RL-Course 2024/25: Final Project Report

Mika Thormann (PPO), Simon Hanrath (SAC)

February 27, 2025

1 Introduction

In this report, we present two reinforcement learning algorithms, Soft Actor-Critic (SAC) and Proximal Policy Optimization(PPO). We evaluate their performance on a 2D hockey environment, where two agents control paddles to hit a puck into the opponent's goal. The observation space is 18 dimensional and includes positions, velocities, and angular movements of both players and the puck. The hockey environment supports continuous and discrete action spaces, where agents can either output real-valued forces for movement, rotation, and shooting or select from predefined discrete actions mapped to equivalent movements. The code for both agents is available on GitHub [1].

2 Soft Actor-Critic

2.1 Method

The Soft Actor-Critic (SAC) algorithm introduced by Haarnoja et al. is an off-policy reinforcement learning algorithm that belongs to the maximum entropy framework [1]. SAC optimizes the policy by combining two objectives: maximizing the expected cumulative reward and maximizing the entropy of the policy. The entropy term encourages the policy to explore more widely with the aim of improved robustness and better generalization. The objective function is thus defined as

$$J_\pi = \sum_{t=0}^{\infty} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right], \tag{1}$$

where the entropy of the policy is defined as

$$\mathcal{H}(\pi(\cdot|s_t)) = -\mathbb{E}_{a_t\sim\pi} \left[\log \pi(a_t|s_t) \right] \tag{2}$$

and its influence on the policy can be adjusted via the α parameter.

The agent encompasses several neural networks. The first is the policy network or actor. It is trained to predict a stochastic distribution over the action space given the current state by maximizing the objective function. The actor decides directly which action the agent takes. We can show that the objective function is equivalent to

$$J_\pi = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[Q_\theta(s_t, a_t) - \log \pi(a_t|s_t) \right]. \tag{3}$$

[1] GitHub Repository: RL_Hockey. Available at: `https://github.com/SimonHanrath/RL_Hockey`

We see that in order to train the actor, we need to have an estimate of the Q-function. The Q-value network, or critic, provides this signal by estimating the expected cumulative reward for a given state-action pair. SAC uses two separate Q-value networks to mitigate overestimation bias, an issue in value-based reinforcement learning where Q-values can become inflated, leading to suboptimal policies. The critic is updated to minimize the difference between the predicted Q-value and the target value derived from the soft Bellman backup equation

$$J_Q = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_{\theta_i}(s_t,a_t) - (r_t + \gamma V(s_{t+1}))\right)^2\right]. \tag{4}$$

Instead of learning a separate value network, which was used in the original formulation of SAC, later versions of the algorithm replace it with the minimum of the two Q-value estimates. This modification eliminates unnecessary redundancy and stabilizes training. The value function is then implicitly estimated as

$$V(s_{t+1}) = \mathbb{E}_{a_{t+1}\sim\pi}\left[\min(Q_{\theta_1}(s_{t+1},a_{t+1}), Q_{\theta_2}(s_{t+1},a_{t+1})) - \alpha\log\pi(a_{t+1}|s_{t+1})\right]. \tag{5}$$

To further stabilize training, we also maintain target versions of the two Q-value networks. These target networks are updated via Polyak averaging, a technique where the target parameters are slowly updated towards the learned critic parameters rather than being directly copied [cite here]

$$\theta_i^{\text{target}} \leftarrow \tau\theta_i + (1-\tau)\theta_i^{\text{target}} \tag{6}$$

This update mechanism prevents strong fluctuations in Q-value estimates and improves convergence and overall training stability.

2.1.1 Automating Entropy Adjustment

As we have seen, the objective function and thus also the policy of SAC depends on the choice of the temperature α. Unfortunately, determining the optimal temperature is not trivial and it needs to be specifically chosen for the given task. Instead of manually tuning the temperature, Haarnoja et al. suggest learning it automatically by optimizing a loss function that adjusts the temperature to match the policy's entropy to a target entropy value H_{target} [2]. The loss function is defined as

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi}\big[-\alpha\cdot(\log\pi(a_t \mid s_t) + H_{\text{target}})\big]. \tag{7}$$

Minimizing this objective ensures that α increases when the policy is too deterministic (low entropy) and decreases when it is too stochastic (high entropy). Haarnoja et al. propose a heuristic for setting H_{target} as the negative of the action space dimensionality, which for our action space corresponds to -4.

