Speeding Up PPO Training for Pokémon Red

Caio Valente
github.com/valentecaio
12th June 2024

I. Introduction

In recent years, deep learning has made significant advances in various domains, including natural language processing, computer vision, and reinforcement learning. One fascinating application of reinforcement learning is training agents to play video games, which presents a complex and dynamic environment for learning. This paper focuses on training an agent to play Pokémon Red using the Proximal Policy Optimization (PPO) algorithm, with modifications to improve convergence speed and performance.

The original implementation from Peter Whidden [1] leveraged the PyBoy emulator API within a Gym environment, employing a Convolutional Neural Network (CNN) policy optimized using the Adam optimizer. The input to the network was based on the rendered images from the game. The primary objective was to maximize a reward function incorporating various game parameters, such as the number of frames visited, the total level of Pokémon in the party, and the frequency of healing events. Despite the promising framework and impressive results, the training process was notably slow, hindered by CPU-bound computations, the lengthy nature of game episodes and the slow convergence of the algorithm.

To address these challenges, this research explores several modifications to accelerate convergence, including dynamic learning rates, varying batch sizes, and incorporating weight decay using AdamW. Among the tested approaches, a constant large batch size with a constant learning rate proved most effective, reducing substantially the necessary training time. However, this configuration resulted in the agent falling into a local minimum, prioritizing exploration and combat over capturing new Pokémon.

II. Related Work

The application of deep reinforcement learning (DRL) to video game environments has gained considerable attention, with notable successes in playing Atari games and other complex benchmarks. One of the pioneering works in this field is the Deep Q-Network (DQN) algorithm, introduced by Mnih et al. (2015) [2] and implemented by Roderick et al. (2017) [5], which demonstrated the capability of DRL agents to surpass human-level performance in various Atari 2600 games [3]. DQN utilizes a convolutional neural network to approximate the Q-value function, enabling the agent to learn optimal policies directly from raw pixel inputs.

Subsequent advancements have built upon the DQN framework, addressing limitations such as instability and slow convergence. Proximal Policy Optimization (PPO), proposed by Schulman et al. (2017) [4], has emerged as a robust alternative, combining the benefits of trust region policy optimization and importance sampling to achieve stable and efficient training. PPO has been successfully applied to a wide range of tasks, including robotic control, autonomous driving, and game playing. The PPO algorithm uses a policy gradient approach to optimize the policy and value functions. It maintains stability and efficiency by clipping the probability ratios during updates to ensure the new policies do not deviate excessively from the old ones.

While significant progress has been made in mastering simpler games like Atari, more complex games such as Pokémon Red, an RPG game, still require further exploration. These games present a greater challenge due to their non-repetitive gameplay. The agent must navigate a narrative, handle recurrent events with straightforward dynamics like battles, and explore the map to advance the plot of the game. This complexity requires sophisticated learning strategies capable of dealing with both the game's strategic and exploratory aspects.

More recently, Peter Whidden proposed a PPO trained agent capable of beating the first gymnasium of Pokemon Red [1]. His original algorithm focuses on training an agent to play Pokémon Red using the algorithm within a custom Gym [7] environment. The primary objective is to maximize the agent's performance by rewarding various in-game achievements and penalizing setbacks, facilitating the agent's ability to explore, battle, and progress through the game effectively. The PPO model should maximize the total reward of the player for any action in any given state.

Whidden released his training code and a pretrained agent obtained through 439 million training steps, establishing it as the current state-of-the-art for this particular game. This extensive training process allowed the agent to develop strategies for exploring the game world, engaging in battles, and advancing through the storyline. However, the agent is not optimal - while it can successfully navigate up to and pass the first gym, it encounters difficulties and often gets stuck beyond that point. This limitation highlights the ongoing challenges and potential areas for improvement in developing reinforcement learning agents for complex RPG games like Pokémon Red.

III. Proposed Technique

The pretrained model provided by the original author was trained for 439M steps, which would take approximately six days to run. However, it was trained running 128 episodes in parallel, on a machine with 128 cores and enough RAM to accommodate such a large rollout buffer size, likely leading to faster convergence and better generalization than what could be achieved on my 12 cores laptop. This means that achieving similar results on my machine could take up to 60 days.

Instead, my goal was to develop a model that could converge faster, within one day, achieving average results compared to the original setup.

To achieve such, I used the same base algorithm as Whidden and experimented with the following hyperparameters: batch size, learning rate, entropy coefficient and weight decay.

The batch size refers to the number of training examples used in one iteration to update the model's parameters. A larger batch size typically provides a more stable estimate of the gradient, leading to smoother and potentially faster convergence, but requires more memory and computational resources. A smaller batch size may introduce more noise into the gradient estimation, which can sometimes help the model escape local minima but often results in less stable training.

The original implementation used the stable_baselines3 library [6], which lacked support for non-constant batch sizes. This work extended this library to include dynamic batch sizing and applied it in my experiments. The maximum batch size I was able to test was limited by my GPU memory (RTX 4050, 6G).

The learning rate determines the step size at each iteration while moving towards a minimum of the loss function. A higher learning rate allows the model to converge faster but risks overshooting the minimum, leading to divergence or oscillation. A lower learning rate provides more precise adjustments to the model parameters, which can result in more stable convergence but at the cost of longer training times. Dynamic learning rates adjust over time to balance these trade-offs, often starting high to speed up initial training and decreasing to refine the model's performance.

The entropy coefficient controls the entropy bonus added to the reward function, encouraging exploration by preventing the policy from becoming too deterministic too quickly. A higher entropy coefficient promotes more exploratory actions, helping the agent to discover potentially better strategies by exploring a broader range of actions. However, excessive exploration can slow down the convergence to an optimal policy. A lower entropy coefficient reduces exploration, allowing the model to exploit known good actions but risking premature convergence to suboptimal policies.

The weight decay is a hyperparameter of the AdamW optimizer. It is a regularization technique that helps prevent overfitting by penalizing large weights. This penalty term is added to the loss function, encouraging the model to keep the weights smaller and more generalized. In contrast to traditional L2 regularization, which directly modifies the gradients, AdamW decouples weight decay from the gradient updates, leading to more effective regularization. Applying weight decay can improve generalization by preventing the model from fitting noise in the training data. However, if set too high, it may hinder the model's ability to learn complex patterns in the data, resulting in underfitting.

In this work, I tried six variations of hyperparameters in the original PPO algorithm and compared their performance to the original retrained model.

