Proceedings of Machine Learning Research 133:361–395, 2021 NeurIPS 2020 Competition and Demonstration Track

Measuring Sample Efficiency and Generalization in Reinforcement Learning Benchmarks: NeurIPS 2020 Procgen Benchmark

Sharada Mohanty[a b,c] MOHANTY@AICROWD.COM
Jyotish Poonganam[a b] JYOTISH@AICROWD.COM
Adrien Gaidon[d e] ADRIEN.GAIDON@TRI.GLOBAL
Andrey Kolobov[d f] AKOLOBOV@MICROSOFT.COM
Blake Wulfe[d e] BLAKE.WULFE@TRI.GLOBAL
Dipam Chakraborty[d c] DIPAM@AICROWD.COM
Gravydas emetulskis[d g] GRAZVYDAS@THREETHIRDS.AI
Joo Schapke[d h] JOAOSCHAPKE@GMAIL.COM
Jonas Kubilius[d g] JONAS@THREETHIRDS.AI
Jurgis Paukonis[d g] JURGIS@THREETHIRDS.AI
Linas Klimas[d g] LINAS@THREETHIRDS.AI
Matthew Hausknecht[d f] MATTHEW.HAUSKNECHT@MICROSOFT.COM
Patrick MacAlpine[d f] PATMAC@GMAIL.COM
Quang Nhat Tran[d i] QUANGTRAN@TEMPLE.EDU
Thomas Tumiel[d j] TTUMIEL@GMAIL.COM
Xiaocheng Tang[d k] XIAOCHENGTANG@DIDIGLOBAL.COM
Xinwei Chen[d j] O.XLNWEL@OUTLOOK.COM
Christopher Hesse[l] CSH@OPENAI.COM
Jacob Hilton[l] JHILTON@OPENAI.COM
William Hebgen Guss[l] WGUSS@OPENAI.COM
Sahika Genc [m] SAHIKA@AMAZON.COM
John Schulman[l] JOSCHU@OPENAI.COM
Karl Cobbe [l] KARL@OPENAI.COM

Editors: Hugo Jair Escalante and Katja Hofmann

[a] These authors have equal contribution.
[b] AIcrowd
[c] AIcrowd Research
[d] The names of these authors are ordered alphabetically.
[e] Toyota Research Institute
[f] Microsoft Research
[g] Three Thirds
[h] Institute of Informatics, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
[i] Temple University
[j] AIcrowd Community
[k] DiDi Labs, Mountain View, CA
[l] OpenAI
[m] Amazon Web Services Artificial Intelligence

Abstract

The NeurIPS 2020 Procgen Competition was designed as a centralized benchmark with clearly defined tasks for measuring Sample Efficiency and Generalization in Reinforcement Learning. Generalization remains one of the most fundamental challenges in deep reinforcement learning, and yet we do not have enough benchmarks to measure the progress of the community on Generalization in Reinforcement Learning. We present the design of a centralized benchmark for Reinforcement Learning which can help measure Sample Efficiency and Generalization in Reinforcement Learning by doing end to end evaluation of the training and rollout phases of thousands of user submitted code bases in a scalable way. We designed the benchmark on top of the already existing Procgen Benchmark by defining clear tasks and standardizing the end to end evaluation setups. The design aims to maximize the flexibility available for researchers who wish to design future iterations of such benchmarks, and yet imposes necessary practical constraints to allow for a system like this to scale. This paper presents the competition setup and the details and analysis of the top solutions identified through this setup in context of 2020 iteration of the competition at NeurIPS.

Keywords: generalization in reinforcement learning, sample efficiency in reinforcement learning

1. Introduction

Procgen Benchmark is a collection of 16 procedurally generated environments designed to benchmark sample efficiency and generalization in reinforcement learning (Cobbe et al., 2020a). Since all content is procedurally generated, each Procgen environment intrinsically requires agents to generalize to never-before-seen situations. Critical elements like level difficulty, level layout, and in-game assets are randomized at the start of every episode. These environments therefore provide a robust test of an agents ability to learn in many diverse settings. By aggregating performance across so many diverse environments, Procgen Benchmark provides high quality metrics to judge RL algorithms. Furthermore, Procgen environments are easy to use[1] and the environments are computationally lightweight. Individuals with limited computational resources can easily reproduce baseline results and run new experiments, and this ability to iterate quickly can help accelerate research.

In this paper, we present the competition design and results of the NeurIPS 2020 Procgen Benchmark. The goal of this competition was to demonstrate the feasibility of using Procgen Benchmark to collectively measure progress on sample efficiency and generalization in Reinforcement Learning. Prior to our competition, reinforcement learning competitions have focused on the development of policies or meta-policies which perform well on a complex domain or across a select set of tasks (Kidziński et al., 2018; Nichol et al., 2018; Guss et al., 2019a). However, to the best of our knowledge, our competition is the first to directly isolate and focus on generalization in reinforcement learning across a broad set of procedural generated tasks. Additionally while the recent MineRL competition utilizes a procedurally generated environment, the competition is more restricted in its procedural generation engine than Procgen and focuses on algorithmic invariance to domain shift as opposed to true generalization across a task distribution (Guss et al., 2019a).

1. All environments are open-source and can be found at https://github.com/openai/procgen

2. Competition

2.1. Environments

This competition builds upon 16 procedurally generated environments which were publicly released as a part of the Procgen Benchmark (Cobbe et al., 2020a). 4 hold-out test environments were created for the evaluations of this competition and were used for the end to end evaluation of the submissions in the competition. Although other environments such as MineRL (Guss et al., 2019b), Malmo (Johnson et al., 2016), and Jelly Bean World (Platanios et al., 2020) make use of procedural generation, Procgen's novelty is in including many *diverse* procedurally generated environments. Further, the Arcade Learning Environment (Bellemare et al., 2013) is very widely used to judge RL algorithms, but it doesn't require agents to meaningfully generalize. This is a significant flaw as generalization is critical in many real world tasks, and it's important that RL benchmarks reflect this reality.

In all environments, procedural generation controls the selection of game assets and backgrounds, though some environments include a more diverse pool of assets and backgrounds than others. When procedural generation must place entities, it generally samples from the uniform distribution over valid locations, occasionally subject to game-specific constraints. Several environments use cellular automata (Johnson et al., 2010) to generate diverse level layouts.

2.2. Metrics

To compare submissions based on a single score across multiple Procgen environments, we calculate the mean normalized return. Following the original Procgen paper, we define the normalized return to be $R_{norm} = (R - R_{min})/(R_{max} - R_{min})$, where R is the raw expected return and R_{min} and R_{max} are constants chosen (per environment) to approximately bound R. As each of the Procgen environments have a clear score ceiling, it was possible to establish these constants. Using this definition, the normalized return is (almost) guaranteed to fall between 0 and 1. Since Procgen environments are designed to have similar difficulties, it's unlikely that a small subset of environments will dominate this signal. We use the mean normalized return since it offers a better signal than the median, and since we do not need to be robust to outliers.

The score computation in Round-1 uses a weighted metric on top of the mean normalized returns. When computing the cumulative round-1 score, same weight is provided to the normalized return from the evaluations of the single test environment as that is provided to the normalized return from the evaluations of all the 16 public environments. This is intentionally designed to incentivize participants to submit their training phase code while they experiment independently with the publicly released Procgen environments.

As mentioned in Section 2.3.1.2, considerations around sample-efficiency are imposed (when necessary) by limiting the total number of timesteps during the training phase to 8M timesteps. And considerations around generalizability are imposed (when necessary) by limiting the total number of levels an agent has access to during the training phase.

2.3. Tasks

The competition was divided into four separate rounds : Warm-Up Round, Round-1, Round-2, Final Exhaustive Evaluation. The evaluation of the submissions for this competition was done across two independent phases : **Training Phase** and **Rollout Phase**.

2.3.1. Training Phase and Rollout Phase

2.3.1.1. Rollout Phase

The rollout phase focuses on evaluating a trained model (a checkpoint) against a set of Procgen environments. The trained model used in this phase, was a carry-forward asset generated after the successful execution of the corresponding Training Phase (2.3.1.2). The environments in each evaluation set were drawn from either the 16 **publicly released environments** or the 4 **hold-out test environments** created specifically for this competition. Whenever the hold-out test environments were used for the evaluation, the corresponding Training Phase (2.3.1.2) had to be invoked by design - as the hold-out test environments were not accessible to participants in advance.

2.3.1.2. Training Phase

The training phase focuses on orchestrating the user submitted code to train against one of the Procgen environments. The training phase always generated a model checkpoint which was subsequently used in the corresponding rollout phase (2.3.1.1). The authors would like to re-iterate the fact that, whenever a hold-out test environment is used in the rollout phase, the corresponding training phase has to be invoked by design.

Two key things that are taken into consideration during the training phase are **sample efficiency** and **generalizability**.

As was shown by the MineRL Competition, limiting the number of environment steps in training yields resource- and sample-efficient submissions that perform well despite this limitation. We likewise address considerations around sample efficiency by limiting the number of timesteps allowed during the training phase to the same 8M timesteps (Guss et al., 2019a).

Considerations around generalizability are addressed by limiting the number of levels (of a particular Procgen environment) that the submitted solutions have access to during the training phase, to 200 Levels.

2.3.2. Rounds

2.3.2.1. Warm Up Round

The goal of the Warm Up Round was to encourage participants to explore the resources (environments, starter kit, tutorials, baselines) released as a part of the competition. This round only considered only the *coinrun* environment.

The submission repository structure (included in the starter kit) required the participants to include the code for their training phase and rollout phase. Pre-agreed entrypoints for both the phases were specified in the submission repository structure. Participants were not allowed to include any trained checkpoints in their submissions. Files larger than a

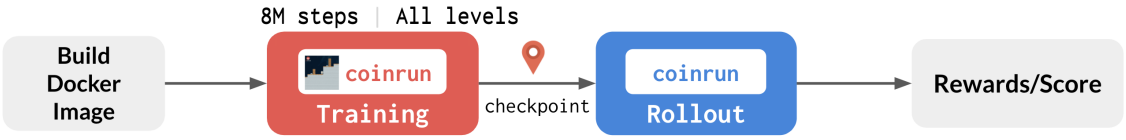

Figure 1: Evaluation workflow for the warm up round

threshold size were automatically scrubbed from the submission repository by the AIcrowd evaluators.

On receipt of the submissions, the AIcrowd evaluators orchestrate the submitted code for the training phase on the *coinrun* environment. Considerations for Sample Efficiency are imposed by limiting the available training timesteps to 8M timesteps. Considerations for Generalization were not taken into account in this round. After the successful execution of the training phase, the trained model was carry-forwarded to the corresponding rollout phase of the submitted code, which was evaluated on the *coinrun* environment on 1000 randomly sampled levels to aggregate the final scores. Figure 1 illustrates the evaluation workflow for Warm-Up Round.

2.3.2.2. Round-1

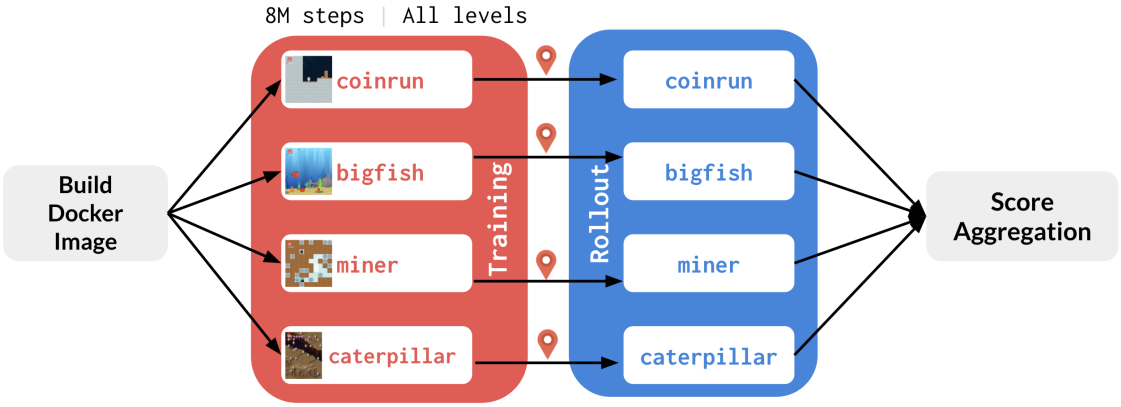

Figure 2: Evaluation workflow for the Round 1. Caterpillar is a hold-out test environment that participants did not have access to throughout the competition

Round-1 extends the problem setting of the Warm-Up Round by introducing parallel evaluations across multiple Procgen environments.

