us scale out SpeedyZero for higher throughput with low network latency overhead.

Efficient On-Node Communication with Shared Memory: Tradition message passing communication between different processes on the same machine serializes data on the sender side and deserializes them again on the receivers (known as data ser/des). This process also entails multiple memory copies. The data ser/des and memory copies make message passing extremely slow when the amount of data we need to transfer is huge. We notice that in our settings, the majority of data transferred can be expressed using NumPy arrays and many of these arrays are written once but read multiple times. Therefore, we develop a special shared memory object store for on-node communication. It avoids data ser/des for NumPy arrays and supports non-copy, lock-free reads for them. With this shared memory object store in hand, processes in SpeedyZero can communicate with higher bandwidth and lower latency.

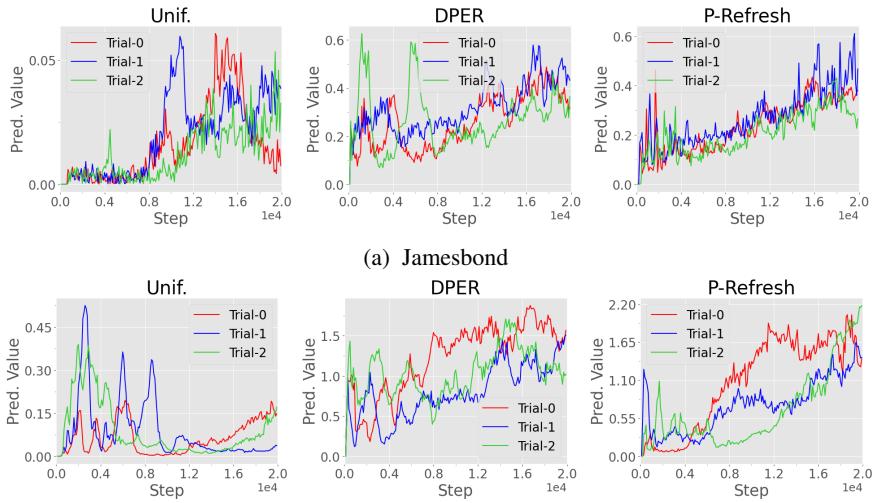
Data Transfer Optimizations: We empirically find that batch generation latency on the reanalysis node and priority refresh latency on the data node affect the final performance a lot. We also notice that a considerable amount of time is spent on CPU-GPU data transfer in the two components. Therefore, we reduce the CPU-GPU data transfer during batch reanalysis by storing all MCTS latent states on GPU. For priority refresh (Sec. 4.3), we store the observations in the replay buffer on GPU to avoid loading them every time the priorities are re-computed. These two optimizations help SpeedyZero better utilize GPU VRAM to reduce the latency on critical components.

Data transfer can also overlap with computation in many cases to further reduce latency and improve throughput. In SpeedyZero, we overlap network transmission with computation by sending and receiving network packages in separate processes, allowing workers to continue their jobs when the package is flying through the network. Moreover, on the trainer node, we also overlap batch loading with training by preloading the next batch into GPU.

4.3 Algorithm Improvements

Unstable Model-Based Training: We empirically find that the predicted values may behave unstably, especially during the initial training stage. The instability issue is largely due to the accelerate training speed and a reduced amount of training steps due to the requirement of unchanged sample efficiency. We remark that the issue is not severe in EfficientZero since EfficientZero uses a much longer overall training time than SpeedyZero does. As shown in Fig. 2, the predicted values climb abnormally high at the beginning, and it takes many steps before the values decrease back to the normal range. Notably, this phenomenon exists when using either prioritized experience replay (DPER) (Horgan et al., 2018) or uniform sampling from the replay buffer. The surge in predicted values at the beginning prevents proper policy improvements since MCTS relies on predicted values. These oscillating value predictions further slow down the training progress.

