RL-Course 2023: Final Project Report

RLcochet: Filip Radović, Khai Gandini, Vera Milovanović

August 13, 2023

1 Introduction

In this report, we describe our approach to the development of reinforcement learning (RL) agents for the Hockey environment. The main goal is to develop the agents that performs well irrespective of who is their opponent, although we evaluate their performances against the weak basic opponent provided by the environment.
The Hockey environment is a two-player hockey-like game. Both of the players control one agent and the goal is to score a goal against another player. The state space is 18-dimensional and one state contains information about the positions of the players and puck, their velocities and remaining time to hold the puck for both of them. The actions one player can take are 4-dimensional, and they contain information about the position and angle change the agent wants to make and whether it should keep the puck. The environment offered both, continuous and discrete action space. The reward obtained from the environment at each time step t was given by $r_t = r_t^{win} + r_t^{dist}$ where

$$r_t^{win} = \begin{cases} 10 & \text{if agent scored} \\ -10 & \text{if opponent scored} \\ 0 & \text{otherwise} \end{cases}, \quad r_t^{dist} = \begin{cases} f * dist(\text{agent}, \text{puck}) & \text{if the agent scored} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

with f being some factor defined by the environment and dependent on the game mode.
All the algorithms we have implemented were model-free off-policy Actor-Critic algorithms for continuous state and action space. Each of the team members implemented one algorithm:

- **Deep Deterministic Policy Gradient** - Filip Radović
- **Twin Delayed Deep Deterministic Policy Gradient** - Khai Gandini
- **Soft Actor-Critic** - Vera Milovanović

2 Method

2.1 Deep Deterministic Policy Gradient

The methods and their evaluation of the implementation of Deep Deterministic Policy Gradient agent is reported in this section.

2.1.1 Algorithm

In reinforcement learning (RL) one is interested in learning a policy which maximizes the expected returns from the start distribution. Deep Deterministic Policy Gradient (DDPG) was introduced in 2015

by Lilicrap et al. as an Actor-Critic algorithm which tries to solve this problem. [9] Like other Actor-Critic algorithms, it learns both a policy function (actor) $\pi^{\theta_1} : S \to A$ and a state-action value function (critic) $Q^{\theta_2} : S \times A \to \mathbb{R}$ with S and A being the state and action space respectively. The actor and the critic are two neural networks represented by their parameters, θ_1 and θ_2, respectively. Additionally, θ_1' and θ_2' are used for the target networks, as their use makes the algorithm more stable.
The algorithm uses the replay buffer to store the transitions experienced while following some behavior policy. The transitions are of the following form: $(s_t, a_t, r_t, s_{t+1}, d_t)$ with $s_t, s_{t+1} \in S$ being the state at time step t and $t+1$ respectively, $a_t \in A$ being the action taken at the time step t by the agent, $r_t \in \mathbb{R}$ being the reward received at the time step t and $d_t \in \{0, 1\}$ being the done value at the time step t. The behavior policy is usually defined as $\beta = \pi^{\theta_1} + \varepsilon$ where ε is some noise. The noise term ε used in this project is generated by Ornstein-Uhlenbeck process. The actor and critic functions are learned by updating the parameters which minimize their respective losses:

$$L(\theta_1) = -\frac{1}{N} \sum_{i}^{N} Q^{\theta_2}\left(s_i, \pi^{\theta_1}(s_i)\right) \tag{2}$$

$$L(\theta_2) = \frac{1}{N} \sum_{i}^{N} \left(Q^{\theta_2}(s_i, a_i) - y_i\right)^2 \tag{3}$$

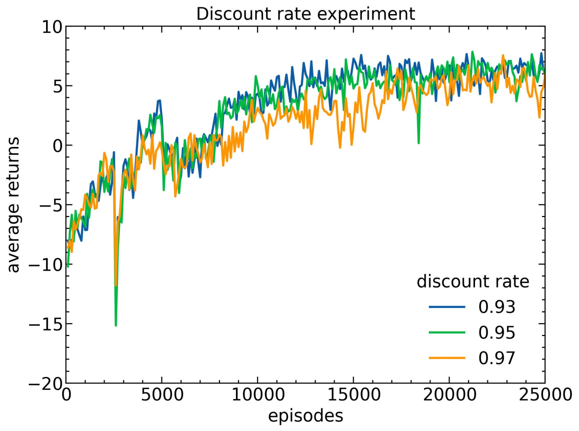
where $y_i = r_i + \gamma(1-d_i)Q^{\theta_2'}\left(s_i', \pi^{\theta_1'}(s_i')\right)$ (also known as TD target) with $\gamma \in [0, 1]$ being the discount factor.
The algorithm works as follows. For each episode, the agent generates the actions according to the behavior policy β and the replay buffer is updated with new policies. After each episode, several fit iterations are done. In each fit iteration, a mini-batch of transitions is sampled from the replay buffer and the actor and the critic parameters θ_1 and θ_2 are updated by minimizing their respective losses. After that, the soft update of the target parameters is done:

