Resetting the Optimizer in Deep RL: An Empirical Study

Kavosh Asadi
Amazon

Shoham Sabach
Amazon

Rasool Fakoor
Amazon

Abstract

We focus on the task of approximating the optimal value function in deep reinforcement learning. This iterative process is comprised of approximately solving a sequence of optimization problems where the objective function can change per iteration. The common approach to solving the problem is to employ modern variants of the stochastic gradient descent algorithm such as Adam. These optimizers maintain their own internal parameters such as estimates of the first and the second moment of the gradient, and update these parameters over time. Therefore, information obtained in previous iterations is being used to solve the optimization problem in the current iteration. We hypothesize that this can contaminate the internal parameters of the employed optimizer in situations where the optimization landscape of the previous iterations is quite different from the current iteration. To hedge against this effect, a simple idea is to reset the internal parameters of the optimizer when starting a new iteration. We empirically investigate this resetting strategy by employing various optimizers in conjunction with the Rainbow algorithm. We demonstrate that this simple modification unleashes the true potential of modern optimizers, and significantly improves the performance of deep RL on the Atari benchmark.

1 Introduction

Value function optimization lies at the epicenter of large-scale deep reinforcement learning (RL). In this context, the deep RL agent is equipped with a parameterized neural network that, in conjunction with large-scale optimization, is employed to find an approximation of the optimal value function. The standard practice is to initialize the optimizer only once and at the beginning of training [1]. With each step of gradient computation, the optimizer updates its internal parameters, such as those that are needed to compute the gradient momentum or adaptive per-parameter step size [2], and updates the parameters of the network using these internal parameters.

A distinctive aspect of value function optimization is an RL-specific technique known as forward bootstrapping [3]. Whereas in more typical regression problems, the loss function is a measure of the discrepancy between a fixed target and the agent's current approximation, in RL the loss function is usually the discrepancy between the agent's current approximation and a second approximation that is obtained by performing one or multiple steps of look ahead. Thus, in contrast to typical regression problems, the regression target depends on the agent's own approximation, and so, it changes continually over the span of learning.

We first argue that in the presence of forward bootstrapping the value function optimization process can best be thought of as a sequence of optimization problems where the loss function being minimized changes per iteration. We demonstrate that in this case the objective function being minimized is comprised of two inputs: target parameters that remain fixed during each iteration, and optimization (or online) parameters that are adjusted to minimize the loss function during each iteration. The loss function can thus change arbitrarily from one iteration to the next one.

Preprint. Under review.

Leaning on this insight, we challenge the standard practice of employing the optimizer without ever resetting its internal parameters. In particular, similar to Bengio et al. [4] we ask if the internal parameters accumulated by the optimizer in the previous iterations can still be useful in computing a good update to the parameters of the neural network in the current iteration. Could it be the case that in most cases relying on gradient computations pertaining to the previous iterations is just contaminating the internal parameters, and ultimately negatively affecting the performance of the RL agent? We answer these question affirmatively through various experiments and ablations.

We then propose a simple modification where we reset the internal parameters of the optimizer at the beginning of each iteration. Stated differently, rather than only resetting the optimizer at the beginning of the entire learning process, we reset the optimizer whenever we move to the next iteration. We show that this remarkably simple augmentation significantly improves the performance of the competitive Rainbow algorithm [5] when used in conjunction with various optimizers. Most notably, under the standard Adam optimizer [6], we observe that resetting unleashes the true power of Adam and ultimately results in much better reward performance, suggesting that resetting the optimizer is a favorable choice for practitioners in value-function optimization.

2 Deep RL as Iterative Optimization

We argue that popular deep RL algorithms can be viewed as iterative optimization algorithms where the loss function being minimized can change per iteration. Note that this is true even given a fixed replay buffer. To this end, first recall that the DQN algorithm [7] decouples the learning parameters into two set of parameters: the target parameters θ, and the optimization (or online) parameters w that are adjusted at each step. DQN updates these two parameters in iterations. More specifically, each iteration is comprised of an initial step, $w^{t,0} = \theta^t$, performing multiple updates with a fixed θ^t:

$$w^{t,k+1} \leftarrow w^{t,k} + \alpha\big(r + \gamma \max_{a'} q(s', a'; \theta^t) - q(s, a; w^{t,k})\big)\nabla_\theta q(s, a; w^{t,k}) \;, \tag{1}$$

and then a synchronization of the two learning parameters, $\theta^{t+1} \leftarrow w^{t,K}$, prior to moving to the next iteration $t + 1$ where updates are performed using θ^{t+1}. Here K is a hyper-parameter whose value is commonly set to 8000 for DQN and its successors [5].

Observe that the effect of changing the optimization parameters w on $\max_{a'} q(s', a'; \theta^t)$ is ignored during gradient computation despite the fact that an implicit dependence exists due to synchronization. In fact, the update cannot be written as the gradient of any loss function that only takes a single parameter as input [8], and that the objective function being minimized here must be comprised of two separate inputs w and θ. We now define:

$$H(\theta, w) = \frac{1}{2} \sum_{\langle s,a,r,s' \rangle \in \mathcal{B}} \big(r + \gamma \max_{a'} q(s', a'; \theta) - q(s, a; w)\big)^2 \;, \tag{2}$$

where $\mathcal{B}$ is the experience replay buffer containing the agent's environmental interactions. Observe that the update (1) could be thought of as performing gradient descent using $\nabla_w H$ on a single sample $\langle s, a, r, s' \rangle$ where $\nabla_w H$ is the partial gradient of H with respect to w. Therefore, by viewing the K gradient steps as a rough approximation of exactly minimizing H with respect to the optimization parameters w, we can write this iterative process as:

$$\theta^{t+1} \approx \arg\min_w H(\theta^t, w) = \arg\min_w H_t(w) \;. \tag{3}$$

In this optimization perspective, what is typically referred to as the online parameter is updated incrementally because the objective function (2) does not lend itself into a closed-form solution in presence of non-linear function approximation such as neural networks. Notice also that in practice rather than the quadratic loss, DQN uses the slightly different Huber loss [7]. Moreover, follow-up versions of DQN use different loss functions, such as the Quantile loss [9], sample experience tuples from the buffer non-uniformly [10], use multi-step updates [11, 12], or make additional modifications [5, 13]. That said, the general structure of the algorithm follows the same trend in that it proceeds in iterations, and that the loss function being minimized can change per iteration.

3 Solving the Sequence of Optimization Problems with Adam

So far we have shown that popular deep RL algorithms could be thought of as a sequence of optimization problems where we approximately solve each iteration using first-order optimization algorithms. However, using vanilla SGD is rarely effective in the context of reinforcement learning.

For example, the original DQN paper [7] used the RMSProp optimizer [14] that maintains a running average of the second moment of the gradients to compute an adaptive per-parameter step size. Similarly, Rainbow [5] used the Adam optimizer [6], which could be thought of as the momentum-based [15] version of RMSProp. More formally, suppose that our goal is to minimize a certain objective function $J(w)$. Then, after computing a stochastic gradient $g^i = \nabla J(w^i)$, Adam proceeds by computing a running average of the first and the second moments of the gradient as follows:

$$m^i \leftarrow \beta_1 m^{i-1} + (1-\beta_1) g^i \ , \qquad \text{and} \qquad v^i \leftarrow \beta_2 v^{i-1} + (1-\beta_2)(g^i)^2 \ ,$$

where $(g^i)^2$ applies the square function to g^i element-wise. Notice that $m^0 = 0$ and $v^0 = 0$, and so the estimates are biased towards 0 in the first few steps. Define the function $\text{power}(x, y) = x^y$. To remove this bias, Adam performs a debiasing step as follows:

$$m^i \leftarrow m^i / \big(1 - \text{power}(\beta_1, i)\big) \ , \qquad \text{and} \qquad v^i \leftarrow v^i / \big(1 - \text{power}(\beta_2, i)\big) \ ,$$

before finally updating the network parameters: $w^i \leftarrow w^{i-1} - \alpha m^i / (\sqrt{v^i} + \epsilon)$, with a small hyper-parameter ϵ that prevents division by zero. We now present the pseudocode of DQN with the Adam Optimizer. Notice that in this pseudocode we do not present the pieces related to the RL agent's environmental interactions to primarily highlight pieces pertaining to how θ and w are updated.

Algorithm 1 pseudocode for DQN with (resetting) Adam

Input: $\theta^0,\ T,\ K$
Input: $\beta_1, \beta_2, \alpha, \epsilon$ ▷ Set Adam's hyper-parameters
~~$i = 0, m^0 = 0, v^0 = 0$~~ ▷ Initialize Adam's internal parameters
for $t = 0$ **to** $T - 1$ **do**
  $w^{t,0} \leftarrow \theta^t$
  $i = 0, m^0 = 0, v^0 = 0$ ▷ Reset Adam's internal parameters
  for $k = 0$ **to** $K - 1$ **do**
    $g^i \leftarrow \nabla H_t(w^{t,k})$
    $i \leftarrow i + 1$
    $m^i \leftarrow \beta_1 m^{i-1} + (1-\beta_1) g^i$ and $v^i \leftarrow \beta_2 v^{i-1} + (1-\beta_2)(g^i)^2$
    $m^i \leftarrow m^i / (1 - \text{power}(\beta_1, i))$ and $v^i \leftarrow v^i / (1 - \text{power}(\beta_2, i))$ ▷ Adam's debiasing
    $w^{t,k+1} \leftarrow w^{t,k} - \alpha m^i / (\sqrt{v^i} + \epsilon)$
  end for
  $\theta^{t+1} \leftarrow w^{t,K}$
end for
Return θ^T

From the pseudocode above, first notice that if the counter i is not reset, then the debiasing quantities $1 - \text{power}(\beta_1, i)$ and $1 - \text{power}(\beta_2, i)$ quickly go to 1 and so the debiasing steps will have minimal effect on the overall update, if at all. In absence of resetting, the optimization strategy could then be thought of as initializing the first (m) and the second (v) moment estimates at each iteration t by whatever their values were at the end of the previous iteration $t - 1$. This seems like an arbitrary choice, one that deviates from design decisions that were made by Adam [6].

This choice makes some sense if the optimization landscape in the previous iterations closely resembles that of the current iteration, but it is not clear that this will always be the case in deep RL. In cases where this is not the case, the agent can waste many gradient updates just to "unlearn" the effects of the previous iterations on the internal parameters m and v. This is a contamination effect that plagues RL optimization as we later demonstrate. Fortunately, there is a remarkably easy fix for this problem. Note, that the Adam optimizer is fully equipped to deal with reinitializing the moment estimates to 0, because the introduced bias due to resetting is dealt with adequately by performing the debiasing step. We next show that resetting can hedge against contamination. Finally, note that this is an inexpensive and convenient fix in that it adds no computational cost to the baseline algorithm, nor does it add any new hyper-parameter to the algorithm.

4 Experiments

For our case study, we chose the standard Atari benchmark [9], and also chose the popular Rainbow algorithm [5], which fruitfully combined a couple of important techniques in learning the value

function. This combination resulted in Rainbow being the state-of-the-art algorithm, one that still remains a competitive baseline. We used the most popular implementation of Rainbow, namely the one in the Dopamine framework [1], and followed exactly the same experimental protocol as the Dopamine did. Whenever we change a parameter from the Dopamine protocol, we will explicitly mention the modification.

Note that other than the popular Dopamine baseline [1], we checked the second most popular implementations of DQN and Rainbow on Github, namely Github.com/devsisters/DQN-tensorflow and Github.com/Kaixhin/Rainbow, and found that resetting is absent in these implementations as well. Lack of resetting is thus not an oversight of the Dopamine implementation, but the standard practice in numerous Deep RL implementations.

It is also important to note that our primary emphasis in this paper is not to develop the next state-of-the-art deep RL agent. Instead, we aim to highlight crucial details pertaining to the optimizer that are either overlooked, or treated as an afterthought, when training deep RL. We believe that gaining a better understanding of the internal behavior of modern optimizers in the context of RL can pave the way for designing more principled approaches to deep-RL optimization.

4.1 Rainbow with Resetting Adam

In the first part, our desire is to compare the behavior of Rainbow with and without resetting in conjunction with its default optimizer, namely Adam. Here we present our experiments on a select set of Atari games. Note that we will present comprehensive results on the full set of 55 Atari games later on, and that at this point our goal is to keep the number of games small so performing ablation studies remains manageable. We also ran these experiments on a single seed, but later on we report a comprehensive set of experiments where we used 10 different seeds.

For this experiment, as well as the next ablation studies, we randomly chose 12 Atari games: 'Amidar', 'Asterix', 'BeamRider', 'CrazyClimber', 'DemonAttack', 'Gopher', 'Phoenix', 'Zaxxon','Breakout', 'Hero', 'Seaquest', and 'Kangaroo'. A key hyper-parameter in the implementation of Rainbow is the number of gradient updates per iteration (K) whose default value is 8000 in most deep RL papers [7, 16, 5]. We are interested to see the impact of this hyper-parameter on the performance of Rainbow with and without reset. We now present our results for $K = 8000$ in Figure 1.

From this figure, we can see that with $K = 8000$ resetting the optimizer is usually competitive with, and sometimes better than, not resetting it. But we are interested in understanding the impact of changing K, and in particular when we decrease this value. Smaller K values provide an even rougher approximation of solving the optimization problem the agent is faced with at each iteration. We hypothesize that because the contamination effect described before is now eliminated with resetting, the agent does not need to unlearn the internal parameters, and so smaller values of K will perform better.

Before presenting this result, notice that regardless of the value of K we perform the same number of gradient updates to the optimization (online) network across training. Stated differently, a smaller value of K will correspond to a larger value T in Algorithm 1 because for the sake of a fair comparison we always keep the multiplication of $K \times T$ fixed for all K values in Rainbow with or without resetting. We now repeat the above experiment with $K = 1000$ and present the result in Figure 2.

Observe that with $K = 1000$, resetting the Adam optimizer under the Rainbow agent results in a significantly better performance on most of the 12 randomly-chosen games. This is because the contamination effect is no longer present when resetting. In contrast, when resetting is not performed, the agent in a sense wastes the first couple steps of the optimization at each iteration on unlearning the internal parameters accumulated from the previous iterations. With resetting, however, the agent actually works even better with small K because fewer number of steps are needed to reasonably solve each iteration.

