Policy Gradient Methods in Continuous Control Tasks

Sabina Elkins[*] McGill University
Jonathan Colaço Carr McGill University
Cesare Spinoso Di Piano McGill University
Étienne Demers McGill University

1 Introduction

This paper provides an overview of our work on the Mujoco Hopper-v2 task. We discuss the algorithms we implemented, provide some theoretical background on algorithms not presented in class, and show our results. Reinforcement learning in the context of teaching agents to move in simulated physical environments is a subject that has been studied extensively [13, 10, 5, 1]. Many algorithms have shown good performance on a variety of tasks and are well documented online. We therefore decided to focus our efforts on understanding and implementing some of the current state-of-the-art algorithms and their variants. Much of our work is based on the documentation found on OpenAI's Spinning Up website [1]. Our results are in line with the benchmarks found in the documentation, and our best-performing agent was trained with the SAC algorithm. We also explored a few methods to improve the performance of our vanilla SAC agent, such as using a learning rate scheduler and learning the temperature coefficient rather than treating it as a hyperparameter (explained in Section 2.5). Our best agent achieved a final performance of 3694.42 and sample efficiency of 1230.33[2].

2 Methods

This section highlights the main reinforcement learning algorithms explored. We decided to use policy gradient methods to train our agent, since these methods are known to perform well in continuous and high-dimensional action spaces [13, 10], as is the case for the Mujoco Hopper environment. Each subsection will introduce the algorithm at hand, reference any implementations used as guides, and note any hyperparameter tuning done.

2.1 Vanilla Policy Gradient (VPG)

As a baseline for other policy-gradient methods, we first considered the vanilla policy gradient (VPG) algorithm [14]. This on-policy method updates the policy's parameters via gradient ascent, using the following gradient of the cost function $J(\pi_\theta)$:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right],$$

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy. For this algorithm, we applied hyperparameter tuning to the actor and critic network architectures ($\{(64, 64), (400, 300), (100, 50, 25)\}$), activation functions ($\{ReLU, \tanh\}$), and learning rates ($\{3\times10^{-4}, 3\times10^{-3}\}$ for the actor, $\{1\times10^{-3}, 1\times10^{-2}\}$ for the critic) based on [7]. The optimal hyperparameters found were a network architecture of $(64, 64)$ and the $ReLU$ activation function for both the actor and the critic, as well as a learning rate of 3×10^{-4} for the actor and 1×10^{-3} for the critic. Although simple to implement, our VPG agent suffered from poor sample efficiency and a relatively low final performance (see Table 1 in Section 3), motivating us to consider more sample efficient algorithms.

2.2 Proximal Policy Optimization (PPO)

In order to improve the policy more quickly, the next method considered was the Proximal Policy Optimization (PPO) [11]. In particular, we implemented the PPO-Clip variant, whereby the objective function is 'clipped' in order to

[*]All authors contributed equally.
[2]This number differs from the one on the leaderboard.

ensure that the new and old policy stay relatively close together in policy space. To do so, we define the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, and define the cost function as follows:

$$J(\theta) = \mathbb{E}\left[\min\left\{r_t(\theta)A_t,\, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)A_t\right\}\right],$$

where A_t is the advantage at time step t. The 'clip' operation modifies the surrogate objective function by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. The minimum of the clipped and unclipped objectives is taken to obtain a lower bound for the unclipped objective.

For tuning the PPO algorithm (and all our subsequent algorithms), we used the same hyperparameter grid as VPG except we chose not to include searches for the learning rates as they significantly increased our search space without providing any noticeable improvements. The optimal hyperparameters found were a network architecture of $(64, 64)$ and the $ReLU$ activation function for both the actor and the critic.

2.3 Deep Deterministic Policy Gradient (DDPG)

As the Mujoco Hopper action space is continuous, we also decided to investigate the Deep Deterministic Policy Gradient (DDPG) method [9]. This agent is specifically designed for continuous action spaces and has been shown to outperform VPG and PPO in these cases [1].