$$\theta_i' = \tau\theta_i + (1-\tau)\theta_i' \quad \forall i \in \{1, 2\} \tag{4}$$

with $\tau \in (0, 1]$ being the soft update hyperparameter. Due to the time constraints, some hyperparameters weren't fitted. Both learning rates for the actor and the critic network were set to 0.0003. The batch size was set to 256 and the soft update hyperparameter was set to 0.005.
The actor and critic were two neural networks with one hidden layer and its hidden size was 256. Non-linearities used in the networks were ReLU.
The evaluation of the agent was done every 500 episodes, and one evaluation against a single opponent would consist of 200 episodes. The number of maximum time steps per episodes was 250.

2.1.2 Modifications and their Evaluation

All of the following hyperparameter tuning and modifications were evaluated on the Hockey environment. All the following metrics were obtained by playing against the weak basic opponent which is one of two basic opponents provided by the environment (the other one is the strong opponent). Three game modes are also provided by the environment: *TRAIN_DEFENSE*, *TRAIN_SHOOTING* and *NORMAL* mode. All of the following experiments follow one basic curriculum which makes the agent to learn in *TRAIN_DEFENSE* mode for the first 2500 episodes. Then, it learns in *TRAIN_SHOOTING* mode for the next 2500 episodes. After that, for each episode the game mode is sampled from the uniform distribution over all possible game modes.
The only environment-specific modification which cannot be generalized to the other environment was



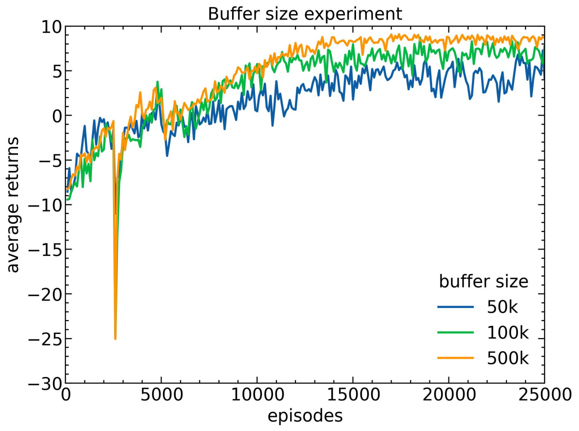


Figure 1: The results of hyperparameter tuning. The sharp falls of average returns in the episode 2500 happen because of the switch between the defense and shooting mode.

the exploitation of one symmetry with respect to the $y = 0$ axis in the environment. Let the mirrored version $m_S(s)$ of a state $s \in S$ be the state which is obtained by multiplying the parts of the state related to the y-axis and angles by -1. Analogously, let the mirrored version $m_A(a)$ of an action $a \in A$ be the action obtained by multiplying the parts of the action related to the y-axis and angles by -1. Then the following holds: $Q(s, a) = Q(m_S(s), m_A(a))$ and $m_A(\pi(s)) = \pi(m_S(s))$. Encoding this information into the actor and the critic makes the use of the collected experience more efficient.

Firstly, the tuning of the **discount rate** was done. The results of tuning can be seen in the Figure 1. The tuning was done with the buffer size of 50000 and the batch size of 128. Three values of the discount rate were tested: $\{0.93, 0.95, 0.97\}$. None of the evaluated values proved to be better than the others, so in the subsequent experiments the discount rate was set to 0.95.

The another hyperparameter tuning was of the **buffer size**. With the standard batch size 256, three values for the buffer size were tested: $\{5 \times 10^4, 1 \times 10^5, 5 \times 10^5\}$. The increase in average returns is notable when increasing the buffer size. Thus, for the buffer size 5×10^6 is chosen.

One of the modifications was the addition of the **resetting** functionality. The results of modification tests can be seen in the Figure 9. It was shown that resetting the actor and critic networks after certain number of environment steps while keeping the replay buffer can lead to finding the policies with larger returns. [10] Since in the implementation of DDPG that was used in the project there was no direct relation between the number of environment steps and gradient updates, but rather between the number of episodes and gradient updates, a reset would be performed after every 2×10^5 gradient updates. The use of the resetting did not show any improvement in the performance of the agent.

Another significant modification of the algorithm was the addition of the **DR3 regularization** term to the action-state value function. [8] The regularization term is actually the dot product between critic's last layer feature representations of the subsequent state-action pairs normalized by regularization coefficient c. The critic loss becomes then:

$$L(\theta_2) = \frac{1}{N} \sum_{i}^{N} \left(Q^{\theta_2}(s_i, a_i) - y_i \right)^2 + c \cdot \phi^{\theta_2}(s_i, a_i) \phi^{\theta_2}(s'_i, a'_i) \tag{5}$$

Three different values for c were tested: $\{0, 0.1, 0.001\}$ with $c = 0$ corresponding to no regularization. No benefits have been noted when using the regularization, so it was not used.