We now repeat this experiment for multiple other values of K, so taken together we use the value of K from the list $[1, 500, 1000, 2000, 4000, 6000, 8000]$ to get a more comprehensive understanding of the effect of K on the performance of each case. Moreover, akin to the standard practice in the literature [17], rather than looking at individual learning curves per game, we look at the human-normalized performance of the agents on all games, namely: $\frac{\text{Score}_{\text{Agent}} - \text{Score}_{\text{Random}}}{\text{Score}_{\text{Human}} - \text{Score}_{\text{Random}}}$. To compare the

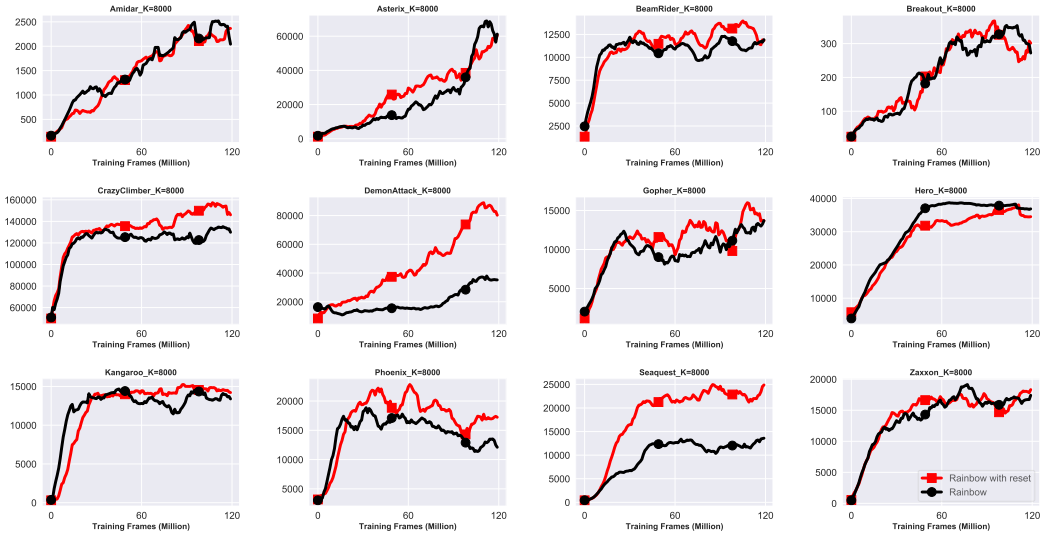

Figure 1: Performance of Rainbow with and without resetting the Adam optimizer and with a fixed value of $K = 8000$ on 12 randomly-chosen Atari games. Overall, resetting the Adam optimizer does not result in performance degradation for this value of K.

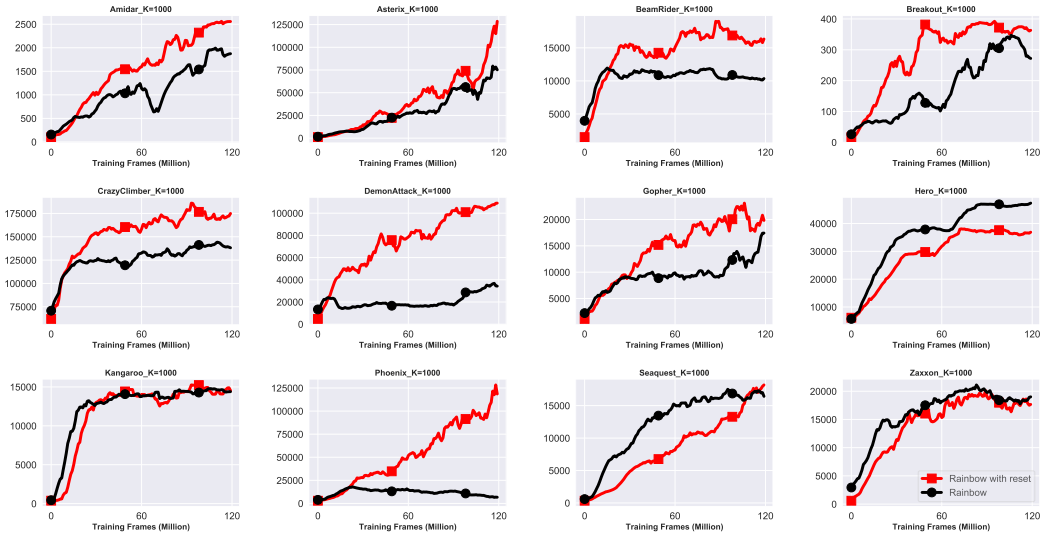

Figure 2: Performance of Rainbow with and without resetting and $K = 1000$. Observe that with a smaller value of $K = 1000$, resetting can more efficiently harness Adam.

performance of the two agents across all 12 games, we compute the median of this number across the games and present these results for each value of K in Figure 3.

From Figure 3 we can see very clearly that Rainbow with resetting is dominating Rainbow without resetting for all values of K but $K = 1$. Notice that in the extreme case of $K = 1$ the Adam optimizer is in effect not accumulating any internal parameters because we reset at each step. Therefore, in essence, when $K = 1$ the update would be more akin to vanilla gradient descent and thus not effective at all. However, with larger values of K, it is clearly better to reset the optimizer at each iteration. Another interesting trend is that in fact when we reset the optimizer, smaller values of K than the default 8000 become the most competitive. This is in contrast to Rainbow without reset where the value of K does not influence the final performance.

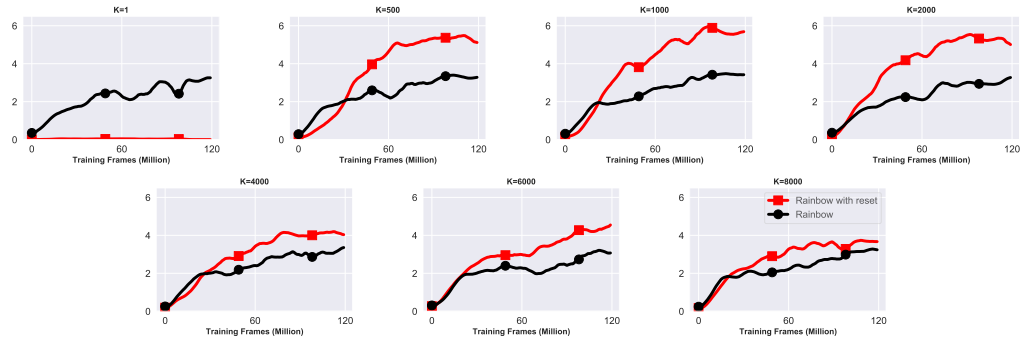

Figure 3: A comparison between Rainbow with and without resetting on the 12 Atari games for different values of K. The Y-axis is the human-normalized median. Observe that for all but $K = 1$ it is clearly better to reset the Adam optimizer. Notice, also that the best performance with resetting is obtained by values of K that are much smaller than the default 8000 used in numerous papers.

We further distill the performance of each combination of with or without resetting and value of K by computing the area under the curve in plots illustrated in Figure 3. We present this result in Figure 4.

We clearly see that resetting can better harness the power of Adam and we also see that an inverted-U shape manifests itself with the best performance achieved for intermediate values of K. In the case of no-resetting notice that performance is basically flat as a function of K.

Also, to further contextualize our results, we add another baseline to this figure where we reset the optimizer at random steps. Whereas in the original resetting case, we reset the optimize right at the beginning of each iteration, in this case we just randomly reset the optimizer after each update to the optimization parameter w, and the value of K determines the probability ($\frac{1}{K}$) with which we reset the Adam optimizer. With this choice, on expectation we have the same number of resets performed by the original per-iteration resetting, but now resetting is stochastically performed at any given step. We present this result in Figure 4 where this additional baseline is indicated in blue.

We see that even randomly resetting the Adam optimizer provides some benefits relative to the Adam with no resetting. However, performance is improved most by deterministically resetting the optimizer at the beginning of each iteration. Please see the Appendix for individual learning curves.

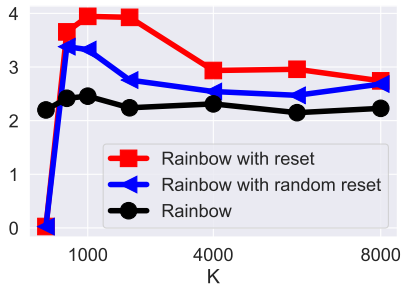

Figure 4: The positive effect of resetting the Adam optimizer in the context of Rainbow. Resetting the optimizer at the beginning of each iteration (red) yields the best performance. Some improvement is also observed even when resetting with probability $\frac{1}{K}$ after each update of the optimization (online) parameter w (blue). Rainbow without resetting (black) is least competitive.

4.2 Experiments with Other Optimization Algorithms

In the previous subsection, we demonstrated that we can significantly boost the reward performance by employing the simple resetting idea. Is this effect specific to the Adam optimizer or can we make a broader conclusion by showing that the same positive effect manifests itself when we use the Rainbow agent in conjunction with alternative optimization algorithms?

To answer this question, we first investigate the setting where we use the RMSProp algorithm. First presented in a lecture note by Hinton [14], this optimization algorithm can be thought of as a reduction of Adam, where we use the gradient itself instead of the first moment estimate. In other words, RMSProp corresponds to Adam with $\beta_1 = 0$ and also skipping the debiasing step for the second moment estimate. This optimizer was also employed in the original DQN algorithm [7].

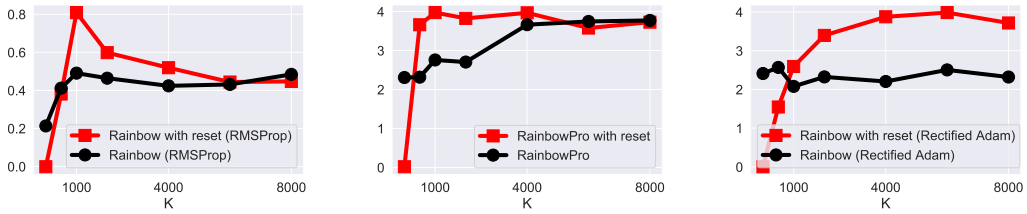

Figure 5: A Comparison between the performance of Rainbow with and without resetting when used in conjunction with various optimizers. In all 3 cases, namely when using RMSProp (left), using proximal update (center), and using Rectified Adam (right), performance is improved by resetting the optimizer at the beginning of each iteration.

Similar to Figure 4, we ran Rainbow RMSProp with and without resetting on the same 12 games, computed the human-normalized median, and then distilled the learning curves by computing their area under the curve for each value of K. Our result are presented in Figure 5 (left).

Notice that while overall we see a performance degradation when we move from Adam to RMSProp, we still observe that resetting the optimizer per iteration can provide a positive effect in terms of reward performance.

We next look at the proximal optimizer introduced by Asadi et al. [18], where they endow Rainbow with proximal updates. This loss ensures that at each iteration the optimization parameter gravitates towards the previous target parameter. Again, we employ resetting in the context of their introduced algorithm, namely Rainbow Pro. The result is presented in Figure 5 (center).

We see a similar effect whereby resetting the optimizer at each iteration can also improve the Rainbow Pro algorithm and make the overall performance less sensitive to the choice of K.

To conclude this section, we next consider the Rectified Adam optimizer introduced by Liu et al. [19]. This recent variant of Adam is especially interesting because it controls the variance of the Adam optimizer early in training by employing step-size warm up in conjunction with a rectifying technique. With resetting, the optimization algorithm basically starts from scratch many times, and so the rectification technique of Liu et al. [19] can be crucial in further improving performance. We present this results in Figure 5 (right).

Observe that equipping the agent with resetting is having a positive impact on the Rectified Adam optimizer akin to what we showed in the case of Adam, RMSprop, and proximal updates. Overall, it is quite clear that resetting the optimizer can nicely hedge against the gradient contamination effect and ultimately yield superior performance in a general sense.

4.3 Comprehensive Experiments on 55 Atari Games

So far we focused on performing smaller ablation studies on a subset of Atari games using only a single seed, but to strengthen our claims we now desire to perform a comprehensive experiment on 55 Atari games using 10 seeds.

In this case, our goal is to evaluate 3 different agents. As our benchmark, we have the original Rainbow algorithm with the Adam optimizer, without resetting, and the original value of $K = 8000$. We refer to this as the standard Rainbow agent. Our next agent is Rainbow with the Adam optimizer and resetting. In light of our results from Figure 4 we chose the value of $K = 1000$ for this agent. Finally, we have Rainbow with Rectified Adam and resetting. Similarly, and in light of Figure 5, we chose the value of $K = 4000$ for this agent. In Figure 6, we present the mean and median performance of the 3 agents on the 55 Atari games, where we ran each combination of game and agent for 10 independent random seeds.

From Figure 6, we can see that resetting the Adam optimizer is clearly boosting the performance of the Rainbow agent in terms of both the mean and the median performance of the agent across all games. This trend is also valid when using the Rectified Adam optimizer and again demonstrates that the contamination effect is hedged against by employing this simple resetting strategy. Moreover, we present another comparison in terms of the per-game asymptotic improvements of the two resetting agents against the Rainbow agent in Figure 7. We again see that asymptotic improvement manifests

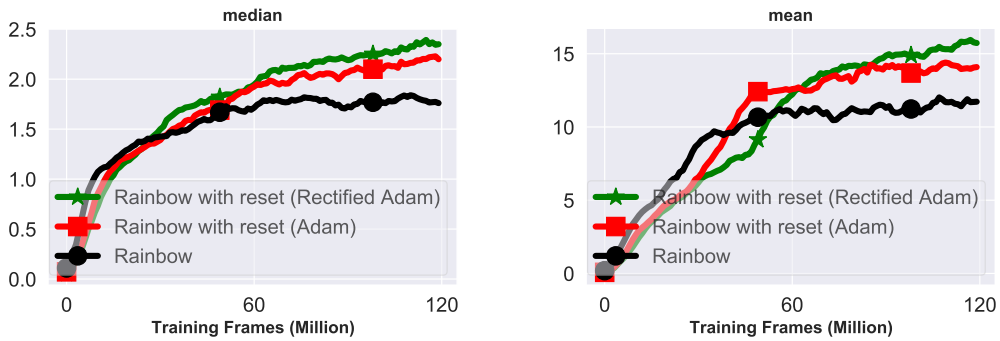

Figure 6: A comparison between Rainbow without reset (black) against two resetting agents, namely Rainbow with restting Adam (red) and Rainbow with resetting Rectified Adam (green). Results are averaged over 10 random seeds, where we human-normalize the results and take their median (left) and mean (right) over 55 games. Resetting the optimizer is clearly favorable.

itself when resetting. It is, therefore, quite clear that resetting is more favorable approach, and that it could be considered as a better option than not resetting.

4.4 Continuous Control

To broaden the scope of our results we conducted additional experiments in continuous control environments using MuJoCo physics simulator [20] on five standard benchmark tasks. Specifically, in this section, we used Soft Actor-Critic (SAC) [21], an off-policy actor-critic algorithm with a stochastic policy. It is worth noting that SAC and Rainbow/DQN differ in several aspects. Chief among these differences is that SAC directly learns the policy by parameterizing it using a neural network. More importantly, SAC utilizes soft target updates (also known as Polyak updates) to update the target parameter, while Rainbow/DQN uses hard target updates (see Section 2 for more details). So it is less clear when to reset the optimizer. We show our results in the Appendix, where the Adam optimizer was reset for both the critic and actor networks every 5000 steps.

It is evident from these results that resetting the Adam optimizer is not as helpful with SAC as it is with Rainbow, but there is still a small positive effect. Again, this is because it is not clear when to reset when Polyak updates are utilized. We leave further exploration of resetting in continuous control to future work.

5 Related Work

In this paper, we showed that common approaches to value function optimization could be viewed as iterative optimization algorithms. This view is occasionally discussed in previous work [22] but in different contexts, perhaps earliest in Fitted Value Iteration [23, 24], Fitted Q Iteration [25, 26], and related approximate dynamic-programming algorithms [27, 28, 29]. More recently, Dabney et al. [30] presented the concept of the value improvement path where they argued that even in the single-task setting with stationary environments, the RL agent is still faced with a sequence of problems, and that the representation-learning process can benefit from looking at the entire sequence. This view was further explored to show that ignoring this non-stationary aspect can lead to capacity [31] and plasticity loss [32]. These works, however, do not consider the effect of RL non-stationarity on the internal parameters of the optimizer. Notice that this kind of non-stationarity is due to the solution not the environment, and stands in contrast to the non-stationary setting that arises when the MDP reward and transitions change, a setting that is well-explored [33, 34, 35, 36, 37].