All the submissions were evaluated against 3 public Procgen environments (*coinrun*, *bigifsh*, *miner*) and one 1 hold-out test environment (*caterpillar*), which participants did not have access to throughout the duration of the competition.

The normalized score for a single submission-environment pair was computed as described in Section 2.2.

The cumulative score of each submission in this round was determined by the mean normalized score of the submission across all the 4 Procgen environments used in the Rollout phase. For reference, the normalized score for a single submission-environment pair was computed as described in Section 2.2.

$$Score = \frac{1}{6} \cdot R_{norm}^{coinrun} + \frac{1}{6} \cdot R_{norm}^{bigfish} + \frac{1}{6} \cdot R_{norm}^{miner} + \frac{1}{2} \cdot R_{norm}^{caterpillar}$$

Similar to Warm-Up Round, considerations for Sample Efficiency are imposed by limiting the available training timesteps to 8M timesteps. Considerations for Generalization were not taken into account in this round.

2.3.2.3. Round-2

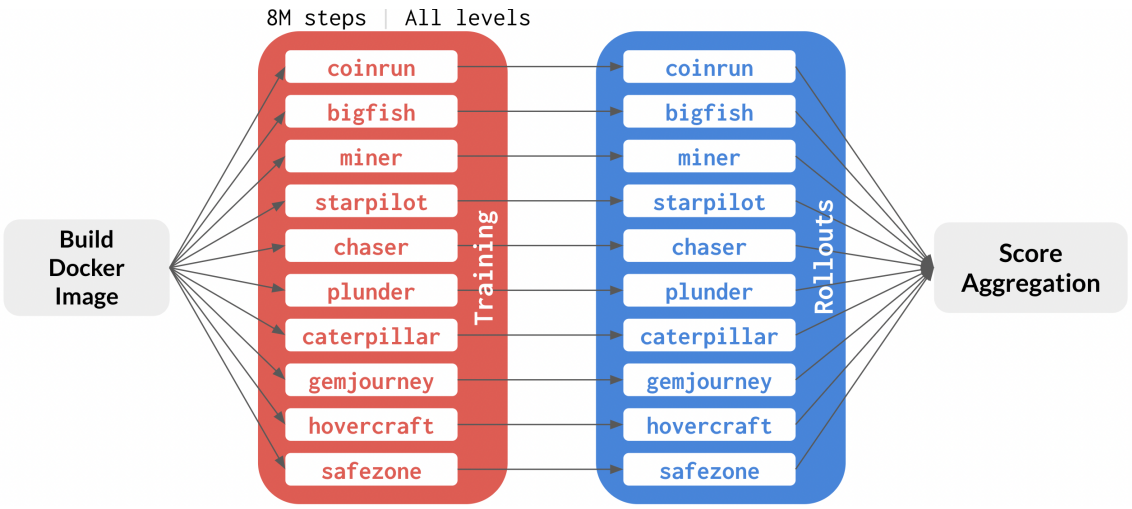

Figure 3: Evaluation setup for the Round 2. *caterpillar*, *gemjourney*, *hovercraft*, *safezone* are hold-out test environments that participants did not have access to throughout the competition.

Round-2 extends the problem setting as described in Round-1 by introducing 3 additional hold out test environments that participants did not have access to, throughout the competition. All the submissions in Round-2 were evaluated on 6 public environments (*coinrun*, *bigfish*, *miner*, *starpilot*, *chaser*, *plunder*) and 4 hold out test environments (*caterpillar*, *gemjourney*, *hovercraft*, *safezone*) for both the Training Phase and the Rollout Phase.

This round continued to impose Sample Efficiency considerations by limiting the training phase to 8M timesteps. Considerations for Generalization were introduced in this round, where the training phase was limited to 200 levels for each of the environments, while the trained models were evaluated on 1000 randomly sampled levels during the Rollout phase.

The cumulative score of each submission in this round was determined by the mean normalized score of the submissions in the rollout phase across 6 public environments (*coinrun*, *bigfish*, *miner*, *starpilot*, *chaser*, *plunder*) and 4 hold out test environments (*cater-*

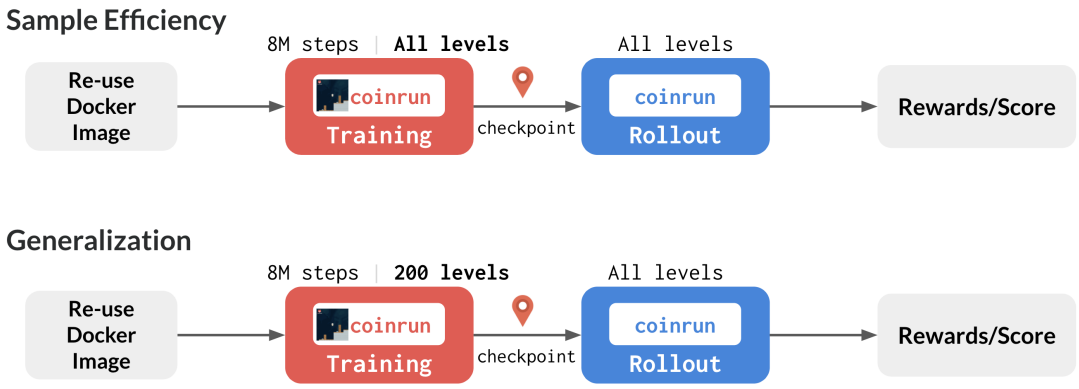

Figure 4: Evaluation workflow for sample efficiency and generalization tracks

pillar, *gemjourney*, *hovercraft*, *safezone*). For reference, the normalized score for a single submission-environment pair was computed as described in Section 2.2.

$$Score = \frac{1}{12} \cdot \sum_{Env}^{PublicEnvs} R_{norm}^{Env} + \frac{1}{8} \cdot \sum_{Env}^{Hold-outEnvs} R_{norm}^{Env}$$

2.3.2.4. Final Exhaustive Evaluation

The top-10 teams from Round-2 were subject to a final exhaustive evaluation. The final exhaustive evaluation is an extended and robust version of the problem formulation of Round-2.

The top-10 teams had the option to specify separate submissions for both the Sample Efficiency and Generalization tracks.

All eligible submissions were evaluated on all the 16 public environments and 4 hold out test environments for both the Training Phase and the Rollout Phase. The submissions were evaluated separately for Sample Efficiency and for Generalization. When measuring Generalization, the constraints of 8M timesteps from Sample Efficiency were implicitly added for operational reasons. Considerations around Generalization were imposed by limiting the number of levels during the training phase to 200 levels for each of the 20 procgen environments. During the rollouts phase the trained models for each environment were evaluated on 1000 randomly sampled levels.

All eligible submissions were evaluated over 3 trials for the Sample Efficiency Track and 3 trials for the Generalization Track. In the Sample Efficiency track, no limits on the number of levels were imposed during the training phase, while in the Generalization track limited the number of levels during the training phase to 200 levels.

The score for each trial was determined by the mean normalized score of the submissions in the rollout phase across all the 20 procgen environments.

$$score_{trial_n} = \frac{1}{20} \cdot \sum_{Env}^{All\ \ Envs} R_{norm}^{Env}$$

The final score for each of the tracks were determined by the maximum score across the three task specific trials for both the Sample Efficiency and the Generalization track.

$$score_{generalization} = \max_{1 \leq n \leq 3} \{score_{trail_n}^{generalization}\}$$

$$score_{sample\_efficiency} = \max_{1 \leq n \leq 3} \{score_{trail_n}^{sample\_efficiency}\}$$

2.4. Competition Statistics

Throughout the competition, a total of 545 individual participants, spread across 44 countries, registered for the competition. After round 1, 50 teams (83 individual participants) were eligible to participate in the round 2. Finally, 10 teams containing 18 individual participants qualified for the final exhaustive evaluation. Across the whole competition, we evaluated a total of 4805 submissions resulting in over 172,000 trained checkpoints throughout the competition and across different evaluation configurations.

3. Methods

The methods used by the top-10 teams can be broadly cateogorized into a set of "base algorithms". The list of "base algorithms" used by the participants is shown in Table 3. The competitors were limited to using a single NVIDIA V100 GPU and 2 hours of training for their models with 8M steps. We used the mean normalized return to compare submissions based on a single score across multiple Procgen environments. The mean normalized rewards across multiple rollouts per environment for the top ten submission for sample-efficiency and generalization tracks are shown in Figure 5 and Figure 6. In both tracks, the trends for the mean normalized rewards for rollouts were similar for *jumper, caveflyer, maze* and *fruitbot* for most of the top ten submissions while the final normalized rewards varied significantly in *leaper* and *dodgeball* for several competitors.

In the following, we provide the key implementation details and modifications for each of the top-10 teams. We grouped the participants' methods into three categories based on the base algorithm: 1) Phasic Policy Gradient (PPG) (Cobbe et al., 2020b), 2) Proximal Policy Optimization (PPO) (Schulman et al., 2017), and 3) other, including Policy-on Policy-off Policy Optimization (P3O) (Fakoor et al., 2020), Reactor (Gruslys et al., 2017), and Soft Actor-Critic (SAC) (Haarnoja et al., 2018).

3.1. Base Algorithm: Phasic Policy Gradient

The top two teams in generalization category used variations of PPG (Cobbe et al., 2020b). Their PPG-based methods also improved sample-efficiency, ranking both teams among the top three. PPG is an extension of PPO that, among other improvements, allows for greater sample reuse by (partially) separating policy and value-function learning into separate phases (Cobbe et al., 2020b).

3.1.1. Team: Gamma

We used the value function from the latest PPO iteration is better instead of the older value targets in the auxiliary phase. We discovered that recalculating the value targets on

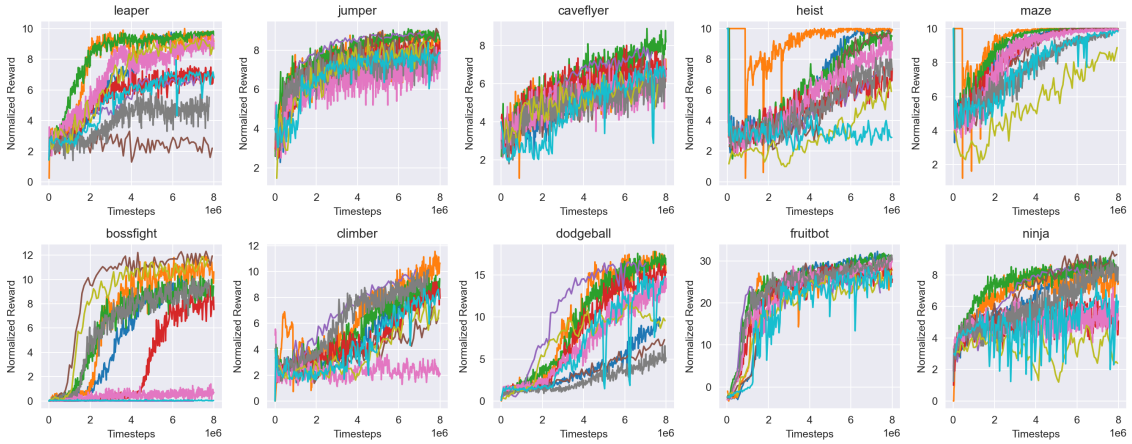

Figure 5: The mean normalized rewards across training per environment for top ten teams in sample-efficiency track.

