Nitro: Boosting Distributed Reinforcement Learning with Serverless Computing

Hanfei Yu
Stevens Institute of Technology
hyu42@stevens.edu

Jacob Carter
Louisiana State University
jcar116@lsu.edu

Hao Wang
Stevens Institute of Technology
hwang9@stevens.edu

Devesh Tiwari
Northeastern University
d.tiwari@northeastern.edu

Jian Li
Stony Brook University
jian.li.3@stonybrook.edu

Seung-Jong Park
Missouri University of Science and Technology
seung-jong.park@mst.edu

ABSTRACT

Deep reinforcement learning (DRL) has demonstrated significant potential in various applications, including gaming AI, robotics, and system scheduling. DRL algorithms produce, sample, and learn from training data online through a trial-and-error process, demanding considerable time and computational resources. To address this, distributed DRL algorithms and paradigms have been developed to expedite training using extensive resources. Through carefully designed experiments, we are the first to observe that strategically increasing the actor-environment interactions by spawning more concurrent actors at certain training rounds within ephemeral time frames can significantly enhance training efficiency. Yet, current distributed DRL solutions, which are predominantly server-based (or serverful), fail to capitalize on these opportunities due to their long startup times, limited adaptability, and cumbersome scalability.

This paper proposes *Nitro*, a generic training engine for distributed DRL algorithms that enforces timely and effective boosting with concurrent actors instantaneously spawned by serverless computing. With serverless functions, *Nitro* adjusts data sampling strategies dynamically according to the DRL training demands. *Nitro* seizes the opportunity of real-time boosting by accurately and swiftly detecting an empirical metric. To achieve cost efficiency, we design a heuristic actor scaling algorithm to guide *Nitro* for cost-aware boosting budget allocation. We integrate *Nitro* with state-of-the-art DRL algorithms and frameworks and evaluate them on AWS EC2 and Lambda. Experiments with Mujoco and Atari benchmarks show that *Nitro* improves the final rewards (*i.e.*, training quality) by up to 6× and reduces training costs by up to 42%.

PVLDB Reference Format:
Hanfei Yu, Jacob Carter, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. *Nitro*: Boosting Distributed Reinforcement Learning with Serverless Computing. PVLDB, 18(1): 66 - 79, 2024.
doi:10.14778/3696435.3696441

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IntelliSys-Lab/Nitro-VLDB25.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:10.14778/3696435.3696441

1 INTRODUCTION

Deep reinforcement learning (DRL) has achieved remarkable success in various fields, including gaming AI [10, 35, 65, 70], robotics [13, 78], and system scheduling [12, 45, 49, 55]. Unlike supervised learning, which trains on readily labeled data, DRL algorithms produce, sample, and learn from fresh training data online through time-consuming and resource-intensive trials and errors. This process presents fundamental challenges for both data sampling and resource provisioning. A few distributed DRL algorithms and infrastructures have been proposed to accelerate training by utilizing large-scale distributed computing [17, 21, 23, 31, 43, 50, 51, 72].

Existing distributed DRL solutions heavily rely on serverful computing infrastructures (*e.g.*, virtual machines (VMs) or bare-metal machines), suffering from low learning efficiency when training with large-scale resources [38]. Specifically, due to the lengthy initialization and startup overheads, serverful infrastructures fail to scale promptly at runtime to satisfy DRL algorithms' dynamic demands for training data and resources [74, 75]. Recent distributed DRL algorithms [17, 43] and frameworks [38, 39, 77] have been developed to optimize the training and resource efficiency by parallelizing training with large clusters and high-end workstations.

However, existing studies ignore the opportunities to accelerate DRL training by jointly optimizing training data sampling efficiency and computing resource provisioning. We are the first to observe that strategically and timely increasing the volume of data sampled by DRL actors during specific training periods can drastically accelerate training, which can be supported by existing theoretical studies on DRL training process [29, 66]. In this paper, we define the increase of actor-environment interactions via spawning more concurrent actors as "*boosting*." The time frameworks for boosting DRL training are typically *ephemeral* and *unpredictable* (§2.3). However, existing serverful solutions fail to capture such boosting opportunities due to their long startup times, limited adaptability, and clumsy scalability.

Thus, we propose *Nitro*, a generic training engine for DRL with serverless computing. *Nitro* is applicable to general actor-learner architectures by abstracting actors as lightweight serverless functions for both on-policy and off-policy algorithms. Serverless functions are known for their auto-scaling, which allows DRL actors to adjust data sampling strategies dynamically according to the training

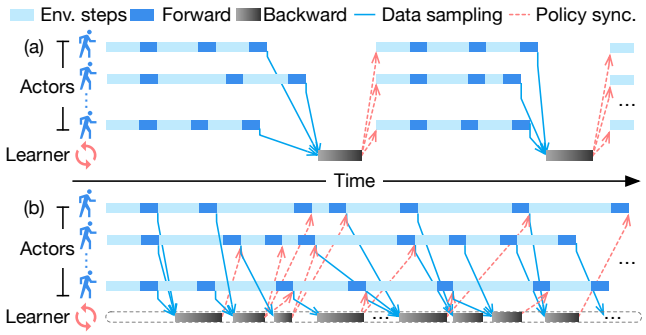


Figure 1: Actor-learner architectures for distributed DRL training, including (a) on-policy (synchronous) training and (b) off-policy (asynchronous) training.

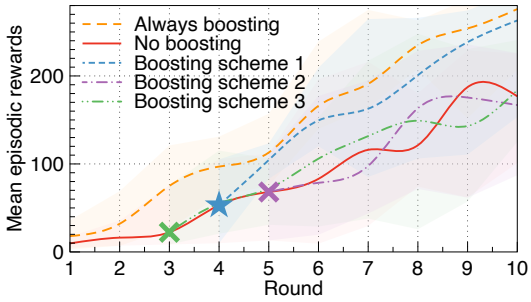


Figure 2: Boosting Proximal Policy Optimization (PPO) training in the Hopper environment by increasing the number of actors from default 1 to 16. ★ indicates a *boostable round*, and × indicates non-boostable rounds for boosting.

demands. Specifically, we inspect the neural network updates of learner policy and compute a boosting score by approximating the Hessian matrix to detect the boosting opportunities. By evaluating the runtime boosting efficiency at different rounds through DRL training, we devise a cost-aware heuristic algorithm to guide *Nitro* to optimize training efficiency at a minimal resource cost. The main contributions of this paper are summarized as follows:

- We design *Nitro*, a generic serverless training engine for both distributed on-policy and off-policy DRL algorithms by jointly optimizing training data sampling efficiency and resource provisioning with timely and effective DRL boosting. *Nitro* is prototyped with AWS EC2 and Lambda.
- We verify the effectiveness and robustness of boosting and devise a boosting score to detect and quantify the boosting opportunities in real-time DRL training. We also design a cost-aware heuristic algorithm to guide the budget allocation of *Nitro* through training.
- We evaluate the *Nitro* prototype by integrating with state-of-the-art (SOTA) DRL algorithms and frameworks. Extensive experiments with Mujoco and Atari benchmarks show that *Nitro* improves the final reward (*i.e.*, training quality) by up to 6× and reduces the training cost by up to 42%.

2 BACKGROUND AND MOTIVATION

2.1 DRL Actor-learner Architectures

DRL aims to optimize a policy π parameterized with θ by maximizing the expected return. The DRL agent learns to maximize the cumulative reward $J(\pi) := \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{n}\gamma^t r_t\right]$, where τ is a trajectory, r_t is the reward at timestep t, and γ is the discount factor. Trajectory τ is a specific data type to describe the training data used by DRL. One trajectory consists of a series of state-action pairs that a DRL agent experienced through environment interactions.

Actor-learner architectures are one of the most performant and efficient large-scale approaches to enable distributed DRL training [16, 17, 21, 23, 31, 43, 72]. Fig. 1 illustrates the actor-learner architecture for DRL training workloads. In actor-learner, an agent is divided into two sub-modules, *i.e.*, one *learner* and multiple *actors*. Each training *round* consists of two steps: 1) each actor interacts with a copy of the same environment under the guidance of a policy and submits the sampled data to the learner, and 2) the centralized learner computes gradients using the sampled data, updates its policy, and synchronizes the new policy to multiple actors. Though synchronous (on-policy in Fig. 1(a)) actor-learner training is more stable [18], recent solutions have shifted to asynchronous (off-policy in Fig. 1(b)) due to higher sampling efficiency [16, 17, 43].

2.2 Boosting Distributed DRL

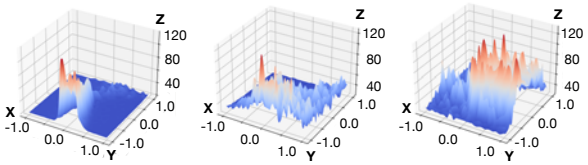
Drawing inspiration from actor-learner architectures that separate the learner and actors, we propose that augmenting the number of concurrent actors during certain rounds (*i.e.*, bootable rounds) could accelerate distributed DRL training. We define boostable rounds as those that have a high potential for achieving large reward gains and significant data quality improvement, compared to other rounds when provided with the same budget. Our hypothesis posits that provisioning a greater number of concurrent actors at boostable rounds results in an increased volume of data samples, obtained through interaction with the DRL environment. This potentially accelerates the learner's ability to escape local optima traps [29].

To verify this hypothesis, we should address the following three questions: **Q1**: Do such *boostable rounds* exist for accelerating DRL training? **Q2**: Why can boosting accelerate the training? **Q3**: When do boostable rounds appear, and how do we capture them?

