RL-Course: Final Project Report

Kexbot: Onno Eberhard, Sebastian Koch

March 17, 2021

1 Introduction

Playing hockey is a complex task. It requires multiple complex behavioral patterns to score goals and defend your own goal when playing an opponent who wants to do the same thing.
For the hockey tournament we implemented two off-policy, model-free reinforcement learning algorithms: Twin Delayed DDPG (TD3) and Soft Actor-Critic (SAC) are state-of-the-art algorithms which perform well on continuous action tasks.
The Hockey environment is one such continuous action environment. The environment state is also continuous and can be described by this simplified state vector:

$$state = \begin{bmatrix} \text{Player 1 Position} \\ \text{Player 2 Position} \\ \text{Player 1 Velocity} \\ \text{Player 2 Velocity} \\ \text{Puck Position} \\ \text{Player 1's remaining time to hold the puck} \\ \text{Player 2's remaining time to hold the puck} \end{bmatrix} \tag{1}$$

At every timestep the agents can choose a continuous action of the form

$$a = \begin{bmatrix} \text{x change} & \text{y change} & \text{angle change} & \text{keep puck} \end{bmatrix}^T \tag{2}$$

For the agent to learn to score goals and to defend its own goal, we use the following reward function provided by the environment.

$$\text{Reward}(s) = \begin{cases} 10 & \text{if agent scored} \\ -10 & \text{if opponent scored} \\ -\frac{30}{\text{max\_dist}} \cdot \text{dist}(a, \text{puck}) & \text{if the puck is on the agent's side} \\ & \text{and not moving or coming towards the agent} \end{cases} \tag{3}$$

The reward function punishes the agent if it concedes a goal and rewards the agent when it scores a goal. To incentive the agent to play the puck if the puck is in the agents half, a penalty is added that gives negative reward for not being close to the puck.

2 Approach 1: Onno

2.1 Simple algorithms

Premature optimization is the root of all evil — Donald Knuth

In accordance with Occam's Razor, one of the fundamental principles in machine learning, we wanted to start with very simple algorithms and only increase the complexity if necessary. Instead of starting on the Hockey environment directly, we used CartPole-v0, Pendulum-v0 and LunarLanderContinuous-v2 (provided by OpenAI Gym [2]) as stepping stones to solve first.
The simplest of the chosen environments is CartPole, where the available actions are "left" and "right". This environment defines *solved* as achieving an average total reward of 195 or greater. A random agent achieves an average score of 22.4. A trivial rule-based agent which playes "left" and "right" alternatingly achieves a score of 37.9 on average. The observation space of CartPole is continuous, so a learning agent needs to use some sort of function approximation. Maybe the simplest form of function approximation is state aggregation. Here, the observation space is simply split up into pieces such that each piece constitutes a single state. To choose the boundary points of these states we let a random policy explore the observation space, plotted the resulting histograms of the collected observations and split them such that the resulting states would all be visited approximately equally. The final agent was then trained using the standard Sarsa algorithm, which can be used here, because state aggregation turns a continuous observation space into a discrete and finite one. The trained agent achieved an average score of 10.0. Clearly this method is not sufficient for this environment (at least with the hyperparameters we chose: learning rate $\alpha = 0.1$, exploration probability (ε-greedy) $\varepsilon = 0.1$, discount factor $\gamma = 0.9$).
A slightly more complex form of function approximation is linear function approximation, which is actually a strict generalization of state aggregation. In state aggregation, the state-action pair can be written as a one-hot vector $\mathbf{x}(s, a)$ with $|\mathbf{x}|$ being the number of state-action pairs. The approximate Q-function is then $\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s, a)$. In general linear function approximation, the feature vector $\mathbf{x}$ is not restricted to being one-hot. However, it must still be chosen manually (feature engineering) such that the weights $\mathbf{w}$ can be learned to yield a good value function. We chose the feature vector $\mathbf{x}(s, a) = (s(2a - 1), s^2(2a - 1))^\top$. With the hyperparameters $\alpha = 0.1$, $\varepsilon = 0.1$ and $\gamma = 0.5$ we achieved an average reward of 27.3, which is slightly better than random. To train the agent, we used Episodic Semi-gradient Sarsa [10, p. 244]) with the update rule