Table 1: Ranking of participants for generalization and sample efficiency and their base algorithms. The Baseline provided by the organizers replicated the PPO implementation as described in (Cobbe et al., 2020a)

Team or Individual	**General. Rank**	**Sample Eff. Rank**	**General. Mean Norm. Reward**	**Sample Eff. Mean Norm. Reward**	**Base Algorithm**
TRI	1	1	0.6083	0.7680	Phasic Policy Gradient
MSRL	2	7	0.5290	0.6700	Proximal Policy Opt.
Alpha	3	4	0.5193	0.7071	-
ttom	4	9	0.4939	0.6386	Proximal Policy Opt.
Gamma	5	3	0.4898	0.7231	Phasic Policy Gradient
zero	6	8	0.4699	0.6431	Soft Actor-Critic
Xiaocheng Tang	7	5	0.4523	0.6918	Proximal Policy Opt.
Joao Schapke	8	2	0.4447	0.7342	Policy-on Policy-off Policy Opt.
Paseul	9	10	0.3963	0.5847	-
three_thirds	10	6	0.3694	0.6916	Reactor & Soft-Actor Critic
Baseline	**11**	**11**	**0.2002**	**0.3695**	**Proximal Policy Opt.**

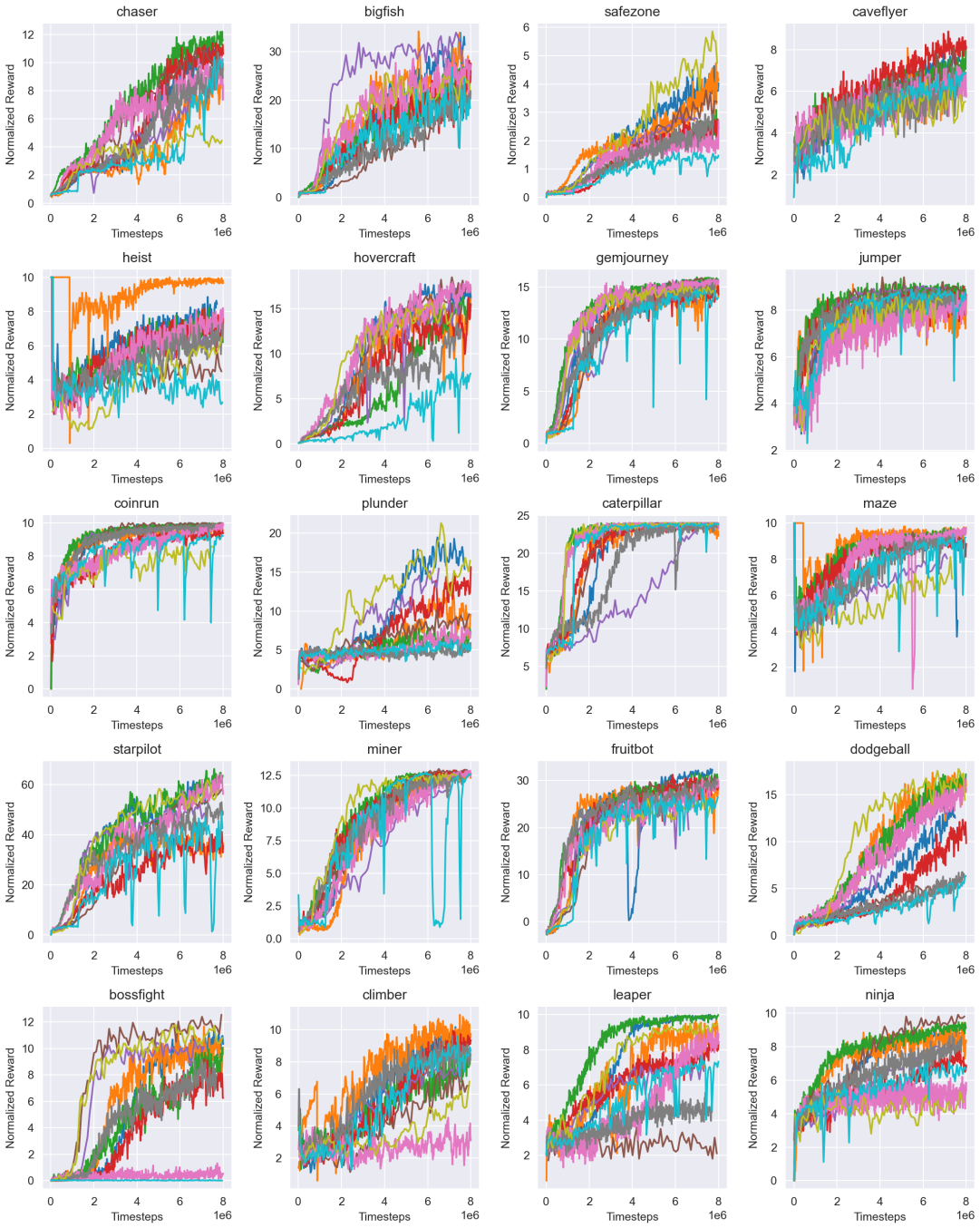

Figure 6: The mean normalized rewards during training phase across rollouts per environment for Team MSRL in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

the entire replay buffer using GAE (Schulman et al., 2016) boosts sample efficiency. We augment the observations in the replay buffer during the auxiliary phase. Crucially, we found that keeping a decent percentage of frames un-augmented is important for policy stability. We used random translate, and colored cutout augmentations from (Laskin et al., 2020) and apply them consistently across frame stacks. Using highest probability action during inference led to the agent getting stuck in one place when a wrong action led to the same state. However, we thought that the stochastically choosing actions in unseen environments led to too much randomness and premature death. Thus, we reduced the softmax temperature during inference which led to improvements in the scores for all environments. Hyperparameter tuning played a very crucial role in the competition. The performance of the final submission is shown in Figure 17.

3.1.2. Team: TRI

For computational reasons, we elected to use the single-network variant of PPG, though unlike the original paper we left the value head of the network attached during the policy phase, and used a smaller loss coefficient for the value objective. We adapted PPO and PPG to use data augmentation. We evaluated a subset of the augmentations such as translation, vertical and horizontal flipping, rotation, conversion to grayscale, and color cutout, finding that random translation worked most consistently across environments. We found that augmentation during the auxiliary phase worked best, and used that. We used a reward shaping penalty. Reward normalization, which involves transforming the rewards of the agent with the goal of normalizing the learning targets of the value network, had a significant impact on performance. Finally, we performed coarse grid searches for a subset of hyperparameters. The generalization and sample-efficiency results of the final submission are shown in Figure 8 and Figure 18 respectively.

3.2. Base Algorithm: Proximal Policy Optimization

One third of the top ten winners used PPO (Schulman et al., 2017) with modifications to the original IMPALA network along with variations on exploration and regularization. All the participants used some form of hyperparameter tuning.

3.2.1. Team:MSRL

Our approach focused on improving the performance of the basic PPO with the IMPALA agent architecture, as described in the original Procgen paper (Cobbe et al., 2020a), purely using data augmentation, L2 regularization of the agent network parameters, and hyperparameter tuning. We didn't find data augmentation, as well as batchnorm and dropout, to be helpful in improving performance. However, L2 regularization and careful hyperparameter choice improved PPO's performance significantly. They allowed our agent to vastly outperform the competition's PPO baseline and to approach the performance of more sophisticated techniques without any changes to the basic PPO algorithm. We tuned all hyperparameters of the rllib's PPO implementation, the number and size of the IMPALA architecture's residual blocks, and the L2 regularization coefficient via a series of hyperparameter searches using the Distributed Grid Descent algorithm (Loynd et al., 2020) running on Microsoft Azure Batch service (AZB). The best-performing parameter combination we

discovered is in Appendix A.3.3. Tuning the discount factor γ, GAE λ, L2 regularization coefficient, and the learning rate for a given architecture was particularly important. The performance of the final submission is shown in Figure 20.

3.2.2. Individual:ttom

The biggest single improvement to performance came from modifying the model. The baseline IMPALA CNN used residual blocks with 16, 32, and 32 channels each. Increasing this width to 32, 64, and 64 channels drastically improved performance. Average pooling the final convolutional layer before flattening and passing into a fully connected layer also improved performance and drastically reduced the number of network parameters. A good learning rate and an approximate one-cycle (Smith and Topin, 2018) schedule greatly sped up early training, getting the algorithm to a reasonable level of performance (close to the final level) after 4M timesteps for all environments. Each algorithm and model variation had parameters tuned individually. Adding an intrinsic reward signal (Random Network Distillation, (Burda et al., 2018)) did result in marginally faster training but performance remained approximately the same. The implementation of RND did not apply across episode boundaries, as originally implemented, which may have negatively affected the results. The performance of the final submission is shown in Figure 23.

3.2.3. Individual:Xiaocheng

We used Random Network Distillation (RND) bonus to encourage exploration. We observed better performance when the intrinsic reward is treated as a life-long novelty computed in a non-episodic setting regardless of the episodic nature of the tasks. Similar to the Never-Give-Up (NGU) agent, we used separate parameterizations to learn varied degrees of exploration and exploitation policies, such that the exploitative policy focuses on maximizing the extrinsic reward and the exploratory ones seek for novelty bonus. We implemented this family of policies under the PPO framework. We apply the mixup regularization proposed for supervised learning in the context of reinforcement learning to increase data diversity and to induce a smoother policy which can generalize better. In particular, we augment the training data by sampling new observations from the convex hull of distinct observations and the corresponding training targets. We added an auxiliary loss term. The auxiliary loss for training the value function can be derived in the similar fashion. To adaptively adjust the agent's behavior accordingly, we employed a meta-controller implemented as a multi-arm UCB bandit with ϵ_{UCB}-greedy exploration to adapt to the changes of the reward through time. The performance of the final submission is shown in Figure 24.

3.3. Base Algorithm: Policy-on Policy-off Policy, Reactor, or Soft Actor-Critic

A modified version of the the Policy-on Policy-off Policy Optimization (P3O) (Fakoor et al., 2020) algorithm ranked second in sample-efficiency. P3O has an off-policy optimization phase in which samples are used with an policy gradient objective modulated by a clipped importance sampling term, hence, resulting in a more sample-efficient approach. The two other base algorithms used were Reactor (Gruslys et al., 2017) and Soft Actor-Critic (SAC) (Haarnoja et al., 2018).

3.3.1. Team:Three-thirds

Reactor (Gruslys et al., 2017) combines the best of PPO and Rainbow and improves on them: it is an Actor-Critic architecture that uses a multi-step off-policy Q-learning, distributional Q-learning in the critic, and a prioritized experience replay for sequences. We modified the vanilla Reactor slightly. Instead of using LSTM, we stacked the last two images. We passed the total reward the agent had accumulated in the current episode as an additional input channel to provide additional information about the game state to the initial convolutional layers. We modified the IMPALA encoder, $[24, 40, 48]$ channels in the convolutional layers, tuned for the competition resource constraints. We used a second Q-network head as in SAC (Haarnoja et al., 2018), which helped decrease the overestimation of the Q-value. Unlike SAC, both Q-networks shared the encoder due to limited computational resources. Sampling from the experience buffer using a mixture of prioritized and uniform distributions as in Reactor, because vanilla prioritized sampling turned out unstable. We used dynamic rescaling of the entropy coefficient τ as in SAC so that the desired entropy level is maintained. The performance of the final submission is shown in Figure 21.