We answer Q1 through a preliminary experiment of ten-round training with PPO algorithm [63] in the Mujoco [68] Hopper environment. Fig. 2 demonstrates boosting DRL training by increasing the number of actors at a *boostable round*. The no-boosting scheme launches only one actor by default to sample 256 timesteps per round. In the experiment, we design three boosting schemes. Each scheme boosts DRL with 16 actors and uniformly samples in total 4,096 (16×256) timesteps at a specific round and resumes one actor at other rounds. Scheme 1 performs boosting at Round 4, Scheme 2 at Round 5, and Scheme 3 at Round 3.[1] Note that all three boosting schemes require the same number of total actors (timesteps), thus under the same budget. We also include an always-boosting baseline with 16 actors throughout the ten rounds as an upper-bound.

Scheme 1 achieves significantly higher episodic rewards, Schemes 2 and 3 fail to accelerate the training due to boosting at non-boostable rounds. The experiment demonstrates that Round 4 is a *boostable round*. Compared to non-boostable rounds (*i.e.*, Rounds 3

[1]For a fair comparison between boosting schemes, we checkpointed and replayed the same learner policy at each boosting round, and individually retrained PPO for the remaining rounds.



(a) 256 t (1 actor). (b) 2048 t (8 actors). (c) 4096 t (16 actors).

Figure 3: 3D landscapes of the DRL surrogate objective boosted with 256 (1 actor), 2048 (8 actors), and 4096 (16 actors) trajectory timesteps at the boostable round in Fig. 2. X and Y axes show the two-dimensional perturbations on neural network parameters, and the Z axis shows the episodic rewards. Red and blue areas represent high and low rewards.

and 5), boosting DRL at Round 4 leads to higher boosting benefits (*i.e.*, episodic reward increase) given the same budget.

The existence of boostable rounds leads us to Q2—why can boosting accelerate the training? To figure it out, we plot and analyze the 3D landscapes of the *surrogate objective*, which is commonly employed by modern DRL algorithms to facilitate training [42, 61, 63, 67, 72]. Since it is difficult to directly optimize the true rewards, DRL algorithms tend to design a surrogate objective to guide the training, where the algorithms optimize the surrogates instead of true rewards. Therefore, we investigate how boosting (*i.e.*, increasing the volume of trajectory sampling) impacts the reward surfaces of DRL surrogate objective.

Following existing neural network visualization techniques [29, 36, 66, 73], we add two-dimensional perturbations (ranging from -1.0 to 1.0) to neural network parameters, where two perturbation directions are based on top two Hessian eigenvectors [73]. We evaluate the episodic rewards of each set of perturbations within a grid size of 30×30. Fig. 3 shows the reward surfaces of the surrogate objective boosted with 256 (1 actor), 2048 (8 actors), and 4096 (16 actors) trajectory timesteps at the boostable round, respectively. Boosting brings more diverse trajectories for the DRL neural network to learn by increasing the actor-parallelism in real-time. The surface of the surrogate objective gradually develops more high-reward regions (red areas) when the volume of sampled trajectories grows [29, 66], indicating that boosting can discover higher rewards and accelerate the training process. We provide a theoretical analysis in §6.1 to further justify the boosting performance gains.

Finally, the existence and effectiveness of boostable rounds naturally lead to Q3—when do they appear, and how to capture them?

2.3 Characteristics of Boostable Rounds

To answer Q3, we conducted a series of experiments across different DRL environments, algorithms, and runs in Fig. 4. Our observations indicate that boostable rounds are *ephemeral* and *unpredictable* in DRL training. Specifically, we devise a metric in §5, *boosting score*, that identifies boostable rounds in the ten-round training. Training rounds with a higher score indicate a potential for achieving higher boosting benefits.

Concretely, Fig. 4(a) shows the boosting scores when training PPO across three environments: Hopper, Humanoid, and Walker2d,

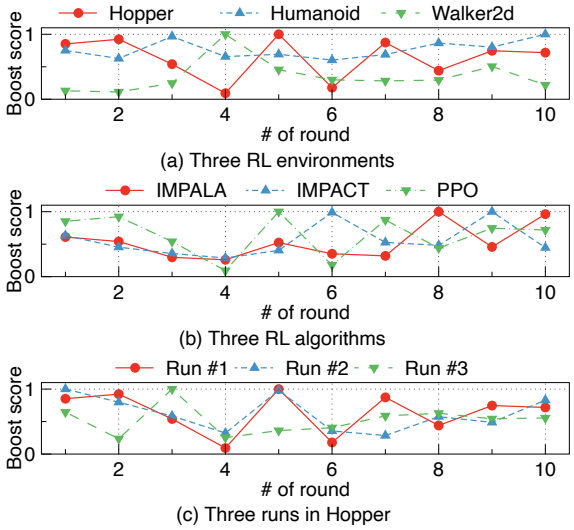


Figure 4: Boosting opportunities vary from environments (a), algorithms (b), and different runs in the same setting (c).

from Mujoco [68] benchmark. The result demonstrates that boosting opportunities vary from task to task. Fig. 4(b) shows the boosting scores for three popular DRL algorithms, including IMPALA [17], IMPACT [43], and PPO, training in the Hopper environment. Boosting opportunities can vary from algorithm to algorithm while training in the same environment. Fig. 4(c) shows the boosting scores for three different runs of PPO in the Hopper environment. Even for the same reinforcement learning (RL) task and algorithm, boosting opportunities are still stochastic and unpredictable.

Through the experiments, we can conclude that capturing boostable rounds in DRL training is challenging. The ephemerality and unpredictability of boosting opportunities make potential boosting solutions with high scaling overheads (*i.e.*, serverful backends) or predictions infeasible. We must seek a timely design that seizes the opportunities immediately upon detection for effective boosting.

2.4 Serverful *vs.* Serverless Distributed DRL

Serverless computing is becoming a promising computing infrastructure for Machine Learning (ML) training [11, 19, 59, 71] due to its fine-grained resource allocation, pay-as-you-go pricing, and agile scalability. Fig. 5(a) depicts the startup time of AWS Lambda functions and three popular types of EC2 instances. In contrast to the lengthy startup of VMs, functions deployed on serverless computing platforms (*e.g.*, AWS Lambda [5]) can be initialized within seconds, exhibiting a natural fit for instant boosting in DRL training.

Existing distributed DRL solutions [16, 17, 21, 23, 31, 43, 72] train on serverful clusters, such as VM-based cloud platforms. However, *serverful DRL solutions fail to catch the potential boost opportunities due to VMs' long startup time and clumsy scalability.*

During the idle window of VM initialization, actor-learner architectures must either pause the learner (synchronous) or replay old trajectories for learner update (asynchronous). Both approaches slow down the training process significantly. Fig. 5(b) shows the mean episodic reward curves with one-time boosting using serverless and two VM approaches (*i.e.*, VM pause and VM reply) with

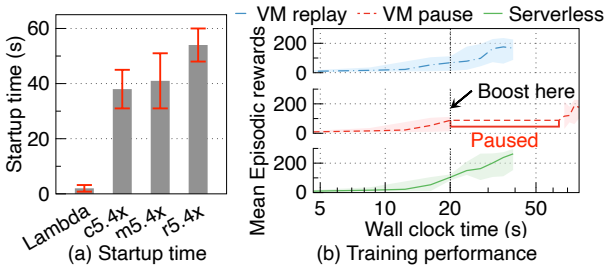


Figure 5: Startup time of AWS Lambda and three popular EC2 instances (`c5.4xlarge`, `m5.4xlarge`, and `r5.4xlarge`).

the same experimental setup in §2.2. We train PPO in the Hopper environment and boost with 16 actors for all three methods. For the serverless boosting, we launch 16 AWS Lambda functions as serverless actors. The two VM-based baselines both launch an AWS EC2 `c5.4xlarge` with 16 CPU cores, where each CPU core hosts an actor process. VM pause baseline stops the learner and waits for the VM startup, whereas VM replay baseline updates the learner with old trajectories from previous training rounds while waiting for the VM startup. When using VMs to boost training, *pausing* the learner incurs a long waiting time and severely increases the training time, while *replaying* old trajectories hinders the training performance. Unlike VMs, serverless functions can be launched in seconds and instantly bring new trajectories for learner update, thus achieving smooth and efficient boosting.

3 OBJECTIVES AND CHALLENGES

Nitro is carefully designed to achieve the three goals:

Effective and timely boosting. Due to the ephemerality and unpredictability (§2.2), serverful solutions can hardly seize the boosting opportunities instantly. *Nitro* exploits *serverless computing* to power and scale DRL actors for timely boosting. Correspondingly, *Nitro* should be scalable to support massive concurrent functions.

Accurate boosting opportunity detection. *Nitro* is keen on accurately spotting boosting opportunities for the stochastic training process of DRL tasks. We design *Nitro* to compute a metric, *i.e.*, boosting score, through the training to guide the boosting.

Cost efficiency. In actor-learner architectures, training cost comes from both sides of the learner and actor. *Nitro* should accelerate the DRL training while maintaining low cost on running the learner and actor. Our cost-efficient boosting design is two-fold: 1) *Nitro* carefully justifies the budget for each boostable round based on real-time training process. 2) *Nitro* enforces boosting decisions to reduce the computing infrastructure cost using serverless computing as existing works [11, 19, 20]. To realize the above objectives, we must answer the following fundamental questions:

How to enable efficient boosting with serverless computing on the actor-learner architecture? Existing actor-learner solutions solely rely on serverful implementation and deployment. While serverless computing is promising for instant boosting, the lack of GPU accelerators makes a pure serverless actor-learner architecture unfeasible [27, 28]. Therefore, we must carefully craft *Nitro* to achieve high boosting efficiency with co-design of serverful and serverless computing.

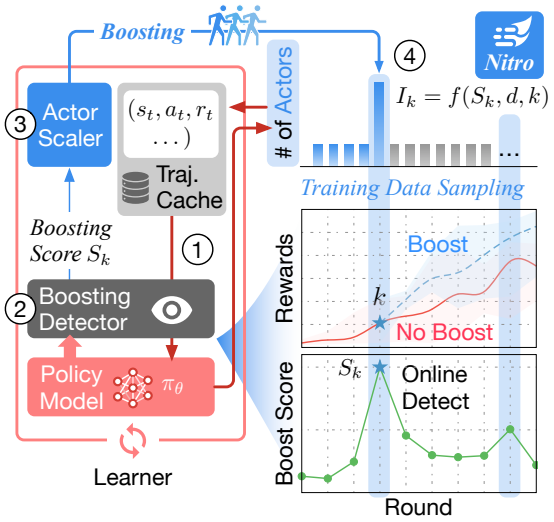


Figure 6: *Nitro*'s workflow.