The next challenge encountered during the project was how to efficiently train the agent. The basic curriculum learning introduced at the beginning of the section was not sufficient to prepare the agent for

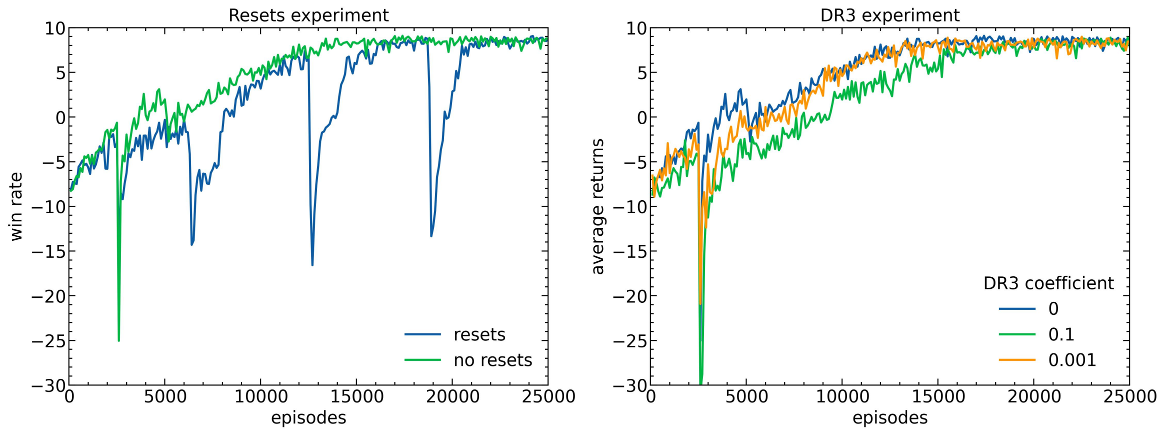


Figure 2: The results for two modifications.

the tournament. Also, it was not defined what is the right use of the basic opponents provided by the environment. The idea implemented as the experiment in this project was to create a buffer containing the opponents. One of the possible options was to train the agent first against the weak opponent, then the strong opponent and at the end to start the training using self-play. This way of training was shown to lead to chase cycles in which the advanced version of the agent can defeat the intermediate version and lose against the weak version of itself while the intermediate version wins against the weak one. [11]
The solution that was proposed by us was to create an **prioritized opponent buffer** (POB) O which would initially contain only the weak opponent. After reaching α_{wr} win rate against the weak opponent, the strong opponent would be added to the buffer. Then after reaching α_{wr} win rate against the both of opponents in the buffer, a clone of the agent at that time would be added. Then again, a new clone of the agent would be added to the buffer after α_{wr} win rate has been attained and so on. The opponent buffer is used in the following way: on every n_{update} episodes during the training an opponent is sampled according to some strategy from the buffer to play against the agent which is being trained for the next n_{update} episodes. Sampling the most difficult opponent for the agent which is being trained, i.e. the opponent that will minimize its expected returns seemed like a good sampling strategy. At this point, the problem of finding the strategy for sampling the opponent looks like this:

$$o^* = \underset{o \in O}{\operatorname{argmin}}\ \mathbb{E}_\pi[R_1 \mid o] = \underset{o \in O}{\operatorname{argmax}}\ (-\mathbb{E}_\pi[R_1 \mid o]) \tag{6}$$

with π being the policy followed by the agent at the moment of the sampling. This problem can be then formulated as a multi-armed bandit problem where the opponents are the arms and their expected rewards change over the time so the problem is non-stationary. In order to solve this problem, Discounted UCB (D-UCB) is used. [7] [4] D-UCB returns the opponent which maximizes the sum of discounted empirical average X and discounted padding function c:

$$I = \underset{o \in O}{\operatorname{argmax}}\ X(\gamma, o) + c(\gamma, B, \xi, o) \tag{7}$$

Due to the time constraints, the parameters of the discounted padding function were not tuned, but γ was set to 0.95, while B and ξ we set to 1.
For the tournament, the agent was trained for 2×10^5 episodes with $n_{update} = 50$ and it ended its training with eight opponents in the buffer which means that the last opponent in the buffer was able to defeat all other opponents from the buffer with $\alpha_{wr} = 0.95$ win rate.