2.1.2 Self Play for SAC

Policies trained in fixed-opponent environments may exploit specific weaknesses rather than develop generally useful strategies and are limited by the environment's complexity. To address this problem we decided to pursue self-play. Basic self-play in which we train against previous versions of the agent shows mixed results. The training is unstable and we observe strategy cycling. The agent would repeatedly discover and exploit weaknesses in its opponent, rather than converging to a robust strategy. This resulted in non-transitive learning dynamics, where each new policy would counter the previous one but fail to generalize.
In order to stabilize training and improve robustness we decided to employ a League-Based Training setup with adaptive opponent sampling, inspired by Google Deepminds AlphaStar [5]. We create a pool

containing many different opponents, including the ones provided by the environment, previously trained agents, agents trained against older versions of the league, and the past version of the agent currently training. Before each episode, we sample one of the agents from the league, where the sample probability is determined by the win rate against the opponent. The exact probability of sampling opponent i is given by:

$$P_i = \frac{M - W_i + 1}{\sum_{j=1}^{N}(M - W_j + 1)} \tag{8}$$

where M is the number of recent games we consider and W_i is the number of wins in these last M games. This prioritization aims to ensure a curriculum where the agent initially trains against a mix of opponents to build robust foundational skills, then increasingly faces stronger ones more often while still occasionally encountering weaker opponents to retain past strategies and robustness.

2.2 Evaluation

2.2.1 Evaluation of the Basic SAC Agent

To assess the performance of our SAC agent we train it against both the strong and the weak basic bot that are provided by the hockey environment. Figure 1 shows three training runs against both opponents. After 5000 episodes of training, the SAC agent outperforms both opponents consistently. We also see a relatively minor deviation between training runs against the same opponent, which indicates the stability of the SAC agent during training. As expected, the agent outperforms the weak opponent more quickly than the strong opponent.

Figure 1 also shows the effect of the automated entropy adjustment described in Section 2.1.1. We see that for both the automatic entropy adjustment and $\alpha = 0$ the training converges quickly to an optimal performance. However, the agent trained with automatic entropy adjustment shows a more robust performance against other opponents. For a fixed α, increasing its value results in slower training improvement.

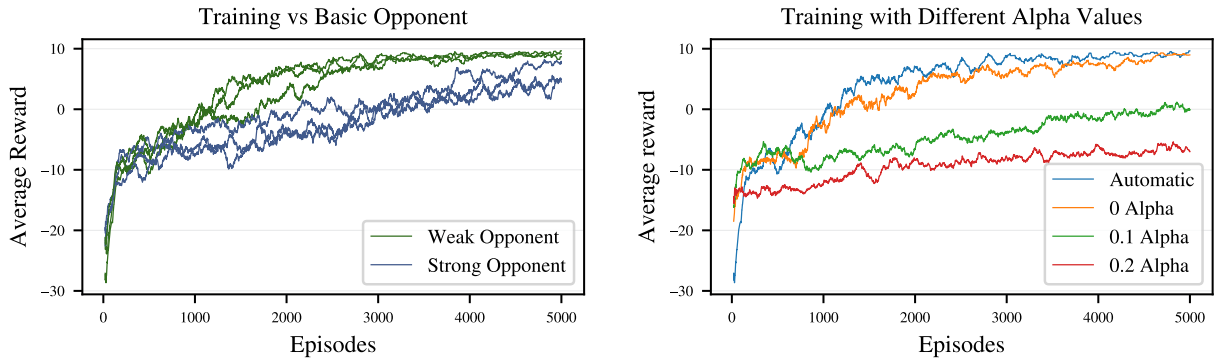

Figure 1: Comparison of training performances: (Left) Three training runs of our SAC agent with automatic entropy adjustment against both strong and weak basic opponents with rewards averaged over the last 100 episodes. (Right) Training runs with different alpha values showing how the average reward evolves.

2.2.2 Evaluation of SAC with League Training

As described in Section 2.1.2 we also train our agent against a diverse set of opponents, including all agents provided by the environment, two SAC agents that have been trained against earlier versions of the league, for each of the environment opponents a SAC agent specifically trained against it, one random agent and a copy of the agent currently being trained from 20 episodes ago. In Figure 2 we see that after the training our agent outperforms all opponents except its own version from 20 episodes ago, which is to be expected. We also see the expected correlation between the frequency of matches and the agent's reward against each opponent. Notably, games are distributed somewhat evenly across the opponents in the beginning, but towards the end of training, matches against the agent's previous self dominate. Effectively the league turns into self-play with occasional games against other weaker opponents to ensure a stable and robust strategy.