To hedge against the contamination effect, we proposed to reset the optimizer per iteration. Resetting the optimizer is a well-established idea in the optimization literature, and can be found for example in a paper by Nesterov [38]. Adaptive versions of resetting are also common and perform well in practice [39, 40]. Restarting the step-size (learning rate) also has some precedence in deep learning [41, 42, 43]. See also the standard book on large-scale optimization [44] as well as a recent review of this topic in the context of optimization [45].

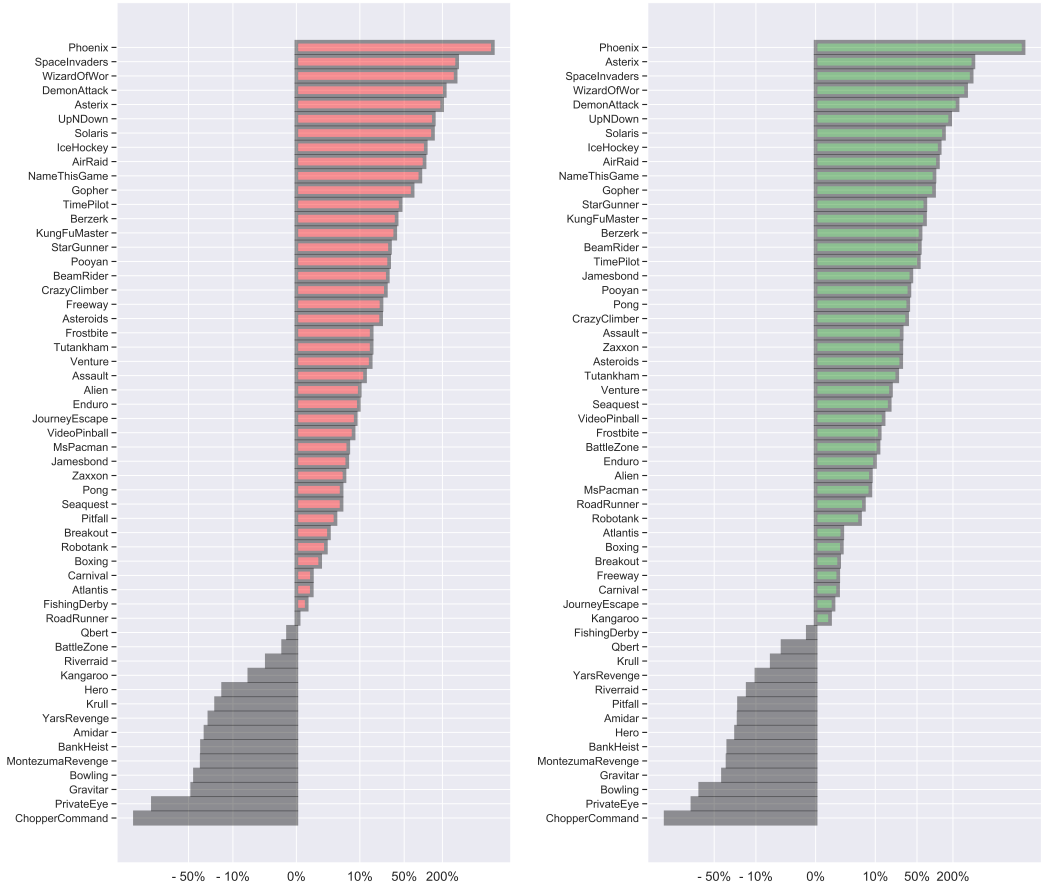

Figure 7: A comparison between the asymptotic performance improvement of the resetting agents against Rainbow without resetting. In particular, we show improvements of Rainbow with resetting Adam against Rainbow (left), and Rainbow with resetting Rectified Adam against Rainbow (right).

Resetting RL ingredients other than the optimizer is another line of work that is related to our paper. Earliest work is due to Anderson which restarts some network parameters when the magnitude of error is unusually large [46]. Another example is Nishkin et al. [47] who proposed to reset parts of the network weights to deal with an effect named primacy bias whereby the agent exhibits overfitting to its first few environmental interactions. Primacy bias is also attributed to the use of ReLU activation in the network architecture [48]. Perhaps the most similar work to our paper is that of Bengio et al. [4] who notices the harmful effect of contamination, and proposes a solution based on computing the Hessian and a Taylor expansion, one that is unfortunately too expensive to be applied in conjunction with deep networks owing to their large number of weights.

6 Conclusion

In this paper we argued that value-function optimization could best be thought of as solving for a sequence of optimization problems where the loss function can change per iteration. We questioned the standard practice of using modern optimization algorithms for solving this sequence of problems while being agnostic to changes in the loss function. We showed that this can result into a contamination effect, which is easily and cheaply hedged against by resetting the internal parameters of the optimization algorithm.

A more general conclusion of our work is that, to fruitfully apply optimization techniques to deep RL, it is imperative to have a deeper understanding of the internal process of these optimization algorithms. Simply applying optimization techniques to RL might not always be effective because these optimization techniques are often designed for different settings than RL, and are also expected to operate under different assumptions than ones that are typically made in RL.

References

[1] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018.

[2] Sebastian Ruder. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747, 2016.

[3] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT press, 2018.

[4] Emmanuel Bengio, Joelle Pineau, and Doina Precup. Correcting momentum in temporal difference learning, 2021.

[5] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In AAAI Conference on Artificial Intelligence, 2018.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In International Conference on Learning Representations, 2015.

[7] 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.

[8] Hamid Reza Maei. Gradient temporal-difference learning algorithms. 2011.

[9] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In International conference on machine learning, pages 449–458. PMLR, 2017.

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

[11] Kristopher De Asis, J Hernandez-Garcia, G Holland, and Richard Sutton. Multi-step reinforcement learning: A unifying algorithm. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018.

[12] Yunhao Tang, Remi Munos, Mark Rowland, Bernardo Avila Pires, Will Dabney, and Marc Bellemare. The nature of temporal difference errors in multi-step distributional reinforcement learning. Advances in Neural Information Processing Systems, 35:30265–30276, 2022.

[13] Kavosh Asadi, Neev Parikh, Ronald E Parr, George D Konidaris, and Michael L Littman. Deep radial-basis value functions for continuous control. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, pages 6696–6704, 2021.

[14] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2):26–31, 2012.

[15] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. Ussr computational mathematics and mathematical physics, 4(5):1–17, 1964.

[16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence, volume 30, 2016.

[17] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In International conference on machine learning, pages 1995–2003. PMLR, 2016.

[18] Kavosh Asadi, Rasool Fakoor, Omer Gottesman, Taesup Kim, Michael Littman, and Alexander J Smola. Faster deep reinforcement learning with slower online network. Advances in Neural Information Processing Systems, 35:19944–19955, 2022.

[19] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. arXiv preprint arXiv:1908.03265, 2019.

[20] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 5026–5033. IEEE, 2012.

[21] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 1861–1870. PMLR, 10–15 Jul 2018.

[22] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In Learning for Dynamics and Control, pages 486–489. PMLR, 2020.

[23] Geoffrey J Gordon. Stable function approximation in dynamic programming. In Machine learning proceedings 1995, pages 261–268. Elsevier, 1995.

[24] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. Journal of Machine Learning Research, 6, 2005.

[25] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16, pages 317–328. Springer, 2005.

[26] András Antos, Csaba Szepesvári, and Rémi Munos. Fitted q-iteration in continuous action-space mdps. Advances in neural information processing systems, 20, 2007.

[27] Dimitri Bertsekas and John N Tsitsiklis. Neuro-dynamic programming. Athena Scientific, 1996.

[28] Michail G Lagoudakis and Ronald Parr. Least-squares policy iteration. The Journal of Machine Learning Research, 4:1107–1149, 2003.

[29] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. Reinforcement learning: State-of-the-art, pages 45–73, 2012.

[30] Will Dabney, André Barreto, Mark Rowland, Robert Dadashi, John Quan, Marc G Bellemare, and David Silver. The value-improvement path: Towards better representations for reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, pages 7160–7168, 2021.

[31] Clare Lyle, Mark Rowland, and Will Dabney. Understanding and preventing capacity loss in reinforcement learning. arXiv preprint arXiv:2204.09560, 2022.

[32] Evgenii Nikishin, Junhyuk Oh, Georg Ostrovski, Clare Lyle, Razvan Pascanu, Will Dabney, and Andre Barreto. Deep reinforcement learning with plasticity injection. In Workshop on Reincarnating Reinforcement Learning at ICLR 2023.

[33] Sindhu Padakandla, Prabuchandran KJ, and Shalabh Bhatnagar. Reinforcement learning algorithm for non-stationary environments. Applied Intelligence, 50:3590–3606, 2020.

[34] Erwan Lecarpentier, David Abel, Kavosh Asadi, Yuu Jinnai, Emmanuel Rachelson, and Michael L Littman. Lipschitz lifelong reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, pages 8270–8278, 2021.

[35] Yash Chandak. Reinforcement Learning for Non-stationary problems. PhD thesis, University of Massachusetts Amherst, 2022.

[36] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. Journal of Artificial Intelligence Research, 75:1401–1476, 2022.

[37] Jelena Luketina, Sebastian Flennerhag, Yannick Schroecker, David Abel, Tom Zahavy, and Satinder Singh. Meta-gradients in non-stationary environments. In Conference on Lifelong Learning Agents, pages 886–901. PMLR, 2022.

[38] Yurii Nesterov. Gradient methods for minimizing composite functions. Mathematical Programming, 140:125 – 161, 2012.

[39] Brendan O'Donoghue and Emmanuel J. Candès. Adaptive restart for accelerated gradient schemes. Foundations of Computational Mathematics, 15:715–732, 2012.

[40] Bao Wang, Tan M. Nguyen, Andrea L. Bertozzi, Richard G. Baraniuk, and Stanley J. Osher. Scheduled restart momentum for accelerated stochastic gradient descent, 2020.

[41] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. ArXiv, abs/1711.05101, 2017.

[42] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017.

[43] Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. arXiv preprint arXiv:1810.13243, 2018.

[44] Amir Beck. First-order methods in optimization. SIAM, 2017.

[45] Sebastian Pokutta. Restarting algorithms: Sometimes there is free lunch. In Integration of AI and OR Techniques in Constraint Programming, 2020.

[46] Charles Anderson. Q-learning with hidden-unit restarting. Advances in Neural Information Processing Systems, 5, 1992.

[47] Evgenii Nikishin, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. The primacy bias in deep reinforcement learning. In International Conference on Machine Learning, pages 16828–16847. PMLR, 2022.

[48] Zaheer Abbas, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C Machado. Loss of plasticity in continual deep reinforcement learning. arXiv preprint arXiv:2303.07507, 2023.

7 Appendix

7.1 Complete Results from Section 4.1

We first show individual learning curves for Rainbow (Adam optimizer) with and without resetting and different values of K.

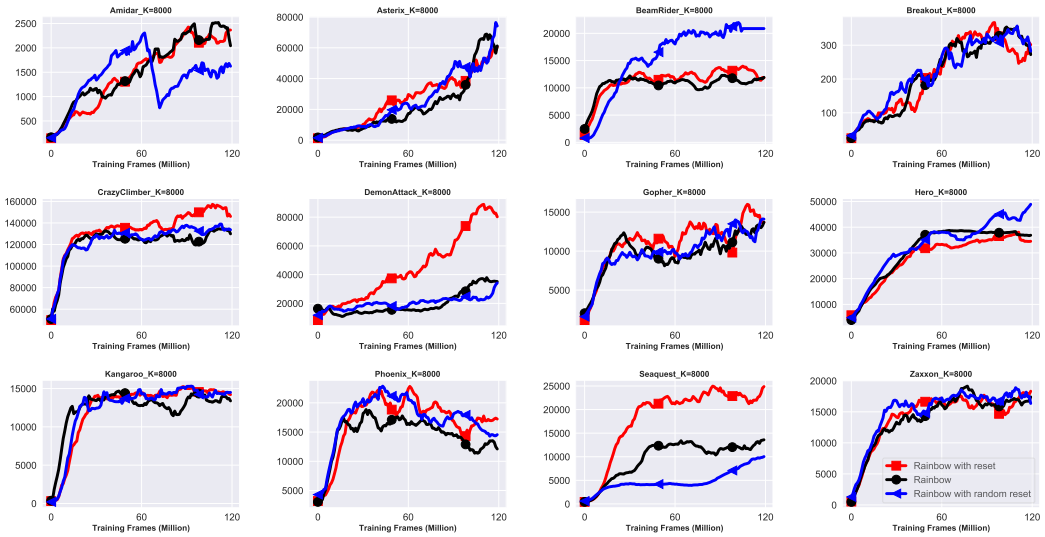

Figure 8: Performance of Rainbow with and without resetting the Adam optimizer and with a fixed value of $K = 8000$ on 12 randomly-chosen Atari games. See section 4 for a description of the random resetting case.

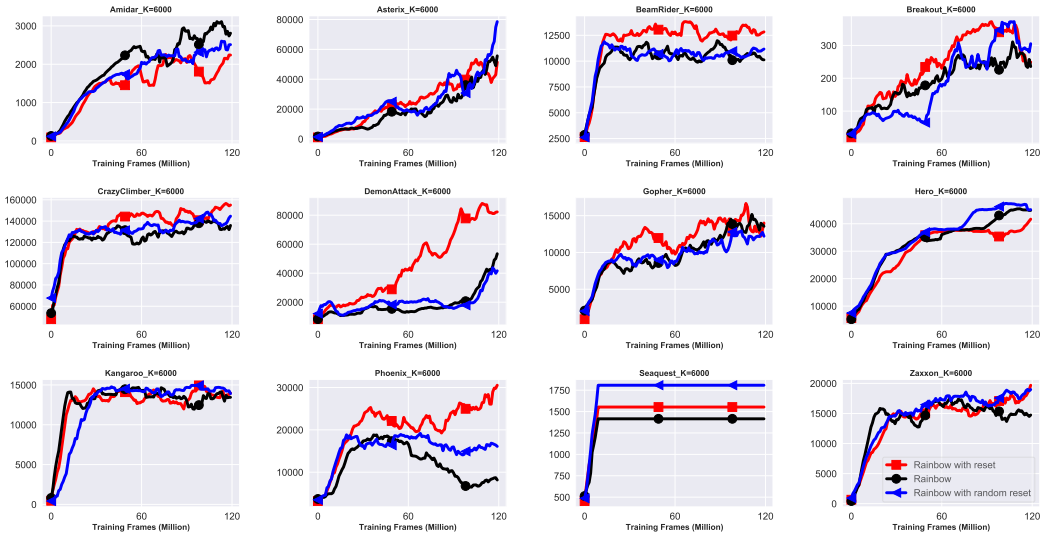

Figure 9: $K = 6000$.