2.2 Twin Delayed Deep Deterministic Policy Gradient (TD3)

While DDPG combines actor-critic stability with neural networks for learning efficient policies in continuous action spaces, its original architecture induces issues such as Q-value overestimation and value function instability, thus inspiring subsequent works like Twin-Delayed DDPG (TD3)[3], which forms the basis of our modifications.

2.2.1 Vanilla DDPG

Given an environment, we aim to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ over actions A conditioned on the environments states $\mathcal{S}$ such that our agent maximizes the expected reward r with respect to its policy. DDPG[9] learns the Q-function approximation Q and policy π jointly using an actor-critic architecture modeled via neural networks parameterized with θ_Q and θ_π, respectively. The actor network is responsible for learning the policy, i.e. mapping states to actions, while the critic network approximates the Q-function which is used to update the actor's policy by following its gradient.
The loss function used for the critic network θ_Q penalizes the TD error between its estimated Q-value $Q(s,a)$ and the actual reward r plus the γ-discounted Q-value of the target critic $Q_{\text{target}}(s', \pi(s'))$ for the next state s' and the action generated by the actor policy π:

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (r + \gamma Q_{\text{target}}(s', \pi(s')))\right)^2\right] \tag{8}$$

where state s, action a, reward r, and the next state s' are sampled from the replay buffer $\mathcal{D}$, which stores the agent's experiences. The target critic Q_{target} is simply a copy of the critic network θ_Q and is gradually updated towards the current network via the soft update method and can be controlled with the hyperparameter τ:

$$\theta_{\text{target}} = \tau \cdot \theta_Q + (1-\tau)\cdot \theta_{\text{target}} \tag{9}$$

Lastly, the actor network's parameters θ_π are updated using gradient ascent on the expected Q-value with respect to the actor's parameters, which encourages it to learn actions that lead to higher Q-values. The update for the actor's parameters can be expressed as:

$$\nabla_{\theta_\pi} J \approx \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q(s,a|\theta_Q)|_{a=\pi(s)} \nabla_{\theta_\pi}\pi(s|\theta_\pi)\right] \tag{10}$$

where J represents the expected return wrt. the policy, and the gradients are approximated using samples from the replay buffer $\mathcal{D}$.

2.2.2 Twin-Delayed DDPG (TD3)

While DDPG is able to effectively approximate the Q-value in many cases and thus optimize its policy, there have been several limitations observed with the original implementation. Most noticeably, it suffers from Q-value overestimation bias, causing the policy to exploit the accumulating error and leading to overall unstable policies and generally high variance[3]. TD3 introduces the following improvements on vanilla DDPG that aim to tackle these problems:
Clipped Double Q-learning. TD3 employs a set of two critics (Q_1, Q_2) instead of a single critic network to combat Q-value overestimation. Using the minimum of the two Q-values for target updates (clipping) reduces the accumulation of biased values. The loss function for the critic update thus becomes:

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (r + \gamma \boldsymbol{\min_{i=1,2} Q_{\theta'_i}}(s', \pi(s')))\right)^2\right] \tag{11}$$

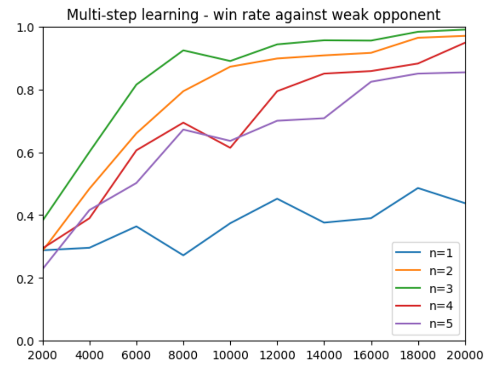


Figure 3: Different values for n. Setting $n > 1$ improves performance significantly over $n = 1$ (single-step).

Although this update rule might reversely induce underestimation bias, this is preferred over overestimation bias, since it does not get directly propagated through the policy update [cite].

Delayed Policy Updates. In the original DDPG, the actor's policy is updated after every Q-function update, which can lead to unstable policies, especially when the critic networks have not fully stabilized since their update. Therefore, [] propose delaying the policy update to every other critic update, which curbs learning from a value estimate with high variance. While decreasing the frequency even further would reduce the accumulation of error as well, training would be slowed down considerably as well.

Target Policy Smoothing Regularization. To address the issue of overfitting to narrow spikes in the Q-value estimation, TD3 adds random noise ϵ to the target policy during the critics updates, which essentially has a smoothing regularization effect on similar actions and helps avoid getting stuck in local patterns. For simplification, we henceforth use y to denote the TD target used in the loss function:

$$y = r + \gamma Q_{\text{target}}(s', \pi(s') + \boldsymbol{\epsilon}), \quad \epsilon \sim \text{clip}\left(\mathcal{N}(0,\sigma), -c, c\right) \tag{12}$$

2.2.3 Modifications

Based on the improved TD3 version of the DDPG algorithm, which mainly focused on rectifying the overestimation bias, there are numerous other extensions that can be added to enhance the actor-critic architecture. Similar to the extensions on DQN as examined in the Rainbow paper [6], we can also apply some of them to TD3. Here, we further extend the TD3 network with Multi-step Learning and Prioritized Experience Replay, which were the two modifications observed to have the most impact on DQN, and analyze their effect on TD3.