A. Training Procedure

The training process involves alternating between emulating the game and updating the network. Emulation is performed on the CPU, with one emulator running per core in parallel. Each training step represents some frames being emulated. In each step, the player does an action (a button press and release) from the available actions in his action space, which are: Button A, Button B, Up, Down, Left or Right.

An episode of the game is a fixed number of steps that is considered as a round of the game. After the episode is over, the game is reset to the initial state and the training continues with the new trained network.

The rollout buffer stores the game states and rewards of the recent emulation steps and is crucial for training. It is the memory bottleneck of the algorithm, and it's used in the training process to update the network. As the buffer can easily get large, a batch size equal to 128 is specified in the algorithm. This means that only a subset of 128 states are randomly selected in the buffer at each training stage.

It would be ideal to have an episode length long enough to finish the storyline of the game, and rollout buffers large enough to store all steps from these episodes. This would allow the algorithm to converge in less iterations. However, this type of game has a long gameplay, and thus would require unachievable amounts of memory for the buffer and long processing times for emulating full episodes, which is a CPU-bound task.

To be able to fit the model in memory and train it in reasonable time, the episode length was originally set to 24576 steps, which should be long enough for the player to beat the first gym in the game. The rollout buffer size was set to a value of n_cpus * episode_length // 8, being n_cpus the number of available CPU cores. This configuration means that the model is trained eight times per episode in each CPU, balancing the need for frequent updates with computational resource constraints.

B. Algorithm Configuration

The same CNN policy from the original work was used 1. It consists of a single network shared between the value and policy functions, and employs a simple convolutional neural network (CNN) architecture inspired by the original Deep Q-Network (DQN) paper by Mnih et al. (2015) [2]. This CNN consists of three convolutional layers followed by activation functions (ReLU), a flattening layer, and a single fully connected layer.

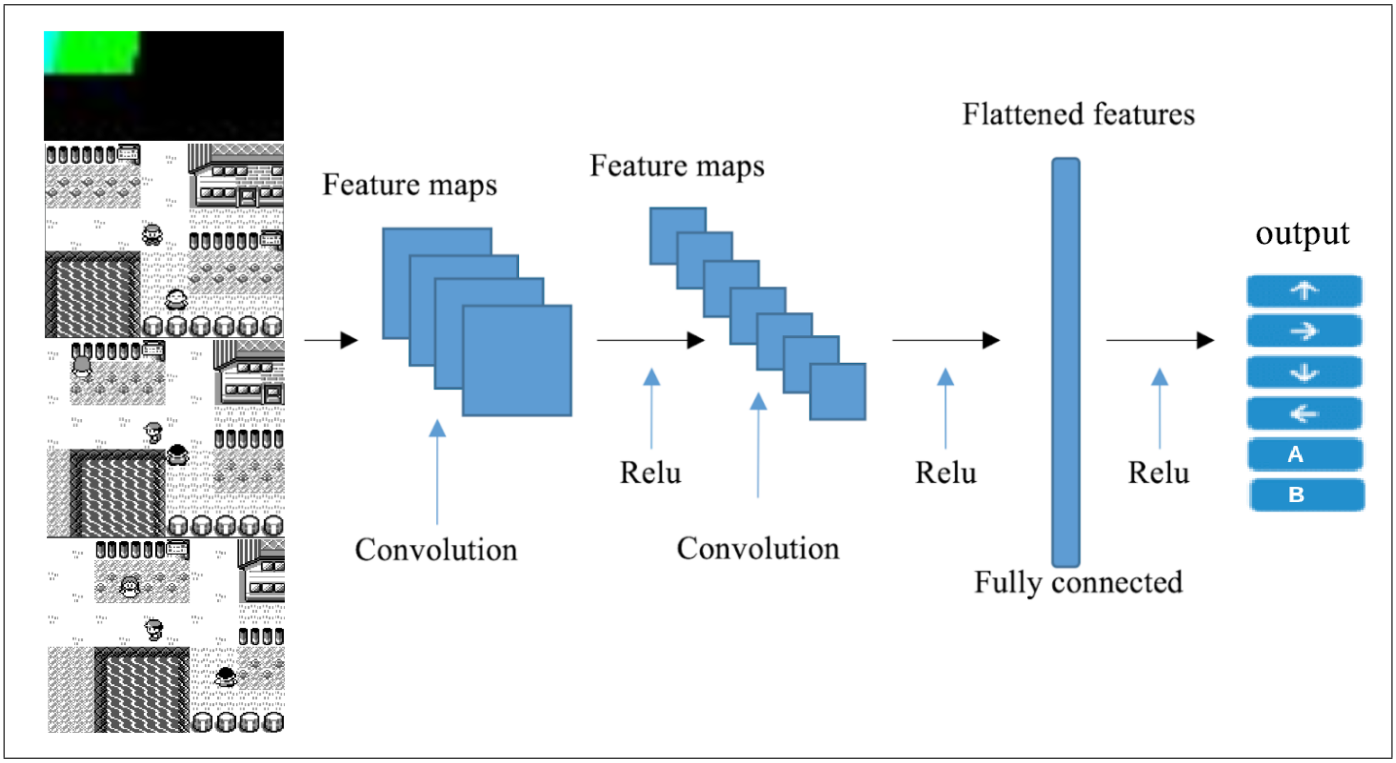

Fig. 1. The CNN network used for policy and value estimation in the PPO algorithm.

The State input of the network is an image containing the last three frames of the game downscaled and concatenated to a pre-processed health points (HP) bar, facilitating short-term memory and enhancing HP-related rewards. The optimizer used for training is the Adam optimizer with default parameters, excluding weight decay.

The original proposed network used the following constant hyperparameters:

- Learning rate: 3×10^{-4}
- Batch size: 128
- Entropy coefficient: 0
- Weight decay in Adam optimizer: 0
- Number of epochs: 3
- Gamma: 0.998

In this work, I experimented with the first four hyper-parameters, varying them to find the best configuration for faster convergence without losing performance. The number of epochs and gamma were kept constant.

C. *Reward Calculation*

The same reward calculation proposed originally was used, based on several metrics related to Pokémon gameplay:

- Event reward: Rewards for specific in-game events being triggered, such as discovering a new area of the map. This should increase exploration of the map.
- Total party level reward: Reward proportional to the cumulative levels of all Pokémon in the player's party. This should motivate the agent to capture new Pokemon and level them up by winning battles.
- Healing reward: Rewards for the healing actions performed, proportional to the amount of HP restored. This should motivate the agent to use healing items and to visit Pokemon centers, which is important since they work as checkpoints when the player loses a battle.
- Maximum opponent level reward: Rewards for defeating high-level opponents. As the level of the opponents increase throughout the game, this should motivate the agent to follow the storyline.
- Gym badges reward: Rewards for earning gym badges. Should motivate the agent to progress through the game.
- Exploration reward: The most important reward, based on the number of unique frames the player has explored. It should directly impact the exploration of the map. The algorithm uses a KNN algorithm to compare the current screen's pixels to a list of previously seen screens. If the screen is new, it is added to the list, encouraging the agent to explore new areas of the game.
- Dying penalty: Penalties (negative rewards) are applied when the player loses a battle, causing a reset to the last visited Pokémon Center. This is particularly important since the player can be sent far back in the game after losing a battle.