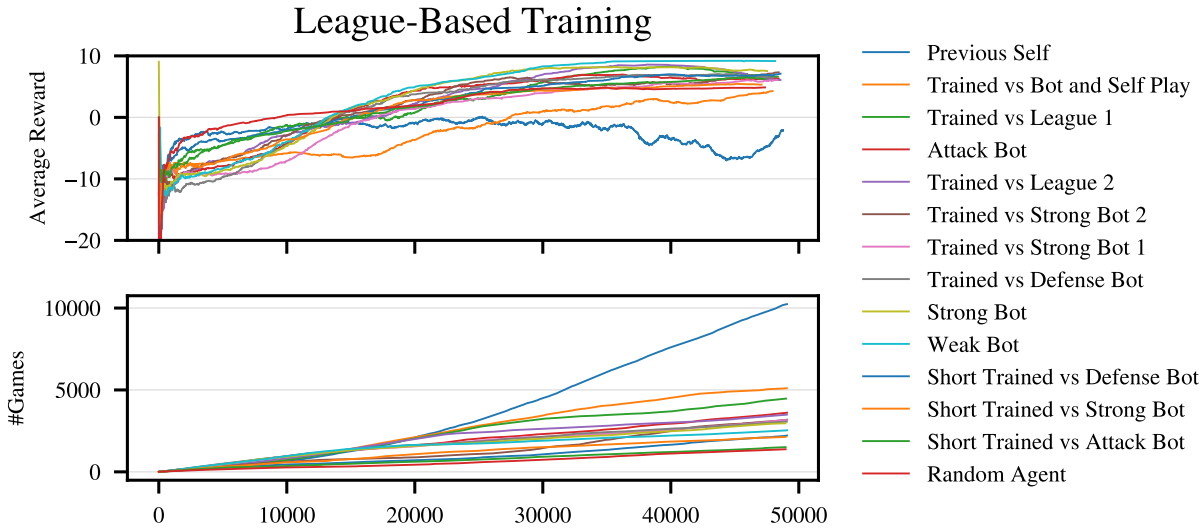

Figure 2: The plots show the training of our SAC agent against the league of opponents. The first plot shows the average reward (over the last 500 episodes) per opponent over the episodes played. The second plot shows the number of games played against each of the opponents over the episodes played.

3 Proximal Policy Optimization

3.1 Method

The Proximal Policy Optimization (PPO) algorithm was first introduced by Schulman et al.[4]. It is built on the concept of Trust Region Policy Optimization[3] (TRPO), aiming to incorporate its ideas in a less complex and widely applicable way. Both are on-policy gradient methods that work with the principle of gradient descent to improve an action policy. The ground objective of these policy gradient methods is to minimize the loss

$$L_{PG}(\theta) = -\hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]. \tag{9}$$

where $\pi_\theta(a|s)$ is the policy and $\hat{A}_t$ is the estimated advantage of the taken action a_t, compared to the previous prediction of that states value. So in general, this loss function encourages high policy probabilities for actions that lead to a larger reward than expected. We could now define a training loop that

iteratively collects data using the current policy and then updates it based on $L^{PG}(\theta)$. While this simple approach looks promising, it often fails in practice, because gradient updates become too large and make the training process unstable. PPO tries to solve this issue by replacing the policy in the loss function with a clipped ratio.

3.1.1 The Clipped Policy Ratio

As an on-policy algorithm, PPO is designed to always be trained on the current version of the policy. But in practice, it relies on collecting batches of data using its current policy. This data is then used for multiple updates of the policy. While this speeds up the training process dramatically, it also introduces an unwanted bias in the loss function. This bias originates from the advantage estimates being computed based on the old policies' actions while updating a newer different policy. To account for this in the loss function, the ratio between the current policy θ and the policy used to collect the data θ_{old} is used to weigh the advantage. This ratio tracks how much the new policy deviates from the old one and leads to more stable updates.

$$L_{CPI}(\theta) = -\hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] = -\hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]. \tag{10}$$

While this change covers the bias problem, it is still possible to receive large gradients when the ratios change drastically during the update steps. To restrain the magnitude of the gradient, PPO introduces a clipping mechanism to limit how much the policy can change between data collection steps. If the policy ratio already changed for more than ϵ, the gradient will be zero for future updates. This is until new data is collected and the ratio naturally resets to 1 because θ_{old} is updated to the current policy. So we end up with the final objective

$$L_{CLIP}(\theta) = -\hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip} \left(r_t(\theta), 1 - \epsilon, 1 + \epsilon \right) \hat{A}_t \right) \right] \tag{11}$$