Multi-Step Learning. As seen in (11), TD3 computes the TD error based on the immediate reward r and the Q_θ-value estimate of taking a single step given the next state s'. The idea of multi-step learning is to simply take n consecutive time steps and replace the single-step reward r with the accumulated, γ-discounted rewards after n steps and the next state s' with the state after n steps s_{t+n}:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}, \tag{13}$$

$$y = \boldsymbol{R_t^{(n)}} + \boldsymbol{\gamma_t^n} \min_{i=1,2} Q_{\theta_i'}(\boldsymbol{s_{t+n}}, \pi(\boldsymbol{s_{t+n}})) \tag{14}$$

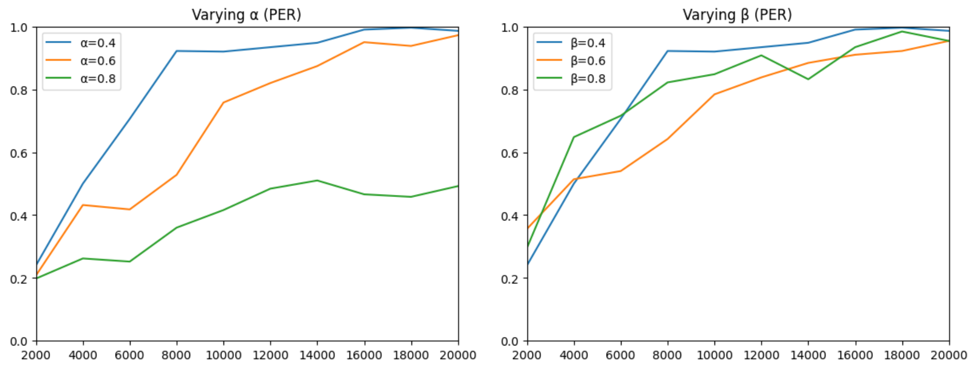


Figure 4: Performance comparison with different values of α and β.

By instead incorporating multiple consecutive rewards and values into the update process, multi-step learning enables the agent to capture delayed consequences of actions over longer time steps, and thus to hopefully reason about long-term dependencies and goals. However, when using multi-step returns, the i.i.d assumption between experiences becomes violated, as the value estimate for one experience becomes dependent on the values of previous experiences in the sequence. Empirically however, when using a moderate number of steps n it has been shown to boost performance regardless[6].
Fig. 3 shows performance on different choices of n (1-5), where $n = 1$ reduces to the single-step learning TD3. We observe that the choice of n has a significant impact on performance, with $n = 3$ learning at a much faster pace and converging to a win rate of >0.9 after only 12000 episodes, while $n = 1$ struggles even after 20000 episodes.

Prioritized Experience Replay (PER). While TD3 samples experiences $e_i = (s, a, r, s')$ from the replay buffer $\mathcal{D}$ uniformly, Prioritized Experience Replay tries to assign a priority to each experience based on the size of its TD error. Experiences with higher TD errors are then more likely to be sampled, thus giving higher weight to experiences that the agent can potentially learn more from.
When sampling a mini-batch of experiences from the replay buffer, we set the probability of selecting an experience e proportional to its priority:

$$P(e) = \frac{p_e^\alpha}{\sum_i p_i^\alpha}, \tag{15}$$

where α is a hyper-parameter that controls the prioritization level. When $\alpha = 0$, all experiences have the same probability of being selected, reducing the sampling again to a uniform distribution.
To compensate for the bias introduced by prioritized sampling, we adjust the weights as follows:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(e_i)}\right)^\beta, \tag{16}$$

where N is the size of the replay buffer and β controls the compensation for the bias.
Similar to the experiments conducted with n for multi-step learning, we test different values of α and β to understand their impact on the TD3 algorithm's performance. Fig. 4 shows the performance of TD3 with different values of α and β. We observe that selecting appropriate values for these hyper-parameters can lead to significant improvements in the learning speed and overall performance. We first fix β to 0.4 to find a good choice for α. A lower α (0.4) converges much faster, while a high α (0.8) learns much slower. Fixing now $\alpha = 0.4$, we explored different values for β. While β does not seem

to impact performance as much, we can still see an overall improvement with a tuned β (0.4) as well. Overall, our results indicate that $\alpha = 0.4$ and $\beta = 0.4$ provide a good balance between prioritization and compensating for the bias, resulting in faster convergence and higher win rates.