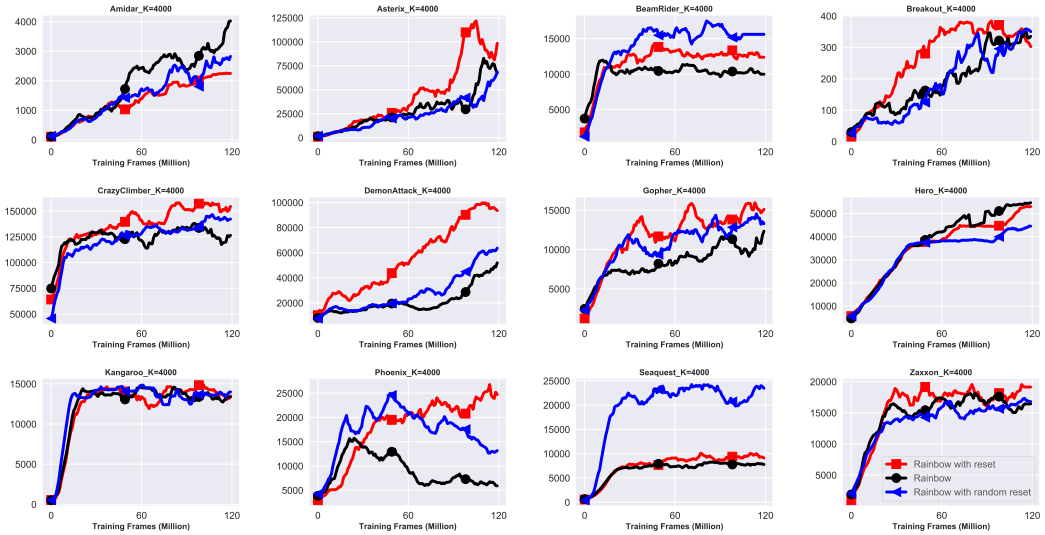

Figure 10: $K = 4000$.

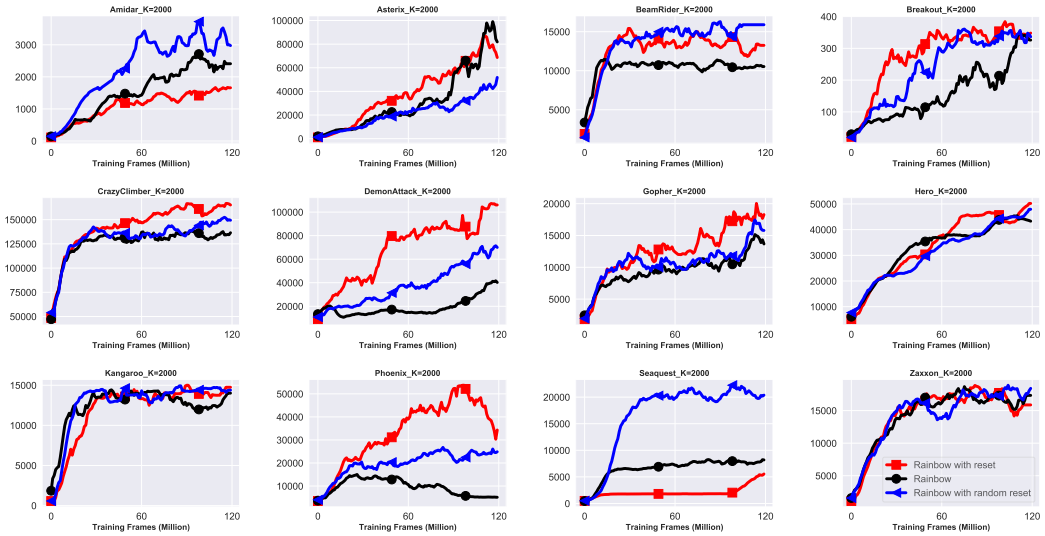

Figure 11: $K = 2000$.

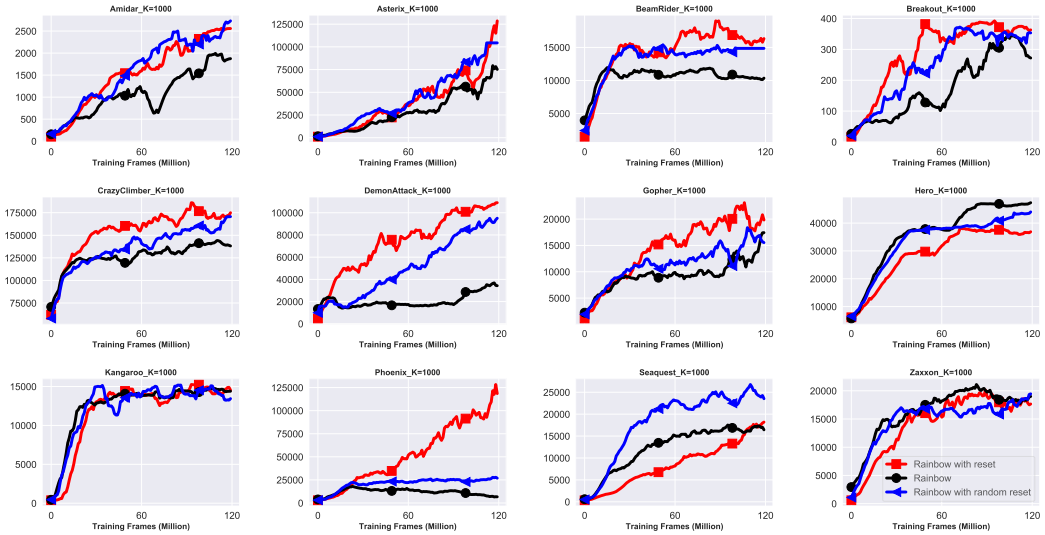

Figure 12: $K = 1000$.

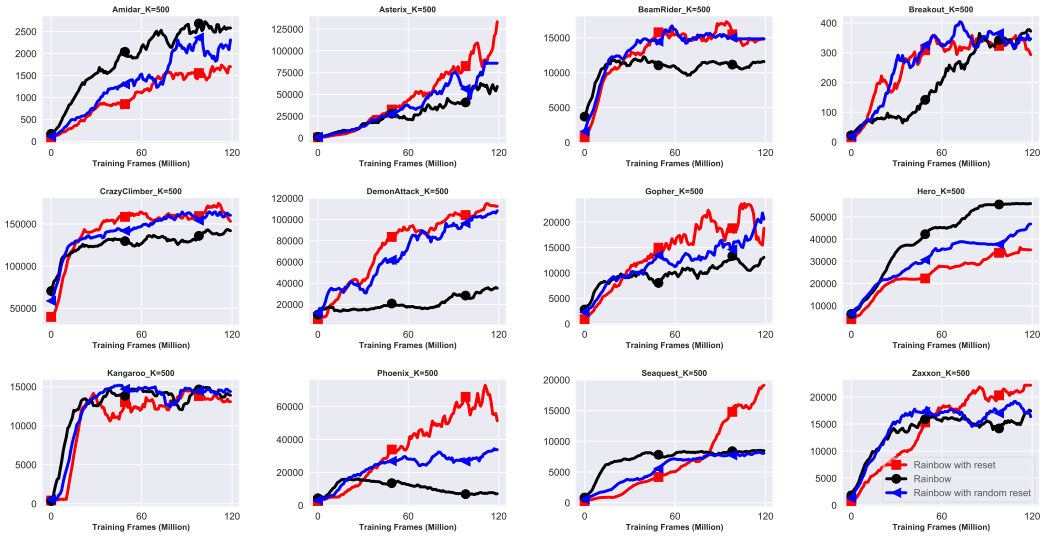

Figure 13: $K = 500$.

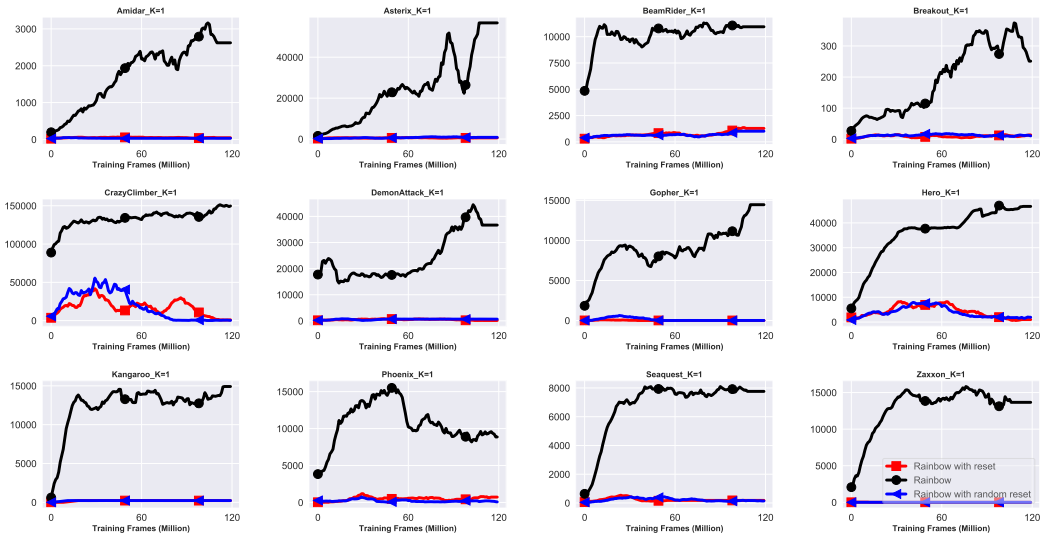

Figure 14: $K = 1$.

We now take the human-normalized median and mean of the results on 12 games and present them for each value of K.

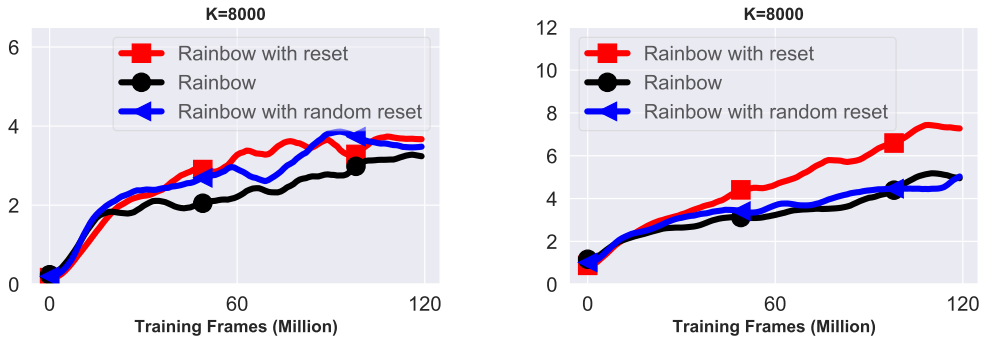

Figure 15: Human normalized median (left) and mean (right) for $K = 800$.

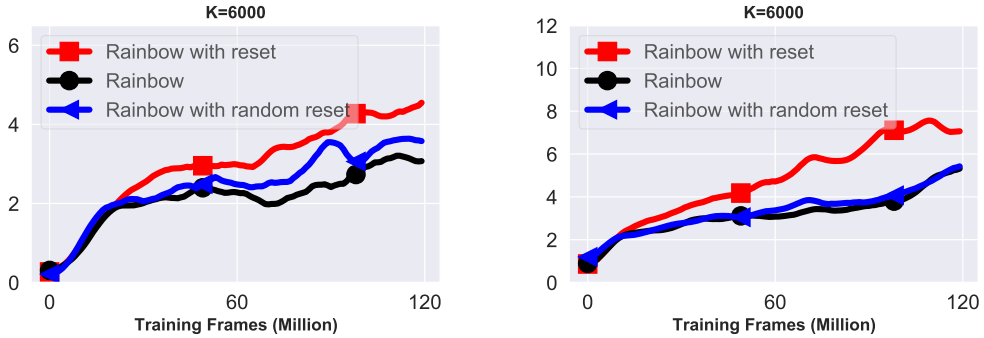

Figure 16: $K = 6000$.

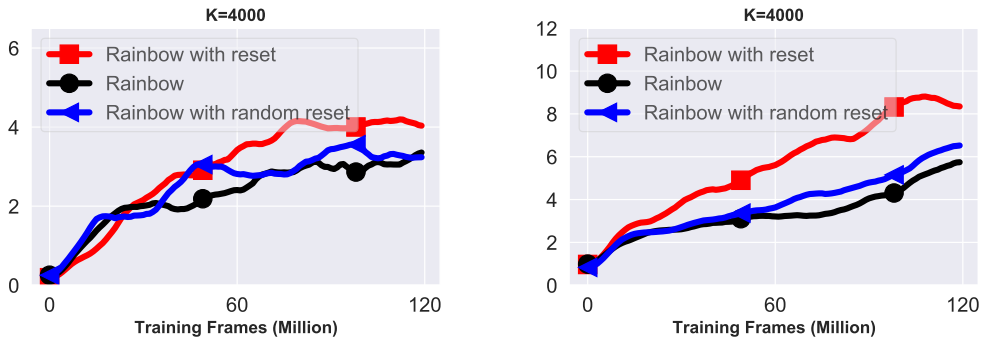

Figure 17: $K = 4000$.

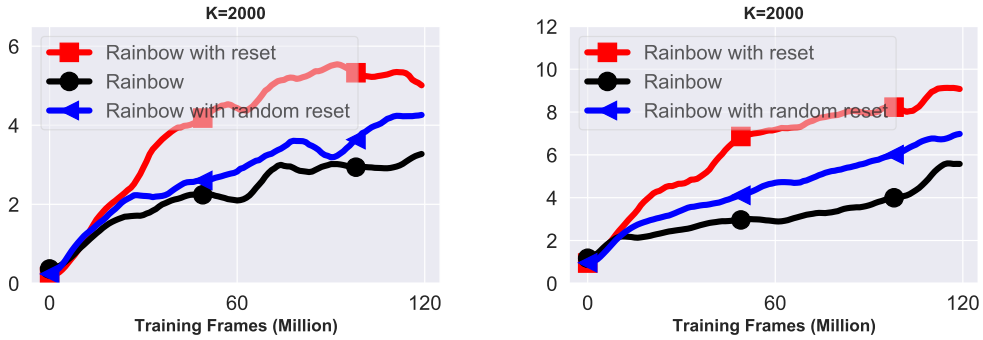

Figure 18: $K = 2000$.

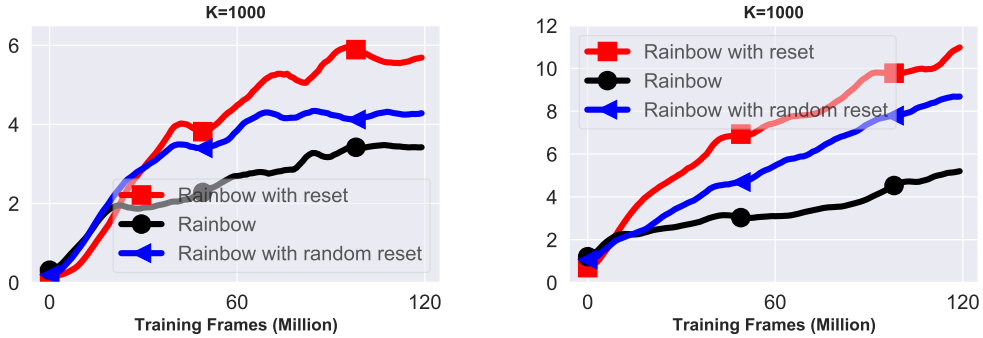

Figure 19: $K = 1000$.