3.3.2. Individual:Joao Schapke

P3O is a synchronous algorithm and its two phases increases the compute time of standard policy gradients. Due to the time constraints of the competition this was a limitation to the performance of the algorithm. In order to improve the compute efficiency we implemented a distributed adaptation of the original algorithm inspired by the IMPALA (Espeholt et al., 2018a) algorithm: many actors with a single learner, and additionally, a single replay buffer. In the final distributed approach we removed the on-policy phase of the algorithm, making it an off-policy policy gradient algorithm. Used prioritised sampling to draw samples with a strong signal. The Procgen environment, similarly to Atari, does not have the Markov property. The partial observation obtained of the environment does not contain all the information needed to make optimal actions, and past observations may contain information (e.g. speed and direction of projectiles or opponents) needed for good performance. We implemented an approach that squishes past frames into a single frame. This is done by adding the weighted 1-lag difference of the past grayscaled frames. The performance of the final submission is shown in Figure 22.

3.3.3. Team: Zero

First, we adapted SAC for discrete action space and refine it with several improvements on DQN (Mnih et al., 2015). We then introduced a new convolutional neural network that not only performed better than the network from IMPALA (Espeholt et al., 2018b) but used fewer resources. We observed that each environment contains some redundant actions never used in practice. We used a shared network and allow only gradients from the Q function back propagate to the encoder which worked best. We used IQN (Dabney et al., 2018), distributional RL algorithm that learns a mapping from probabilities to returns, to replace the Q function in SAC. We used adaptive multi-step improves multi-step learning to allow adaptively select the multi-step target based on the quality of the experiences. We modified IMPALA CNN by adding a channel-wise module (Hu et al., 2020; Woo et al., 2018) to the residual block and replacing all MaxPool with MaxBlurPool (Zhang, 2019).

This architecture had only a quarter of the parameters compared to IMPALA CNN and empirically performed much better because of the channel-wise attention module. The performance of the final submission is shown in Figure 25.

4. Discussion

There were several key commonalities across the methods which resulted in significant improvements in generalization or sample-efficiency:

- *Hyper-parameter tuning*: Almost all participants applied hyper-parameter tuning which resulted in significant improvements in the performance. The methods require a lot of tuning in order to get good performance, and this consumes a great deal of time and is a dull task. A more critical evaluation of improving hyper-parameter tuning for reinforcement learning algorithms would be beneficial for practitioners.
- *Data augmentation*: Many participants used data augmentation for generalization while trying to balance the variations without degrading the sample efficiency. For PPG, keeping a decent percentage of frames un-augmented was important for policy stability.
- *Neural Network*: Modifying the IMPALA neural network from (Cobbe et al., 2020a) also resulted in significant performance changes. Several teams made the IMPALA neural network's CNN layers wider. Another modification was to add channel-wise module to the residual block and replacing all MaxPool with MaxBlurPool which resulted in fewer parameters yet achieved higher rewards.
- *Reward shaping and normalization*: One team found out that the reward normalization, which involves transforming the rewards of the agent with the goal of normalizing the learning targets of the value network, had a significant impact on performance.

We also list a few methods that have not resulted in anticipated performance improvements:

- Adding an intrinsic reward signal (Burda et al., 2018) did result in marginally faster training but performance remained approximately the same.
- Not all data augmentation techniques worked as well and not for all methods. In some cases, after optimizing for the neural network and algorithm parameters, the performance improvement from data augmentation was not significant.
- Using recurrent neural networks slowed the performance too much. A more effective method was to use framestacking.
- DQN worked on some environments but required a lot more experience and resulted in very slow training (e.g., large target network update interval, small learning rate).
- Noisy nets (Fortunato et al., 2019) for exploration did not result in improvements. One reason may be that the PPO update implicitly constrains the policy to not change to much, the additional change due to the noise makes the policy update less efficient.

- Approaches based on auxiliary tasks such as contrastive learning (Srinivas et al., 2020) and deepMDP (Gelada et al., 2019) degraded performance. This may be due the overhead these methods introduce, which significantly reduces the number of training steps, leading to an undertrained model.

5. Conclusion

We ran the NeurIPS 2021 Procgen Competition for measuring Sample Efficiency and Generalization in Reinforcement Learning. We describe the design of a Reinforcement Learning competition using the Procgen Benchmark, which enables end to end training and evaluation of thousands of user submitted code repositories. The end to end training and evaluation framework allows us to add interesting layers of complexities to the benchmark design, like the ability to enforce and measure generalization and the ability to enforce sample efficiency constraints. We summarized the performance and described the submissions of the top teams for both the Sample Efficiency Track and the Generalization Track.

Acknowledgments

We thank the whole team at AIcrowd for the countless hours dedicated towards the conception and execution of this challenge. We would particularly thank the whole DevOps and Support team at AIcrowd to allow such a smooth execution of such an ambitious challenge. We thank Sunil Mallya and Cameron Peron for their support in obtaining the necessary computational resources to execute a challenge of this scale, and for facilitating the provision of AWS compute credits for the participants of this benchmark. We thank Anna Luo, Jonathan Chung and Yunzhe Tao from the Amazon Sagemaker team for their continued support in preparing high quality baselines for participants to get started with Procgen Competition on Sagemaker. We thank the hundreds of participants of this competitions who spent countless hours helping us improve this benchmark and push the state of art for Generalization in RL. We thank Amazon Amazon Web Services for their generous sponsorship in computational resources to enable this competition.

References

Microsoft Azure Batch. https://azure.microsoft.com/en-us/services/batch/.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, jun 2013.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning, 2020a.

Karl Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. *arXiv preprint arXiv:2009.04416*, 2020b.

Will Dabney, Georg Ostrovski, David Silver, and Remi Munos. Implicit quantile networks for distributional reinforcement learning. *35th Int. Conf. Mach. Learn. ICML 2018*, 3: 1774–1787, 2018.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018a.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Boron Yotam, Firoiu Vlad, Harley Tim, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *35th Int. Conf. Mach. Learn. ICML 2018*, 4:2263–2284, 2018b.

Rasool Fakoor, Pratik Chaudhari, and Alexander J. Smola. P3o: Policy-on policy-off policy optimization. In *UAI*, 2020.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2019.

Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G. Bellemare. Deepmdp: Learning continuous latent space models for representation learning, 2019.

Audrunas Gruslys, Will Dabney, Mohammad Gheshlaghi Azar, Bilal Piot, Marc Bellemare, and Remi Munos. The reactor: A fast and sample-efficient actor-critic agent for reinforcement learning. *arXiv preprint arXiv:1704.04651*, 2017.

William H Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The minerl competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*, 2019a.

William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. MineRL: A large-scale dataset of Minecraft demonstrations. In *The 28th International Joint Conference on Artificial Intelligence*, 2019b.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-Excitation Networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(8):2011–2023, 2020. ISSN 19393539. doi: 10.1109/TPAMI.2019.2913372.

Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 10. ACM, 2010.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *The 25th International Joint Conference on Artificial Intelligence*, pages 4246–4247, 2016.

Łukasz Kidziński, Sharada P Mohanty, Carmichael F Ong, Jennifer L Hicks, Sean F Carroll, Sergey Levine, Marcel Salathé, and Scott L Delp. Learning to run challenge: Synthesizing physiologically accurate motion using deep reinforcement learning. In *The NIPS'17 Competition: Building Intelligent Systems*, pages 101–120. Springer, 2018.

Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

Ricky Loynd, Roland Fernandez, Asli Celikyilmaz, Adith Swaminathan, and Matthew Hausknecht. Working memory graphs. In *ICML*, 2020.

Volodymyr Mnih, Kavukcuoglu Koray, Silver David, Andrei A. Rusu, Joel Veness1, Marc G. Bellemare, Alex Graves, Riedmiller Martin, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran1, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Macmillan Publ.*, 2015.

Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

Emmanouil Antonios Platanios, Abulhair Saparov, and Tom Mitchell. Jelly bean world: A testbed for never-ending learning. In *ICLR*, 2020.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates, 2018.

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

Sanghyun Woo, Jongchan Park, Joon Young Lee, and In So Kweon. CBAM: Convolutional block attention module. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 11211 LNCS:3–19, 2018. ISSN 16113349. doi: 10.1007/978-3-030-01234-2_1.

Richard Zhang. Making convolutional networks shift-invariant again. *36th Int. Conf. Mach. Learn. ICML 2019*, 2019-June:12712–12722, 2019.

Appendix A. Supplementary Material

A.1. Generalization Track Graphs

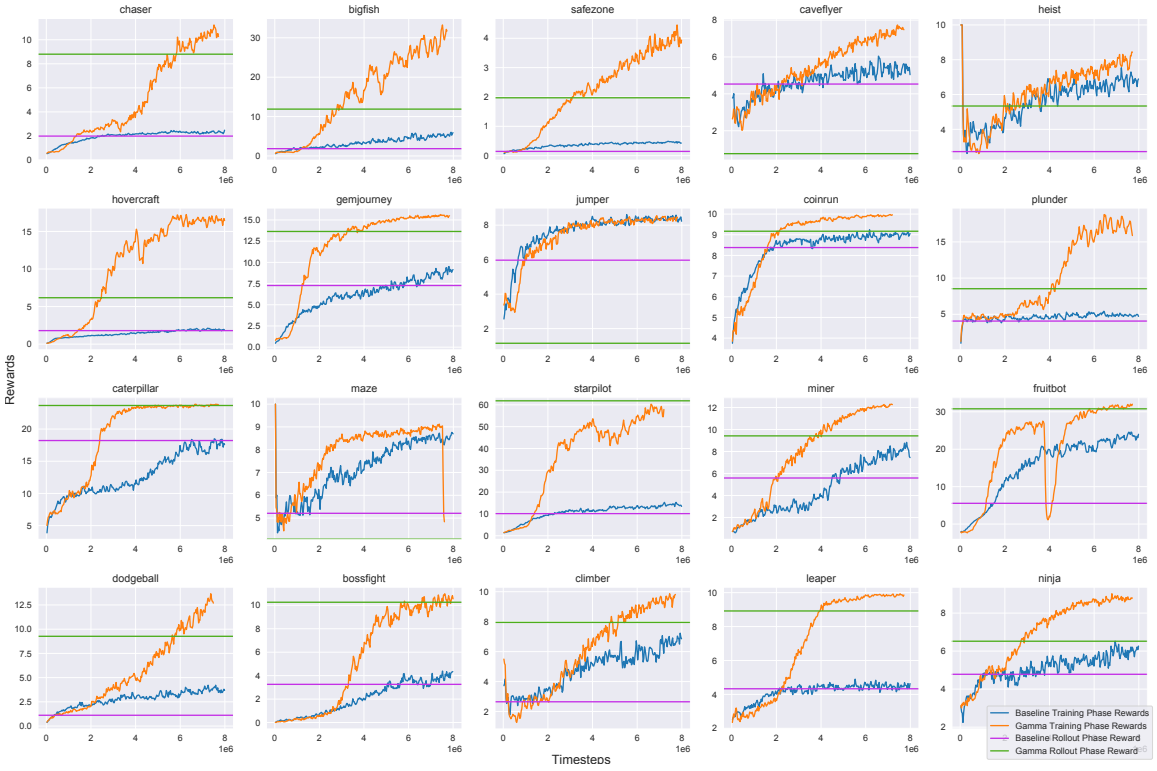

Figure 7: The mean normalized rewards during training phase across rollouts per environment for Team Gamma in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