The DDPG algorithm uses a parameterized *actor* function $\mu(s, \theta^\mu)$ which specifies the current policy. A *critic* $Q(s, a)$ is learned using the Bellman equation as in Q-learning. The actor is updated using the following gradient:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] = \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}\right],$$

where ρ^β is the state-visitation distribution. This has been shown [12] to be the gradient of the policy's performance. Similar to the PPO methodology, we tested various network architectures and activation functions for the Q-function and policy. $ReLU$ activations paired with a network architecture of (100, 50, 25) worked best for both the Q-function and policy networks. While DDPG improved upon PPO, this agent is known to overestimate Q values [1], which we tried to overcome with Twin Delayed DDPG.

2.4 Twin Delayed DDPG (TD3)

Twin Delayed DDPG (TD3) [3] is an algorithm which aims to address one of the shortcomings of DDPG. Recall that DDPG aims to simultaneously learn a value function and the policy which maximises this value function. When the estimated value function of a policy is far from its real value, the learning of the policy can be unstable. For example, if the value function overestimates the value of a particular action, it may incorrectly teach the policy to always take this action. TD3 introduces 3 tricks to mitigate this problem. First, the algorithm learns two Q-functions instead of 1. It uses the smallest value returned by the two functions to form targets, which reduces the risk of overestimation similar to Double Q-Learning [6]. However, only the first Q-function is used when updating the policy via gradient ascent. Second, the policy is updated less frequently than the Q-functions. This allows to obtain a more accurate estimate of the true Q-function before updating the policy. Third, noise is added to the target action, which has a smoothing effect on the Q-functions. This reduces the risk of the policy inadvertently exploiting errors in the Q-functions.

The same hyperparameter search from the DDPG agent was conducted for TD3, and we again found that $ReLU$ activations and (100, 50, 25) architectures worked best for policy and both Q-functions.

2.5 Soft Actor-Critic (SAC)

Unlike previously discussed algorithms, Soft Actor-Critic (SAC) [4] is an off-policy algorithm that learns a stochastic policy. It combines elements from stochastic policy algorithms such as VPG and PPO, as well as elements from TD3 and has thus shown great promise for learning in continuous action spaces. To explain SAC, we first need to introduce entropy regularized reinforcement learning. Recall that the entropy of a probability distribution denoted by $H(P) = \mathbb{E}[-\log P(x)]$ is a measure of the uncertainty of the distribution's outcomes. In entropy regularized reinforcement learning, at each time step, we add an additional reward proportional to the entropy of the policy at that particular time step. The optimal policy is then given by

$$\pi^* = \text{argmax}_\pi \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t))\right)\right].$$

The corresponding equations for the Q-functions are easily obtained (we implement the more recent SAC version which does not require a value function). A higher temperature coefficient α favors a policy with higher entropy and thus a more uncertain or exploratory agent. Playing with this coefficient leads to an exploration vs. exploitation trade-off similar to the one seen in class in the context of ϵ-greedy policies.

In our first implementation of SAC, we used a fixed α, whose value was chosen via hyperparameter tuning ($\{0.1, 0.2, 0.4, 0.8\}$). Hyperparameter tuning provided us with a $(64, 64)$ network architecture with a $ReLU$ activation function and $\alpha = 0.2$. We subsequently implemented a version where α varies throughout the training as described by [5]. In this version α is updated at each training step to maximize an objective based on a target level of entropy and on the current policy. Similarly to TD3, SAC learns two Q-functions and uses Polyak averaging to update the target networks. The key difference between the two is that since the SAC policy is stochastic, there is no need for a target policy: next actions are drawn directly from the current policy.