Priority Refresh: To address the above issue, we propose *Priority Refresh*, in which we actively refresh the priorities of all data points in the replay buffer. As shown in Fig. 1b, a group of *priority refreshers* periodically update the priorities of all data points in the replay buffer. The latest priorities are synchronized to all reanalysis nodes with a constant frequency. Since the goal is to stabilize the values, we use TD errors as the priorities. The key difference between P-Refresh and Distributed Prioritized Experience Replay (DPER) (Horgan et al., 2018) is that DPER only updates priorities of data points that are trained on, while P-Refresh updates priorities of all data points. This difference allows SpeedyZero to effectively use data from old policies, which stabilizes the training and leads to better performance.



(a) Jamesbond

(b) KungFuMaster

Figure 2: The predicted values of different trials when using uniform sampling from the replay buffer (Unif.), distributed prioritized experience replay (DPER), and priority refresh (P-Refresh) in Jamesbond and KungFuMaster. Uniform sampling exhibits very unstable values. Values of DPER are less unstable but still suffer from the same instability issues. Notably, in KungFuMaster, the values of uniform sampling are abnormally high during the initial stage of training and the agent can not get a higher reward until the values go down at around 10k steps. In contrast, P-Refresh shows stable improvement in the predicted values and exhibits much lower variance across different trials.

4.4 Implementation

SpeedyZero is highly optimized for higher training throughput and lower latency on critical modules. In this section, we will introduce some key system efficiency optimizations. For more details about the implementation of SpeedyZero, please refer to the appendix.

Distributed Data Parallel: We use Distributed Data Parallel provided by PyTorch (Li et al., 2020) to amortize batch size on multiple GPUs for faster training.

Data Compression: We compress the batches sent through the network with the lz4 algorithm, reducing the average network bandwidth requirement by $12\times$. Also, lz4 features high compression and decompression throughput, causing negligible overhead to our overall latency.

Replay Buffer Replication: We keep a replica of the replay buffer on each reanalysis node. This grants data loaders fast access to the data they need and also reduces traffic over the network. Each trajectory is only transmitted once while priorities are synchronized throughout the training process.

5 Experiments

5.1 Experiment Setup

Atari 100k Benchmark. The Atari 100k benchmark is proposed for testing the effectiveness of sample efficient RL methods (Kaiser et al., 2019). It contains 26 Atari games that are deemed solvable with a limited amount of samples. In this benchmark, agents are allowed to take at most 100 thousand environment steps, which are equivalent to 400 thousand frames due to a frameskip of 4. EfficientZero is the first method that achieves human performance in terms of both the mean and median of the human normalized score on this benchmark. In our experiments, we test SpeedyZero on the Atari 100k benchmark and also conduct additional experiments on the same set of games with 300k environment steps. Raw performance on each game as well as the mean and median of the human normalized score is reported. Human normalized score is computed as $(\text{score}_{\text{agent}} - \text{score}_{\text{random}})/(\text{score}_{\text{human}} - \text{score}_{\text{random}})$. The baselines we compare against include SimPLe (Kaiser et al., 2019), CuRL (Srinivas et al., 2020), SPR (Schwarzer et al., 2020), MuZero and EfficientZero. All baselines use 100k environment steps.

For the main results in Sec. 5.2 and ablation study in Sec. 5.3, the trainer node is configured with 4 DDP trainers and each DDP trainer receives batches with batch size 128 for training, indicating a total batch size of 512. The model held by Reanalyze workers is updated every 40 training steps. The