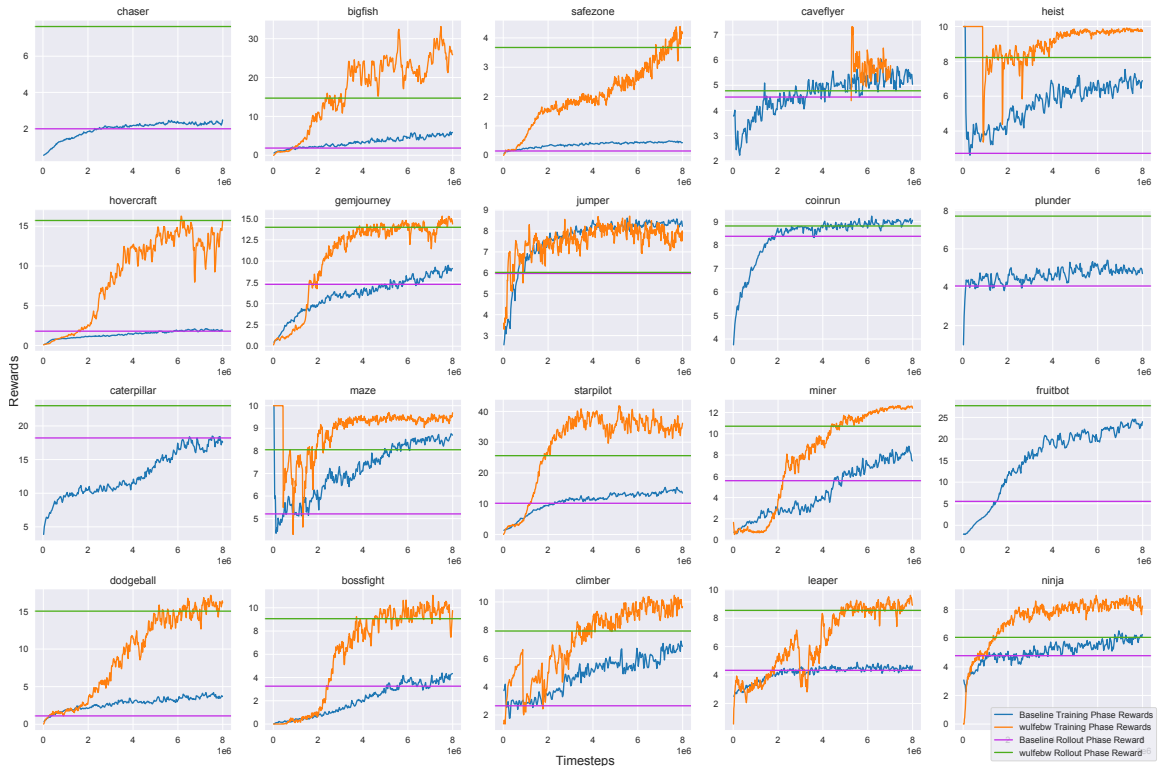

Figure 8: The mean normalized rewards during training phase across rollouts per environment for Team TRI in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

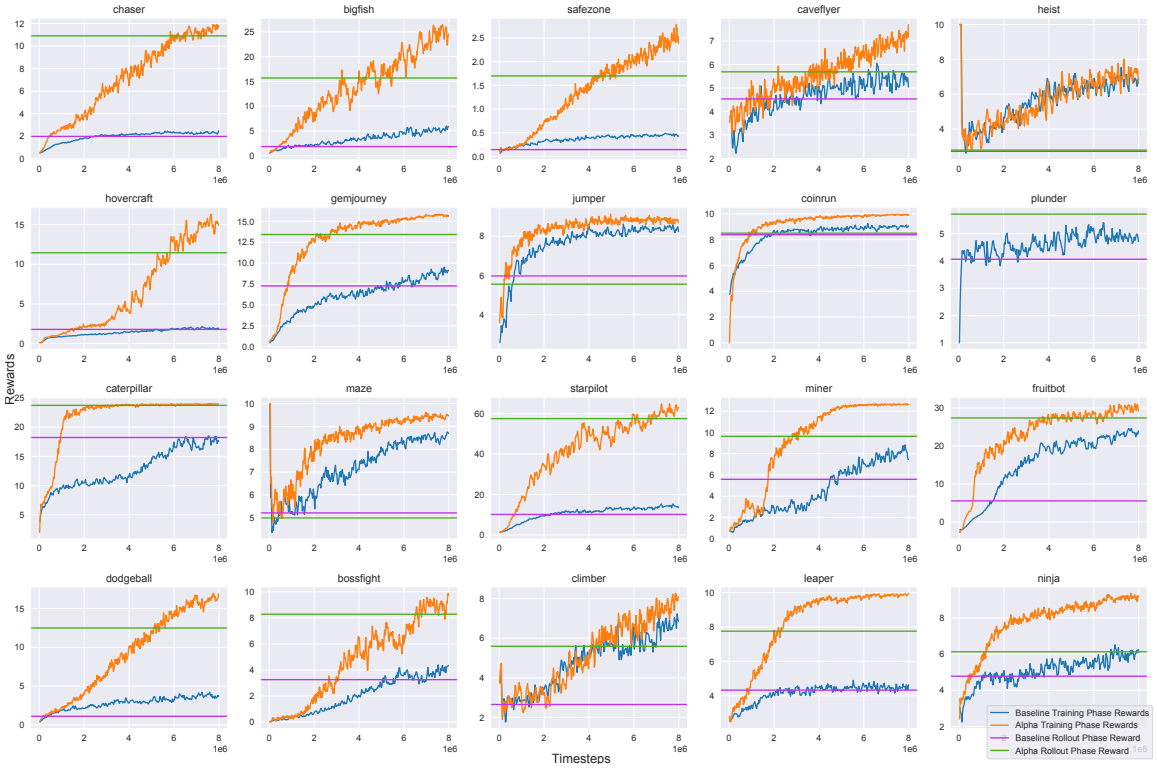

Figure 9: The mean normalized rewards during training phase across rollouts per environment for Team Alpha in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

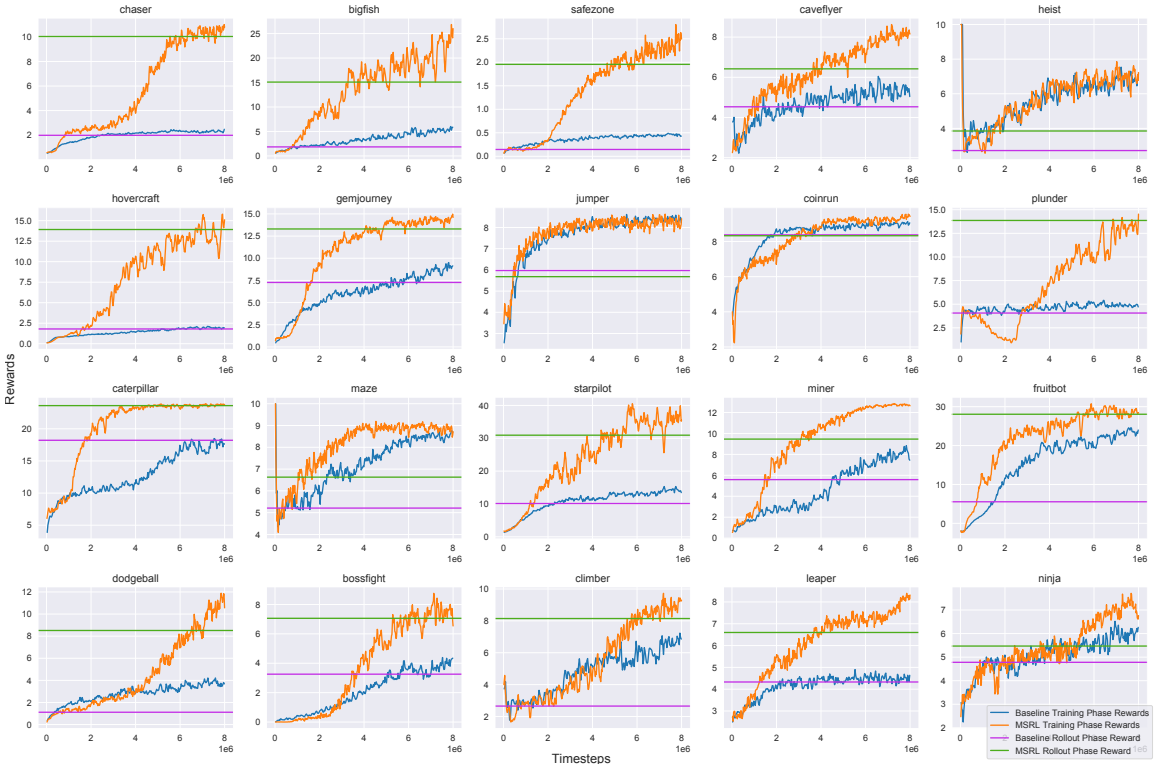

Figure 10: The mean normalized rewards during training phase across rollouts per environment for Team MSRL in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

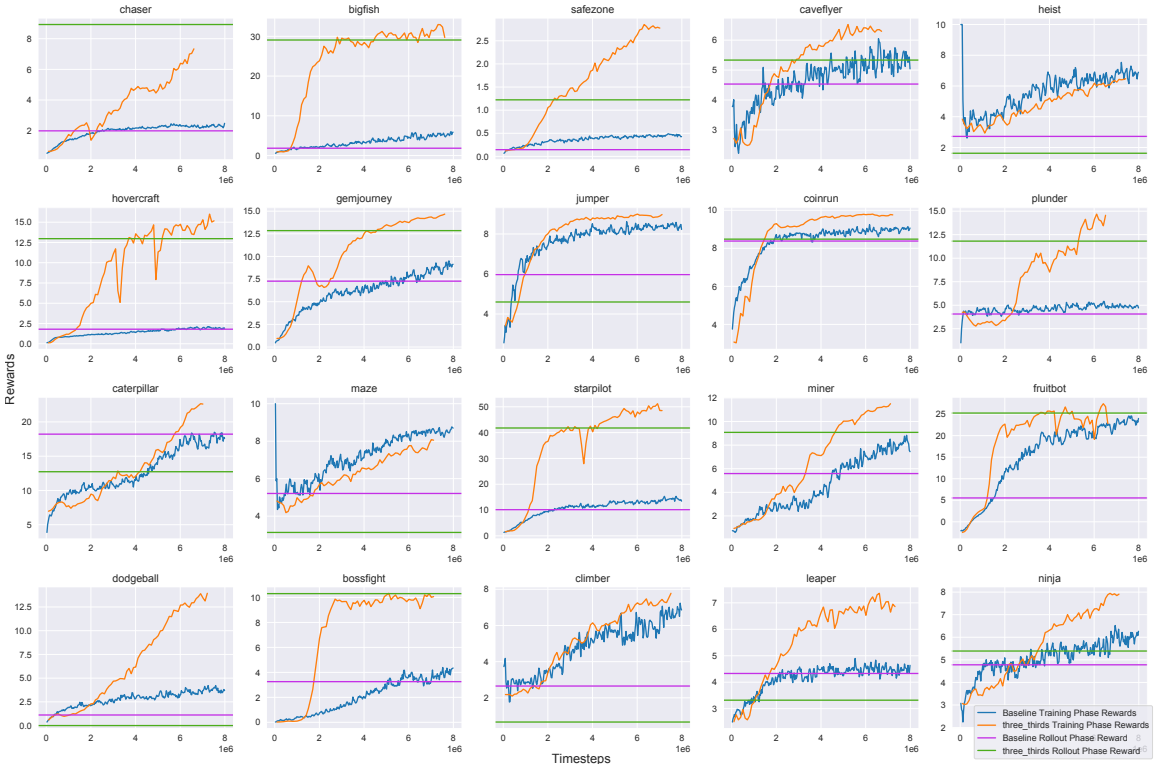

Figure 11: The mean normalized rewards during training phase across rollouts per environment for Team ThreeThirds in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