IV. Experiments and Results

All experiments were conducted under the following fixed hyperparameters:

- Number of CPU cores: 12 (episodes running in parallel).
- Total training steps: 68714496, i.e: 233 episodes per CPU.
- Episode length: 24576 steps.
- Steps per update: 2048, i.e: update model 12 times per episode.

Due to the CPU-bound nature of the training algorithm, the training time for all models was approximately the same. Each model took between 20 and 22 hours to be trained for 233 episodes.

Looking for convergence speed without losing performance, the following models were tried:

1) **Baseline**: The original model with small batch size (128), constant learning rate of 3×10^{-4}, no entropy and no weight decay. This model did not converge in 68M steps and continued to perform closely to random actions after 68M steps of training, as shown in Fig.3.
2) **AdamW**: Variation using AdamW optimizer with weight decay equal to 1×10^{-4}. Performed closely to the baseline model with no notable improvements or deterioration (Fig.3).
3) **Dynamic Learning Rate**: Dynamic learning rate that decreases logarithmically from 5×10^{-3} to 5×10^{-5}; Performed closely to the baseline model with no notable improvements or deterioration (Fig.3).
4) **Large Batch**: Large batch size (50% of rollout buffer size). Performed much better than the others before. Quickly achieved very small loss function values (Figures 8, 9, 10) for both value and policy but got stuck in a local minimum, focusing on training the initial Pokémon without capturing others or visiting Pokemon centers for healing (Fig.6). Of all models, this one demonstrated by far the best map exploration (above 3000 frames), but followed a poor long-term strategy by not capturing new Pokémon. It is the only model that surprisingly learned to not capture any Pokémon, which happened around step 8M (Fig.7). It got so many rewards from exploring the map that it stopped considering other rewards relevant.
5) **Large Batch + Dynamic Learning Rate**: Large batch size (50% of rollout buffer size) and dynamic learning rate that decreases logarithmically from 5e-3 to 5e-5. This model was the most consistent one for this number of steps (68M). It showed the capacity to get total rewards as high as model 4 and learned to heal and capture Pokemon, but explored the map less. Of all models, it got the best healing reward (Fig.6), the best sum of party level reward (Fig.5), and converged the entropy loss function to the smallest value (Fig.8), but did not learn to explore the map as well as models 4 or 6, staying at around 1000 frames only. The total reward seems to have converged to around 260 (with maxes around 300) but more training would be required to be sure.
6) **Large Batch + Entropy**: Large batch size (50% of rollout buffer size) and entropy coefficient of 0.02. Exhibited the most promising results for training longer, showing the highest total rewards and the second highest exploration rate (Fig.4). It learned to heal and capture Pokemon almost as efficiently as model 5, was the only model to pass the 350 total rewards mark (Fig.3), and kept a good exploration rate (around 1800 frames). We can infer from the loss functions (Figures 8, 9, 10) that this model would still need more training to converge, however, it already shows similar rewards to model 5 with a much better exploration score. Despite not fully converging, it indicated potential for being the best model among those tested, contingent on further training.
7) **Dynamic Learning Rate + Dynamic Batch**: Dynamic learning rate that decreases logarithmically from 5×10^{-3} to 5×10^{-5}, and dynamic batch size that increases exponentially from 32 to 6144 (25% of rollout buffer size). Performed particularly worse than all others, converging to a seemingly random and ineffective strategy with small value and policy losses but a relatively large entropy loss, as shown in Figures 8, 9, 10.

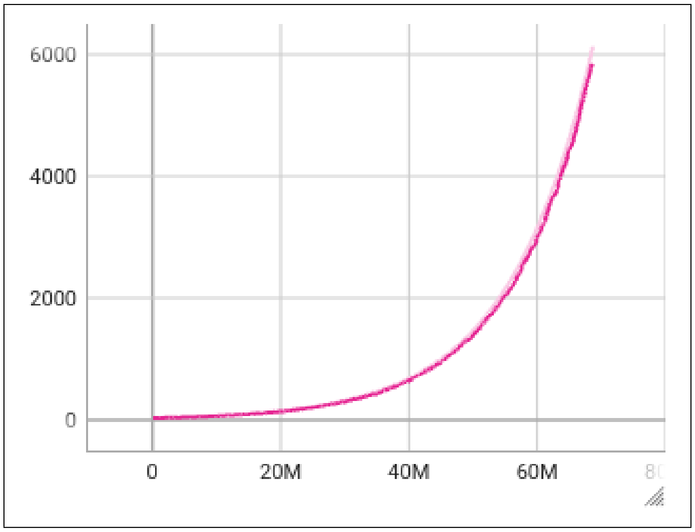

Fig. 2. Exponential batch size on model 7.

V. Conclusions and Future Work

Throughout this research, I aimed to find a reinforcement learning model that could converge within 20 hours of training while achieving average performance compared to the original setup. The experiments involved various configurations of batch size, learning rate, entropy coefficient, and optimizer types. The baseline model (Model 1) did not converge, performing similarly to random actions even after 68 million steps. Models 2 and 3 showed no notable improvements over the baseline, indicating that neither the AdamW optimizer nor the dynamic learning rate alone provided significant benefits, while Model 7 (Dynamic LR + Dynamic Batch size) performed even worse than the baseline, quickly converging to an ineffective strategy.

Among the models, Model 4 (Large Batch) performed significantly better than the initial configurations by quickly achieving small loss values and demonstrating superior map exploration. However, it fell into a local minimum, focusing solely on training the initial Pokémon without capturing new ones or healing. Model 5 (Large Batch + Dynamic LR) emerged as the most consistent model within the 68 million steps, balancing exploration, healing, and capturing Pokémon effectively, though its map exploration was limited. Model 6 (Large Batch + Entropy) showed the highest potential for further training, achieving the highest total rewards and a good balance between exploration and efficient gameplay.