How to detect boosting opportunities online with low overheads? We demonstrated that the boosting score can effectively reveal the boosting opportunities in §2.2. However, boosting real-time training requires the score computation to be online with negligible overheads. We must design the score to be easy-to-compute while providing precise guidance to boosting opportunities.

How to balance cost and efficiency while boosting DRL performance? Given a limited budget, *Nitro* should carefully justify and trade-off budgets on every boostable round. Spending too much on one boostable round may result in an insufficient budget for future boosting opportunities. However, the efficiency of each boostable round varies through training. It is hard to design a cost-aware actor scaler to boost DRL training efficiency at minimal costs.

4 *NITRO*'S OVERVIEW

Nitro is a DRL training engine that accelerates general DRL algorithms with cost-efficient boosting. Fig. 6 shows *Nitro*'s architecture and workflow. *Nitro*'s training cycle is supported by the five system components (Trajectory Cache, Policy Model, Boosting Detector, Actor Scaler, and Serverless Actors), summarized in four steps:

Step (1): Policy update. The learner periodically polls the *Trajectory Cache* and gathers new trajectories to update its policy network. Trajectory Cache is an in-memory data buffer that resides in *Nitro*'s learner server to store trajectories for experience replay. Trajectories submitted by the serverless actors are serialized and stored as key-value pairs in the cache, where the keys are unique function invocation IDs. The update procedure consists of two modes: on-policy and off-policy. In on-policy mode, the learner defers the update and continuously checks if the trajectory cache has enough samples until reaching the target. Off-policy mode updates the policy whenever new trajectories arrive in the cache and can replay old trajectories [23, 31, 40] if the cache is temporarily empty.

Step (2): Boosting opportunity detection. After a new policy is generated, *Nitro* inspects the new policy network to determine whether it needs boosting. The *Boosting Detector* is designed to inspect the boosting opportunities and compute the boosting score of the policy in real-time training. It analyzes the objective landscape

of DRL policy networks and determines if a policy is in boostable status by calculating a boosting score. Upon identifying a boostable round, the Detector notifies the *Actor Scaler* for timely boosting.

Step ③: Actor scaling. After the policy is analyzed, the *Actor Scaler* makes cost-aware actor scaling decisions based on the received boosting score and historical information (*e.g.*, the remaining budget). The decision is then forwarded to a serverless platform for launching concurrent DRL actors and sampling trajectories by rolling out the new policy.

Step ④: Serverless execution. Each *Serverless Actor* receives the new policy weights and synchronizes its own policy. Then, the actors interact with the environment using the new policy and collect new trajectories from the policy. Once complete, each actor individually submits the samples back to *Nitro*'s cache for future policy updates. In *Nitro*, issuing a boosting decision is equivalent to massively increasing the number of concurrently launched serverless actors (increasing samples), thus improving policy learning.

To boost on-policy algorithms [61, 63], *Nitro* defers launching the actors until the learner completes the policy update, ensuring each actor receives the latest policy weights. For off-policy algorithms [16, 17, 43, 50], *Nitro* launches actors while asynchronously updating itself, free from the blocking of actor synchronization. We repeatedly train DRL tasks with *Nitro* in the above four-step manner until the DRL agent achieves the target reward or runs out of monetary budget.

5 *NITRO*'S DESIGN

5.1 Hessian-Based Neural Network Analysis

For a DRL policy neural network model θ, the first derivative of the surrogate objective $L(\theta)$ *w.r.t.* the model parameters is the gradient $g_\theta \in \mathbb{R}$ (definitions in §6.1), used for backward propagation [60]. Differentiating the gradient g_θ yields a square matrix, $H := \nabla^2_\theta L(\theta) = \nabla_\theta g_\theta \in \mathbb{R}$, commonly known as the *Hessian* matrix. The Hessian matrix is a second-order derivative that effectively captures important properties of the deep neural network objective landscape [64, 73]. Existing works have widely employed the Hessian matrix H to analyze the objective surface curvature of neural networks [15, 29, 32, 36, 52, 66] and guide neural network training, *e.g.*, natural gradients [3, 30, 46] and Hessian-based policy gradients [24, 64]. We also leverage Hessian information to design a boosting score to capture boosting opportunities.

5.2 Boosting Opportunity Detection

We design the *Boosting Detector* deployed on *Nitro*'s learner server (Fig. 6) to inspect and analyze the policy model weights to detect opportunities for DRL boosting.

Nitro estimates the eigenvalues of the Hessian matrix to inspect the policy network to measure the local curvature. The eigenvalues of Hessian $\lambda := \{\lambda_1, \ldots, \lambda_n\}$ are proved to indicate the curvature of the objective surface [36]. However, directly forming the full eigenvalues of Hessian can be computationally expensive [73]. Since the objective optimization of neural networks is typically dominated by the eigenvalues of the largest magnitude [36], this inspires us to look for the min-max ratio $R^{concave}$ of Hessian eigenvalues:

$$R^{concave} := -\frac{\max(0, \lambda_{\min})}{\lambda_{\max}}, \tag{1}$$

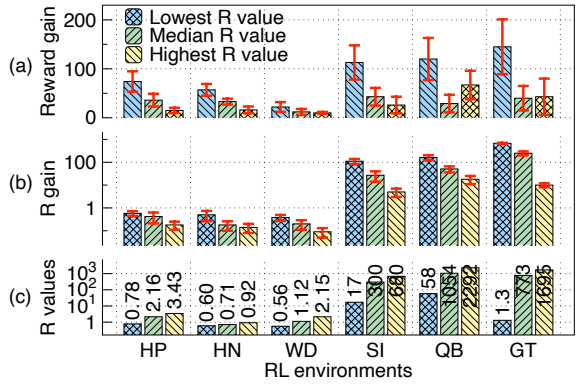


Figure 7: PPO reward gains after boosting at three different rounds in six environments (HP: Hopper, HN: Humanoid, WD: Walker2d, SI: SpaceInvaders, QB: Qbert, GT: Gravitar).

where $\lambda_{\min}$ and $\lambda_{\max}$ are minimum and maximum eigenvalues, respectively. The metric $R^{concave}$ roughly measures how concave the local surface of the objective is.

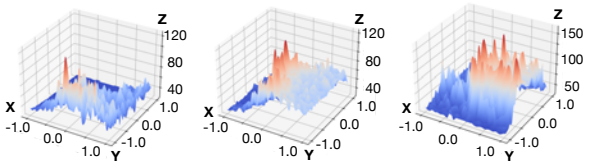
Since DRL studies a reward maximization problem (as opposed to loss minimization in other learning settings), higher convexity in the reward surface is better, corresponding to higher rewards. Hence, we invert Eq. 1 to measure the ratio R for convexity:

$$R := -\frac{\lambda_{max}}{\lambda_{min}}. \tag{2}$$

The intuition behind Eq. 2 is to measure the ratio to approximately quantify the convexity of the objective surface. Existing works have verified the importance and effectiveness of convexity measurement on DRL's surrogate objectives. For example, Li et al. [36] explains the trainability of neural networks by quantifying the (non)convexity of loss, Kakade and Langford [29] detects the mismatch between true rewards and surrogate objectives of DRL by characterizing convexity, and Sullivan et al. [66] explores the performance cliffs in DRL training by measuring convexity.

We then conduct an experiment to verify whether R can be an effective indicator of boosting opportunities in DRL training. The experimental setting is the same as in §2.2. We train PPO in the Hopper environment with ten rounds, where one actor is launched per round to sample 256 timesteps. We select three training rounds out of ten with the lowest, median, and highest R values, respectively. Each round is boosted with 16 actors to sample 4,096 timesteps. Fig. 7 shows the reward gains of boosting the selected three rounds. We define the reward gain as the difference between the reward after and before boosting, likewise R gain. The results in Fig. 7(a) indicate that boosting rounds with a lower R can achieve more benefits (*i.e.*, reward gain) under the same number of actors. The R value of each round also increases after boosting (Fig. 7(b)), indicating the surrogate objective can be improved by boosting to discover higher rewards.

Fig. 8 depicts the 3D reward surfaces of the surrogate objective for the lowest, median, and highest R values in the Hopper environment from Fig. 7. With lower R values, the reward surface for the DRL neural network to explore is poorer (less high-reward regions) due to less diverse trajectories, which indicates that boosting at low-R points can potentially achieve higher benefits.



(a) Lowest R (0.78). (b) Median R (2.16). (c) Highest R (3.43).

Figure 8: 3D landscapes of the DRL surrogate objective for lowest, median, and highest R values in Hopper environment from Fig. 7, respectively. X and Y axes show the two-dimensional perturbations on neural network parameters, and the Z axis shows the episodic rewards. Red and blue areas represent high and low rewards, respectively.

5.3 Cost-aware Actor Scaling

We have demonstrated that measuring the convexity of the surrogate objective (R in Eq. 2) can be an effective metric to guide boosting. However, given a limited training budget, *Nitro* cannot assume every boosting opportunity to have maximum actor-parallelism. We must design a cost-aware algorithm for *Nitro* to manage the boost budget while preserving training efficiency.