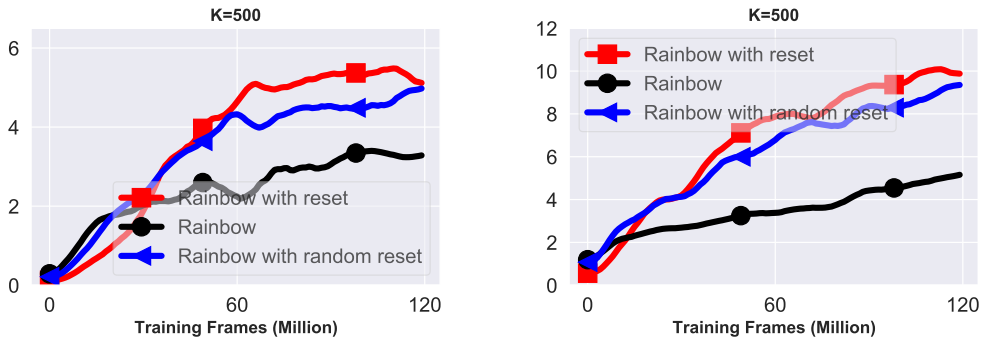

Figure 20: $K = 500$.

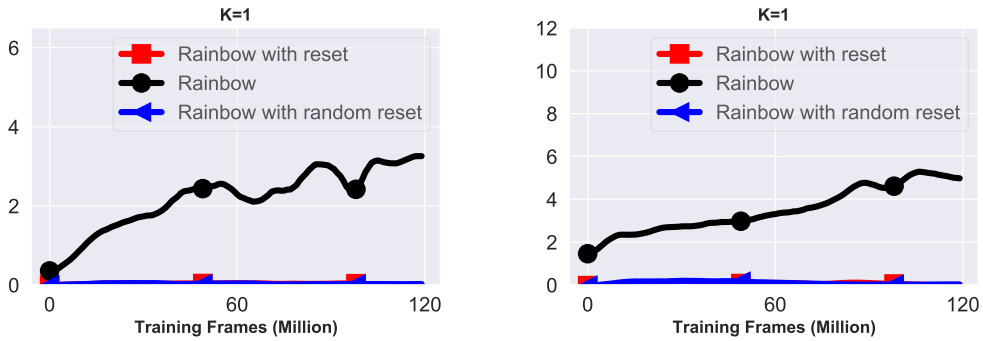

Figure 21: $K = 1$.

We also show area under the curve.

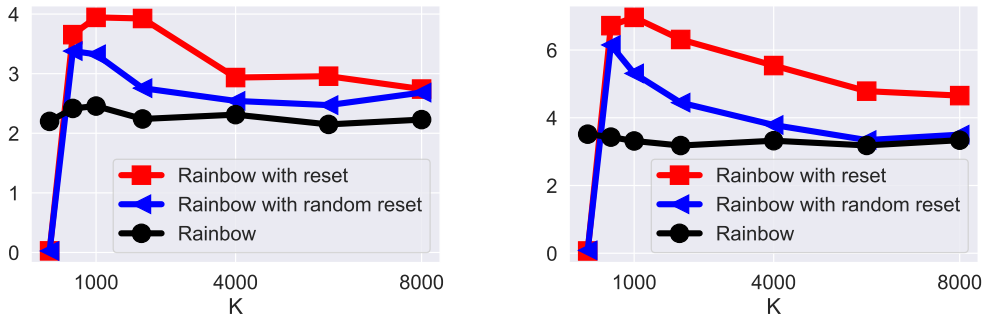

Figure 22: Area under the curve for median (left) and mean (right) of human-normalized performance.

7.2 Complete Results from Section 4.2

We now show complete results from section 4.2 starting with Rainbow RMSProp.

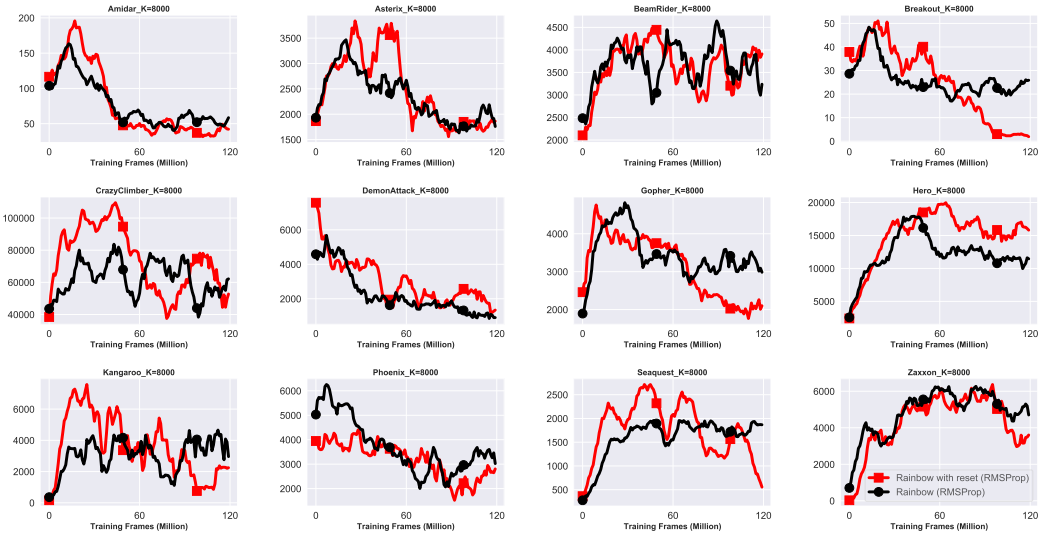

Figure 23: Performance of Rainbow with and without resetting the RMSProp optimizer and with a fixed value of $K = 8000$ on 12 randomly-chosen Atari games.

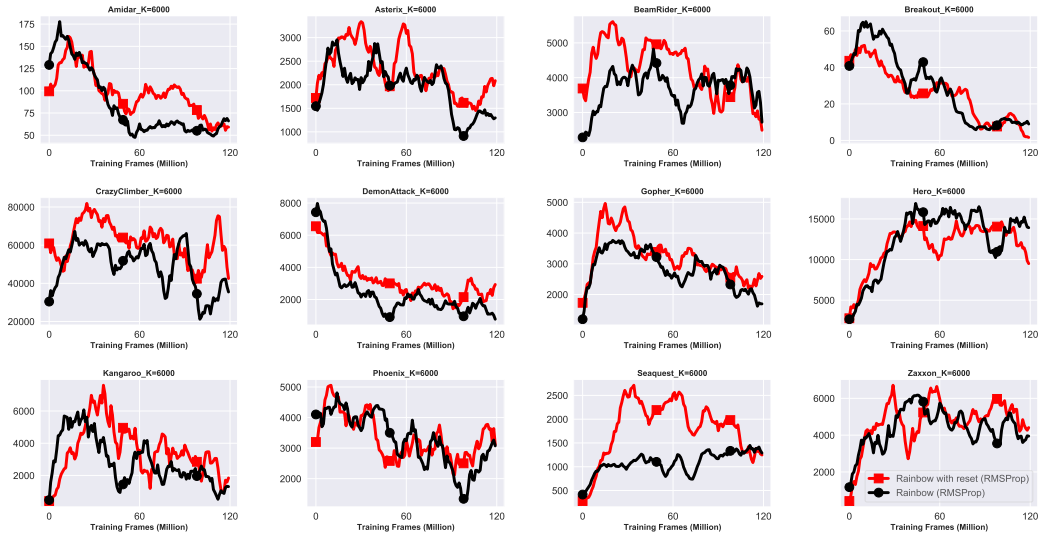

Figure 24: $K = 6000$.

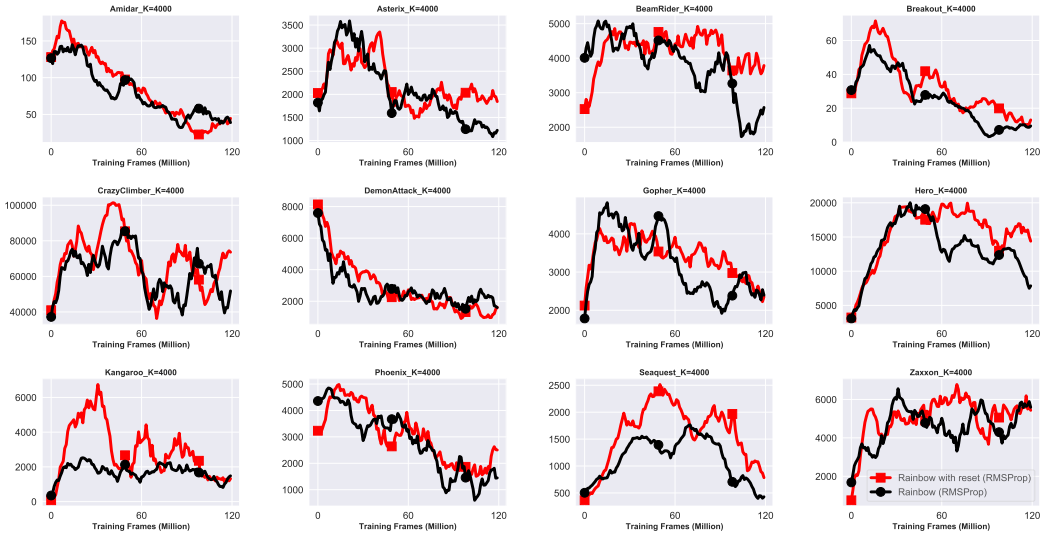

Figure 25: $K = 4000$.

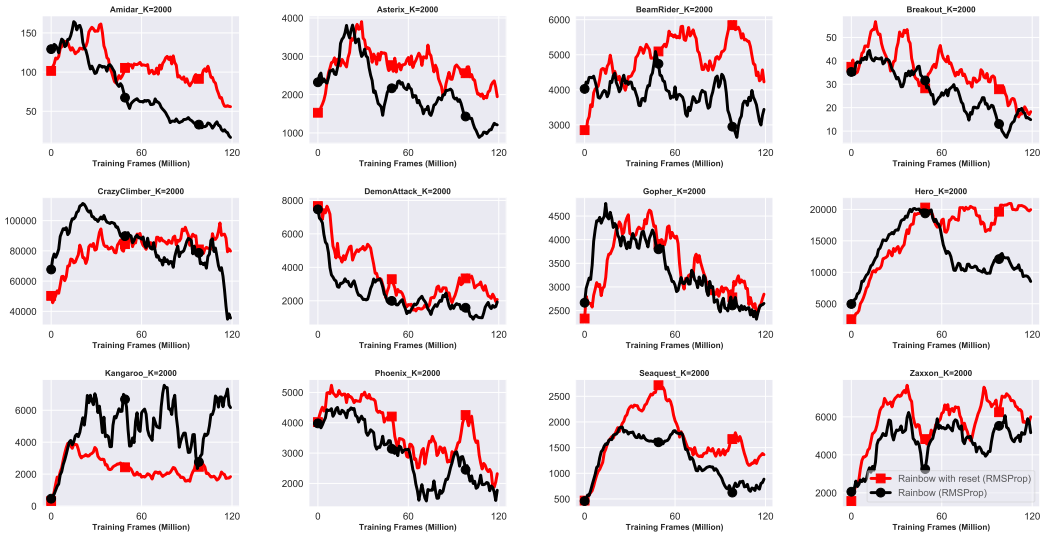

Figure 26: $K = 2000$.

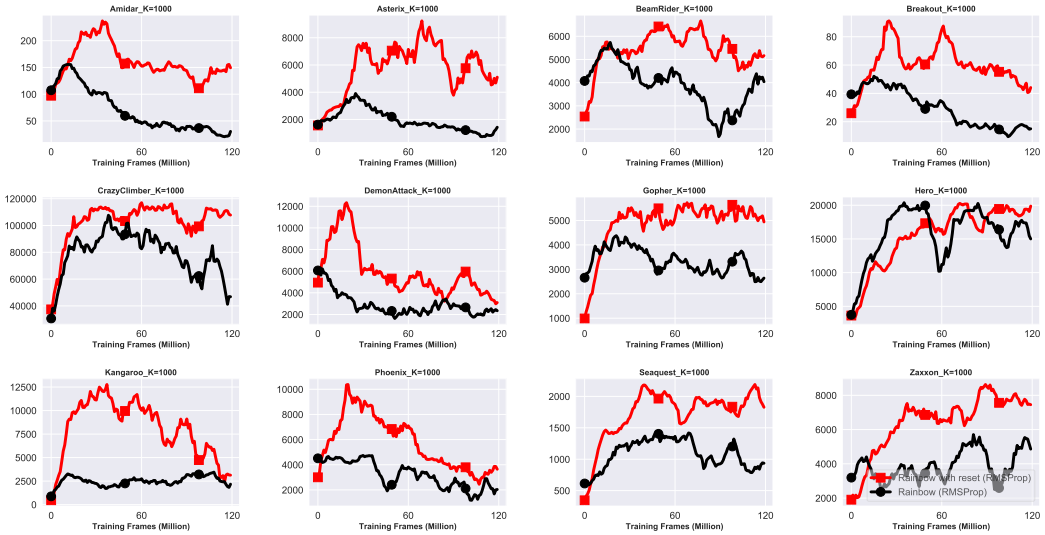

Figure 27: $K = 1000$.

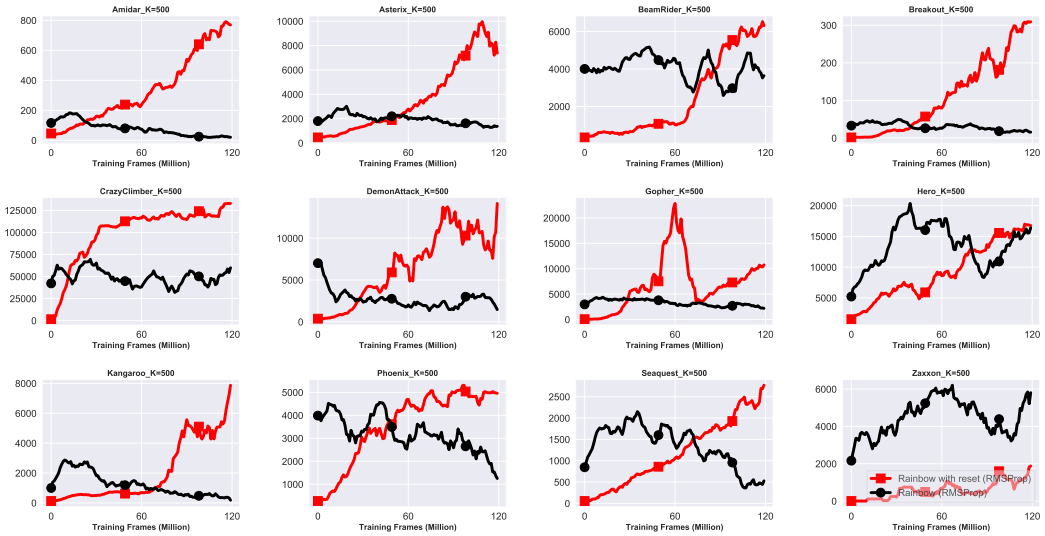

Figure 28: $K = 500$.