2.3 Soft Actor-Critic

The methods and the evaluation of the implementation of Soft Actor-Critic (SAC) agent is reported in this section.

2.3.1 Algorithm

Soft Actor-Critic [5]. is similar to the previously discussed TD3 algorithm, as it learns three functions: a policy (actor) $\pi^{\theta_1} : S \to [0,1]$, a state-action value functions (critic) $Q^{\theta_2}, Q^{\theta_3} : S \times A \to \mathbb{R}$ with S and A being the state and action space respectively. SAC uses clipped double-Q learning to form the targets, but the next-state actions used in the target come from the current policy instead of a target policy. Like in TD3, the shared target is computed using target Q-networks, which are obtained by polyak averaging their parameters. Moreover, $Q^{\theta_2}, Q^{\theta_3}$ are learned by MSBE minimization, by regressing to a single shared target.
However, unlike the previous two algorithms, SAC optimizes an inherently stochastic policy during the training. It means that the policy function maps each state from the sate space S to a probability distribution over actions from the continuous action space A as $\pi^{\theta_1} : S \to [0,1]$. That is, given the state, the agent chooses an action randomly based on the probability distribution, for which we take a Gaussian distribution with the mean and standard deviation obtained by the actor's feed-forward network. The advantage of learning the stochastic policy is that it captures the uncertainty in the environment and should prevent the agent from getting stuck in sub-optimal policies. For taking the actions when evaluating the policy we use remove the stochasticity, by taking an action as $a = \arg\max_a \pi^{\theta_1}$, instead of sampling from the policy distribuion.
However, the main feature of SAC is the modified objective function that incorporates the entropy measure of the policy into the reward. It serves as a regularizer that controls the randomness of the policy, as it encourages exploration by learning a policy that acts as randomly as possible while it is still able to succeed at the task. Hence it tackles the exploration-exploitation dilemma. A higher entropy of the policy encourages more exploration, while a lower entropy results in more exploitation, which is adjusted by the temperature parameter α. The optimal policy is given by

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi} \sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \tag{17}$$

where the entropy is given by

$$H\left(\pi(\cdot|s_t)\right) = \mathbb{E}_{a\sim\pi(\cdot|s_t)}(-\log \pi(\cdot|s_t)). \tag{18}$$

The entropy maximization leads to policies that can (1) explore more and (2) capture multiple modes of near-optimal strategies (i.e., if there exist multiple options that seem to be equally good, the policy should assign each with an equal probability to be chosen). SAC is sensitive to the temperature parameter since the entropy can vary unpredictably both across tasks and during the training as the policy changes[12]. Hence, the authors proposed learnable temperature parameter, which is obtained by minimizing the following objective function:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi_t}\left[-\alpha \log \pi_t(a_t|s_t) - \alpha H_0\right] \tag{19}$$

, where H_0 is a predefined minimum policy entropy threshold, for which the value of negative action space dimension was taken, as proposed in the original paper [5].

2.3.2 Modifications

Since SAC is the off-policy algorithm, it typically collects some experiences in the replay buffer at the beginning of the training process. Usually, these experiences are controlled by random actions that are sampled from uncorrelated Gaussian distribution. In many cases, white noise exploration is not sufficient to reach relevant states. Consequently, not adequate early state space exploration could potentially significantly degrade the convergence rate. The off-policy nature of SAC makes it possible to replace the white noise process by a different random process. In Pink Noise is All You Need paper[2] it has been proposed to use pink noise as the default action noise, as it led to better state space exploration. It is given by

$$|\hat{\epsilon}(f)|^2 \propto f^{-\beta} \tag{20}$$

, where $\beta = 2$ in the case of the pink noise, and $|\hat{\epsilon}(f)|^2$ is the power spectral density (PSD). We incorporate the pink noise by sampling the sequence of actions at the beginning of each episode, taking the action each environment step, and putting the transition to the replay buffer. We discuss the result of the experiment in the next section.

Hyperparameter	Value
number of hidden layers	1
width of the hidden layer	256
nonlinearity	ReLU
target smoothing coefficient	0.05
target update interval	1
number of gradient steps	32
discount rate	0.95
learning rate	10^{-3}
oprimizer	NAdam
max. buffer size	10^5
batch size	128
target update interval	1
automatic entropy tuning	True
buffer action noise	white

Figure 5: Baseline hyperparameters.

We also make use of the MDP symmetry and prioritized opponent buffer (POB) for sampling the opponents during the training, as described in the Section 2.1.2. The latter modification significantly improves the robustness of the agent, as it doesn't overfit to the opponent that it played with last. The parameters taken and results are discussed in the following section.