Since DRL is a complex and fairly stochastic learning process that can be affected by numerous factors (*e.g.*, hyperparameters, network initialization, *etc.*), estimating boosting benefits (*i.e.*, episodic reward increase) of a given boosting scheme ahead of each round is hardly possible [69]. Thus, precisely determining the budget for boosting before each round is infeasible. Therefore, *Nitro* employs a heuristic to determine the number of actor functions per round to achieve cost-efficient training. We design the algorithm based on a key observation: *boosting efficiency decreases through training, suggesting spending more budget in the early training rounds.*

Fig. 9 shows the reward gains and boost efficiency through 50-round training for PPO in the Hopper environment. We sample five points with the lowest R, each from every ten training rounds. Each point is boosted with 4×, 8×, 16×, and 32× budget compared to no boosting, respectively. Boosting efficiency is defined as the achieved reward gain over the spent boosting budget, normalized to $[0, 1]$ in Fig. 9. While learning to accumulate rewards is relatively simple in the early training rounds, finding optimal strategies in the late rounds becomes difficult. As a result, the reward gain and boosting efficiency gradually decrease through training. This observation suggests that *spending the same budget in early rounds is more efficient than reserving for the late rounds.* Existing studies [20, 76] on distributed ML training also discover similar trends, *i.e.*, spending more budgets in the early rounds instead of later ones, which is aligned with our empirical findings.

We use a sliding window of size n to monitor and record the convexity $W_n := \{R_{k-n+1}, \ldots, R_k\}$ in Eq. 2 for the past n rounds. The boosting score S_k is estimated using min-max normalization over the window W_n. To incorporate the observations, we embed an *exponential decay* factor $d \in (0, 1)$ to the convexity metric R, which anneals exponentially to the round number k:

$$S_k := \frac{R_{max} - R_k}{R_{max} - R_{min}} \times d^k,$$

where R_{min} and R_{max} are the minimum and maximum values from the sliding window W_n, respectively. Hence, the score always falls within $[0, 1]$, where a score close to 1 indicates a potentially high boosting benefit and vice versa. Naturally, the number of actors I_k to scale at round k is computed proportionally to the boosting score S_k given by

$$I_k := \text{Clip}\Big(I_{max} \times S_k, I_{min}, I_{max}\Big),$$

where I_{min} and I_{max} are the minimum and maximum number of actors we can launch per round. The clip function ensures that the number of actors per round is always within a certain range, which is adjustable based on the training budget.

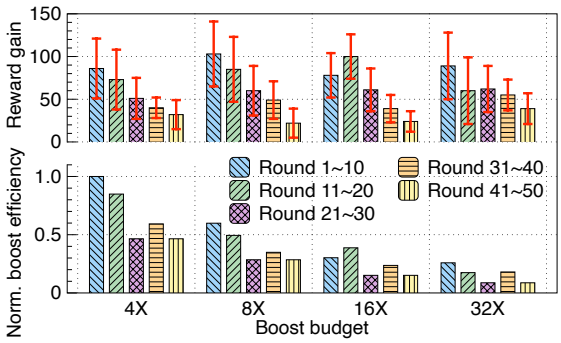


Figure 9: Reward gain and boosting efficiency across different rounds when training PPO in the Hopper environment.

6 *NITRO*'S ANALYSIS

In this section, we present the theoretical analysis to justify the performance gain of *Nitro*'s boosting.

6.1 *Nitro*'s Performance gain

Theorem 1. *Define the surrogate objective*

$$L(\theta) := \mathbb{E}_t\big[log\pi_\theta(a_t|s_t)A_t(s_t|a_t)\big],$$

where (s_t, a_t) is the state-action pair at timestep t and the advantage function $A_t(s_t|a_t) := Q_t(s_t|a_t) - V_t(s_t)$ guides the update of policy π parameterized by θ [62]. The gradient g_θ for updating policy π_θ is differentiated from the objective as follows

$$g_\theta := \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{(s_t,a_t)\in\tau} \nabla_\theta \log \pi_\theta(a_t|s_t)A_t(s_t|a_t)\right],$$

where τ denotes the trajectories containing state-action pairs (s_t, a_t) generated by π_θ.

Intuitively, the gradient g_θ is pushed towards the direction that increases the probability of taking action a_t if the advantage feedback $A_t(s_t|a_t)$ from a_t is positive, meaning that the action a_t yields a higher reward above the average performance $V_t(s_t)$ at state s_t. Otherwise, g_θ is forced to go the opposite direction to avoid a_t if a_t leads to a reward below the average (*i.e.*, $A_t(s_t|a_t)$ is negative).

Recall that we detect boosting opportunities by measuring the convexity metric R (Eq. 2). If the surrogate objective $L(\theta)$ (Theorem 1) has low R values, then its surface is relatively flat [36], indicating the policy π_θ has a less diverse advantage feedback A_t. In this case, the gradient g_θ may struggle to guide what actions lead to good or bad rewards, potentially resulting in policy updates in the wrong direction and causing the optimization process to

become trapped in local optima [29]. Existing works also reveal that limited trajectories can hinder the surrogate objective from approximating the reward distribution [29, 66]. Thus, *Nitro* boosts at training rounds with low R by diversifying the trajectories (*i.e.*, state-action pairs (s_t, a_t)), thus improving the policy update process and training performance.

6.2 *Nitro*'s Complexity

The time complexity of *Nitro* is mainly dominated by the computation of the metric R in Eq. 2, which requires estimating the maximum and minimum eigenvalues of the Hessian matrix of a neural network (§7). This section analyzes the complexity of R computation in *Nitro*.

Recall that g_θ is the gradient of the objective *w.r.t.* a neural network and H denotes its Hessian matrix. Explicitly forming the Hessian matrix by differentiating from g_θ can be computationally costly. Instead, existing methods compute the Hessian-vector product Hv for inspecting traces or eigenvalues:

$$Hv = \nabla_\theta(g_\theta^T v) = \nabla_\theta(g_\theta^T)v + g_\theta^T \nabla_\theta v,$$

where T represents the matrix transpose operation and v is a normalized random vector drawn from the Gaussian distribution. While we can compute Hv using arbitrary random vectors, the Hessian approximation quality depends on the choice of v. Gaussian distributions are proved to generate random vectors with the least computation variances [73]. Therefore, we follow this routine to compute Hv using the chain rule of derivatives. Importantly, Hv has been proved to be easy-to-compute, where the cost of computing Hv is the same as one backward propagation of the gradient g_θ [73]. Additionally, instead of computing Hv using all trajectories, we compute Hv with a subset of trajectories sampled from *Nitro*'s trajectory cache, further reducing the complexity. Therefore, the computational complexity of *Nitro* is negligible to the DRL training complexity itself. We further show that both Hessian matrix approximation and using trajectory subset are necessary for reducing computation overheads in §8.7.

6.3 *Nitro*'s Robustness

We prove that the performance gain of *Nitro* is robust by holding a lower bound on monotonic reward improvement, which guarantees policy update performance during training. Recall that $J(\pi)$ denotes the cumulative rewards achieved by rolling out policy π.

Theorem 2. *When Nitro's policy π updates to a new one π', the following reward improvement lower bound holds:*

$$J(\pi') - J(\pi) \geq -\frac{\gamma \epsilon^{\pi'} \sqrt{2\log(1+\rho)}}{(1-\gamma)^2},$$

where the constant $\epsilon^\pi \doteq \max_s |\mathbb{E}_{a\sim\pi}[A^\mu]|$, γ is the discount factor, and ρ is the surrogate clip threshold [37, 43, 63], respectively.

We refer to the Corollary 1 from [2]. First, we replace Total Variation (TV) divergence with Kullback–Leibler (KL) divergence using Pinsker's inequality [14]. Then, we apply Jensen's inequality [14] on the log function with ρ to reach the form of Theorem 2.

Table 1: Hyperparameters of PPO and IMPACT in evaluation.

Parameter	PPO	IMPACT
Learning rate	0.00005	0.0005
Discount factor (γ)	0.99	0.99
Batch size (Mujoco)	4096	4096
Batch size (Atari)	256	256
Clip parameter	0.3	0.4
KL coefficient	0.2	1.0
KL target	0.01	0.01
Entropy coefficient	0.0	0.01
Value function coefficient	1.0	1.0
Target update frequency	N/A	1.0

7 IMPLEMENTATION

Nitro is designed to be a generic training engine for boosting distributed DRL training with a hybrid serverful and serverless computing solution. For concreteness, we describe its implementation in the context of a combination of AWS services, including AWS EC2 [4] and AWS Lambda [8]. We implement *Nitro* with 3K lines of Python, which will be open-sourced after review. We describe the detailed implementation of *Nitro*'s components and features below.

Boosting Detector. We employ the PyHessian library [73] and use a stochastic Lanczos method to estimate the eigenvalues of Hessian. PyHessian only requires Hessian-vector products for Hessian estimation, which can be calculated directly with PyTorch's automatic differentiation [53]. The computation overhead of the Boosting Detector is trivial compared to *Nitro*'s training time (§8.7).

Learner. We use AWS EC2 instances with GPU accelerators to host the learner. The learner's core logic is implemented by PyTorch [53], including the policy model' neural networks and learning process.

Serverless Actor. We employ AWS Lambda to implement lightweight serverless actor functions. The dependencies of the actor function are installed and packaged as a Docker container image [48]. We store and manage the actor container image in AWS Elastic Container Registry [6] for fast function deployment. Before training DRL tasks, *Nitro* profiles information about the execution time and resource demand of actor functions with the task to run. The profiled results are used to determine the optimal function memory size when deploying actor functions on AWS Lambda. We use AWS Lambda's built-in pre-warming services, provisioned concurrency [9], to further reduce the function startup overhead.

Actor Scaler. We implement the actor scaler in Python with AWS Python SDK Boto3 [7], and use Python's built-in multiprocessing library to invoke actor functions on AWS Lambda concurrently.

Trajectory Cache. We use Redis [58], an in-memory key-value cache, to implement the trajectory cache in *Nitro*. Existing DRL frameworks generally store the trajectory data in a key-value manner (*i.e.*, dictionary in Python). We also follow the key-value data structure of existing DRL frameworks to cache trajectories, which makes key-value databases such as Redis an appropriate choice for the cache. The Redis instance resides on the learner server and provides high-performant communications between the learner and actor functions. Upon completing sampling, actor functions serialize the trajectories using Pickle [54] and submit the serialized sample batch to Redis. Depending on the DRL algorithm, *Nitro*'s learner performs either asynchronous policy update by periodically polling Redis or waits for all actors to perform synchronous update.