These findings suggest that larger batch sizes combined with dynamic learning rates and entropy coefficients, can enhance the training efficiency and performance of reinforcement learning models in complex RPG game environments like Pokémon Red. Further research and extended training could refine these approaches for even better results.

It is important to note that the proposed strategies have not been exhaustively tested. Models may perform better with different parameters, especially those with dynamic learning rates and batch sizes, which could benefit from a deeper study varying minimum and maximum values and using different increasing/decreasing policies, such as linear. Additionally, my tests were limited by the available hardware. A follow-up study using larger batch sizes could be beneficial. For instance, utilizing 100% of the rollout buffer in the same scenario would require approximately 20 GB of RAM and 12 GB of VRAM. This suggests that with slightly better hardware, the models might achieve even more promising results, highlighting potential areas for future research.

References

[1] Peter Whidden. Playing Pokemon Red with Reinforcement Learning (2023). Code: https://github.com/PWhiddy/PokemonRedExperiments.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533 (2015).

[3] Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The arcade learning environment: An evaluation platform for general agents. J. Artif. Intell. Res. 47, 253–279 (2013).

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

[5] Melrose Roderick, James MacGlashan, Stefanie Tellex. Implementing the Deep Q-Network. Res arXiv:1711.07478 [cs.LG] (2017)

[6] Stable-Baselines3 - Reliable Reinforcement Learning Implementations.

[7] Gymnasium - An API standard for reinforcement learning with a diverse collection of reference environments.

VI. Annexes

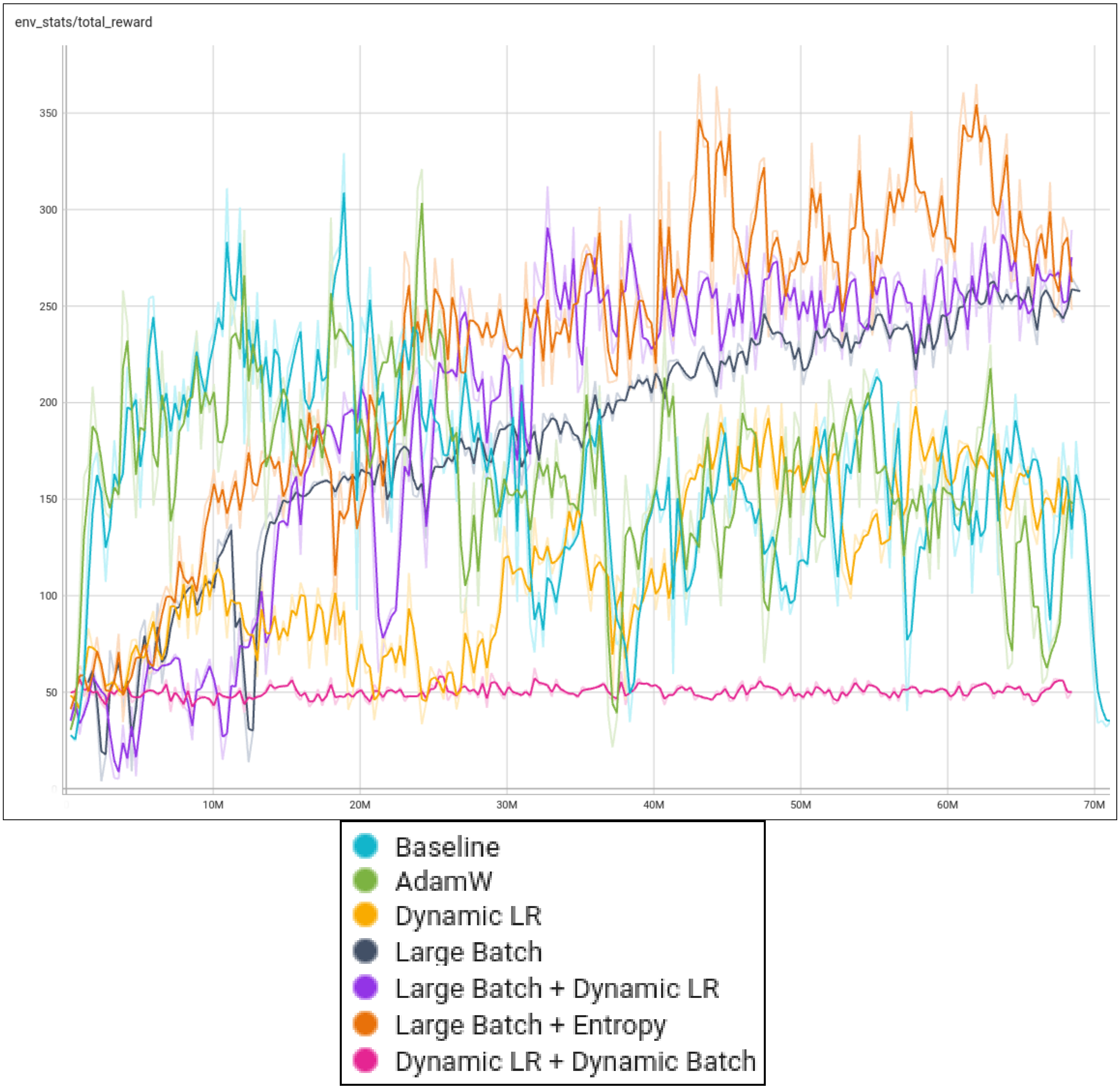

Fig. 3. Total reward during training.