To further improve our SAC agent, we used an exponentially decaying learning rate scheduler for both the policy and action-value networks to try to stabilize training. We also experimented with a 'bottleneck' architecture of (128, 32) in an attempt to force the network to only learn the most important features of the action and state space. In addition, to improve sample efficiency, we implemented the use of a prioritized replay buffer proposed by [2] where two batches of 100 observations each are sampled uniformly from the experience buffer and where the 100 observations with the highest cumulative return are selected. The motivation behind this method is that it should allow the agent to make better use of its experience and thus improve sample efficiency. Finally, to improve the performance of our agent we re-train it starting from its best checkpoint using both a learned α as well as a cosine annealing learning rate. We do this in order to help the networks slowly move away from their current local optima and explore other potentially useful network settings without experiencing catastrophic forgetting [8].

3 Results

3.1 Performance

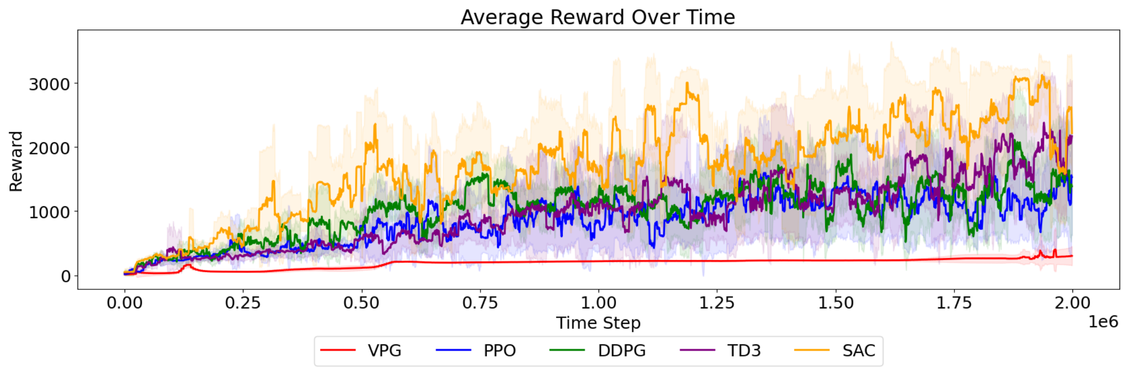
Table 1 shows a collection of metrics for an agent from each algorithm trained with their optimal hyperparameters. The *final performance* is the mean reward calculated on a set of 5 seeds for 50 episodes. The *cumulative training reward* is the sum of training rewards. Note that the reward is not an episodic reward, but the mean reward over 20 episodes, which was calculated every 1000 time steps during training. Finally, the *sample efficiency* metric is discussed in Section 3.2.

Algorithm	**Final Performance**	**Cumulative Training Reward**	**Sample Efficiency** [a]
VPG	1042.28	374 906.26	28.07
PPO	2681.58	1 702 329.46	140.63
DDPG	2544.58	2 056 793.39	171.66
TD3	3039.96	1 950 103.75	152.11
SAC	**3520.60**	**3 354 563.10**	**205.03**

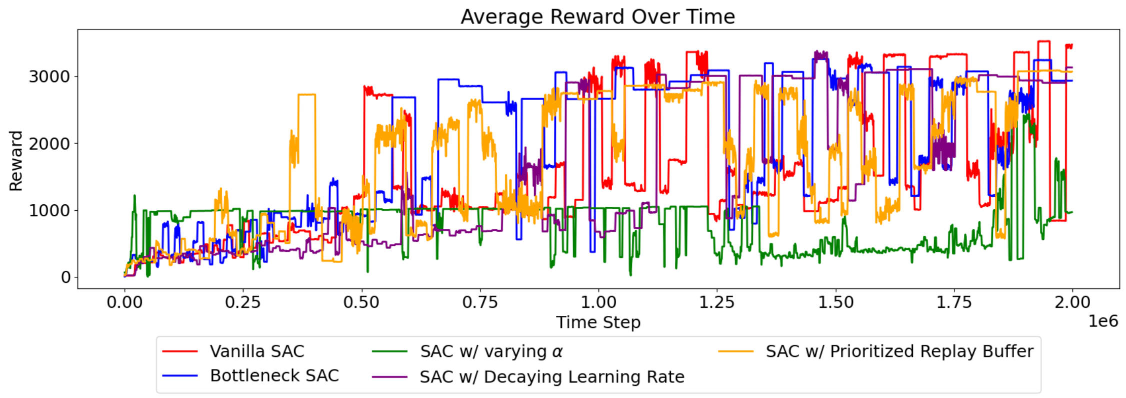
Table 1: Evaluation metrics for the best hyperparameter setting for each agent. We use the best "checkpoint" to measure final performance. The cumulative training reward is calculated as an average over 5 training runs of 2 000 000 steps. Sample efficiency is calculated as described in Section 3.2.