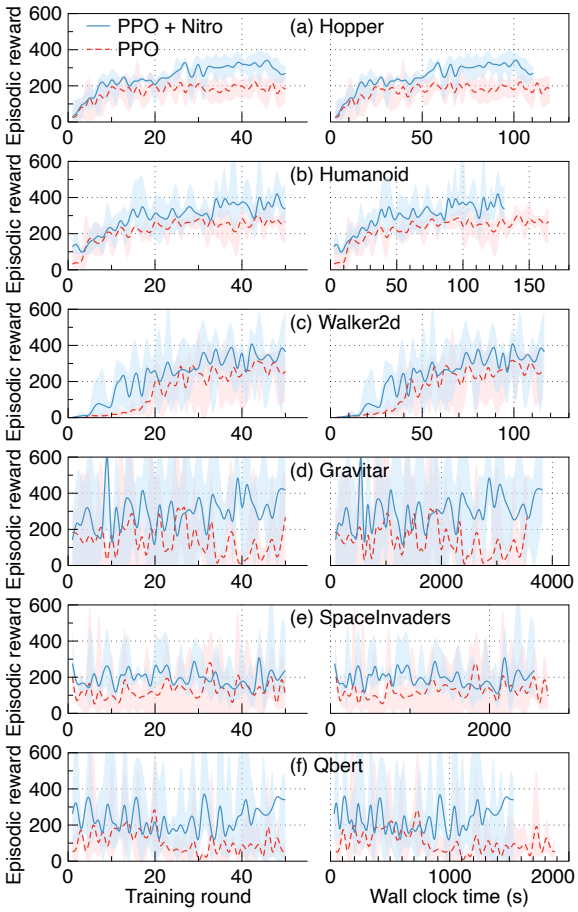


Figure 10: *Nitro* improves PPO training efficiency.

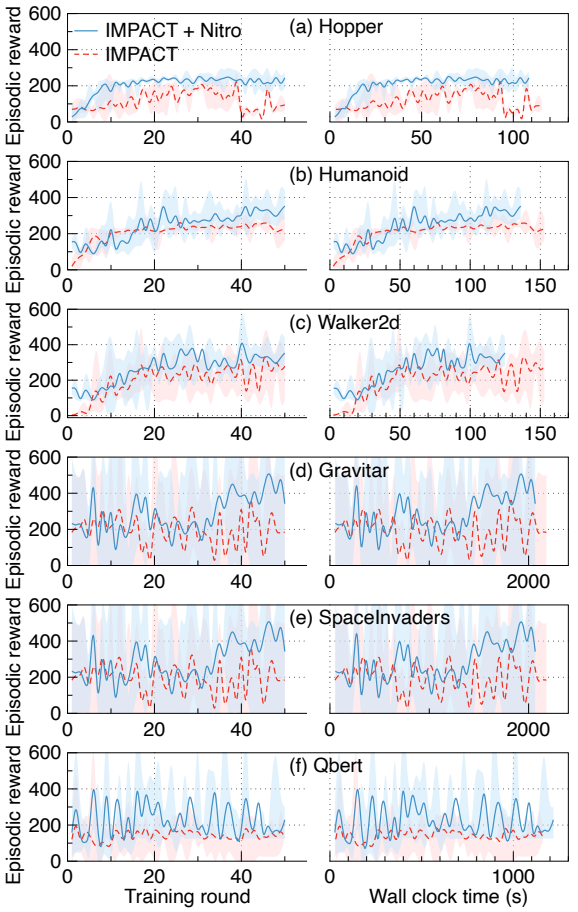


Figure 11: *Nitro* improves IMPACT training efficiency.

8 EVALUATION

This section conducts extensive experiments to evaluate *Nitro*, including overall performance against SOTA baselines (§8.2), effectiveness of actor scaling (§8.3), ablation study (§8.5), scalability (§8.6), sensitivity analysis (§8.4), and breakdowns and overheads of *Nitro* (§8.7). *Nitro* improves the final reward (*i.e.*, training quality) by up to 6× and reduces training cost by up to 42% than baselines.

8.1 Experimental Setup

Testbeds. We deploy all serverful baselines to a cluster of AWS EC2 VMs: one `p3.2xlarge` and one `c5.4xlarge`. The cluster contains one NVIDIA V100 GPU and 16 Intel Xeon Platinum CPUs for training DRL tasks. For *Nitro*, we use the same `p3.2xlarge` instance to host the learner while deploying each actor as a function on AWS Lambda with 1,024 MB memory.[2] The learner VM has 16 GB GPU memory, 60 GB CPU memory, and up to 10 Gigabit bandwidth.

Workloads. Six popular environments from OpenAI Gym are used to evaluate *Nitro* and SOTA baselines, including three continuous-action MuJoCo environments (Hopper, Humanoid, and Walker2d) and three discrete-action Atari environments (SpaceInvaders, Qbert, and Gravitar). For three MuJoCo environments, the policy network consists of two fully-connected layers of 256 hidden units with `Tanh` activation. For three Atari environments, the policy network consists of three convolutional layers of 8×8, 4×4, and 11×11 kernel sizes with `ReLU` activation, respectively. The input sampled from Atari games is a stack of three 84×84 images. In both cases, the critic networks share the same architecture as the policy networks.

***Nitro*'s setting.** We limit the actor allocation range of *Nitro* within [8, 64] during every DRL training round, *i.e.*, $I_{min} = 8$ and $I_{max} = 64$. We set AWS Lambda's provisioned concurrency for *Nitro* to be 64, the same as I_{max} to eliminate cold-starts.[3] Considering the evaluation deployment's cost-efficiency, we choose to verify *Nitro*'s effectiveness at the scale of 64 actors for most experiments. The evaluation results demonstrate that *Nitro* effectively outperforms SOTA baselines. Additionally, we also evaluate *Nitro*'s actor function scalability and large-scale deployment in §8.6, showing that larger scales further accelerate DRL training. We allocate 1,024 MB memory to each serverless actor function and set the timeout limit to 60 seconds. The exponential decay factor d is set to 0.96 when evaluating *Nitro*. The sliding window size n is set to six rounds. We further evaluate the sensitivity of three parameters in §8.4.

[2]We use AWS services in the US East 2 region. The hourly unit prices for `p3.2xlarge` and `c5.4xlarge` are $3.06 and $0.68, respectively. The AWS Lambda function invocation fee is $0.0000166667 per GB-second.

[3]The AWS Lambda provisioned concurrency fee is $0.0000041667 + $0.0000097222 for every GB-second in US East 2 region.

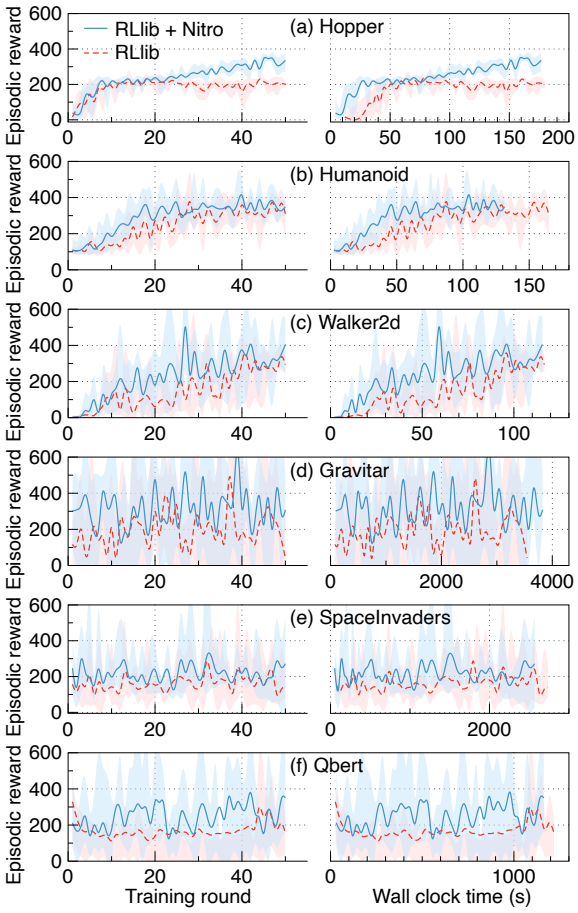


Figure 12: *Nitro* improves RLlib training efficiency.

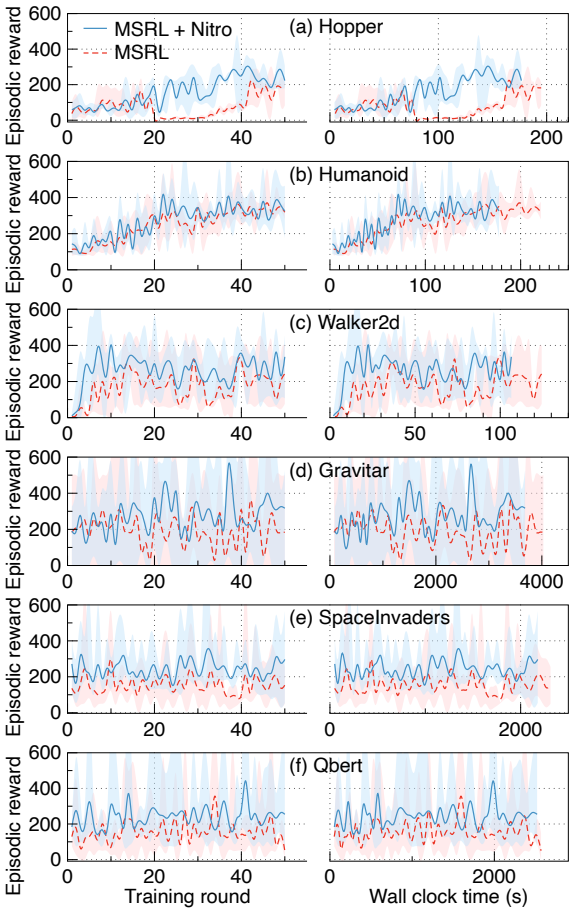


Figure 13: *Nitro* improves MSRL training efficiency.