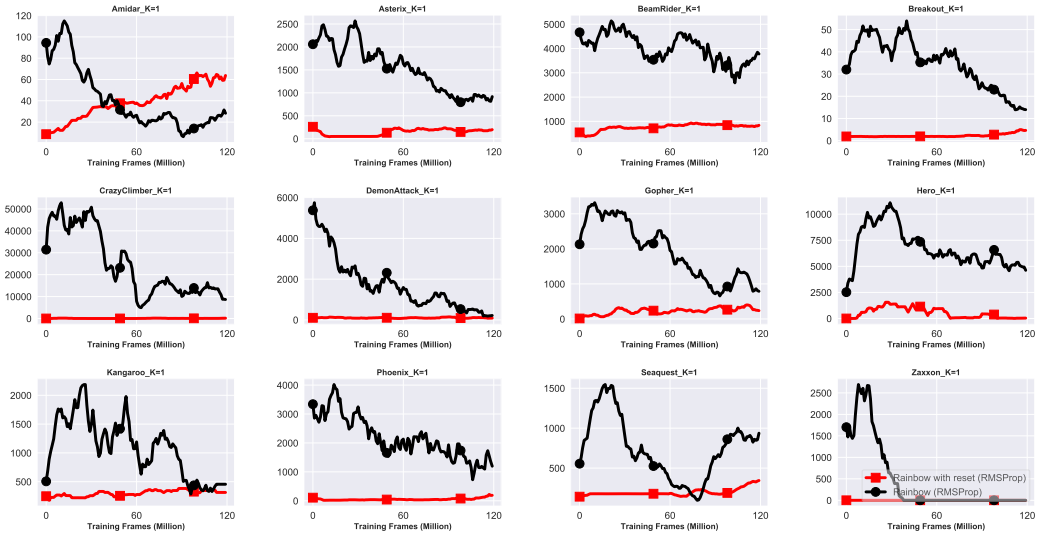

Figure 29: $K = 1$.

We now take the human-normalized median on 12 games and present them for each value of K.

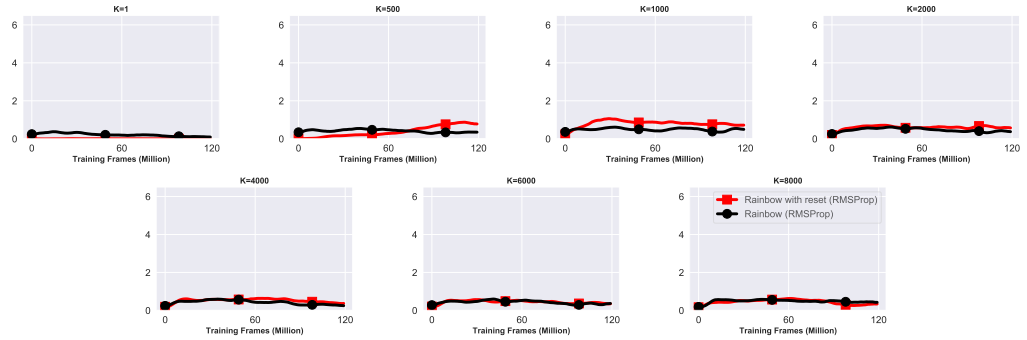

Figure 30: A comparison between Rainbow with and without resetting RMSProp on the 12 Atari games for different values of K.

Overall we can see that RMSProp results in poor performance, but resetting can somewhat improve the performance.

We now move to the Rainbow Pro agent.

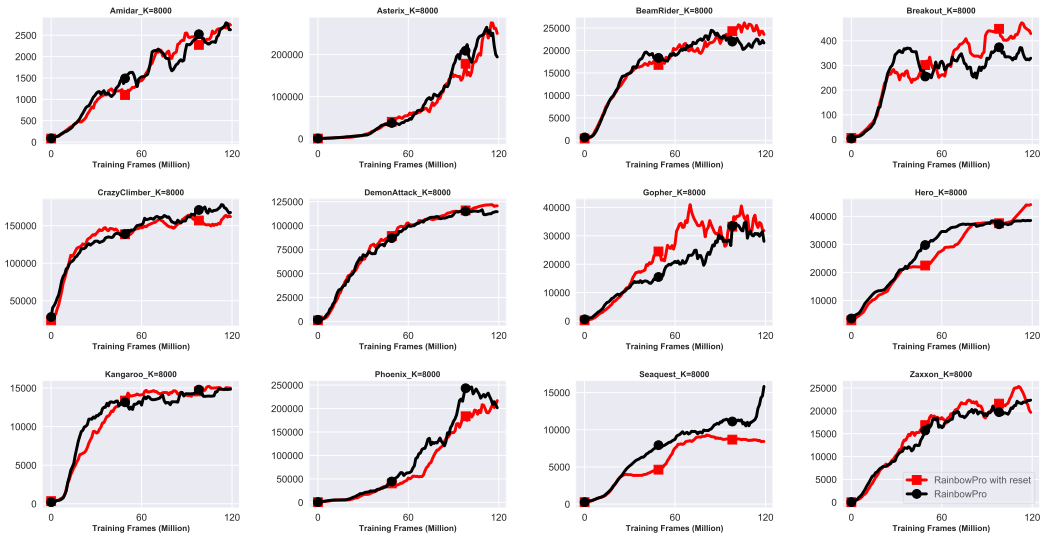

Figure 31: Performance of Rainbow Pro with and without resetting the Adam optimizer and with a fixed value of $K = 8000$ on 12 randomly-chosen Atari games.

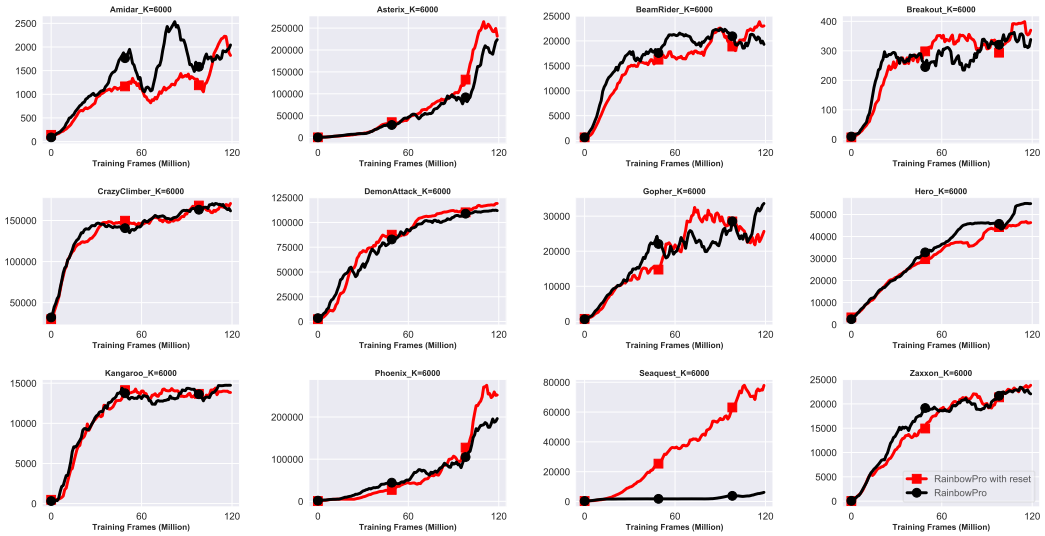

Figure 32: $K = 6000$.

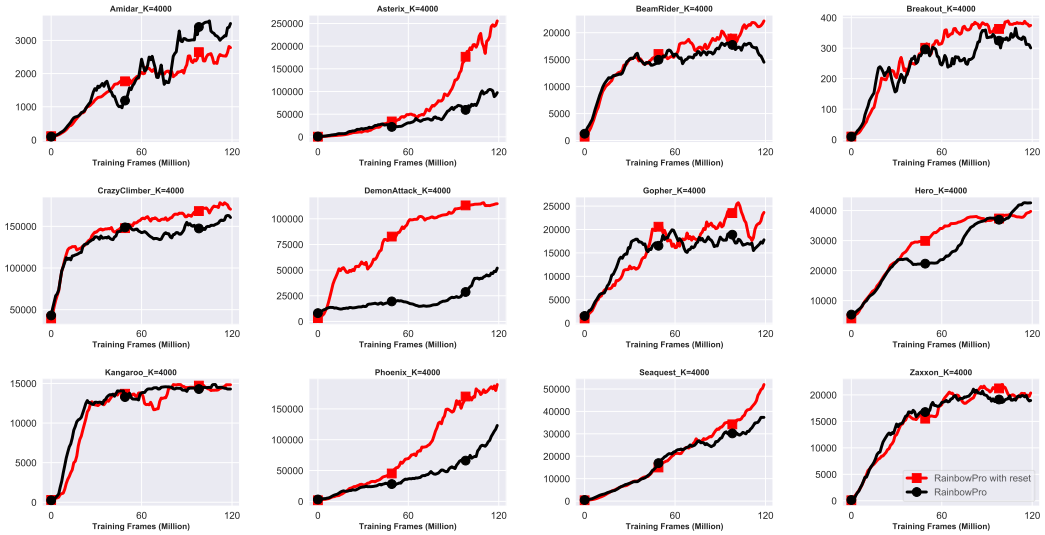

Figure 33: $K = 4000$.

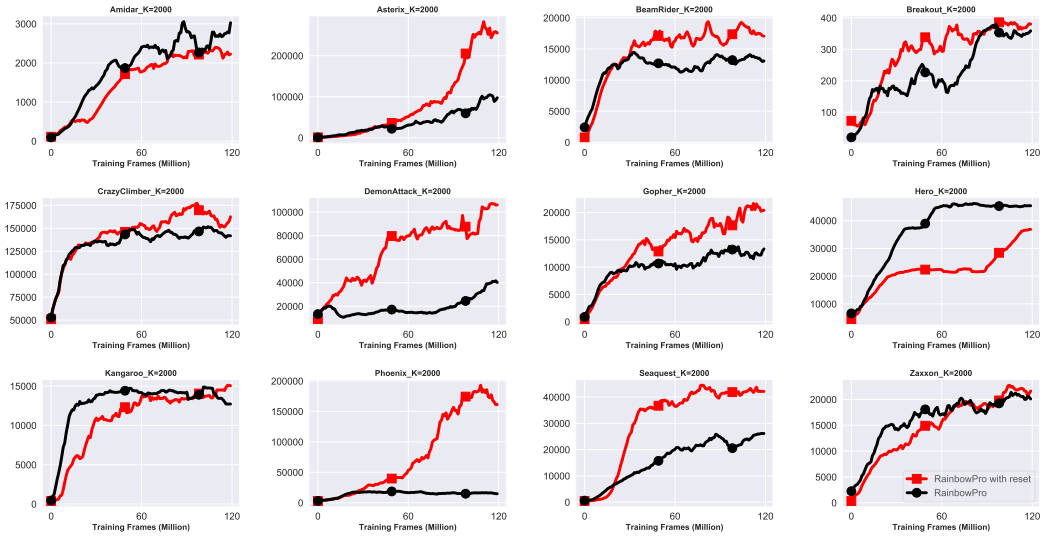

Figure 34: $K = 2000$.

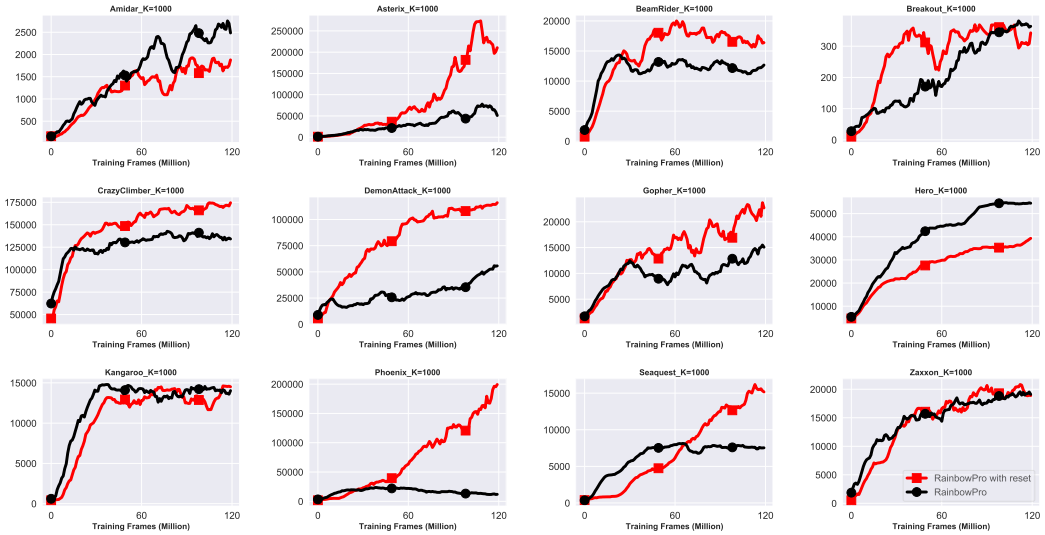

Figure 35: $K = 1000$.

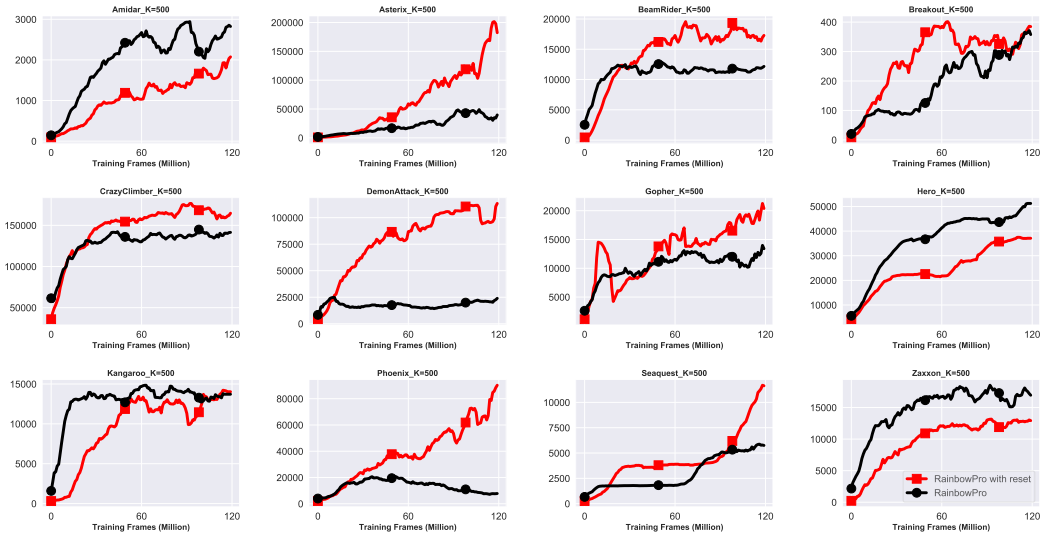

Figure 36: $K = 500$.