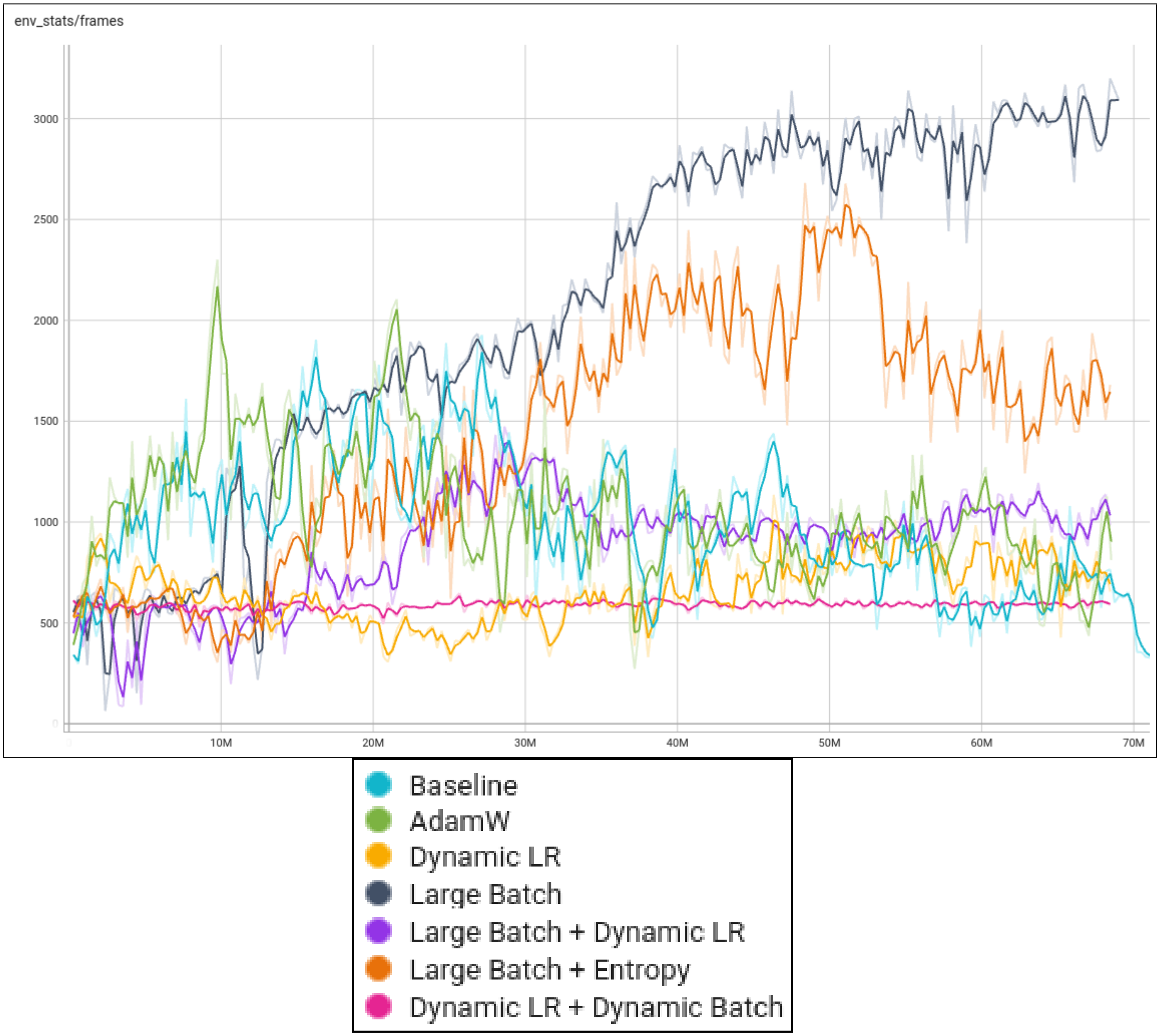

Fig. 4. Number of explored frames during training.

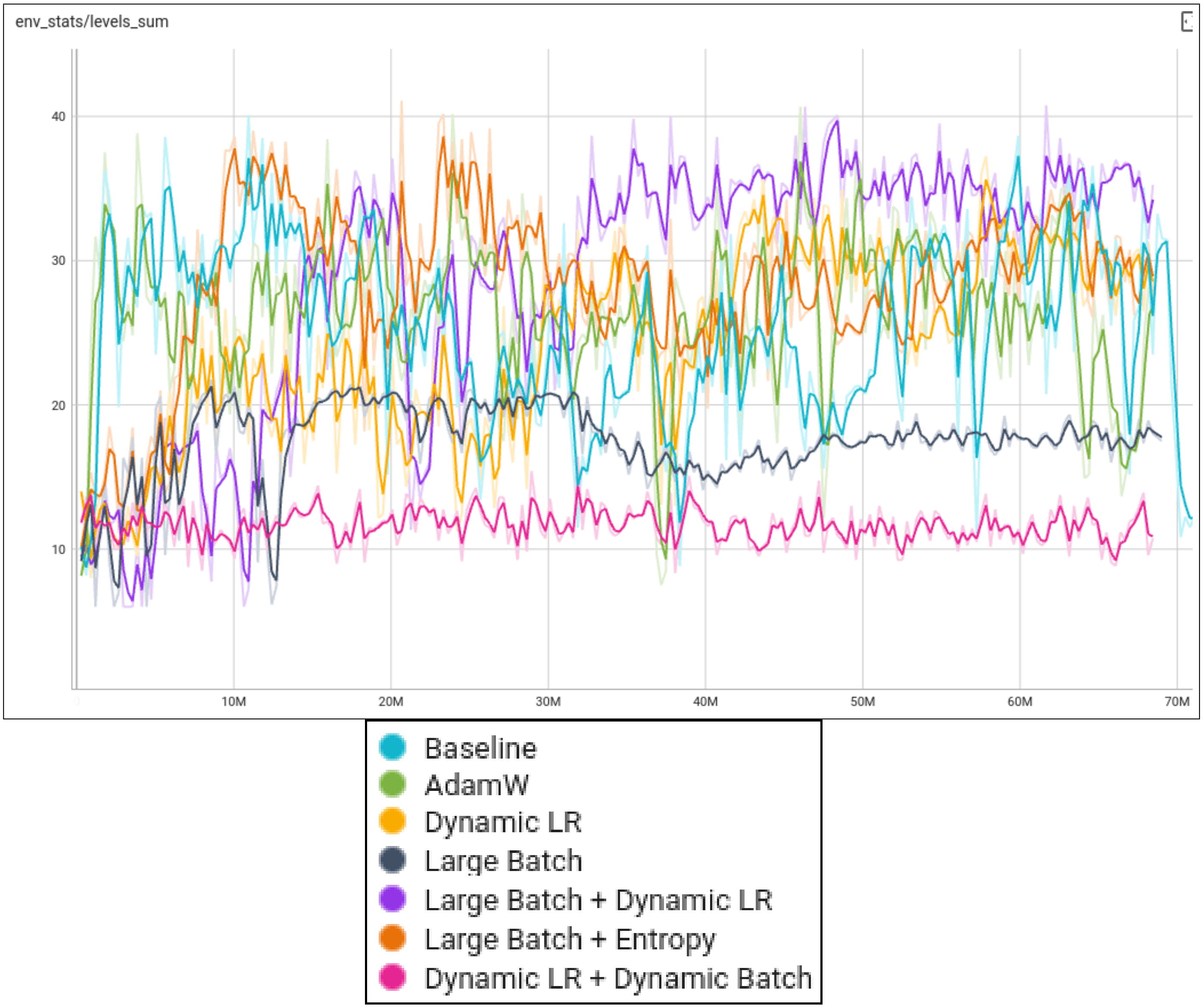

Fig. 5. Healing reward during training.