8.2 Overall Performance

8.2.1 Integrating with DRL Algorithms. We evaluate how *Nitro* boosts SOTA DRL algorithms. Specifically, we integrate *Nitro* with two algorithms, one on-policy and one off-policy: 1) **PPO** [63] is the most famous on-policy DRL algorithm that has been employed in extensive applications [10, 13, 55, 56]. We implement a standard distributed PPO with Generalized Advantage Estimation (GAE) [62] and surrogate objective clipping [63]. 2) **IMPACT** [43] is a SOTA off-policy algorithm. IMPACT itself builds on a long list of improvements over PPO and combines various tricks for asynchronous training, such as V-trace importance sampling [17] and the surrogate target network [41]. We use PPO and IMPACT as baselines and integrate them with *Nitro* for comparison. Table 1 describes the hyperparameter settings of PPO and IMPACT used in the evaluation. We employ the same hyperparameter settings from tuned examples in RLlib [39]. Both PPO and IMPACT training use Adam optimizer [33]. We train each algorithm for 50 rounds in six environments. The results are averaged over ten repeated experiments, each with a different random seed.

Training efficiency. Figs. 10 and 11 show the episodic rewards through training in six environments for PPO and IMPACT, respectively. IMPACT completes training faster than PPO in most environments due to the advantage of being off-policy. *Nitro* outperforms PPO and IMPACT by training faster in statistical efficiency and wall clock time. *Nitro* improves the final reward by up to 5× and 3× than PPO and IMPACT, respectively.

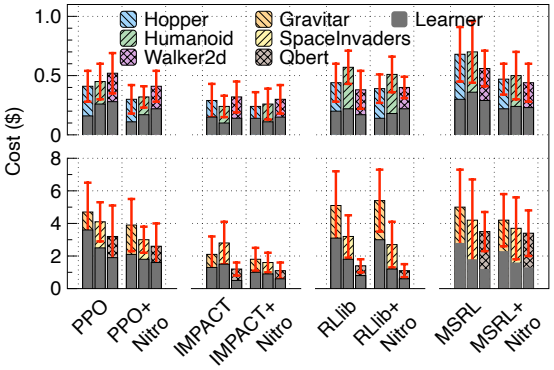


Figure 14: *Nitro* reduces training costs of PPO, IMPACT, RLlib, and MSRL. Grey bars represent the time spent on the learner, and the rest indicate the time spent on actors.

Training cost. Fig. 14 shows the training costs of the two baselines and variants integrated with *Nitro*. *Nitro* reduces training costs by up to 29% and 42% than the PPO and IMPACT, respectively.

8.2.2 Integrating with DRL Frameworks. We also evaluate how *Nitro* improves SOTA DRL frameworks. Two popular DRL frameworks are integrated with *Nitro* in the evaluation: 1) **Ray RLlib** [39]

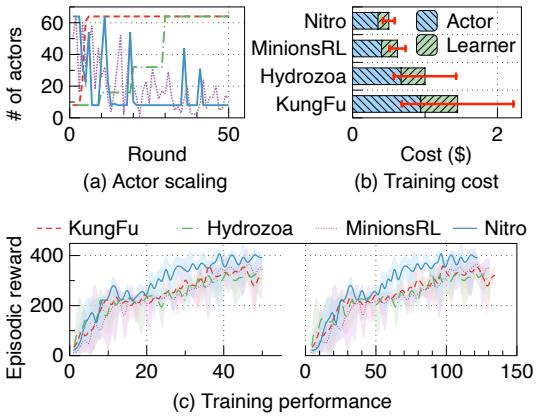


Figure 15: *Nitro* reduces training cost while boosting.

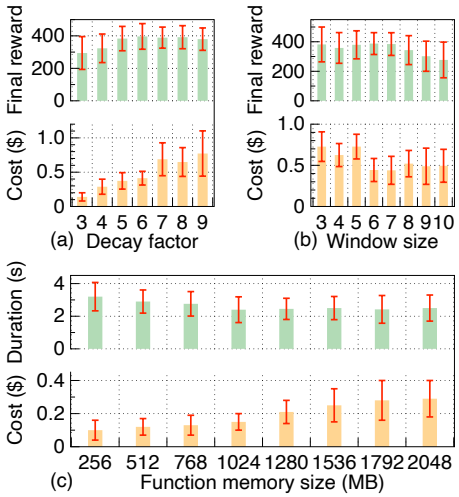


Figure 16: Sensitivity analysis in the Hopper environment.

is an open-source industrial-grade RL library with a comprehensive implementation of algorithms. We replace RLlib's remote RolloutWorkers with serverless actor functions for integration. The original states of RLlib's actors are transformed into input/output data of functions and stored in the distributed cache. 2) **MSRL** [77], the default RL library developed for MindSpore [26], is a distributed RL training system that uses dataflow graphs to execute DRL algorithms. Similarly, we replace the original Workers defined in MSRL with serverless actor functions to execute the algorithm fragments. We run PPO with two frameworks in six environments using the same experimental setting in §8.2.1.

Training efficiency. Figs. 12 and 13 show the episodic rewards through training in six environments for RLlib and MSRL, respectively. In both frameworks, we observe similar improvements with *Nitro*. *Nitro* accelerates PPO training in two frameworks by improving both statistical and training efficiency. *Nitro* improves the final reward by up to 6× and 5× than RLlib and MSRL, respectively.

Training cost. Fig. 14 shows the training costs of the two frameworks and variants integrated with *Nitro*. *Nitro* reduces training costs by up to 21% and 30% than RLlib and MSRL, respectively.

8.3 Actor Scaling

We compare *Nitro*'s actor scaling algorithm against three SOTA dynamic worker scaling schemes for distributed ML and DRL training: 1) **KungFu** [44] employs a dynamic worker scaling algorithm based on gradient noise scale (GNS) for serverful ML training, which only scales up the number of workers proportional to the increase of GNS and never scales down. For a fair comparison, we extend KungFu to a serverless environment by replacing KungFu's workers with serverless functions. 2) **Hydrozoa** [19] doubles the number of serverless workers with a fixed schedule rate (*e.g.*, every ten rounds), which serves as a heuristic baseline for actor scaling. 3) **MinionsRL** [74] employs a DRL-based actor scheduler to dynamically scale serverless actors, which tries to solve the scheduling problem via black-box optimization. We train PPO in the Hopper environment for each scheme with 50 rounds, the same as in §8.2.

Fig. 15(a) depicts how the four schemes scale actors through training. KungFu and Hydrozoa ignore the budget and quickly increase the number of concurrent actors to the maximum, while *Nitro* performs cost-aware boosting via careful actor scaling. Interestingly, MinionsRL also tends to increase the number of actors at early rounds and scale down gradually, which aligns with our observations in §5.3. However, MinionsRL fails to capture the boosting opportunities due to its black-box nature, thus degrading the training performance. Fig. 15(b) shows four schemes' training costs. KungFu and Hydrozoa are not designed for distributed DRL training and ignore training costs, thus incurring significantly higher costs than *Nitro*. While MinionsRL's training cost is less than KungFu and Hydrozoa, its expensive pre-training cost [74] is omitted, yet still incurring higher costs than *Nitro*. As MinionsRL fails to capture boosting opportunities, it scales actors in non-boostable rounds and wastes more budgets. Fig. 15(c) shows the statistical training efficiency and wall clock time of KungFu, Hydrozoa, MinionsRL, and *Nitro*, respectively. *Nitro* achieves higher rewards over the three schemes at minimal costs.

8.4 Sensitivity Analysis

We analyze the sensitivity of three parameters in *Nitro*: decay factor d, sliding window size n, and actor function memory size. We run the same experiment in §8.2, *i.e.*, training PPO in the Hopper environment, but with different parameter values for analysis. The results are reported in Fig. 16. Other combinations of algorithms and environments show similar sensitivity results.

Decay factor d. We set the decay factor to 0.96 in the evaluation. Fig. 16(a) shows the achieved final reward and training cost when gradually increasing the factor from 0.93 to 0.99 in the step of 0.01. When the factor increases, both the final reward and cost increase because *Nitro* allows spending more budget. The final reward stops growing at 0.96 while the cost is still increasing.

Sliding window size n. In our evaluation, we set the sliding window size to 6. Fig. 16(b) shows the final reward and training cost when gradually increasing the window size from 3 to 10. Increasing the window size allows *Nitro* to track longer historical information, thus making the boosting score computation more conservative. Hence, with a larger window size, *Nitro* tends to launch fewer actors when boosting, leading to lower costs but worse final rewards. We

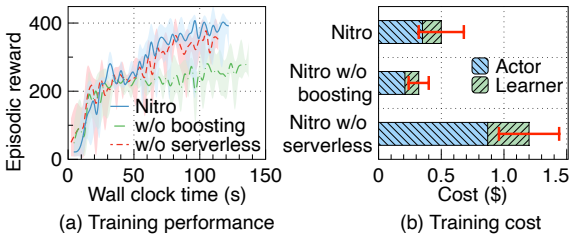


Figure 17: Ablation study of *Nitro*.

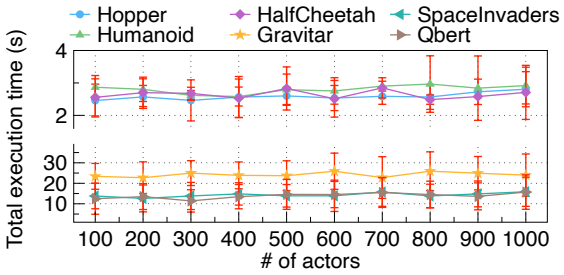


Figure 18: Scalability of *Nitro* with respect to the number of concurrent actors in six environments.

observe that *Nitro* achieves minimal training costs while preserving high performance with a size of 6.