[a] Calculated as described in Section 3.2.

The performance of the five algorithms replicates the performances seen in OpenAI's Spinning Up [1], with VPG being the lowest performing, then DDPG, PPO, TD3, and finally SAC with the highest performance. Note that the metrics reported for SAC in Table 1 and Figure 1a are 'vanilla' SAC, and those in Figure 1b are variants of SAC. Beyond the final performance, we were also able to replicate the training behaviour seen in OpenAI's Spinning Up [1]. Figure 1a shows the 20-step average reward measured every 1000 time steps across 2 million training steps for each of the algorithms averaged over 5 runs. These training rewards follow the trends seen in Open AI, except that (with the exception of VPG) all of the algorithms see much greater variability in their training rewards.



(a) Comparison of the averages and standard deviations of five runs of training rewards for the best agent for each of the algorithms attempted.



(b) Comparison of the training rewards for SAC variants (single run).

Figure 1: Comparisons of the rewards throughout 2 000 000 training time steps, which are 20-step average rewards measured every 1000 time steps.

Figure 1b provides similar mean reward plots for four variants of SAC in comparison with the implementation from the previous figure[3]. We observe that decaying the learning rate does indeed help stabilize the mean reward compared to the vanilla implementation. Using a learned alpha does not appear to help training as the agent gets too often stuck exploiting instead of exploring. This suggests that more fine-tuning is needed to make this method work properly. In addition, using a bottleneck architecture appears to benefit training. This is even more obvious when observing the plot for the cumulative reward (see Figure 4 in the appendix). Finally, the use of the prioritized replay buffer appears to help especially in terms of sample efficiency as this variant is the only one to reach rewards close to 3000 before 500 000 time steps.

For our final agent, we re-train the agent with the optimal SAC hyperparameters starting from its best evaluation checkpoint of around 3520 average rewards for 2 million more time steps with both a cosine annealing learning rate and learned α. In this case, the learned α helps as it drives the agent to exploit more which balances the exploration caused by the changing learning rate. Together, this appears to generally avoid catastrophic forgetting (Figure 5). Using the best agent from this re-training process, we obtain a performance of 3696.60. To give us a final performance boost, we restart this training process from the best checkpoint several times which gives us a final performance of 3815.96.

3.2 Sample Efficiency

To calculate sample efficiency, an agent was trained for 100 000 time steps and we recorded the mean reward every 5000 time steps. We computed the area under the curve and then scaled it by 100 000.[4] This was averaged over

[3]Due to computational constraints, we were not able to re-run these models to take averages.

[4]The *sample efficiency* metric was calculated as described by the TA to our group (we note this because it differs from the assignment's description of it).

5 different seeds. The results of sample efficiency generally follow the same trend as the *final performance*. One exception to this is that DDPG outperforms both PPO and TD3 on this metric. For TD3, this can be explained by the fact that DDPG will get stuck in a local optimum (and so has a higher chance of exploiting early on) and TD3 has safe guards against that (as explained in Section 2.4). For PPO, the training algorithm is meant to keep the agent's policy from varying too sharply from its current estimate, so a smaller sample efficiency is expected.