Game	Random	Human	SimPLE	CuRL	SPR	MuZero	EfficientZero	SpeedyZero			
Environment Steps	/	/	100k	100k	100k	100k	100k	300k	300k	300k	100k
Time	/	/	/	/	/	8.5 hours	8.5 hours	**0.5 hours**	**0.75 hours**	**1 hour**	**0.75hours**
Alien	227.8	7127.7	616.9	558.2	801.5	530.0	1140.3	818.4	734.1	1177.7	543.2
Amidar	5.8	1719.5	88.0	142.1	**176.3**	38.8	101.9	73.5	98.8	112.7	68.7
Assault	222.4	742.0	527.2	600.6	571.0	500.1	1407.3	991.5	906.2	1470.0	616.4
Asterix	210.0	8503.3	1128.3	734.5	977.8	1734.0	16843.8	41993.3	67179.1	49494.6	4226.9
Bank Heist	14.2	753.1	34.2	131.6	**380.9**	192.5	361.9	205.8	217.6	187.3	128.8
BattleZone	2360.0	37187.5	5184.4	14870.0	16651.0	7687.5	**17938.0**	9655.3	6418.1	8906.3	5861
Boxing	0.1	12.1	9.1	1.2	35.8	15.1	44.1	21.2	31.3	47.0	15.5
Breakout	1.7	30.5	16.4	4.9	17.1	48.0	406.5	258.6	340.9	409.8	304.6
ChopperCmd	811.0	7387.8	1246.9	1058.5	974.8	1350.0	**1794.0**	356.7	472.1	453	408.8
CrazyClimber	10780.5	35829.4	62583.6	12146.5	42923.6	56937.0	80125.3	83232.3	98454.9	93455.2	69929.4
Demon Attack	152.1	1971.0	208.1	817.6	545.2	3527.0	13298.0	54994.4	34404.9	39699.7	5407.9
Freeway	0.0	29.6	20.3	**26.7**	24.4	21.8	21.8	0.0	4.0	0.0	13.9
Frostbite	65.2	4334.7	254.7	1181.3	**1821.5**	255.0	313.8	1105.2	561.7	238.9	261.1
Gopher	257.6	2412.5	771.0	669.3	715.2	1256.0	**3518.5**	2465.5	1989.7	2929.4	1481.4
Hero	1027	30826.4	2656.6	6279.3	7019.2	3095.0	**8530.1**	7806.1	7633.0	8461.3	6115.2
Jamesbond	29.0	302.8	125.3	471.0	365.4	87.5	459.4	494.3	351.6	535.5	289.9
Kangaroo	52.0	3035.0	323.1	872.5	**3276.4**	62.5	962.0	411.3	1012.7	1246.9	525.9
Krull	1598.0	2665.5	4539.9	4229.6	3688.9	4890.8	6047.0	6771.6	5659.6	7962.6	5470.5
Kung Fu Master	258.5	22736.3	17257.2	14307.8	13192.7	18813.0	**31112.5**	24159.7	19958.8	21700	17253.5
Ms Pacman	307.3	6951.6	1480.0	1465.5	1313.2	1265.6	1387.0	1152.5	1716.6	3386.8	805.5
Pong	-20.7	14.6	12.8	-16.5	-5.9	-6.7	**20.6**	1.8	3.8	18.2	-4.9
Private Eye	24.9	69571.3	58.3	**218.4**	124.0	56.3	100.0	75.7	55.3	48.5	63
Qbert	163.9	13455.0	1288.8	1042.4	669.1	3952.0	15458.1	9584.7	9034.4	16334.2	4389
Road Runner	11.5	7845.0	5640.6	5661.0	14220.5	2500.0	18512.5	13350.0	11942.1	21692.2	5053.3
Seaquest	68.4	42054.7	683.3	384.5	583.1	208.0	1020.5	494.4	663.7	1047.5	367.1
Up N Down	533.4	11693.2	3350.3	2955.2	**28138.5**	2896.9	16095.7	4582.6	4942.7	10064	2376.1
Normed Mean	0.000	1.000	0.443	0.381	0.704	0.562	1.904	2.440	2.213	2.660	1.038
Normed Median	0.000	1.000	0.144	0.175	0.415	0.227	**1.160**	0.500	0.531	0.904	0.383