Actor function memory size. Fig. 16(c) shows the actor function duration and total function cost with different memory sizes. The actor function duration stops decreasing after the memory size is larger than 1024 MB, while the total function cost continuously grows. We conclude that *Nitro* 's latency in the Hopper environment is not sensitive after reaching a certain function memory size. We plan to verify the sensitivity of other environments in the revision.

8.5 Ablation Study

To verify the effectiveness of two key components: serverless computing and boosting, we compare *Nitro* with two variants of itself: 1) ***Nitro* w/o serverless**. We use a `c5.9xlarge` with 72 CPU cores (smallest VM size above 64 CPU cores) as the actor server to replace the serverless actors. 2) ***Nitro* w/o boosting**. This variant statically launches 16 serverless actors per round without any boosting. We train PPO in the Hopper environment using the same experimental setting described in §8.2. Fig. 17(a) and Fig. 17(b) report the episodic reward and training cost for three baselines, respectively. *Nitro* w/o boosting launches fewer serverless actors than *Nitro*, incurring minimal cost but suffering from performance degradation. *Nitro* w/o serverless achieves similar performance with *Nitro* yet suffers from excessive cost. The results demonstrate that both serverless computing and boosting are necessary to the design of *Nitro*.

8.6 Scalability

We evaluate the scalability of *Nitro*'s serverless actor functions and conduct large-scale testing on *Nitro*.

Actor function scalability. Fig. 18 shows the scalability of *Nitro* with AWS Lambda. We gradually increase the number of concurrent actors from 100 to 1,000. Here, we measure the total execution time of a group of actors, defined as the time starting from invoking actors concurrently until the trajectory cache has received all submissions, to characterize scalability. The total execution time

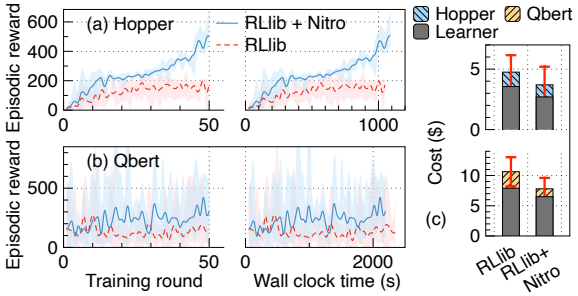


Figure 19: Large-scale testing of *Nitro* with RLlib running PPO in two environments.

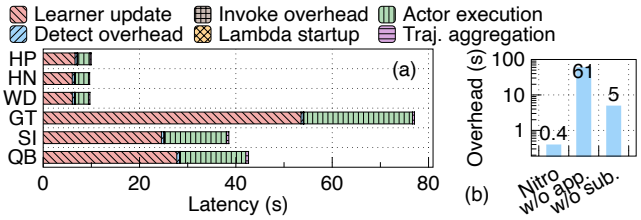


Figure 20: Latency breakdown, including (a) one-round training on six environments and (b) boosting score overhead.

of actors in all environments stays consistent when the number of concurrent actors increases, indicating that *Nitro* can scale to thousand-level serverless functions and achieve further benefits in large-scale training. Executing actors in Atari environments (Gravitar, SpaceInvaders, and Qbert) has a higher latency than Mujoco environments (Hopper, Humanoid, and Walker2d) because passing data of stacked frames takes more time.

Large-scale testing. We conducted a large-scale experiment with a serverful baseline (Ray RLlib) and *Nitro* using the PPO algorithm. The serverful Ray RLlib uses one `p3.8xlarge` VM with four NVIDIA V100 GPUs as the learner node and a `c5.24xlarge` VM with 96 CPU cores hosting actors. *Nitro* also uses the same `p3.8xlarge` VM with four V100 GPUs but can scale up to 128 serverless actor functions in AWS Lambda. This evaluation aims to test a larger-scale setting of the whole system, including more learners, actors, and trajectories with an upgrade of 4× more GPUs and 6× more CPUs. We use two environments, Hopper and Qbert, as examples in the experiment. Fig. 19 presents the episodic rewards through training and total cost. In a large-scale setting, *Nitro* still outperforms RLlib by improving the final reward by 2.4× and 2.5× while reducing training costs by 22% and 27%, for Hopper and Qbert, respectively. The exponentially increasing rewards indicate that *Nitro* can achieve more significant benefits in large-scale settings. While the rewards stopped growing due to reaching the budget limit, they will eventually converge with more training budgets.

8.7 Breakdowns and Overheads

We report the latency breakdowns of *Nitro*'s one-round training.

Latency and overhead breakdowns. Fig. 20(a) characterizes the latency breakdown of six environments used in evaluation with *Nitro*'s one-round training. We evaluate the overheads of *Nitro* with a maximum of 64 concurrent actors. The total overhead of all components incurs less than 5% delay, which is negligible for the one-round latency while providing boosted performance.

Boosting score computation overhead. Fig. 20(b) shows the boosting score overheads of three baselines when running on Hopper environment, including (i) *Nitro* with both Hessian matrix approximation and use of trajectory subset, (ii) *Nitro* without Hessian matrix approximation, and (iii) *Nitro* without using trajectory subset. Hessian approximation and using trajectory subset drastically reduces the boosting score computation time by 100× and 10×, respectively. Both techniques are critical to achieving negligible computation overheads.

9 RELATED WORK

DRL frameworks. Recently, a few open-source DRL training frameworks have been proposed. Acme [22] is a research-oriented DRL framework. Stable-Baselines3 [57] is developed for reliable DRL implementation. CleanRL [25] aims to provide high-quality single-file DRL implementations. SpinningUP [1] focuses on educational purposes for DRL beginners. RLlib [39] provides industry-grade DRL framework. MSRL [77] uses fragmented dataflow graphs to execute DRL algorithms. *Nitro* can be integrated with existing frameworks to enable DRL boosting. We demonstrate this capability by evaluating RLlib and MSRL integrated with *Nitro* (§8.2.2).

Distributed DRL training. A3C [50] firstly introduced a simple actor-leaner prototype. IMPALA [17] is the first off-policy (asynchronous) actor-learner architecture with V-trace correction. IMPACT [43] stabilized DRL training performance by adding a surrogate target network to the actor-learner architecture. SEED RL [16] aimed to accelerate off-policy actor-learner architectures by shifting actor inferences to centralized GPU servers. However, the above distributed RL solutions were all designed for serverful architectures, thus can hardly exploit the agile scalability and fine-grained resource provisioning of serverless computing. Unlike existing serverful solutions, *Nitro* fuses serverless computing into the actor-learner architecture to accelerate DRL training by boosting the number of concurrent actors instantly.

Actor scaling in serverful DRL. Many studies aim to scale out actors in distributed RL using serverful infrastructures. ActorQ [34] proposes scaling and accelerating DRL training according to the quantification of the inference on actors. SRL [47] scaled DRL training to industrial production environment by abstracting the components of actor-learner architectures. Parallel Q-Learning [38] massively parallelizes and optimizes GPU-based environment interactions on a single workstation. Existing solutions scale distributed DRL actors to fully utilize large-scale clusters and workstations. Instead, *Nitro* enables dynamic actor scaling with serverless computing to boost DRL training cost-efficiently.

Serverless ML training. Serverless computing has recently attracted the ML community to design novel training frameworks. Cirrus [11] proposes a serverless framework that simplifies end-to-end ML training. Siren [71] designs a DRL function scheduler to automate distributed ML training on serverless computing platforms. Jiang et al. [28] conducts a comprehensive comparison between server-based and serverless ML training, which indicates serverless training is cost-efficient. However, due to the lack of GPU support, the above serverless approaches cannot match the performance of server-based training [28]. Other works have attempted to enable GPUs for serverless training. Llama [59] proposes a serverless framework for auto-tuning video analytics pipelines, which supports GPUs in their backends. Hydrozoa [19] presents a deep neural network (DNN) training framework on top of Azure Container Instancess (ACIs) with dynamical data and model parallelism.

Serverless DRL training. Stellaris [75] proposes an asynchronous learning paradigm for DRL systems with serverless computing, which focuses on improving learner efficiency instead of actor boosting. MinionsRL [74] is the closest work to us, which also leverages serverless computing to design DRL training systems with dynamic actor scaling. However, MinionsRL falls short in three aspects. First, it is limited in synchronous training and only serves on-policy DRL algorithms, whereas our solution can accelerate both on-policy and off-policy algorithms. Second, it is a data-driven solution relying on a DRL-based scheduler and taking hours of pre-training and excessive pre-training expenses, whereas our work is a system solution that serves as a pluggable enhancement to existing DRL frameworks without any preparation overheads. Third, the DRL-based scheduler of MinionsRL cannot be well-justified since it applies a black-box optimization algorithm. In contrast, *Nitro* leverages provable metrics to explicitly capture boosting opportunities to accelerate DRL training with cost-efficiency. *Nitro* outperforms MinionsRL by training DRL algorithms faster with much less costs.