2.3.3 Experiments

LunarLander

The implementation of the SAC algorithm (which was inspired by the CleanRL blogpost[1]) is checked in the LunarLanderContinuous-v2 environment. It is also continuous action space environment, but simpler to solve compared to the Hockey environment. For the first experiment, we use the hypreparameters proposed in the original SAC paper. The result can be seen in the Figure 6. We conclude that the reward is optimized and the behavior is reasonable. Additionally, we test here if the hypothesis about sampling the actions from the pink noise process would lead to better exploration. The result is compared to the baseline algorithm with the actions sampled from the white noise in the Figure 6. Moreover, using the pink noise for sampling the trajectories leads to faster convergence compared to the case with white noise. As expected, the difference is present only at the beginning of the training, as the initial trajectories are replaced after 1000 environment steps with the newly col-

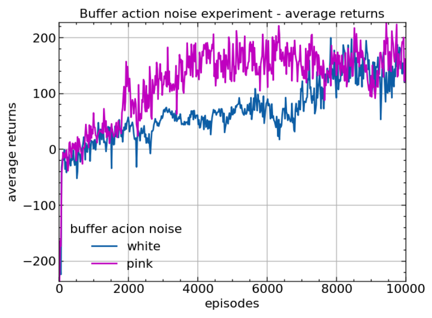


Figure 6: Solving LunarLander environment.

lected ones, using the updated policy.

Hockey environment In the fist part of the secation we talk about the hyperparameter search and the effect of some modifications described in the Section 2.1.2. All of the following experiments are done against weak opponent that is provided by the environment. Additionally, they follow one basic curriculum which makes the agent to learn in *TRAIN_DEFENSE* mode for the first 2500 episodes. Then, it learns in *TRAIN_SHOOTING* mode for the next 2500 episodes. After that, for each episode the game mode is sampled from the uniform distribution over all possible game modes.

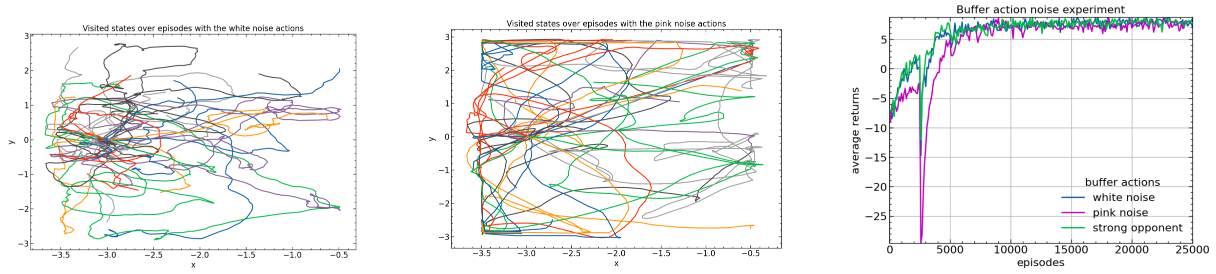


Figure 7: Comparison of the influence of the noise processes in state space coverage (left and middle) and average returns over 20 episodes (right). The sharp falls of average returns in the episode 2500 happen because of the switch between the defense and shooting mode.

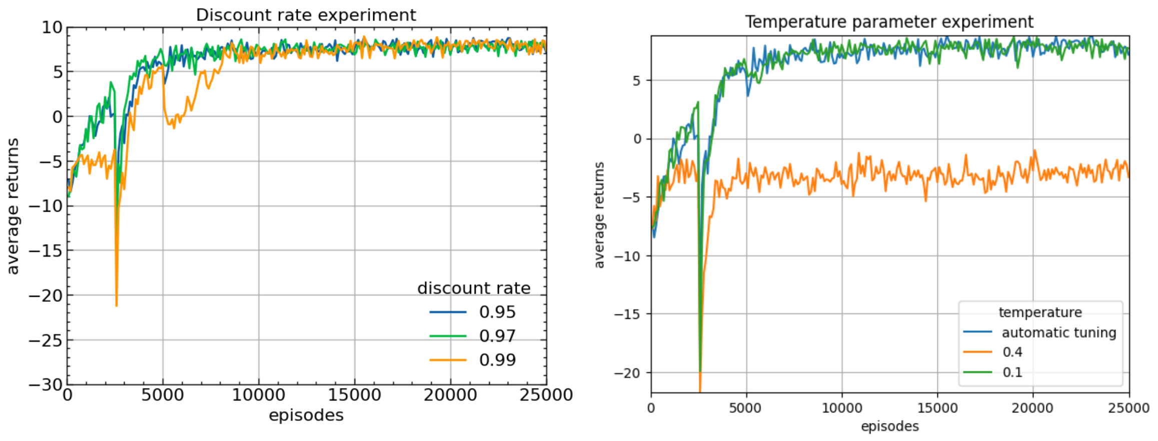


Figure 8: The results of hyperparameter tuning. The sharp falls of average returns in the episode 2500 happen because of the switch between the defense and shooting mode.