Table 1: Scores and running time achieved by SpeedyZero and some baselines on the Atari 100k benchmark. Compared with previous RL methods, SpeedyZero achieves human-level performance and performs best on 12 out of 26 games with 10× shorter training time. The results of SpeedyZero are evaluted with 100 evaluation episodes. The 0.5 hour and 1 hour results of SpeedyZero with 300k environment steps are evaluated with 3 training seeds. The 0.75 hour results of SpeedyZero with 100k and 300k environment steps are evaluated with 16 seeds. The training time of EfficientZero is evaluated under the same machine configuration of the 300k, 30 minutes experiment of SpeedyZero.

models of the priority refreshers and actors are updated every 10 training steps. The total number of training steps is 20k. As shown in Table 1, we run SpeedyZero with three different clusters, resulting in 30min, 45min and 60min total running time due to differences in the machine hardware. The detailed hardware configuration of the three clusters is listed in the appendix. As a reference, EfficientZero uses a batch size of 256 and 120k training steps, taking over 8.5 hours to finish training under the same machine configuration used by the 30min experiments of SpeedyZero.

In the bottleneck analysis of Sec. 5.4, we carry out case studies on large batch size training with limited steps and point out several bottlenecks that prevent SpeedyZero from using larger batch sizes and smaller number of training steps. We compare SpeedyZero and model-free method PPO (Schulman et al., 2017) under the setting of large batch size training on a subset of representative environments. PPO uses 25 million environment steps for training. For SpeedyZero, when using larger batch sizes, we accordingly shrink the interval of updating the target model and the number of training steps. When using a batch size of 1024, the target model update interval is 20 and the number of training steps is 10k. When using a batch size of 2048, the target model update interval is 10 and the number of training steps is 5k.

5.2 Mastering Atari in 30 Minutes

Table. 1 compares the result of SpeedyZero under different cluster configurations and number of environment steps with the result of other methods on the set of games in the Atari 100k benchmark. In all the experiments conducted, SpeedyZero surpasses human performance on the normalized mean scores. Running with 100k environment steps for 45 minutes, SpeedyZero achieves a normalized mean of 1.038 and a normalized median of 0.383. Compared with EfficientZero, which uses 8.5 hours on each game, SpeedyZero achieves 11.3× speedup. We remark that the performance gap is mainly caused by the reduced number of training steps in our setting.

Additionally, We perform experiments with 300k environment steps on three different clusters which allow SpeedyZero to finish 20k training steps in 30, 45, and 60 minutes. In the 30 minutes experiment, which accelerates EfficientZero by 17×, SpeedyZero achieves a normalized mean of 2.440 and normalized median of 0.500. In the 45 minutes experiment, SpeedyZero demonstrates a similar performance to the 30 minutes experiment, with a normalized mean of 2.213 and normalized median of 0.531. In the 60 minutes experiment, SpeedyZero achieves the best performance, with a normalized mean of 2.660 and a normalized median of 0.904. The main reason for the performance

Game	DPER	Uniform	P-Refresh (60min)	P-Refresh (30min)
Pong	2.9	-5.7	**18.2**	1.8
Jamesbond	309.4	87.9	**535.5**	494.3
UpNDown	3759.4	3296.9	**10064**	4582.6
Hero	6836.9	3405.0	**8461.3**	7806.1

Table 2: Ablation study on different training data sampling strategy, i.e., DPER, uniform sampling, and Priority Refresh, on four environments. Unif. is the worst in all environments while Priority Refresh outperforms all the baselines in these environments.

gap between these experiments is the non-uniform speedup of different components on machines with faster GPUs. For example, increasing the training speed while not accelerating the data generating process will influence the "priority staleness", which measures the model version gap between when a batch is sampled from the replay buffer and when it is trained on. As shown in the ablation study 5.3, the priority staleness has a significant effect on the final performance.