For our final agent, we perform an additional grid search using the optimal SAC hyperparameters but with $\alpha \in \{0.1, 0.2, 0.4, 0.8\}$ as well as the learned alpha. We also increase the update frequency of the SAC agent to once per episode. The most sample efficient agent receives a scaled AUC of 1230[5] with $\alpha = 0.8$ (Figure 6). This suggests that a potentially simpler and more effective way of varying α would be via an exponential weight decay where more exploration is favoured at the beginning and more exploitation is introduced when a sufficiently good optima is found.

3.3 Adaptability

To test our algorithm's ability to adapt to a held out task, we start with an SAC agent that was pre-trained for 2 000 000 steps using its best hyperparameter settings[6]. We then modify the gravity of the Mujoco environment and train the agent for an additional 1 000 000 time steps to determine whether it can adapt to the new environment. Although we can't make direct reward comparisons between different tasks, our agents seem to learn efficiently and achieve reasonable performance on these new tasks. Results are shown in Figure 2 and Table 2.

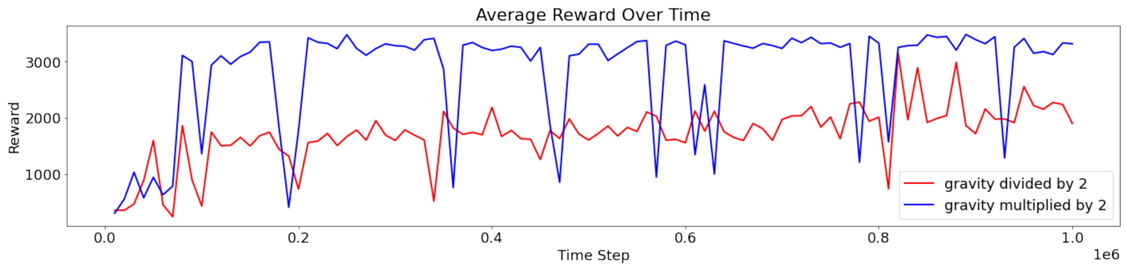


Figure 2: An SAC agent is pre-trained for 2 000 000 time steps and subsequently trained for 1 000 000 time steps in an environment with a different gravity. The plot shows the average rewards at every 1000 time steps in the new environment.

4 Conclusion

In conclusion, of all the policy gradient methods considered, the Soft Actor-Critic agent performed the best in the Mujoco Hopper environment. While multiple SAC variants were tested, the original SAC agent achieved the highest single time step reward after being trained 2 million time steps. It was also the most sample efficient. Further optimizations were explored for both improving the performance from a pre-trained agent and increasing sample efficiency. Moreover, SAC adapts well to varying gravity in the environment, even when trained for only 1 million timesteps. The SAC algorithm and its entropy regularization term showcases a fine balance exploration and exploitation which is useful for the Mujoco Hopper and other reinforcement learning tasks.

[5]This may differ from the leaderboard submission due to discrepancies in calculations e.g. which seed is used.

[6]This agent differs from the leaderboard submission due to performance improvements made after this section was written.

References

[1] J. Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[2] C. Banerjee, Z. Chen, and N. Noman. Improved soft actor-critic: Mixing prioritized off-policy samples with on-policy experience. *CoRR*, abs/2109.11767, 2021.

[3] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. 2018.

[4] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2018.

[5] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. 2019.

[6] H. Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[7] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017.

[8] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, page arXiv:1509.02971, Sept. 2015.