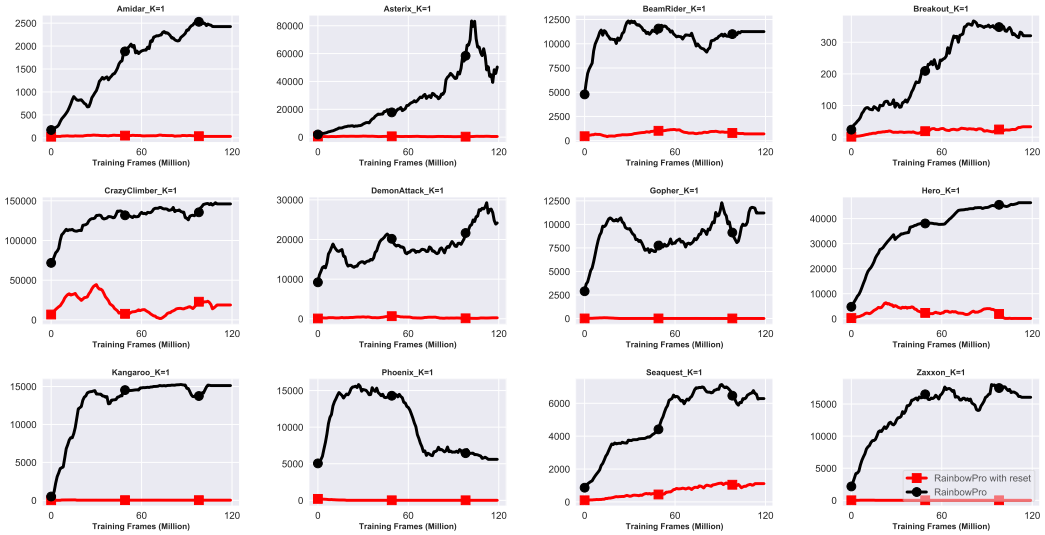

Figure 37: $K = 1$.

We now take the human-normalized median on 12 games and present them for each value of K.

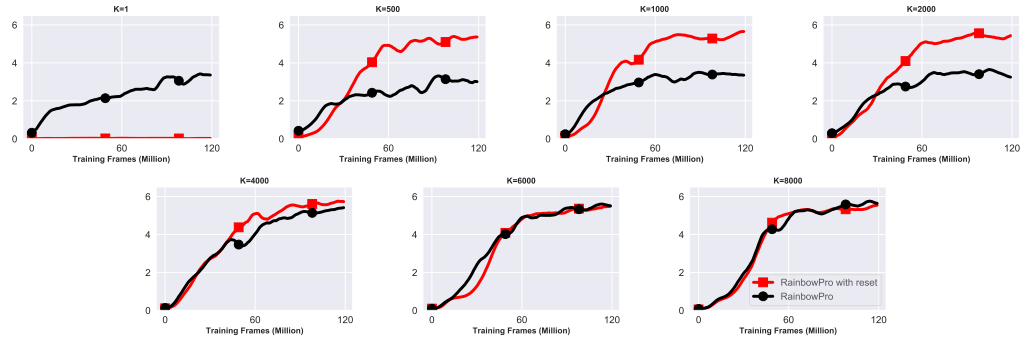

Figure 38: A comparison between Rainbow Pro with and without resetting Adam on the 12 Atari games for different values of K.

Overall we can see that resetting makes RainbowPro less sensitive to the K hyper-parameter.

In the last result of this section, we look at Rainbow with the Rectified Adam optimizer.

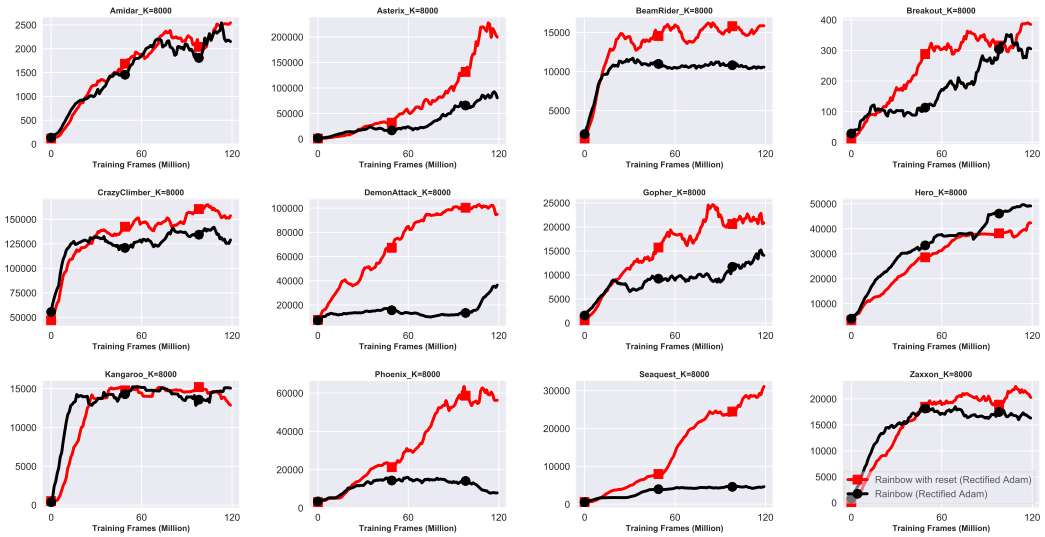

Figure 39: Performance of Rainbow with and without resetting the Rectified Adam optimizer and with a fixed value of $K = 8000$ on 12 randomly-chosen Atari games.

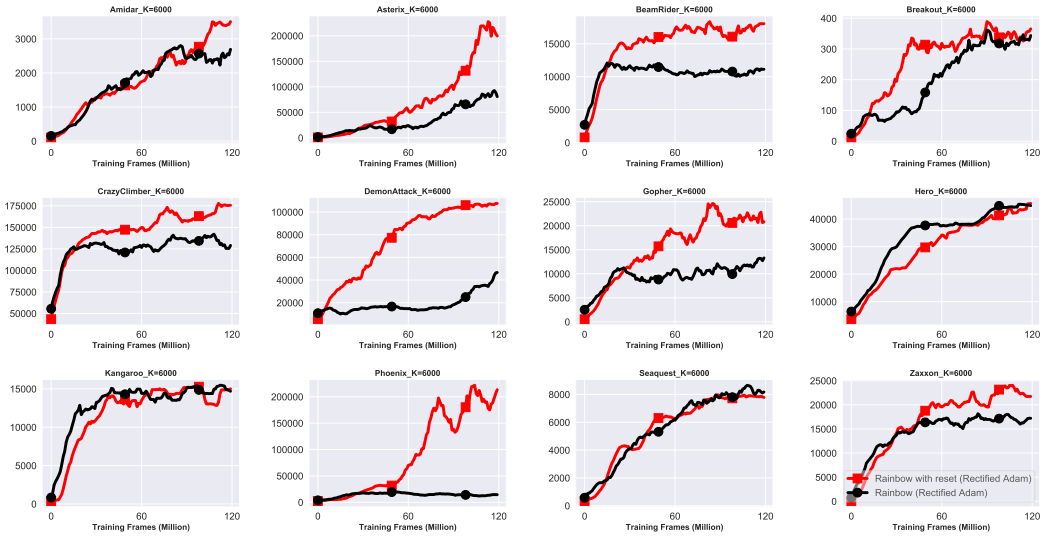

Figure 40: $K = 6000$.

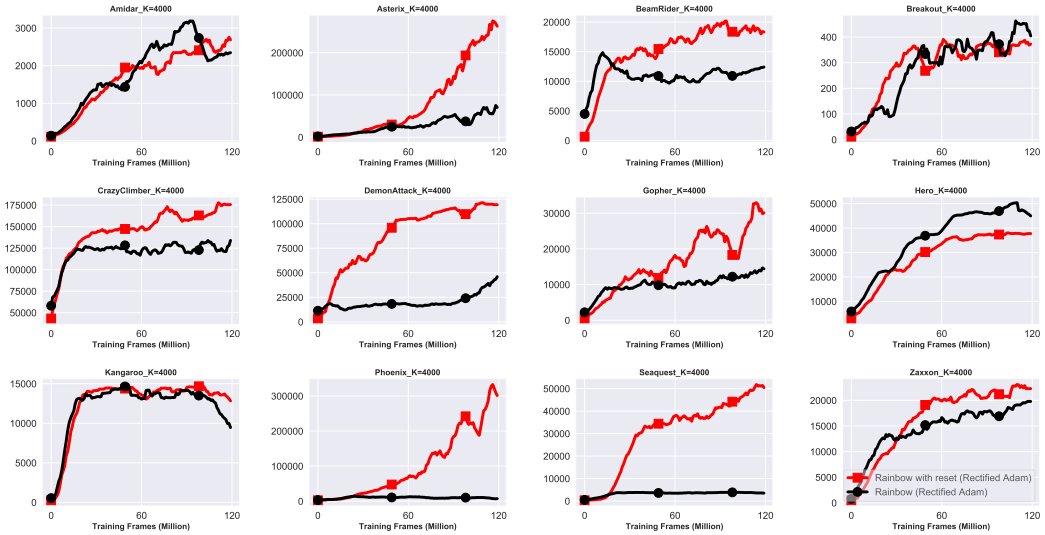

Figure 41: $K = 4000$.

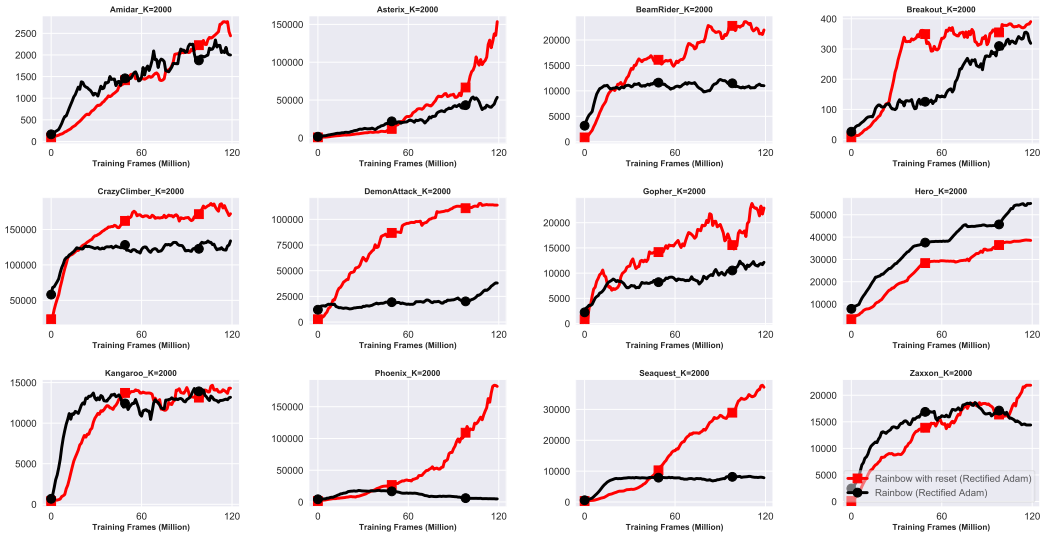

Figure 42: $K = 2000$.

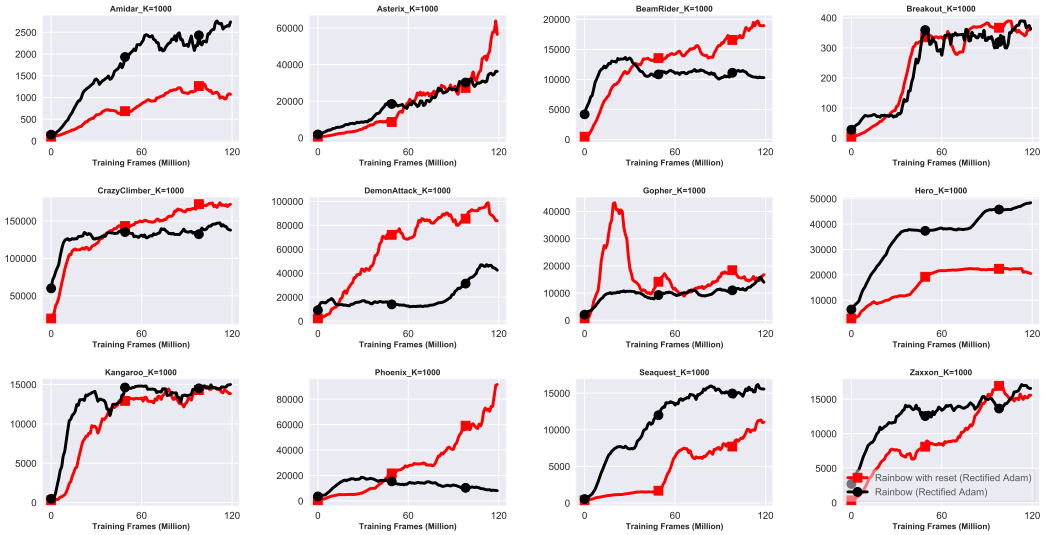

Figure 43: $K = 1000$.

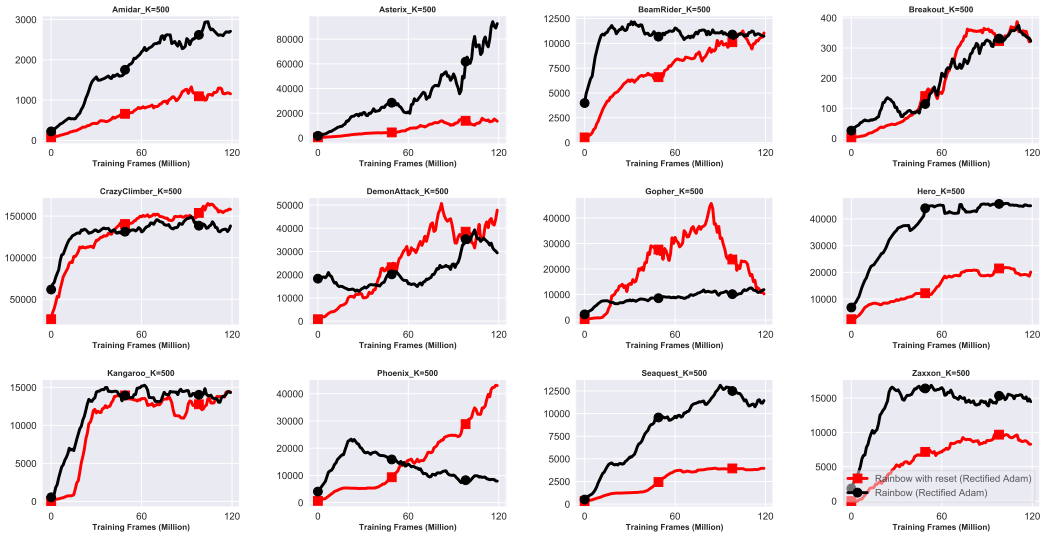

Figure 44: $K = 500$.