3.1.2 Advantage Estimation

The introduced objective can be used to train the policy network for our algorithm. As it includes the mentioned advantage estimates $\hat{A}_t$ for every time step t, we need to also predict these values. As suggested by the original PPO implementation, we do this by simultaneously training a value network to predict the discounted reward of the remaining episode. The advantage is then calculated as the difference between the collected true discounted reward $V_{target}(s_t)$ and the value estimation $V_\theta(s_t)$ as

$$\hat{A}_t = V_{target} - V_\theta(s_t) = \sum_{i=t}^{T} \gamma^{i-t} r_i - V_\theta(s_t) \tag{12}$$

where T is the final time step in a recorded episode. Advantages model, if the action taken was surprisingly good or bad, instead of returning the raw discounted reward. This prevents the policy from further updating for already expected outcomes, stabilizing training, and reducing variance. To receive a prediction for $V_\theta(s_t)$, we optimize the value net $V_\theta(s_t)$ with respect to the squared loss between the estimates and targets.

$$L_V = \hat{\mathbb{E}}_t (V_\theta(s_t) - V_{target})^2 \tag{13}$$

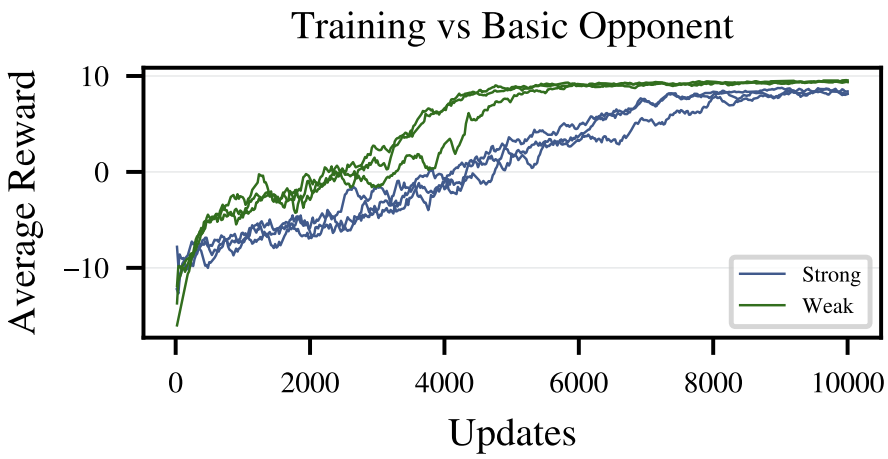

Figure 3: Sample reward evolutions against the basic weak and strong opponent. The curve is smoothed and rewards are collected as the average reward between 20 updates.

3.1.3 Exploration Through Entropy

To prevent our policy from optimizing towards only one earlier learned strategy, without exploring different approaches, we would like to introduce a third loss term. For that, we introduce L_E which is simply the negative entropy of the policy. A higher entropy encourages a policy with less certainty, preventing it from focusing on just a few actions.

$$L_E = -\hat{\mathbb{E}}_t H(\pi) = -\hat{\mathbb{E}}_t \sum_a \pi(a|s_t) \log \pi(a|s_t) \tag{14}$$

3.1.4 Combined loss

As mentioned earlier, we would like to train policy and value network in parallel. So we can define our final loss function as a combination of the three parts. To control the amount of exploration, we include an additional hyperparameter c_E and receive our final loss

$$L_{final}(\theta) = L^{CLIP}(\theta) + L_V(\theta) + c_E * L_E(\theta) \tag{15}$$

3.2 Evaluation

3.2.1 The Training Loop

We trained the PPO agent by letting it compete with a pre-build bot that comes in a weak and strong variant. Before each update, 5000 steps in the environment are recorded and the old data is discarded. Each update includes 4 weight updates on different random batches of 512 steps. These parameters proved to maintain a good balance between sampling efficiency and variance. The reward function sparsely rewards the player for winning or defending their own goal. It also includes a reward for closeness to the ball at every time step when defending. This reward helps to guide the agent, especially in the earlier stages of training. Figure[3] shows that the performance against the weak agent rises much faster and begins to saturate after about 5000 updates, while the performance against the strong agent only begins to saturate after about 10000 updates.