5.3 Ablation Study

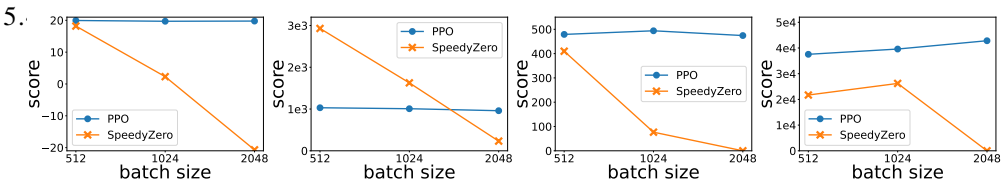
Priority Refresh. As stated in Sec. 4.3, the RL agent's final performance suffers a lot from the unstable predicted values during the initial training stage. Therefore we propose Priority Refresh to stabilize the training process. In this section, we carry out an ablation study on Priority Refresh by comparing Priority Refresh with DPER and uniform sampling.

As shown in Fig. 2, when using uniform sampling, the predicted values remain unstable throughout the training process. When using DPER, although the values sometimes seem more stable, the variance across different trials is still considerably high. In contrast, when using Priority Refresh, training is much more stable within a single run and across multiple runs. The stability of values also significantly affects the final performance of RL agents. In Table. 2, Priority Refresh achieves the highest score among all sampling methods in a set of environments. This suggests the superiority of Priority Refresh in stabilizing training and improving final performance.

Priority Staleness. As data loaders run in parallel with the DDP trainers, there is a time gap between a batch is sampled and is trained on. This means that datapoints in the batch are sampled according to priorities from an old version of the model. This gap is measured by "priority staleness", i.e. the step when the batch is trained on minus the step when it is sampled. We ablate different degrees of priority staleness in Table. 3. When the priority staleness is very small, SpeedyZero has poor performance. We hypothesize that the reason is that a larger priority staleness brings some regularization effect on the sampling probability, hence preventing the training from only focusing on a limited number of datapoints. However, the priority staleness is hard to control in SpeedyZero, since it depends on the latency of multiple components in the system, which could differ a lot on machines with different configurations. We leave the study of optimal priority staleness and improved schemes to control the priority staleness as future work.

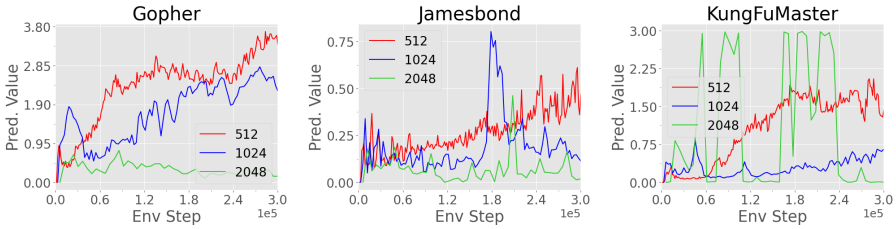
Staleness	15	20	35
Pong	-0.7	1.8	**18.2**
Jamesbond	351.3	494.3	**535.5**
UpNDown	4268.4	4582.6	**10064**

Table 3: Performance in Pong, Jamesbond and UpNDown with different degrees of priority staleness. The result indicates that the priority staleness should not be too small, i.e., the priorities should be recomputed at a proper frequency, for the best performance.



(a) Pong (b) Gopher (c) Jamesbond (d) KungFuMaster

Figure 3: Effect of batch size for SpeedyZero comparing with distributed implementation of PPO. We report the scores of PPO and SpeedyZero on four different games with batch size of 512, 1024, and 2048. Here PPO consumes 25M samples while SpeedyZero uses 300k samples. With an increasing batch size, PPO maintains its performance while the performance of SpeedyZero drops a lot. With a batch size of 2048, SpeedyZero completely fails to learn any useful strategies.



(a) Gopher (b) Jamesbond (c) KungFuMaster

Figure 4: The predicted values on the training batches of a single trial when using different batch sizes in three environments. When the batch size is 512, the values show stable improvement. With a larger batch size, the training shows much slower improvement or even becomes unstable.