Hyperparameters were tuned independently due to the very limited computational and time resources, even though in reality they typically aren't independent. Tuning is done with respect to the baseline model with the hyperparameters given in Table 5. Firstly, **discount rate** is tuned and based on the results in Figure 8. The value of the optimal one is chosen to be 0.95. Secondly, the **temperature parameter** is tuned. We check if the some fixed values are better than the automatic tuning. Based on the results in Figure 8, we choose automatic tuning, as it performs well and we expect that it is more robust once other modifications are added, and also stronger opponents. Varying other hyperparameters by doubling or halving their values don't affect the training process significantly. Therefore, we omit the discussion about them and take the values in Table 5. Lastly, we want to check whether **sampling form the pink noise process** leads to the better exploration of the state space increased expected returns in the early stage on the training process, as it was the case with the LunarLander environment, see Figure 6. We first take a look at the state space coverage, which can be seen in the Figure 7. We conclude that the state space is better explored when actions are sampled from the pink noise process. However, the returns are surprisingly lower, which does not match the behavior in the LunarLander environment. This might be the case as the Hockey environment is much more complex, starting with the fact that there is an opponent whose actions we don't know. Additionally, we try putting strong opponent's actions in the replay buffer, but it doesn't lead to any outstanding improvements, see 7. Therefore, we sample from the white noise.

3 Conclusion

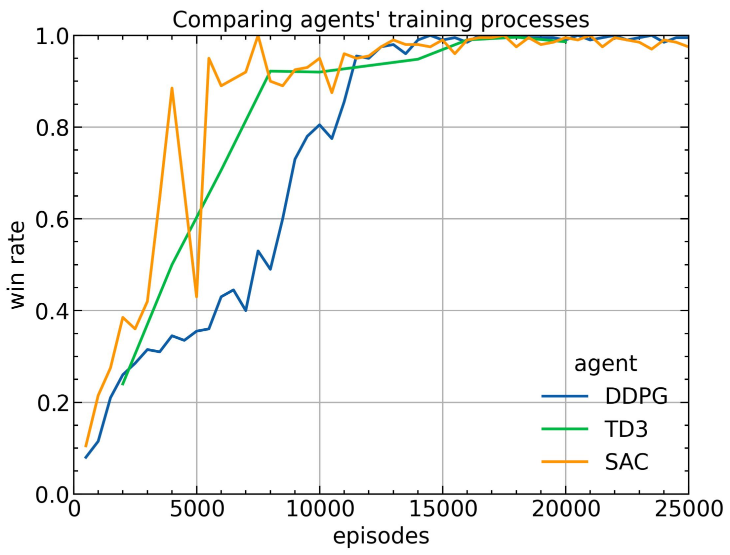


Figure 9: Win rates against weak agent trough the training for each of the implemented algorithms.

In the end we compare the three implemented algorithms and note that all of them manage to learn a policy which allows the agent to win almost in 100% of played matches against the weak opponent. The only notable difference is at the beginning of the training, when TD3 and SAC improve their win rate faster than DDPG.
One of the problems we encountered during the project was the vast hyperparameter space that should have been explored in order to fine suitable values for the problem. Just the use of POB introduced even five new hyperparameters and it requires a lot of computational resources to tune them.
While watching the agents learned using the POB we also noticed that they learned similar policies, they tend to move closer to the center of the field and to shoot directly on the goal. It sometimes seems like it lacks creativity and that it could be fixed by better exploration. After all, the name of this team was derived from the word ricochet, and ironically its agents seem to avoid to make rebounding shots.

References

[1] CleanRL sac. `https://docs.cleanrl.dev/rl-algorithms/sac/`. Accessed: 2023-08-11.

[2] O. Eberhard, J. Hollenstein, C. Pinneri, and G. Martius. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2022.

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

[4] A. Garivier and E. Moulines. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415*, 2008.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[6] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[7] L. Kocsis and C. Szepesvári. Discounted ucb. In *2nd PASCAL Challenges Workshop*, volume 2, pages 51–134, 2006.

[8] A. Kumar, R. Agarwal, T. Ma, A. Courville, G. Tucker, and S. Levine. Dr3: Value-based deep reinforcement learning requires explicit regularization. *arXiv preprint arXiv:2112.04716*, 2021.

[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 preprint arXiv:1509.02971*, 2015.

[10] E. Nikishin, M. Schwarzer, P. D'Oro, P.-L. Bacon, and A. Courville. The primacy bias in deep reinforcement learning. In *International conference on machine learning*, pages 16828–16847. PMLR, 2022.

[11] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[12] L. Weng. Policy gradient algorithms. *lilianweng.github.io*, 2018.