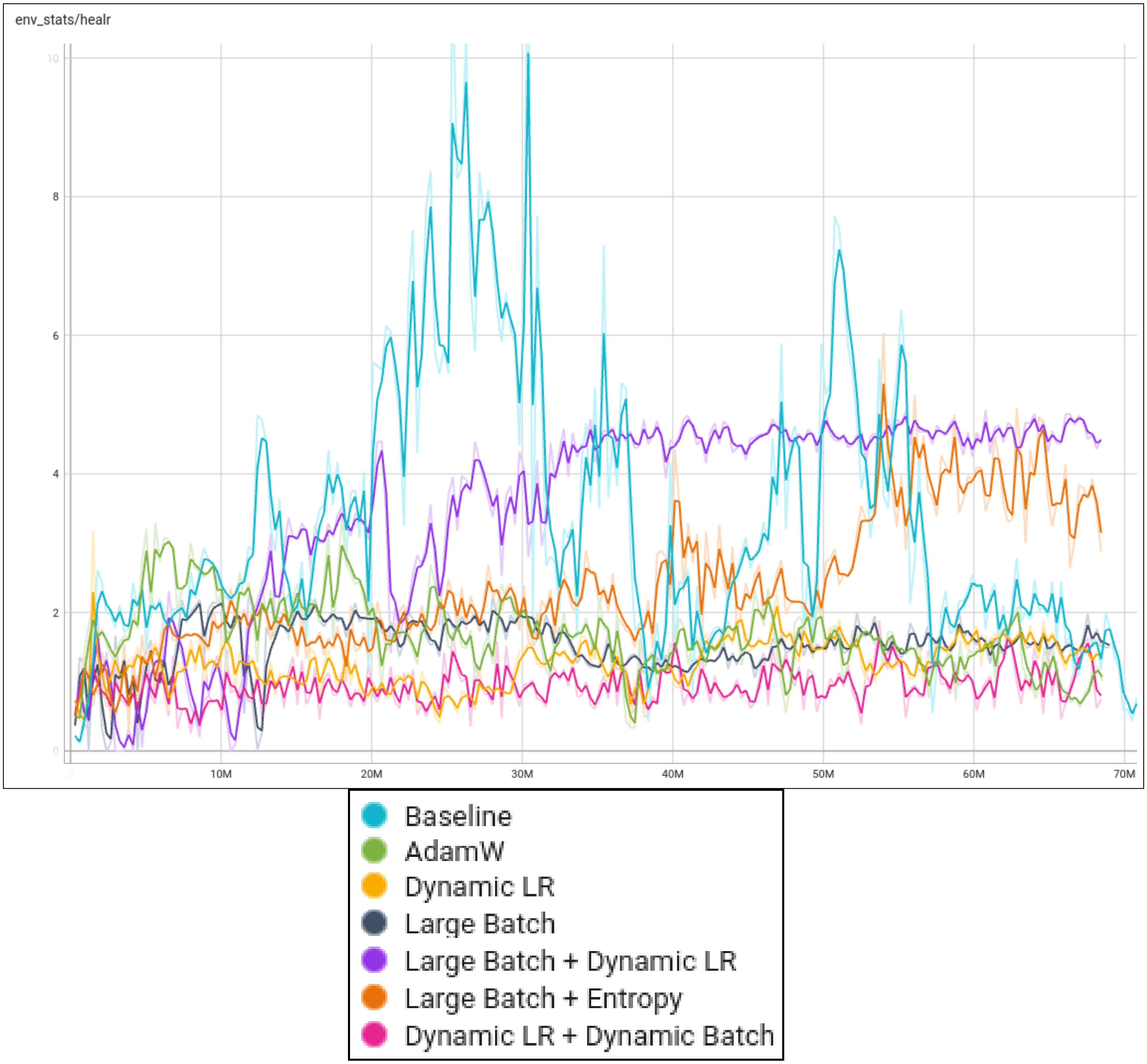

Fig. 6. Sum of levels of Pokémon party during training.