Batch Size	Oracle Staleness	SpeedyZero Staleness	Ratio
512	20	50.8	2.54
1024	10	36.3	3.63
2048	5	37.2	7.44

Table 4: Bottleneck analysis on the reanalyze staleness. Oracle staleness is the optimal staleness that could be achieved by the synchronized version of SpeedyZero. SpeedyZero Staleness measures the actual reanalyze staleness achieved by SpeedyZero. Finally, "Staleness Ratio" is the ratio of SpeedyZero staleness to oracle staleness. A larger staleness ratio indicates a larger gap between synchronized and asynchronized execution. As the batch size increases, the reanalyze ratio becomes larger, indicating a much more severe gap between SpeedyZero and the synchronized version.

Prior works have shown that PPO can be easily parallelized and benefit from large batch size training (Stooke & Abbeel, 2018). However, we find that it is hard to train with larger batch sizes in SpeedyZero. Fig. 3 compares PPO and SpeedyZero when using different batch sizes in a number of selected environments. PPO doesn't show performance degradation and maintains its performance when using larger batch size. However, SpeedyZero demonstrates different degrees of performance degradation in different environments. When using a batch size of 2048, SpeedyZero completely fails to learn useful strategies. We find the Reanalyze staleness as a critical factor influencing the performance of SpeedyZero and blocking the learning of SpeedyZero.

Reanalyze Staleness. Since reanalysis and training are parallelized, DDP trainers often do not receive batches that are reanalyzed by the latest target model but by an old version of the model. This model version gap of the training batches, which we call "Reanalyze staleness", could significantly affect the quality of training. In practice, we find that there could be several reasons resulting in an increased Reanalyze staleness, including improper queue design between the reanalysis node and the trainer node, a large latency of the Reanalyze processes, communication overhead and compression time over the training batches.

When using a larger batch size, which requires a shorter interval to update the target model, the issue of Reanalyze staleness becomes more severe since the latency of components except for the trainers remains the same while the interval between two continuous target model updates is much shorter. Table. 4 reports the ratio of actual SpeedyZero Reanalyze staleness to the oracle Reanalyze staleness when using different batch sizes. As expected, the ratio increases as the batch size increases. Fig. 4 further compares the values during the training process with different batch sizes. The values improve much slower and even become unstable when using larger batch sizes. It is an interesting direction to study how to reduce the Reanalyze staleness for large batch size training of SpeedyZero in future work.

6 Conclusion

In this work, we have developed a fast and sample efficient model-based RL training system, SpeedyZero, and introduced a new priority experience replay method, Priority Refresh, to stabilize training in settings with limited training steps. With the highly optimized system design and P-Refresh combined, SpeedyZero achieves human-level performance on the Atari 100k benchmark with only 300k samples and 30 minutes of training. This work is one step towards the application of RL in real-world scenarios where both sample efficiency and training time are mission-critical. We expect future research to further accelerate SpeedyZero and apply SpeedyZero in real world.

REFERENCES

Karam M Abughalieh and Shadi G Alawneh. A survey of parallel implementations for model predictive control. *IEEE Access*, 7:34348–34360, 2019.

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Max Schwarzer, Ankesh Anand, Rishab Goel, R Devon Hjelm, Aaron Courville, and Philip Bachman. Data-efficient reinforcement learning with self-predictive representations. *arXiv preprint arXiv:2007.05929*, 2020.

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

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

Aravind Srinivas, Michael Laskin, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Philipp Wu, Alejandro Escontrela, Danijar Hafner, Ken Goldberg, and Pieter Abbeel. Daydreamer: World models for physical robot learning. *arXiv preprint arXiv:2206.14176*, 2022.

Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. *Advances in Neural Information Processing Systems*, 34:25476–25488, 2021.

Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.

Yunzhi Zhang, Ignasi Clavera, Boren Tsai, and Pieter Abbeel. Asynchronous methods for model-based reinforcement learning. *arXiv preprint arXiv:1910.12453*, 2019.