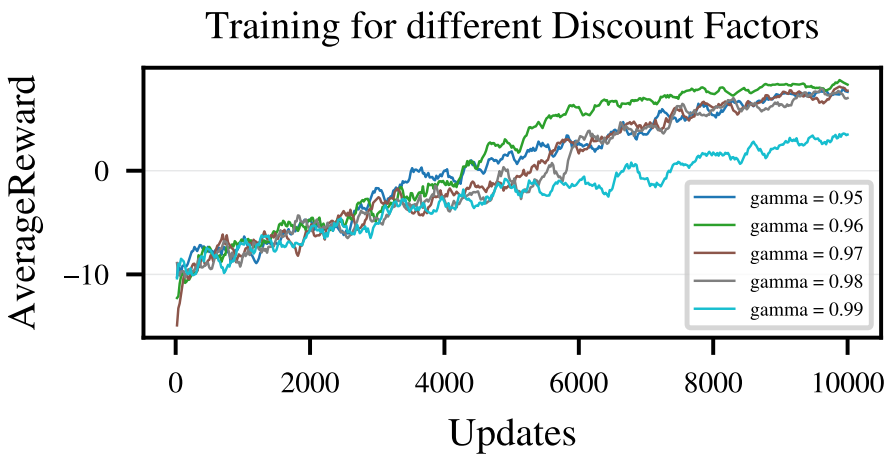

Figure 4: Sample reward evolutions for different discount factors "gamma". The curve is smoothed and rewards are collected as the average reward between 20 updates.

3.2.2 Discount Value Tuning

While we tried to tune many of the hyperparameters, the discount value was the one that improved our training performance the most. We started with a basic value of $\gamma = 0.99$ but realized that lower values work better for our given problem setting. Figure[4] shows that lower values lead to quicker convergence and a better final reward. We did not observe any further improvements when lowering γ further than 0.96, so we kept that value for further experiments.

3.2.3 Network Architecture

We started with a simple 2-layer MLP architecture with a width of 256 for the policy and value networks. Our results show that this approach is sufficient to confidently beat the strong opponent with a win rate of 96% after 10000 updates. Further investigation revealed that even a single-layer network with a width of 128 is sufficient to solve this task and reach on-par performance.
While predicting continuous actions for the 4-dimensional action space seems to be the obvious output choice, we received better results with a 28-dimensional discrete architecture, covering the agent's movement, rotation, and all possible combinations of the two. That is why we opted for this approach in all of our tests.

4 Discussion

In this report, we explored two reinforcement learning algorithms, Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO) by applying them to a 2D hockey environment.

Both SAC and PPO exhibit strong performance in the hockey environment, beating the strong opponent consistently. However, they both have very distinct properties that make them applicable to different tasks. While PPO is generally better suited for discrete action spaces, SAC is made to operate on continuous action spaces. We also observed this in our project setting, choosing an enlarged discrete action space for PPO. So when a discrete action space is not possible or gets too large, SAC might be the better choice. In addition to that, SAC is generally more sample-efficient than PPO because of its off-policy

nature. This makes it the better choice for tasks where simulating the environment is costly. The main advantage of PPO is its on-policy nature, allowing for more aggressive updates and faster convergence. So it should be preferred over SAC in situations where sample efficiency is no problem and when the problem can be solved in a discrete action space.

In our particular case, the league-based training strategy proved useful for the SAC agent. While both algorithms could beat the strong opponent after training against it, they seemed to overtrain the opponent's strategy and performed poorly in novel off-policy scenarios. Here, the varied curriculum of the league greatly improved robustness, as highlighted by its strong performance against all opponents, including the PPO agent, against which the SAC agent trained against the strong bot struggled [1]. With more time at hand, it would have been interesting to also train the PPO agent against a league of opponents. This would have allowed for a fairer comparison between both agents and an assessment of whether one profits more from the league play than the other.

Winrates in %				
	Strong Opponent	**PPO**	**SAC**	**SAC Tournament**
PPO	96.5	-	58.6	6.2
SAC	99.1	41.4	-	30.3
SAC Tournament	95.5	93.8	69.7	-

Table 1: Winrates in % of row agents against column agents. We made them play 1000 matches against each other and recorded match outcomes to receive the winrates.

References

[1] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1861–1870. PMLR, 2018.

[2] 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. *CoRR*, abs/1812.05905, 2018. arXiv:1812.05905.

[3] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, , et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.