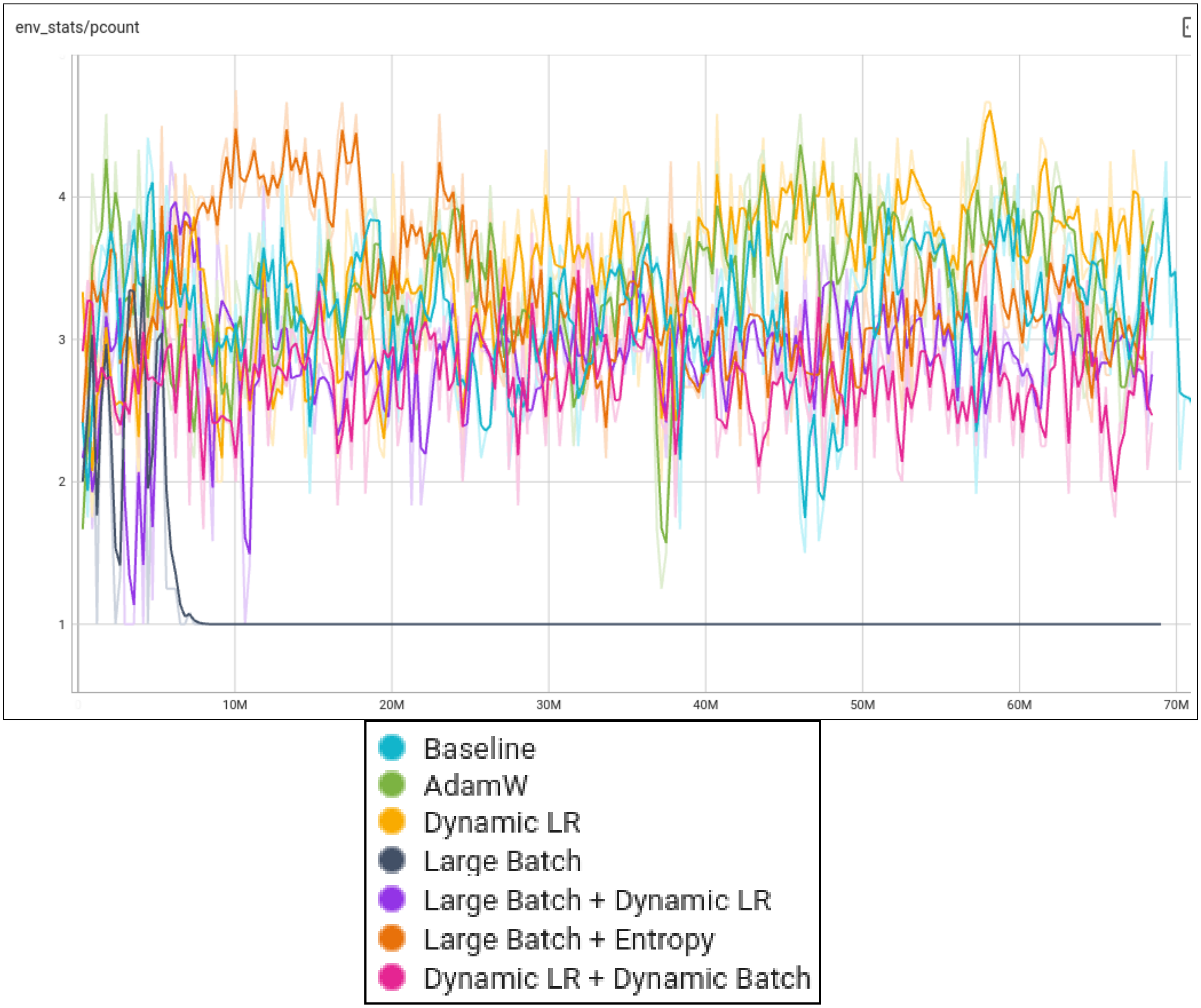

Fig. 7. Number of Pokémon in the party during training.

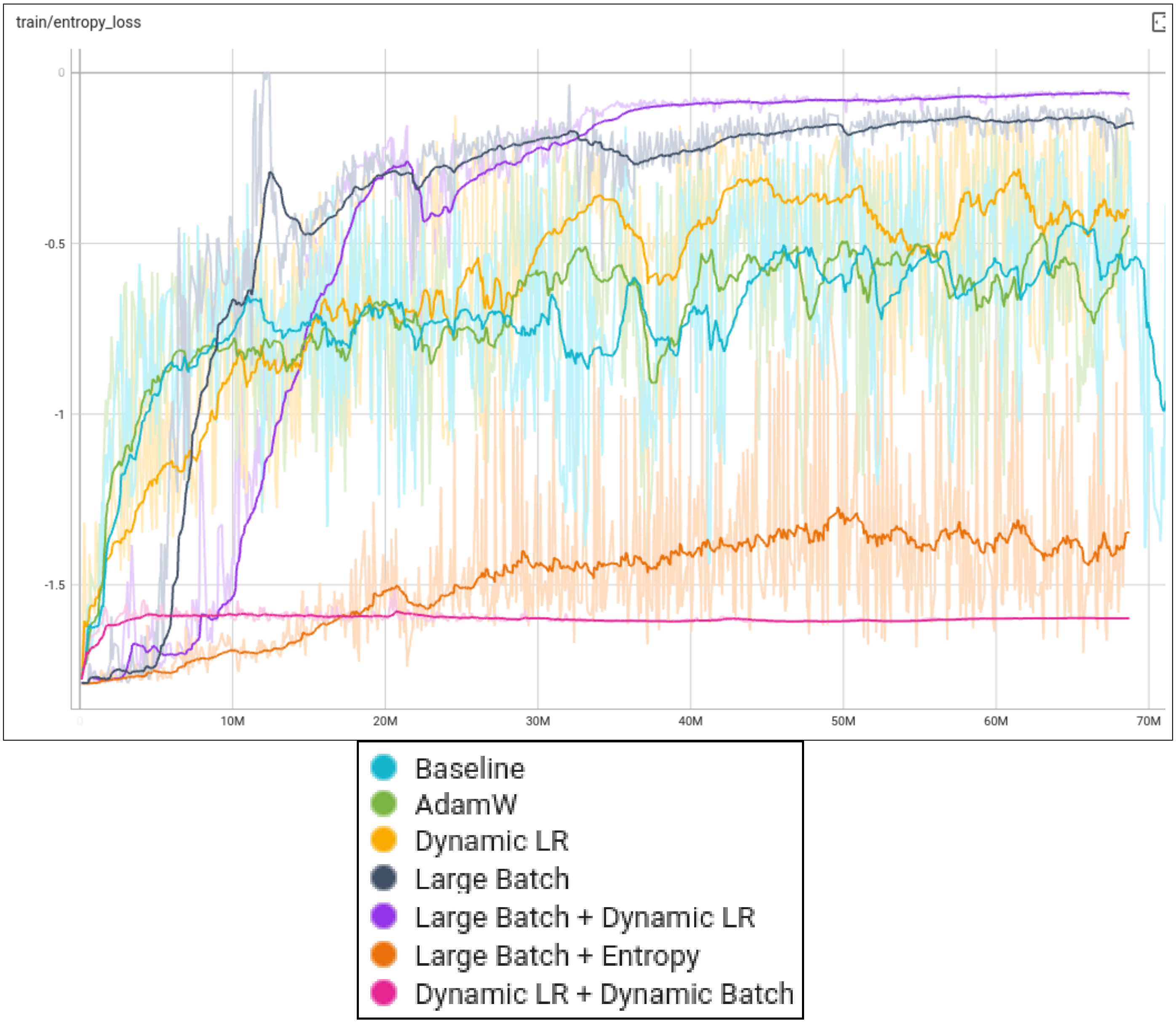

Fig. 8. Entropy function loss (smoothed).