A APPENDIX

A.1 CLUSTER HARDWARE CONFIGURATIONS

This section lists the cluster hardware configurations of the 30min, 45min and 60min experiments. For all these experiments, the training clusters contain five machines, among which one machine is used as the data node, another three as reanalysis nodes, and the last one as the trainer node. For the 30min experiments, each machine in the cluster contains 8 A100 80G GPUs (with NV-Switch), 128 CPU cores, and 1TB of RAM. For the 45min experiments, the trainer node contains 8 A100 80G GPUs (without NV-Switch), 128 CPU cores, and 512GB of RAM. The data node and the reanalysis nodes contain 2 NVIDIA RTX 3090 GPUs, 128 CPU cores, and 512GB of RAM. For the 60min experiments, each machine in the cluster contains 4 A10 GPUs, 128 CPU cores, and 512GB of RAM.

A.2 MODELS AND HYPER-PARAMETERS

SpeedyZero uses the same model as EfficientZero. The model consists of three modules: representation function, dynamics function, and prediction function, which are all represented as neural networks. We list the architecture of each module below:

Layer Type	Configuration	Activation	Batch Norm
Representation Network			
Convolution	kernel 3x3, stride 2, 32 output planes, output resolution 48×48	ReLU	Yes
Residual Block	32 planes	-	-
Residual Downsample Block	stride 2, 64 output planes, output resolution 24×24	-	-
Residual Block	64 planes	-	-
Average Pooling	stride 2, output resolution 12×12	ReLU	Yes
Residual Block	64 planes	-	-
Average Pooling	stride 2, output resolution 6×6	ReLU	Yes
Residual Block	64 planes	-	-
Dynamics Network			
-	Concate the latent states with input actions into 65 planes	-	-
Convolution	kernel 3x3, stride 2, 64 output planes	-	Yes
Convolution	kernel 3x3, stride 2, 32 output planes, output resolution 48×48	-	Yes
-	A residual link adds the input latent states with output	ReLU	No
Residual Block	64 planes	-	-
Prediction Network			
Residual Block	64 planes	-	-
Convolution	kernel 1x1, 16 output planes	ReLU	Yes
-	Flatten	-	-
Linear	D output dimensions (D differs for reward/policy/value)	-	-
Value-prefix Prediction Network			
Convolution	kernel 1x1, 16 output planes	ReLU	Yes
-	Flatten	-	-
LSTM	512 hidden size	ReLU	Yes
Linear	32 output dims	ReLU	Yes
Linear	601 output dims	-	-

Table 5: Network architecture of SpeedyZero.

We stack four historical frames as the input observations, with an interval of 4 frame-skip. The frames are staked on the channel dimension, hence the shape of the input is $96 \times 96 \times 12$.

Common hyper-parameters of SpeedyZero is as shown in Table. 6.

A.3 LARGE BATCH SIZE EXPERIMENT SETUP

For SpeedyZero, We use the linear scaling rule for large batch size training. The hyper-parameters for large batch size training are shown in Table. 7.

For PPO, we only change the batch size for different experiments. The hyper-parameters of PPO are shown in Table. 8. The network architecture we use for PPO is shown in Table. 9.

Parameter Name	Value
Observation down-sampling	96×96
Frame staked	4
Frame skip	4
Reward clipping	True
Terminal on loss of life	True
Max frames per episode	108K
Discount factor	0.997^4
Optimizer	SGD
Max gradient norm	10
Priority exponent (α)	0.6
Evaluation episodes	100
Actor model update interval	10
Unroll steps	5
TD steps	5
Policy loss coefficient	1
Value loss coefficient	0.25
Self-supervised consistency loss coefficient	2
LSTM horizon length	5
Dirichlet noise ratio	0.3
Number of simulations in MCTS	40

Table 6: Common hyper-parameters of SpeedyZero.