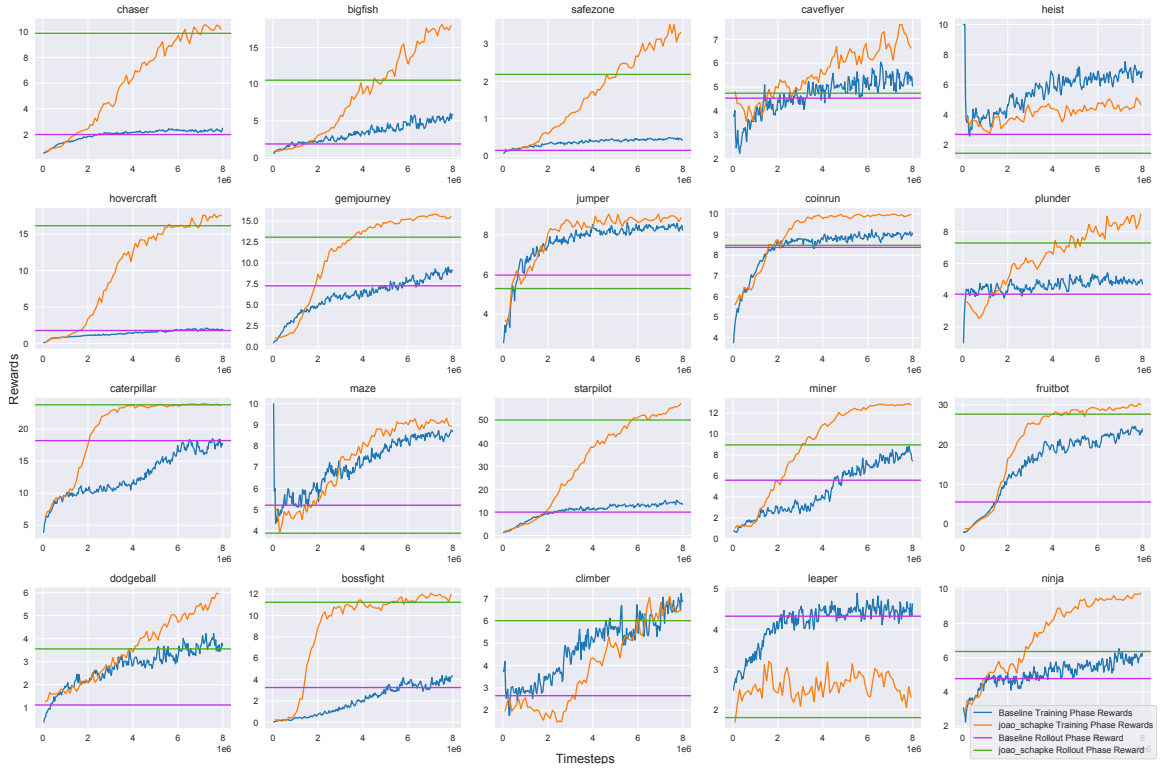

Figure 12: The mean normalized rewards during training phase across rollouts per environment for Individual Joao Schapke in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

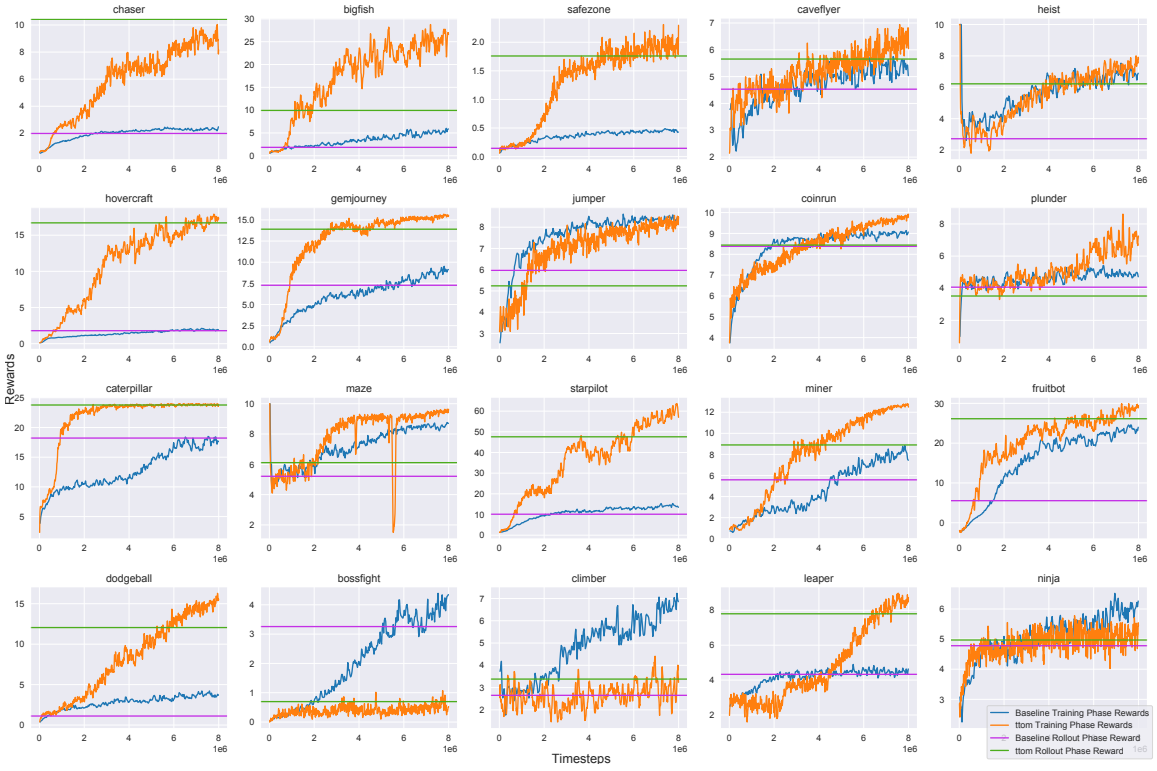

Figure 13: The mean normalized rewards during training phase across rollouts per environment for Individual ttom in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

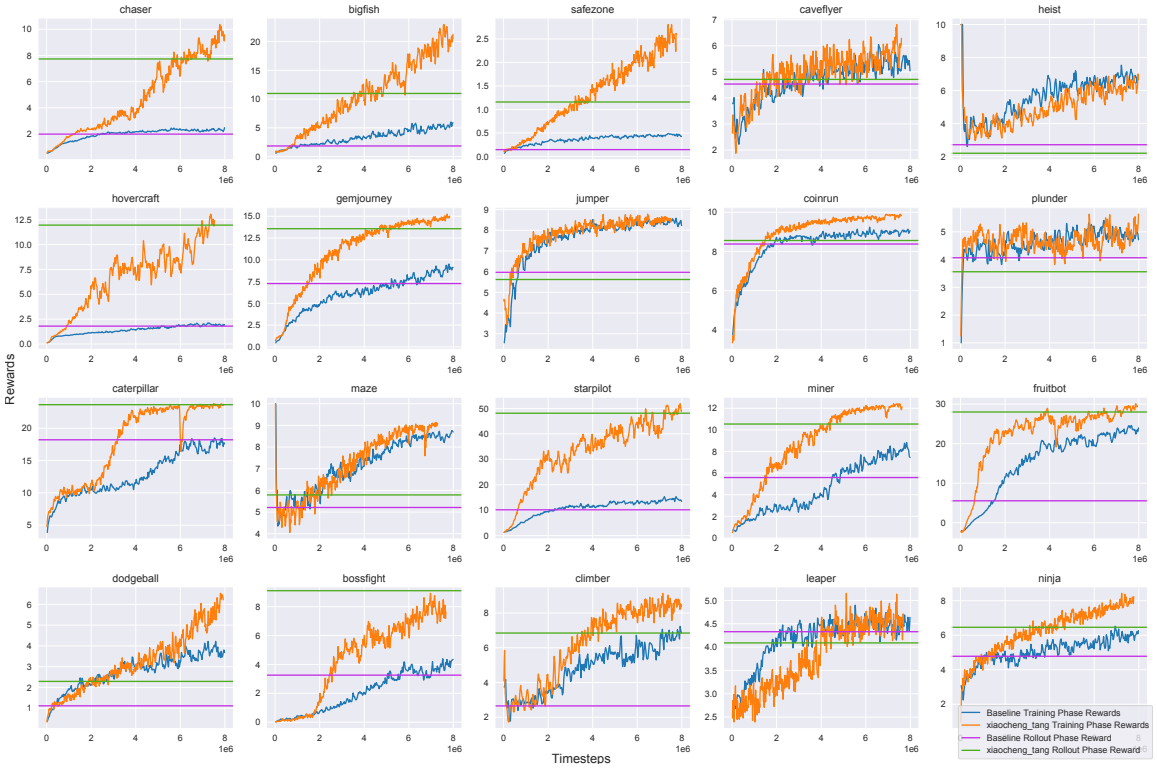

Figure 14: The mean normalized rewards during training phase across rollouts per environment for Individual Xiaocheng Tang in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

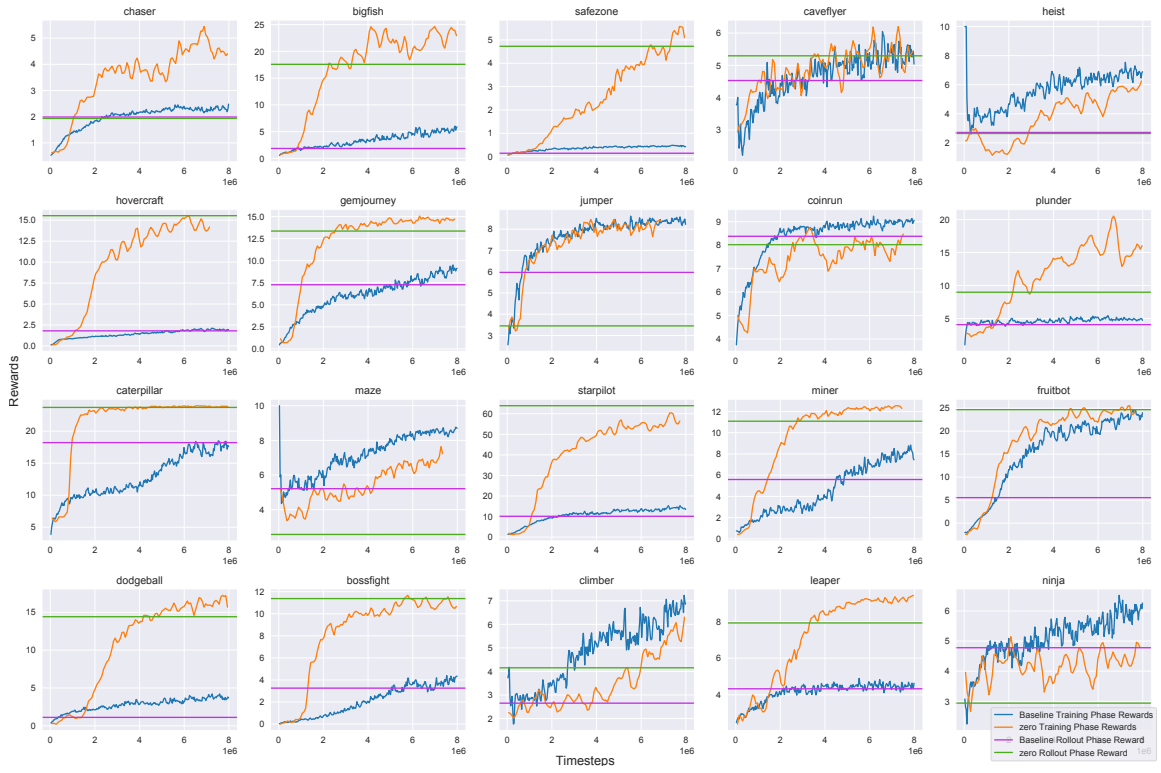

Figure 15: The mean normalized rewards during training phase across rollouts per environment for Team Zero in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