[10] J. Peters and S. Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225, 2006.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[12] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, 06 2014.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[14] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

Appendix

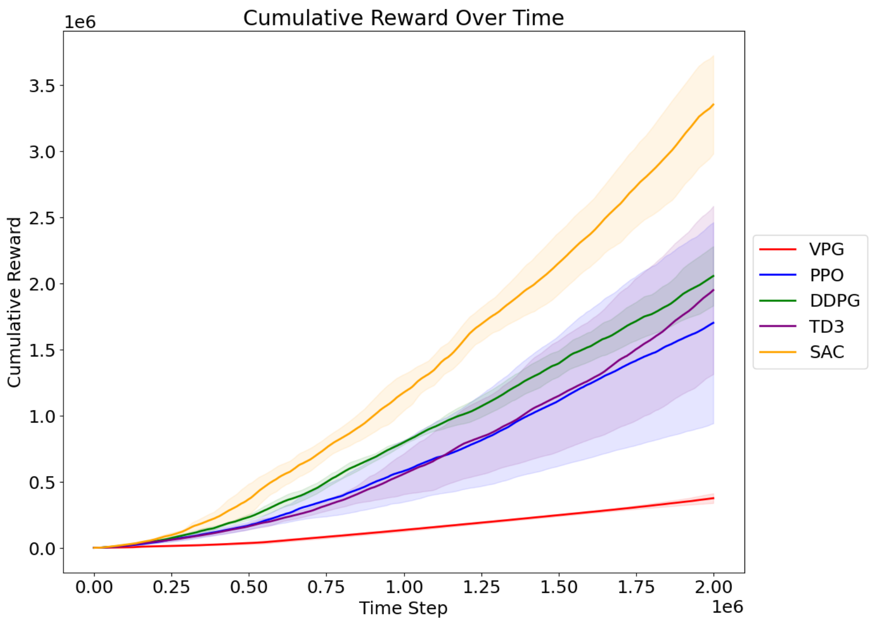


Figure 3: The average cumulative reward during training for each of the five algorithms in Section 2. The rewards are from 5 runs of 2 000 000 training time steps, whose 20-step average reward is measured every 1000 time steps.

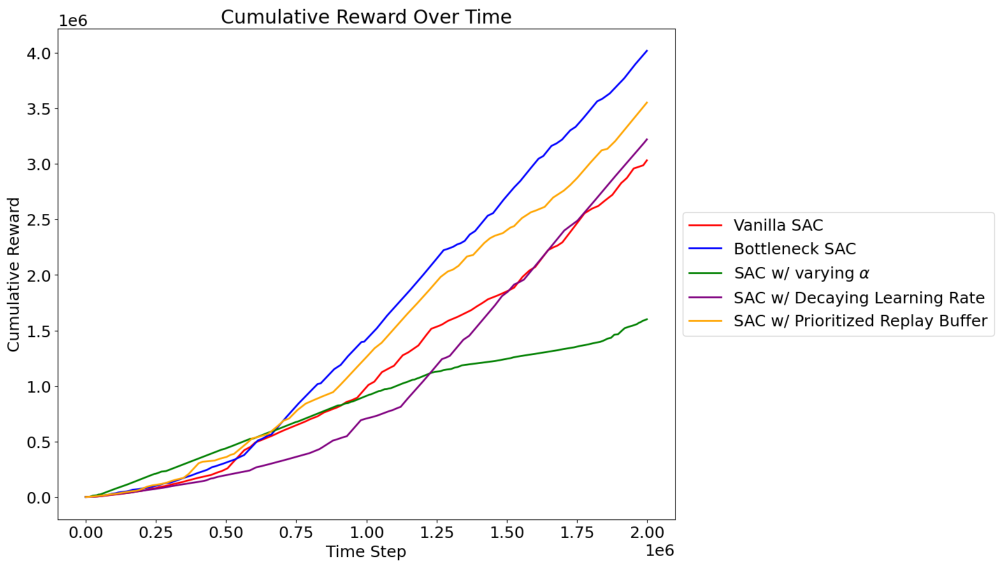


Figure 4: The cumulative reward during training for the improvements made to vanilla SAC discussed in Section 2.5. The rewards are from 1 run of 2 000 000 training time steps, whose 20-step average reward is measured every 1000 time steps.