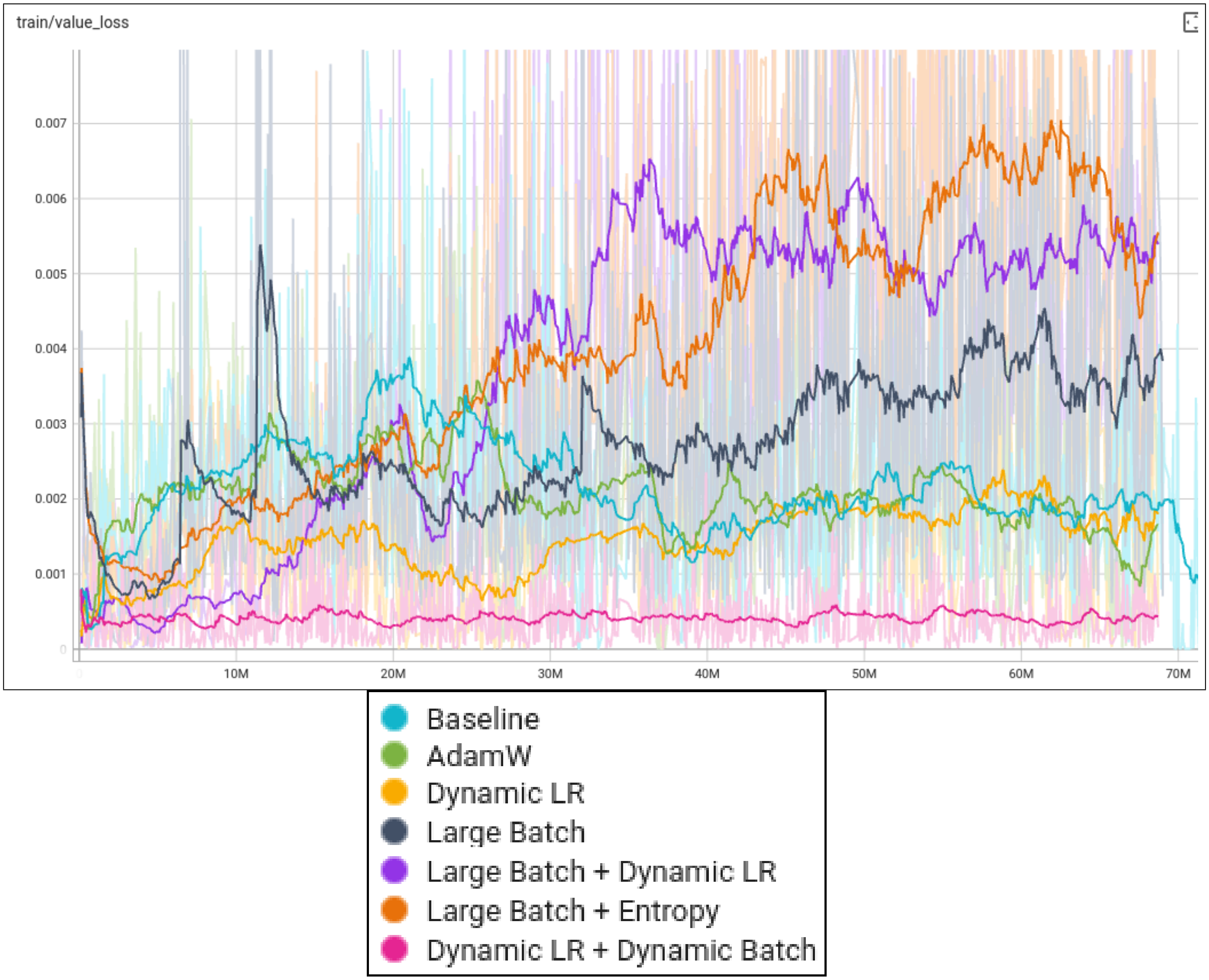

Fig. 9. Value function loss (smoothed).

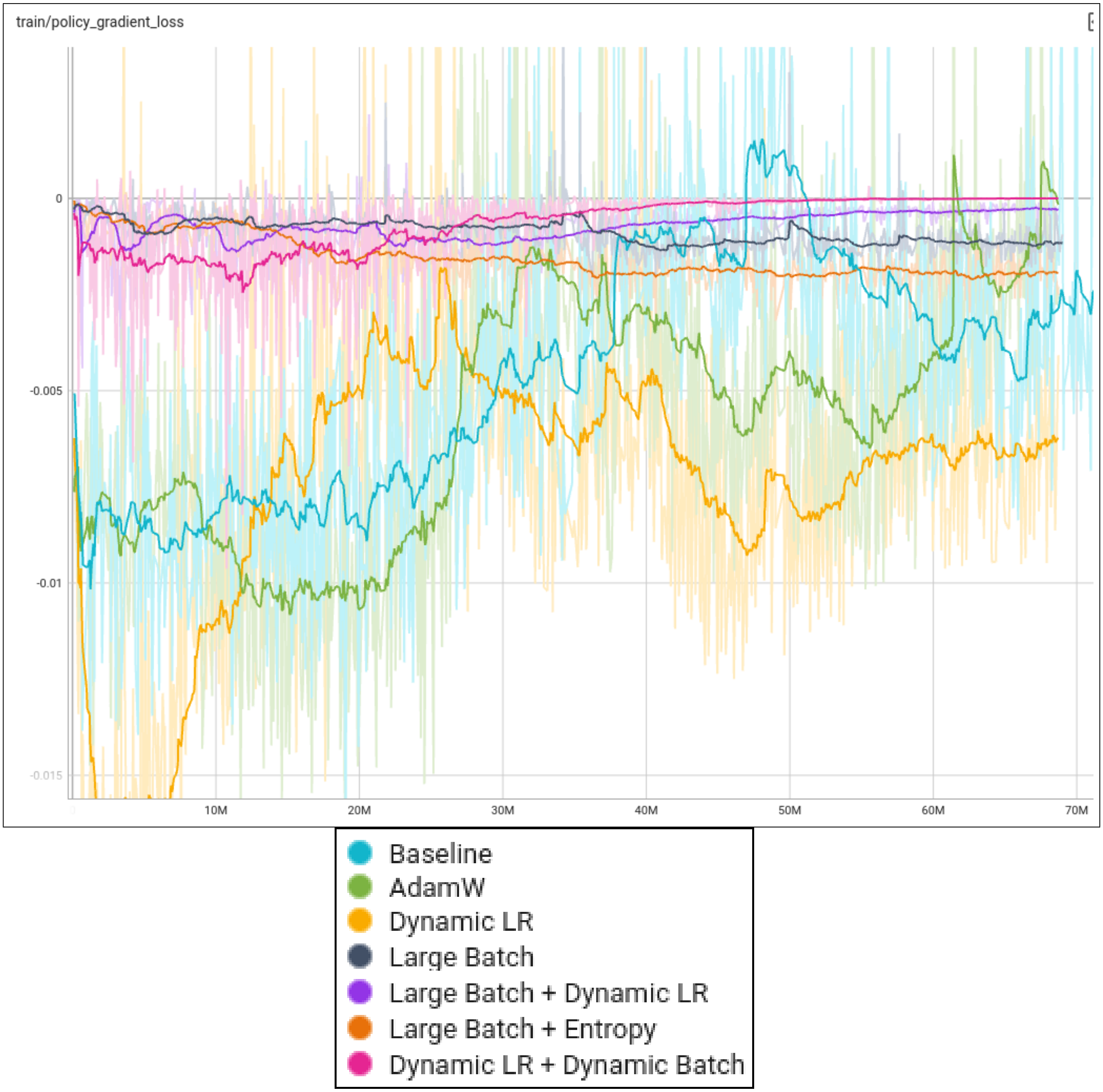

Fig. 10. Policy function loss (smoothed).