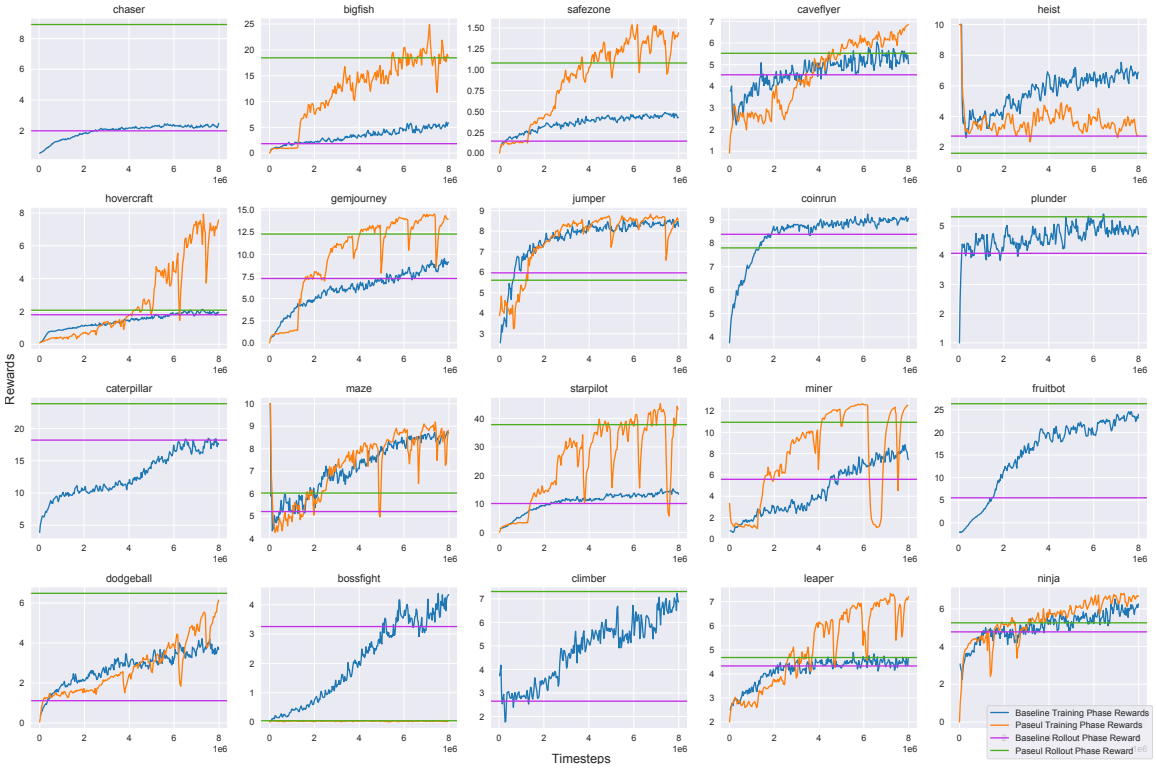

Figure 16: The mean normalized rewards during training phase across rollouts per environment for Team Paseul in generalization track. Note that training score are for the 200 levels, not for the entire distribution of the environment.

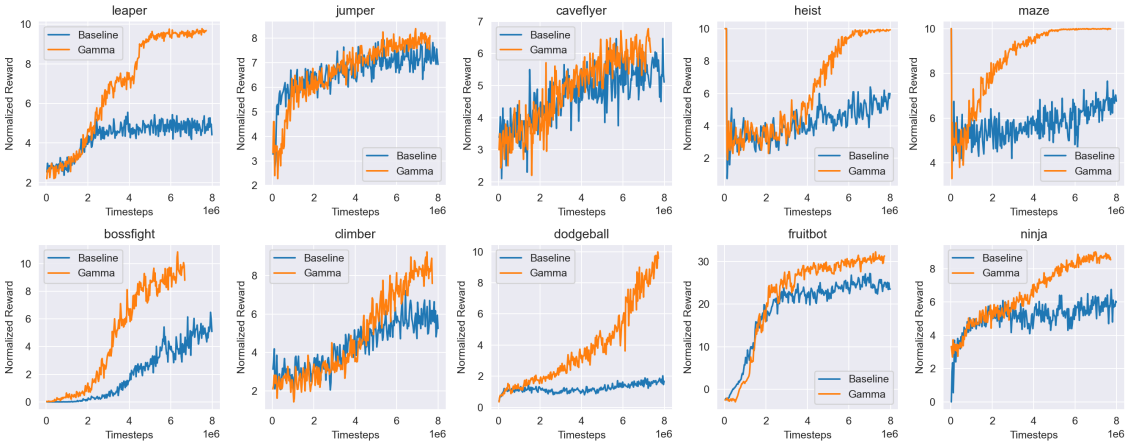

Figure 17: The mean normalized rewards across rollouts per environment for Team Gamma in sample-efficiency track.

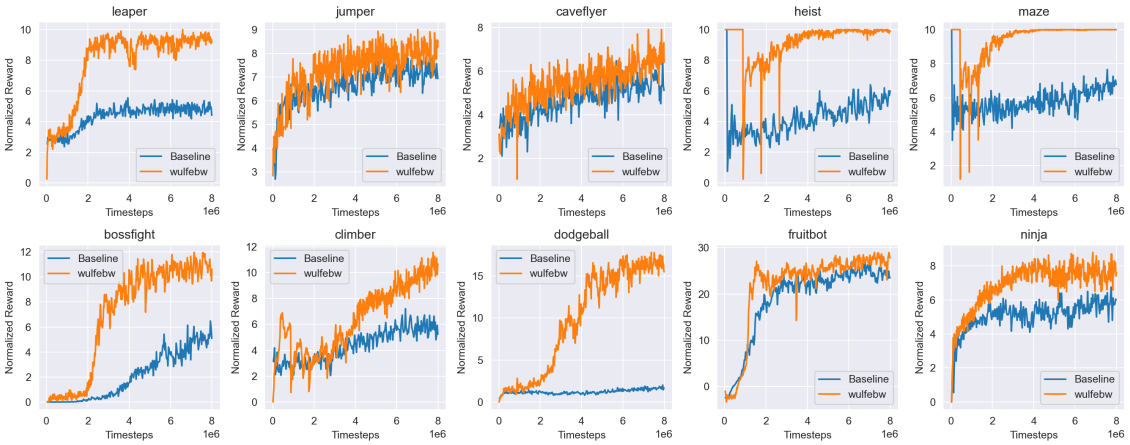

Figure 18: The mean normalized rewards across rollouts per environment for Team TRI in sample-efficiency track.

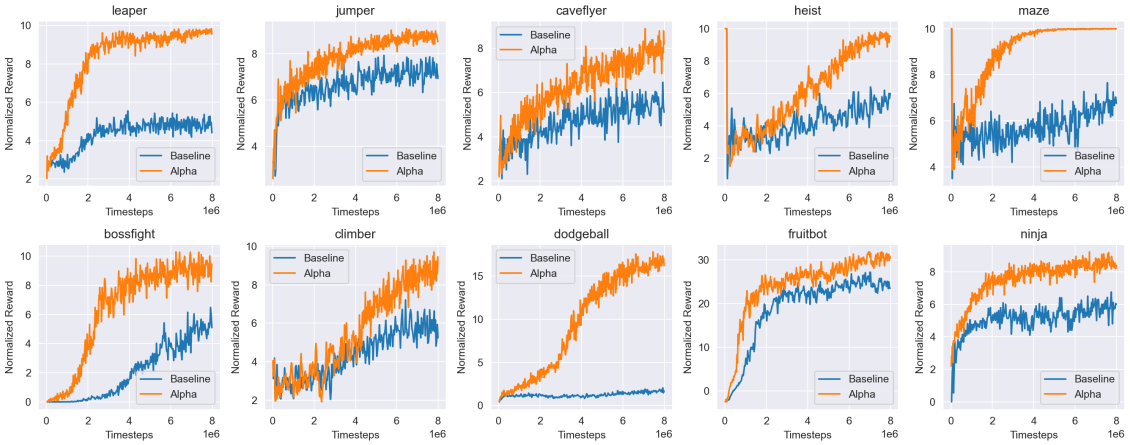

Figure 19: The mean normalized rewards across rollouts per environment for Team Alpha in sample-efficiency track.

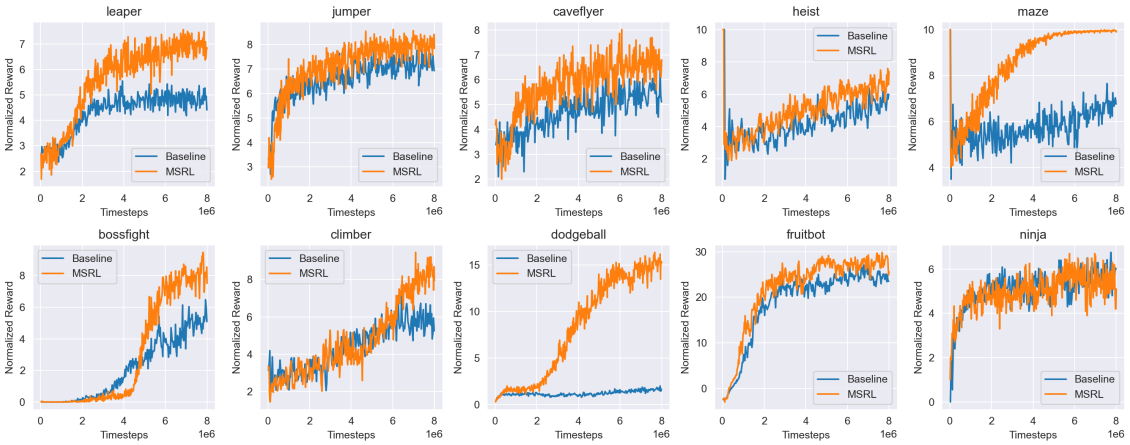

Figure 20: The mean normalized rewards across rollouts per environment for Team MSRL in sample-efficiency track.

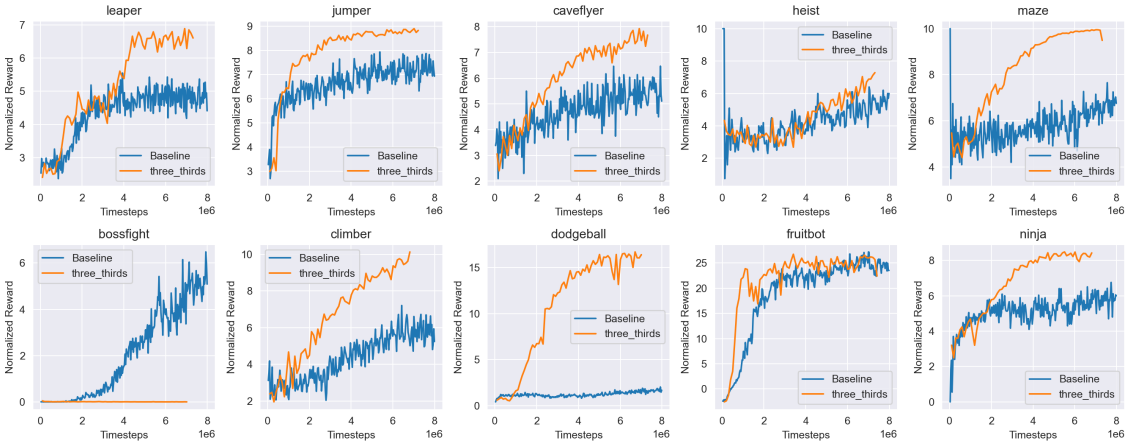

Figure 21: The mean normalized rewards across rollouts per environment for Team Three-Thirds in sample-efficiency track.