Batch Size	Target Model Update Interval	Training Steps	Learning Rate
512	40	20k	0.4
1024	20	10k	0.8
2048	10	5k	1.6

Table 7: Hyper-parameters of SpeedyZero with different batch sizes.

A.4 System Workflow and System Configurations

In this section, we describe the detailed workflow of SpeedyZero. We illustrate this by looking at how one data point (environment step) takes effect in the whole training process. The story begins on the data node. Actors on the data node collect the data point from the environment and put it into the replay buffer. Priorities refreshers periodically recompute the priorities of data points in the replay buffer. Since we have replay buffer replicas on reanalysis nodes, we need to synchronize the data points and their priorities to the replicas. Each data point is only sent once (when the trajectory is finished) while priorities are synchronized periodically throughout the training process.

On the reanalysis node, data loaders sample batches based on previously computed priorities and reformat the batches for ease of GPU reanalysis. The batches are then sent into a shared memory queue between data loaders and Reanalyze workers. The separation of data loading and reanalyzing decouples CPU workload with GPU workload, improving resource utilization. During our implementation, we partition data loaders and Reanalyze workers into several groups. Each group has one shared memory queue, and the data loaders can only communicate with Reanalyze workers in the same group. We use grouping here to achieve higher bandwidth and lower latency. Continue with the workflow, Reanalyze workers will then pop batches from the shared memory queue and reanalyze them using MCTS. The Reanalyze workers will not send the reanalyzed batches to the trainers. Instead, they will push the batches into another shared memory queue, called the batch queue. Batch senders will then take out the batches from the batch queue and send the batches through the network to the trainer node. This additional step overlaps slow network transmission with computation since Reanalyze workers can work on the next batch when the batch senders are sending the current batch.

Parameter Name	Value
Observation down-sampling	84×84
Frame staked	4
Frame skip	4
Learning rate	5e-5
Discount	0.99
Env steps	25M
PPO Epoch	4
GAE parameter (λ)	0.95
Number of actors	8
Entropy coeff.	0.01
VF coeff.	0.5

Table 8: Hyper-parameters of PPO

Layer Type	Configuration	Activation	Batch Norm
Convolution	kernel 8x8, stride 4, 32 output planes	ReLU	-
Convolution	kernel 4x4, stride 2, 64 output planes	ReLU	-
Convolution	kernel 3x3, stride 1, 64 output planes	ReLU	-
Linear	512 ouput dims	ReLU	-
Linear	act_dim output dims	-	-

Table 9: Network architecture of PPO.

Similarly, on the trainer node, the DDP trainers do not receive batches themselves. The batch receivers are responsible for batch receiving and they will put the batches into a trainer-side batch queue. The DDP trainers directly read the batches from this queue and use the batches in training.

In Table. 10, we show some key system configurations used to setup SpeedyZero on our cluster. This can serve as a reference when setting up SpeedyZero on new clusters.

Configuration Name	Value
Data Node	
Number of Actors	3
Number of Priority Refreshers	5
Replay Buffer Capacity (in number of trajectories)	8192
Replay Buffer Capacity (in memory consumption)	~32GB
Reanalysis Node	
Number of Data Loaders	36
Number of Reanalyze workers	24
Number of Batch Senders	32
Number of Data Loader - Reanalyer Queues	8
Data Loader - Reanalyer Queues Capacity (in number of batches, each)	32
Data Loader - Reanalyer Queues Capacity (in memory consumption, each)	~32GB
Number of Batch Storages	8
Batch Storage Capacity (in number of batches, each)	16
Batch Storage Capacity (in memory consumption, each)	~8GB
Trainer Node	
Number of DDP Trainers	4
Number of Batch Receivers	8
Number of Batch Storages	1
Batch Storage Capacity (in number of batches, each)	32
Batch Storage Capacity (in memory consumption, each)	~16GB
Signal Queue Capacity (in number of training signals)	16
Signal Queue Capacity (in memory consumption)	~2GB

Table 10: Key system configuration of SpeedyZero.