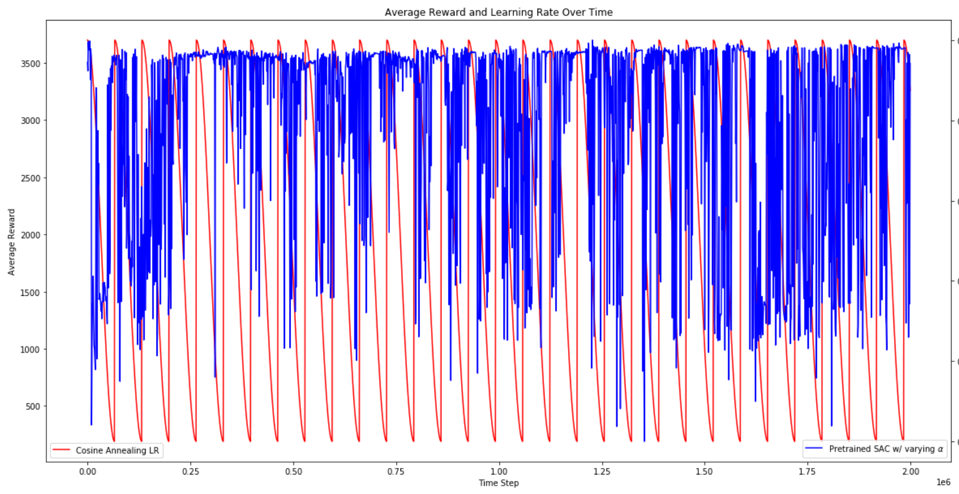


Figure 5: The average reward and learning rate of the SAC agent starting from its best performance checkpoint and using a varying alpha as well as a cosine annealing learning rate.

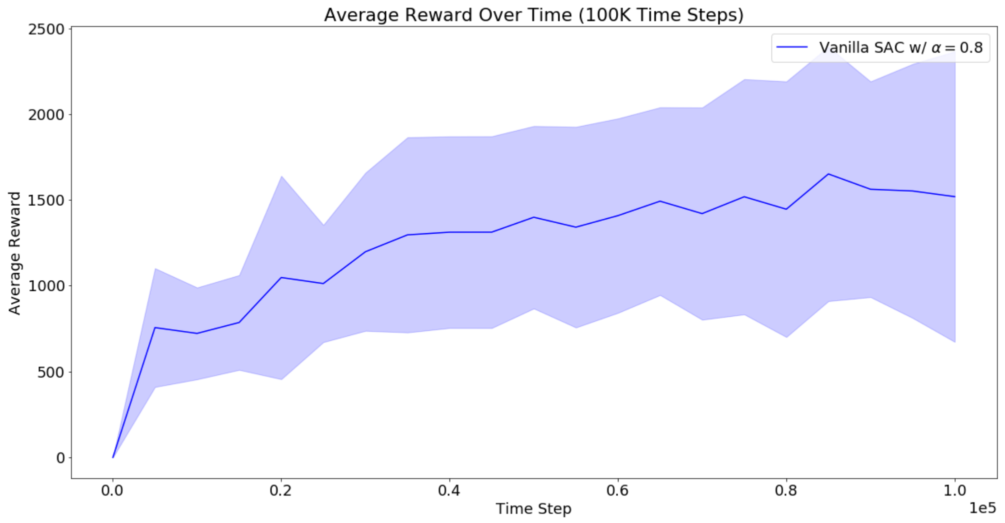


Figure 6: The average reward the Vanilla SAC agent for the first 100K using more frequent training (batch update at the end of every episode) and best α averaged over 5 runs. The error bars correspond the standard deviation of the 5 runs for each time step.

Gravity	Final Performance	Cumulative Training Reward	Sample Efficiency
0.5x default	3144	171 429	922
2x default	3482	281 791	1480

Table 2: Results obtained from training a pre-trained SAC agent for 1 000 000 additional time steps on an environment with modified gravity. Note that the training rewards were sampled at every 10 000 timesteps, rather than the 1000 used for table 1.