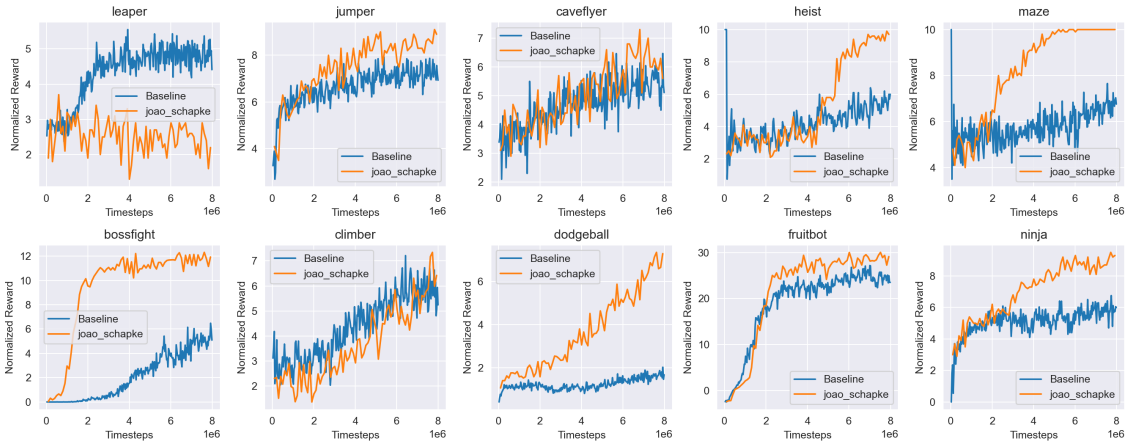

Figure 22: The mean normalized rewards across rollouts per environment for Individual Joao Schapke in sample-efficiency track.

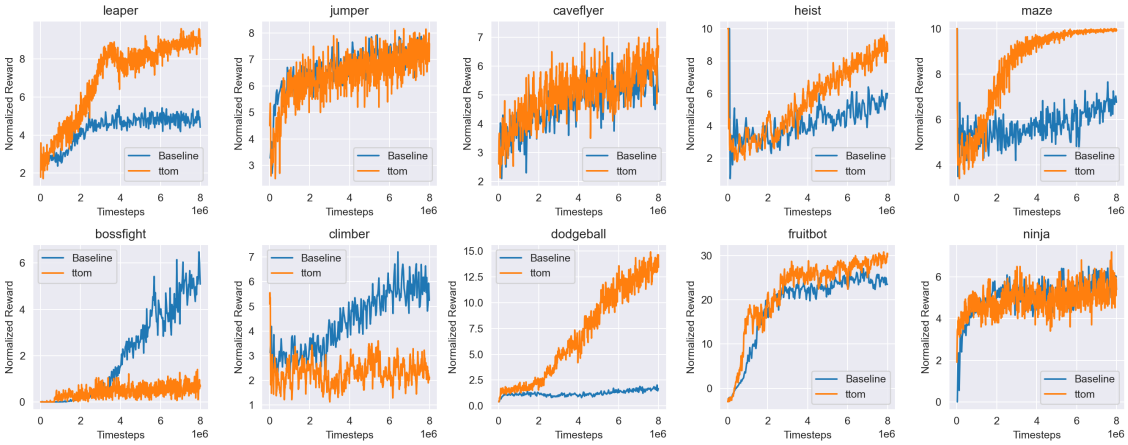

Figure 23: The mean normalized rewards across rollouts per environment for Individual ttom in sample-efficiency track.

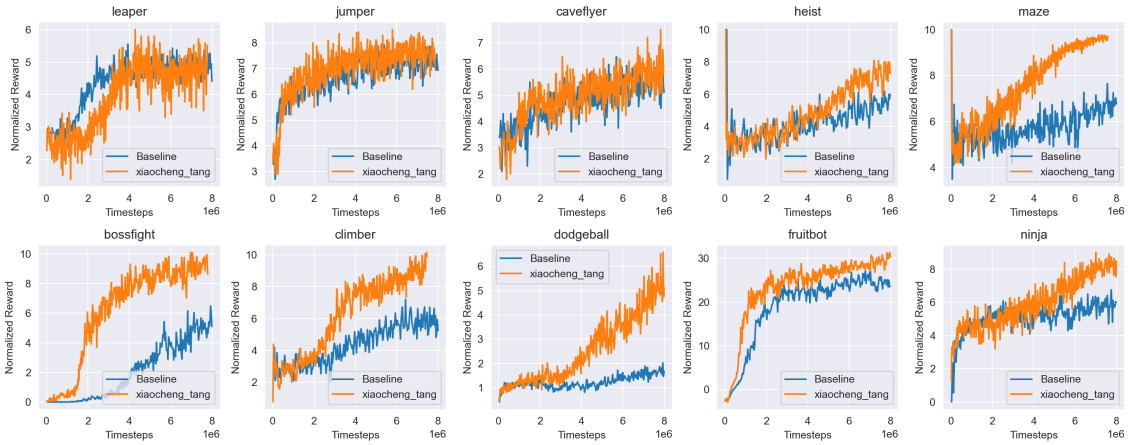

Figure 24: The mean normalized rewards across rollouts per environment for Individual Xiaocheng Tang in sample-efficiency track.

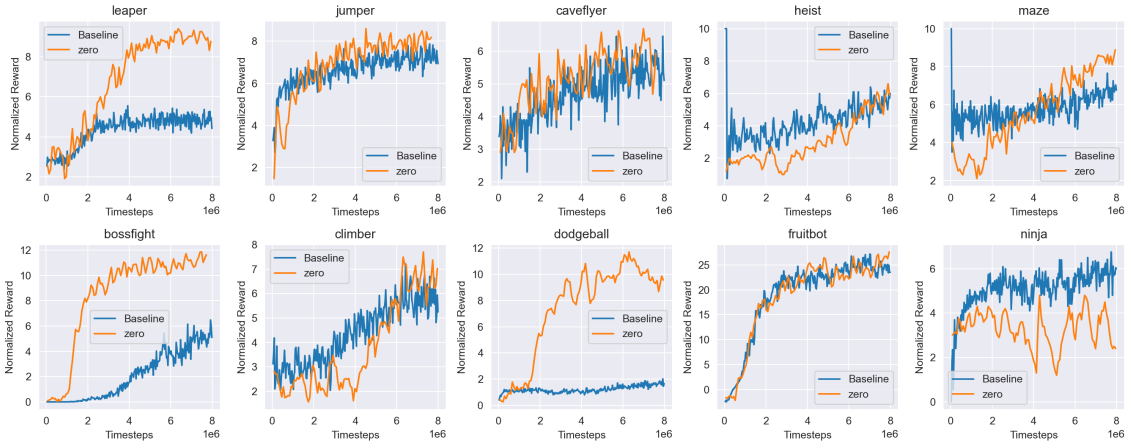

Figure 25: The mean normalized rewards across rollouts per environment for Team Zero in sample-efficiency track.

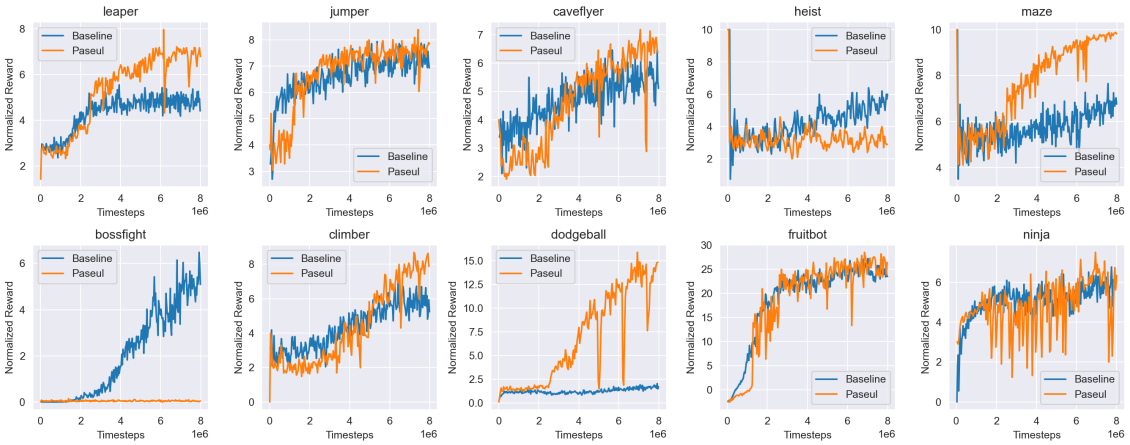

Figure 26: The mean normalized rewards across rollouts per environment for Team Paseul in sample-efficiency track.

A.2. Sample-Efficiency Track Graphs

A.3. Hyper-parameters

A.3.1. Team: Gamma

Number of Parallel Envs	112
Truncated Rollout Length	256
Updates per epoch	8
PPO Minibatch size	3584
Frame Stack	2
Aux Phase Frequency	18
Replay buffer size	500k
γ	0.996
λ	0.95
Reward Normalization	Yes
Replay buffer sampling	Uniform Random [2]

Impala CNN Depths	[32, 64, 64]
Last Dense Layer	512
Preprocessing	Divide by 255
Optimizer	Adam
Gradient Clipping	None
Auxiliary Epochs	7
Aux Minibatch size	2048
Optimizer Epsilon	1×10^{-8}
Learning rate	5×10^{-4}
Learning rate Schedule	Linear
Final learning rate	5×10^{-5}

A.3.2. Team: TRI

Hyperparameter	**Value**
Impala Layer Sizes	32, 48, 64
Rollout Fragment Length	16
Number of Workers	7
Number of Envs Per Worker	125
Minibatch Size	1750
PPG Auxiliary Phase Frequency	32
PPG Auxiliary Phase Number of Epochs	2
Value Loss Coefficient	0.25
Framestack	2
Dropout Probability (Auxiliary Phase Only)	0.1
No-op Penalty	-0.1

A.3.3. Team: MSRL

PPO hyperparameter	Value
Impala layer sizes	32, 64, 64, 128, 128
Rollout fragment length	256
Number of workers	2
Number of environments per worker	64
Number of CPUs per worker	5
Number of GPUs per worker	0.1
Number of training GPUs	0.3
Discount factor γ	0.995
SGD minibatch size	2048
Batch size	2048
Number of SGD iterations	3
SGD learning rate	0.0006
Framestacking	off
Truncate episodes	true
Value function clip parameter	1.0
Value function loss coefficient	0.5
Value function share layers	true
KL-div. coefficient	0
KL-div. target	0.1
Entropy regularization coefficient	0.005
PPO clip parameter	0.1
Gradient norm clip value	1
GAE λ	0.8
L2 regularization coefficient	0.00001

A.3.4. Team: ThreeThirds

Environment steps	8M
Generalization training levels	200
Steps trained/sampled	7x
Rollout length (steps)	32
Batch size (steps)	512
γ	0.995
λ	1
Adam learning rate, momentum	$1.5 \times 10^{-4}, 0$
Target network update freq. (batches)	500
Replay buffer size	400k
Replay prioritization ϵ	0.25
Reward prediction loss coeff.	0.1
Entropy target: initial, final	2.3, 1.0
Temperature learning rate	2.5×10^{-4}
Random exploration ε	0.01

A.3.5. Individual: Joao Schapke

Hyperparameter	Value
Learning rate	0.007
entropy_coeff	0.0
vf_coeff	0.5
gamma	0.995
gae_lambda	0.85
grad_clip	1
rollout_fragment_length	32
train_batch_size	2048

Hyperparameter (P3O)	Value
buffer_size	20000
learning_starts	5000
prioritized_replay_alpha	2
prioritized_replay_beta	1
prioritized_replay_eps	0.0000001

A.3.6. Individual: ttom

Hyperparameter	Value
Truncated Rollout Length	200
Updates per epoch	3
PPO Minibatch size	1024
Frame Stack	2 (modified)
γ	0.99
λ	0.9
Reward Normalization	No
Impala CNN Depths	[32, 64, 64]
Last Dense Layer	256
Preprocessing	Subtract 128, divide by 255
Optimizer	Adam
Gradient Clipping	None
Learning rate	1E-4->3E-5
Learning rate schedule	Cosine