10 CONCLUSION

This paper proposed *Nitro*, a generic DRL training engine for actor-learner architectures that provides strategic boosting with serverless computing. In *Nitro*, actors are abstracted and packaged as lightweight serverless functions for agile scaling while the learner utilizes a GPU server for efficient training. With serverless functions, *Nitro* dynamically adjusts data sampling strategies according to the DRL training demands. To accurately and promptly seize the ephemeral boosting opportunities in real-time boosting, we design a boosting score to inspect the neural networks inside *Nitro*'s learner policy. Inspired by observations from realistic DRL tasks, we devise a cost-aware heuristic algorithm to guide *Nitro* to achieve minimal training cost. Various SOTA DRL algorithms and frameworks are integrated with *Nitro* to evaluate the effectiveness. Experiments on AWS EC2 and Lambda with Mujoco and Atari benchmarks show that *Nitro* improves the final reward (*i.e.*, training quality) by up to 6× and reduces training cost by up to 42%.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedback. The work of Hanfei Yu and Hao Wang was supported in part by NSF 2153502, 2403247, 2403398, and the AWS Cloud Credit for Research program. The work of Devesh Tiwari was supported by NSF 2124897. The work of Jian Li was supported in part by NSF 2148309 and 2337914. The work of Seung-Jong Park was supported in part by NSF 2403248 and 2403399. This work used JetStream2 at IU through allocation CIS220024 and CIS240498 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning. https://spinningup.openai.com.

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. 2017. Constrained Policy Optimization. In *International Conference on Machine Learning (ICML)*.

[3] Shun-Ichi Amari. 1998. Natural Gradient Works Efficiently in Learning. *Neural Computation* (1998).

[4] AWS. 2006. AWS EC2: Secure and Resizable Compute Capacity in the Cloud. https://aws.amazon.com/ec2/.

[5] AWS. 2014. AWS Lambda: Serverless Compute. https://aws.amazon.com/lambda/.

[6] AWS. 2015. Amazon Elastic Container Registry. https://aws.amazon.com/ecr/.

[7] AWS. 2015. AWS SDK for Python (Boto3). https://aws.amazon.com/sdk-for-python/.

[8] AWS. 2018. AWS Lambda: Serverless Compute. https://aws.amazon.com/lambda/.

[9] AWS. 2019. AWS Lambda: Configuring Provisioned Concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[10] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680* (2019).

[11] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *ACM Symposium on Cloud Computing (SoCC)*.

[12] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. Auto: Scaling Deep Reinforcement Learning for Datacenter-scale Automatic Traffic Optimization. In *2018 conference of the ACM special interest group on data communication (SIGCOMM)*.

[13] Xiangyu Chen, Zelin Ye, Jiankai Sun, Yuda Fan, Fang Hu, Chenxi Wang, and Cewu Lu. 2020. Transferable Active Grasping and Real Embodied Dataset. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*.

[14] Imre Csiszár and János Körner. 2011. *Information Theory: Coding Theorems for Discrete Memoryless Systems*. Cambridge University Press.

[15] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. 2017. Sharp Minima Can Generalize for Deep Nets. In *International Conference on Machine Learning (ICML)*.

[16] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2020. Seed RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *International Conference on Learning Representations (ICLR)*.

[17] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *International Conference on Machine Learning (ICML)*.

[18] Shixiang Shane Gu, Timothy Lillicrap, Richard E Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. 2017. Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning. *Advances in Neural Information Processing Systems (NIPS)* (2017).

[19] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. 2022. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. *Proceedings of Machine Learning and Systems (MLSys)* (2022).

[20] Wang Hao, Niu Di, and Li Baochun. 2019. Distributed Machine Learning with a Serverless Architecture. In *Proc. the IEEE Conference on Computer Communications (INFOCOM)*.

[21] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Thirty-second AAAI conference on artificial intelligence (AAAI)*.

[22] Matthew W Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, et al. 2020. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979* (2020).

[23] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *International Conference on Learning Representations (ICLR)*.

[24] Feihu Huang, Shangqian Gao, Jian Pei, and Heng Huang. 2020. Momentum-Based Policy Gradient Methods. In *International Conference on Machine Learning (ICML)*.

[25] Shengyi Huang, Rousslan Fernand JulienDossa Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João GM Araújo. 2022. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *The Journal of Machine Learning Research* (2022).

[26] Ltd. Huawei Technologies Co. 2022. Huawei MindSpore AI Development Framework. In *Artificial Intelligence Technology*.

[27] Jiawei Jiang, Shaoduo Gan, Bo Du, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, Sheng Wang, and Ce Zhang. 2023. A Systematic Evaluation of Machine Learning on Serverless Infrastructure. *The VLDB Journal* (2023).

[28] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *2021 International Conference on Management of Data (SIGMOD)*.

[29] Sham Kakade and John Langford. 2020. A Closer Look at Deep Policy Gradients. In *International Conference on Learning Representations (ICLR)*.

[30] Sham M Kakade. 2001. A Natural Policy Gradient. *Advances in Neural Information Processing Systems (NIPS)* (2001).

[31] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2018. Recurrent Experience Replay in Distributed Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*.

[32] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Learning Representations (ICLR)*.

[33] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[34] Maximilian Lam, Sharad Chitlangia, Srivatsan Krishnan, Zishen Wan, Gabriel Barth-Maron, Aleksandra Faust, and Vijay Janapa Reddi. 2021. ActorQ: Quantization for Actor-learner Distributed Reinforcement Learning. In *Hardware Aware Efficient Training Workshop at ICLR*.

[35] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS Games with Deep Reinforcement Learning. In *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*.

[36] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing the Loss Landscape of Neural Nets. *Advances in neural information processing systems (NIPS)* (2018).

[37] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *ACM Symposium on Cloud Computing (SoCC)*.

[38] Zechu Li, Tao Chen, Zhang-Wei Hong, Anurag Ajay, and Pulkit Agrawal. 2023. Parallel Q-Learning: Scaling Off-policy Reinforcement Learning under Massively Parallel Simulation. In *International Conference on Machine Learning (ICML)*.

[39] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[40] Xingxing Liang, Yang Ma, Yanghe Feng, and Zhong Liu. 2021. PTR-PPO: Proximal Policy Optimization with Prioritized Trajectory Replay. *arXiv preprint arXiv:2112.03798* (2021).

[41] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous Control with Deep Reinforcement Learning. *arXiv preprint arXiv:1509.02971* (2015).

[42] Zuxin Liu, Zhepeng Cen, Vladislav Isenbaev, Wei Liu, Steven Wu, Bo Li, and Ding Zhao. 2022. Constrained Variational Policy Optimization for Safe Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[43] Michael Luo, Jiahao Yao, Richard Liaw, Eric Liang, and Ion Stoica. 2020. IMPACT: Importance Weighted Asynchronous Architectures with Clipped Target Networks. In *International Conference on Learning Representations (ICLR)*.

[44] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[45] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[46] James Martens. 2020. New Insights and Perspectives on the Natural Gradient Method. *Journal of Machine Learning Research (JMLR)* (2020).

[47] Zhiyu Mei, Wei Fu, Guangju Wang, Huanchen Zhang, and Yi Wu. 2024. SRL: Scaling Distributed Reinforcement Learning to Over Ten Thousand Cores. *International Conference on Learning Representations (ICLR)* (2024).

[48] Dirk Merkel et al. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (2014).

[49] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[50] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[51] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1507.04296* (2015).

[52] Jack Parker-Holder, Luke Metz, Cinjon Resnick, Hengyuan Hu, Adam Lerer, Alistair Letcher, Alexander Peysakhovich, Aldo Pacchiano, and Jakob Foerster.

2020. Ridge rider: Finding Diverse Solutions by Following Eigenvectors of the Hessian. *Advances in Neural Information Processing Systems (NIPS)* (2020).

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NIPS)* (2019).

[54] Python. 2008. Pickle — Python Object Serialization. https://docs.python.org/3/library/pickle.html.

[55] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for {SLO-Oriented} Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[56] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI blog* (2019).

[57] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-baselines3: Reliable Reinforcement Learning Implementations. *The Journal of Machine Learning Research* (2021).

[58] Redis. 2009. Redis Official Website. http://redis.io/.

[59] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *ACM Symposium on Cloud Computing (SoCC)*.

[60] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning Representations by Back-propagating Errors. *Nature* (1986).

[61] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *International Conference on Machine Learning (ICML)*.

[62] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2016. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *International Conference on Learning Representations (ICLR)*.

[63] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[64] Zebang Shen, Alejandro Ribeiro, Hamed Hassani, Hui Qian, and Chao Mi. 2019. Hessian Aided Policy Gradient. In *International Conference on Machine Learning*. 5729–5738.

[65] 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. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* (2016).

[66] Ryan Sullivan, Justin K Terry, Benjamin Black, and John P Dickerson. 2022. Cliff Diving: Exploring Reward Surfaces in Reinforcement Learning Environments. In *Nineteenth International Conference on Machine Learning (ICML)*.

[67] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems (NIPS)* (1999).

[68] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A Physics Engine for Model-based Control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[69] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya Tolstikhin. 2020. Predicting Neural Network Accuracy from Weights. *arXiv preprint arXiv:2002.11448* (2020).

[70] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster Level in StarCraft II Using Multi-agent Reinforcement Learning. *Nature* (2019).

[71] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE 2019 Conference on Computer Communications (INFOCOM)*.

[72] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. 2019. DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames. *arXiv preprint arXiv:1911.00357* (2019).

[73] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W Mahoney. 2020. Pyhessian: Neural Networks Through the Lens of the Hessian. In *2020 IEEE international conference on big data (Big data)*.

[74] Hanfei Yu, Jian Li, Yang Hua, Xu Yuan, and Hao Wang. 2024. Cheaper and Faster: Distributed Deep Reinforcement Learning with Serverless Computing. In *Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI)*.

[75] Hanfei Yu, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. 2024. Stellaris: Staleness-Aware Distributed Reinforcement Learning with Serverless Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[76] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proc. the 2017 Symposium on Cloud Computing (SoCC)*. ACM.

[77] Huanzhou Zhu, Bo Zhao, Gang Chen, Weifeng Chen, Yijie Chen, Liang Shi, Yaodong Yang, Peter Pietzuch, and Lei Chen. 2023. MSRL: Distributed Reinforcement Learning with Dataflow Fragments. In *2023 USENIX Annual Technical Conference (ATC)*.

[78] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. 2017. Target-driven Visual Navigation in Indoor Scenes Using Deep Reinforcement Learning. In *IEEE International Conference on Robotics and Automation (ICRA 2017)*.