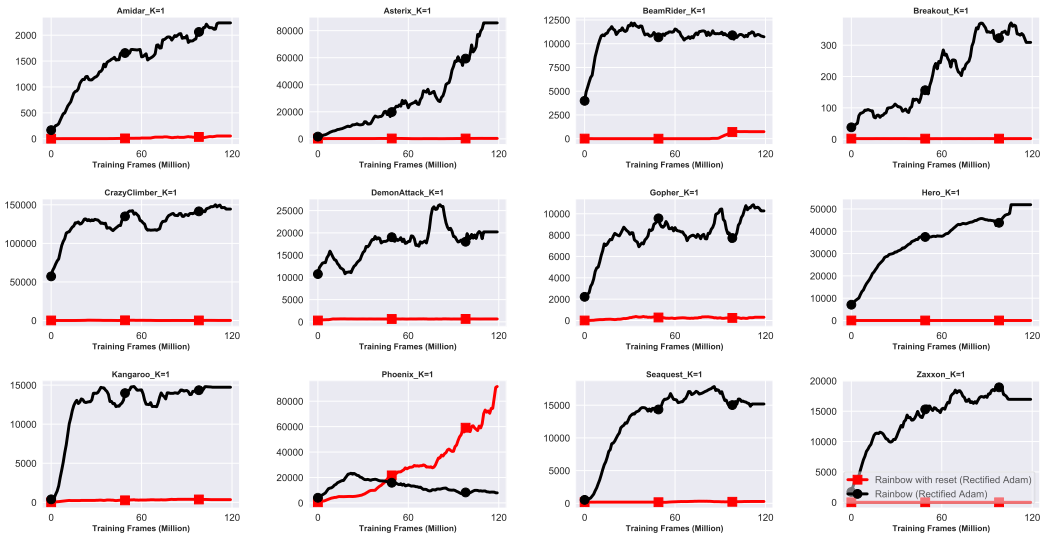

Figure 45: $K = 1$.

We now take the human-normalized median on 12 games and present them for each value of K.

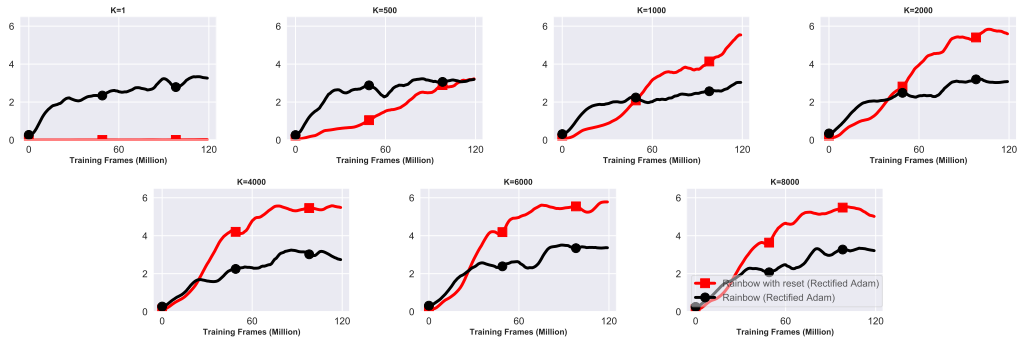

Figure 46: A comparison between Rainbow with and without resetting Rectified Adam on the 12 Atari games for different values of K.

Overall we can see that resetting improves Rainbow with Recitified Adam.

8 Complete Results From Section 4.3

We now show full learning curves for all 55 Atari games and over 10 random seeds. We benchmark three agents: the default Rainbow agent from the Dopamine (no reset), Rainbow with resetting the Adam optimizer, and Rainbow with resetting the rectified Adam optimizer.

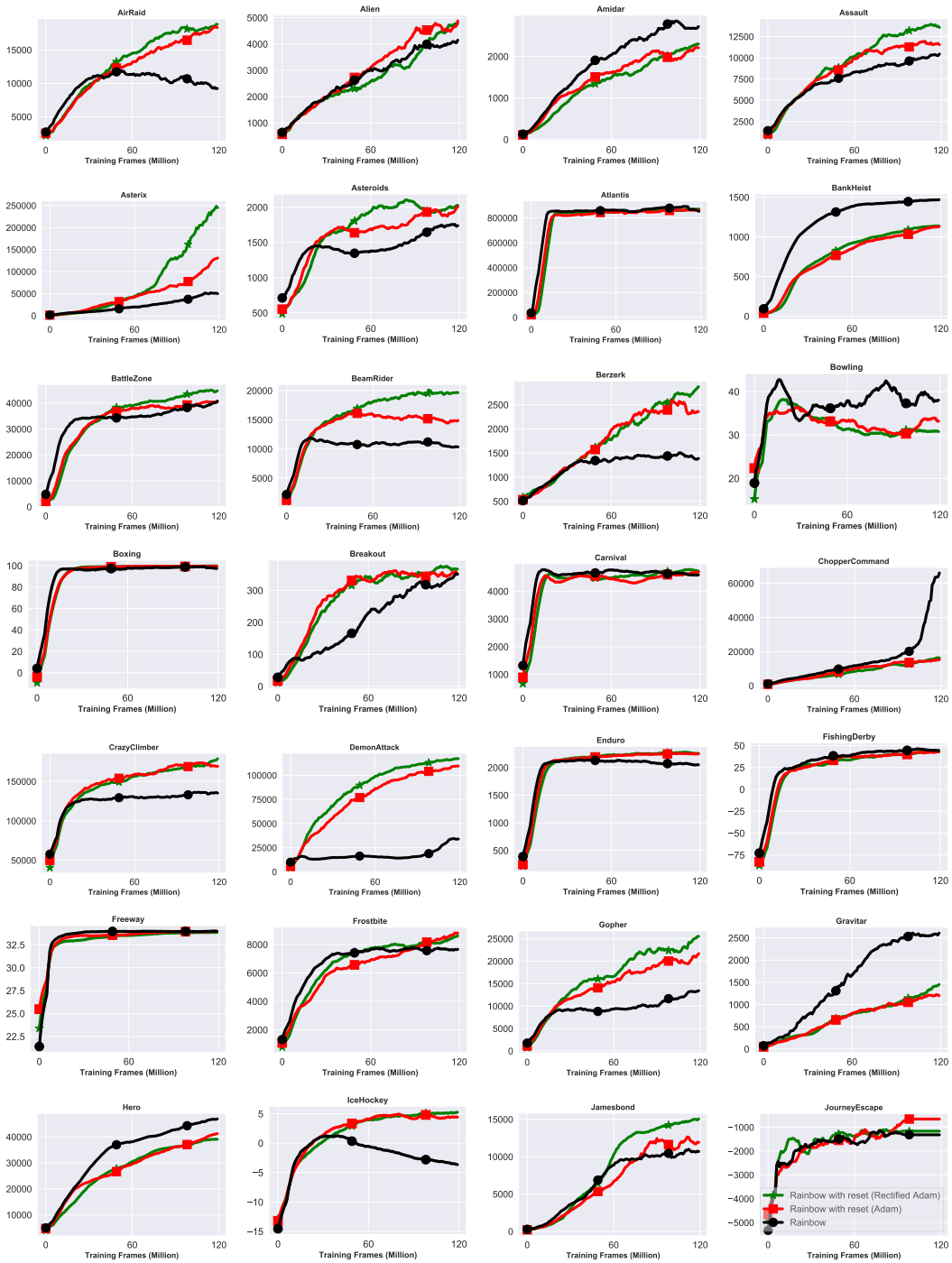

Figure 47: Full learning curves (Part I) averaged over 10 seeds.

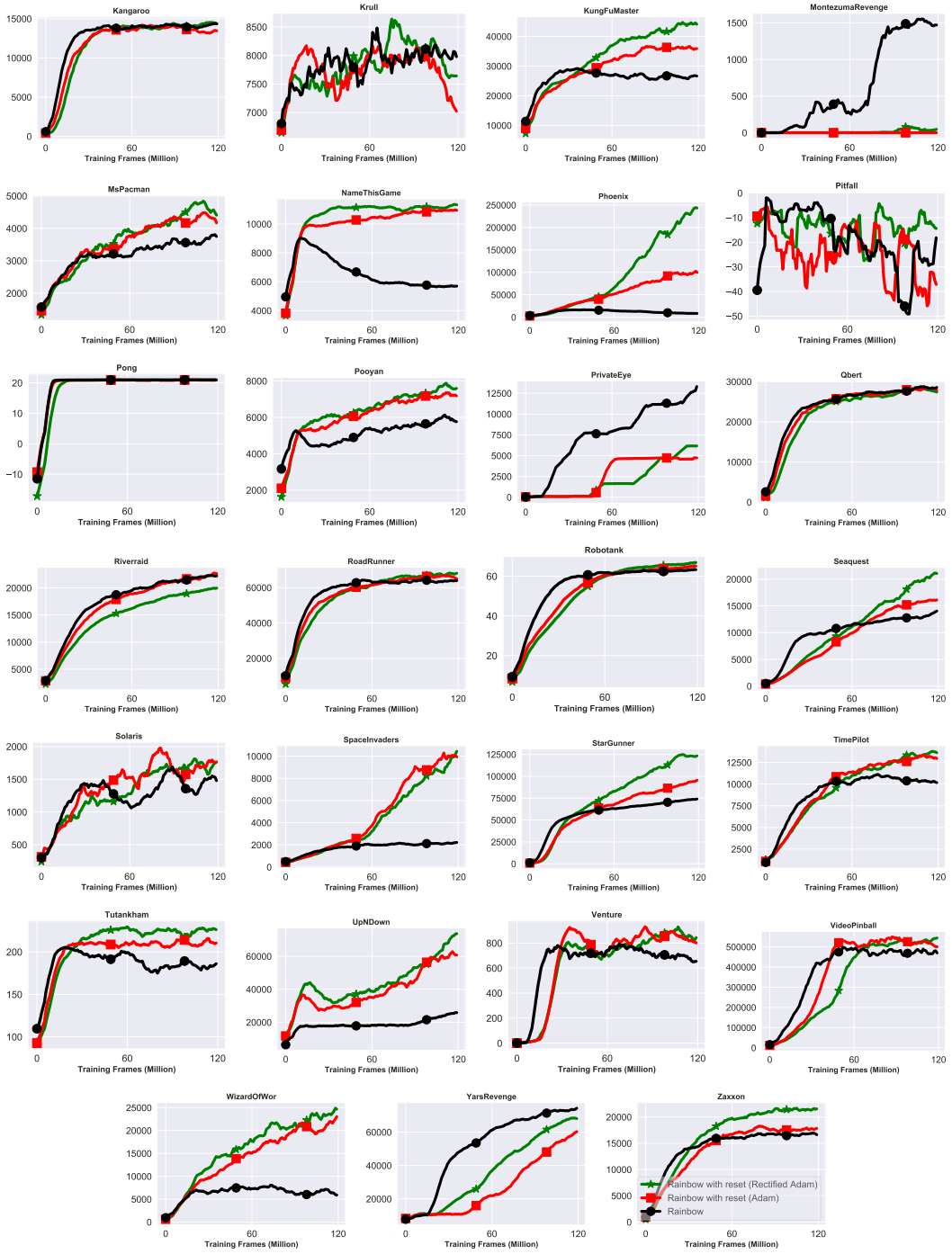

Figure 48: Full learning curves (Part II) averaged over 10 seeds..

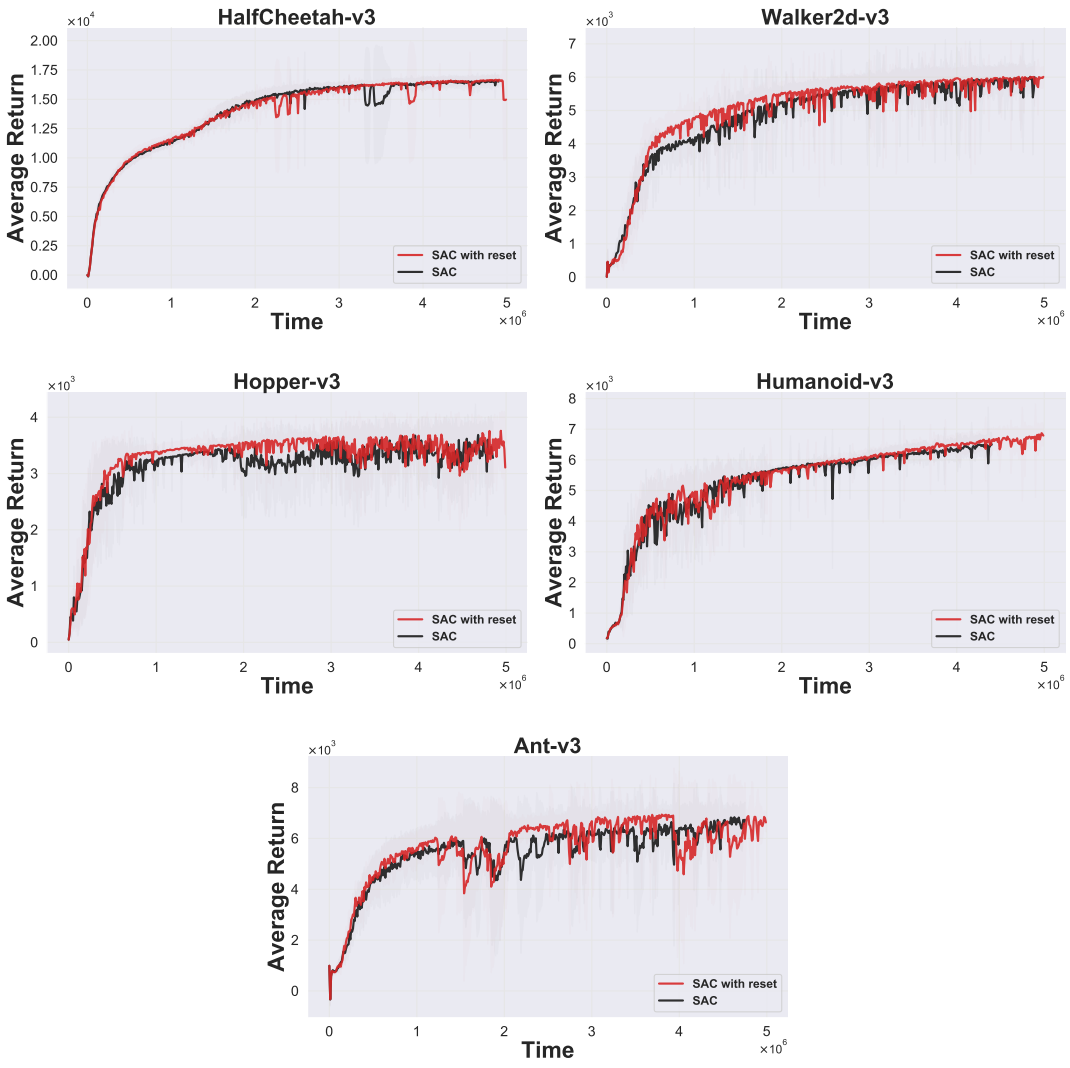

Figure 49: A comparison between Soft Actor-Critic (SAC) with and without resetting Adam on the standard MuJoCo tasks. In this study, both the actor and critic optimizers are reset every 5000 steps. The results are averaged over 10 different seeds.

8.1 Complete Results From Section 4.4

We finally present results on continuous control task with soft actor critic (SAC) and the Adam optimizer, where we reset the optimizers every 5000 steps. Note that, in contrsat to Atari and Rainbow, the target parameter θ is updated using the Polyak strategy, so it is less clear when to reset the optimizer. Thus we chose the simple strategy of resetting the optimizer every 5000 steps. We leave further exploration of resetting with Polyak updates to future work.