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}(S_{t+1}, A_{t+1}) - \mathbf{w}_t^\top \mathbf{x}(S_t, A_t))\mathbf{x}(S_t, A_t). \tag{4}$$

This definitely works better than state aggregation, it is also important to note that the feature vector (and thus the number of learnable parameters) is orders of magnitude smaller than in state aggregation. However, this performance is still abysmal. This is probably because the chosen feature vector is not good enough. The obvious step to overcome the need to hand-engineer good features is to learn good representations automatically using backpropagation and deep neural networks. Instead of simply implementing the Q-function as a neural network and using Semi-gradient Sarsa for training, we implemented the DQN algorithm [8], including an experience replay buffer and a target network. The version we implemented is based on the pseudocode provided in the lecture [7], although it is modified to allow specifying how many steps should pass between updates and by adding an ε-decay parameter. The architecture we used had two hidden layers of size 128. The other hyperparameters are $\gamma = 0.99$, the learning rate is 0.001 (the Adam method is used for optimization) and for exploration we started with $\varepsilon_0 = 1$ and $\varepsilon_t = (0.97)^t \varepsilon_{t-1}$. This method was able to solve the environment; after about 50 episodes the agent performed perfectly (score 200 every time). We observed that the training worked much better when no target network was used, but instead the Q-function that was being learned was used to compute the target values, as is shown in Figure 1 on the left.

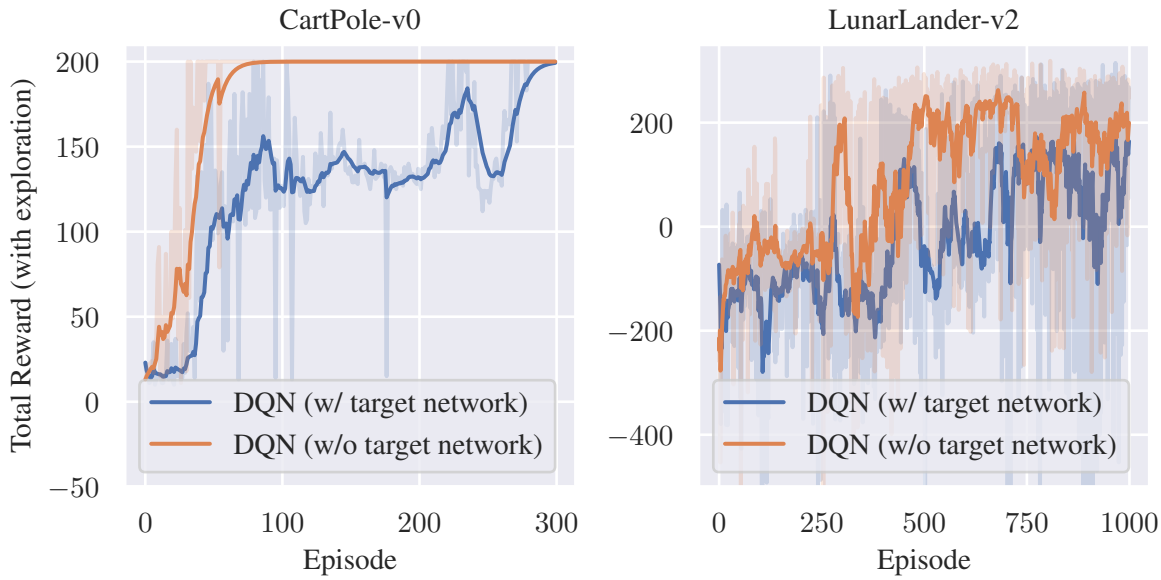


Figure 1: Agents trained using the DQN algorithm. The target network approach does not seem to improve training here.

Having solved the CartPole task, we can now move on to more complex environments. The Pendulum environment cannot be solved using DQN, because it has a continuous action space. The network in DQN takes the observations as input and outputs the predicted Q-value for each available action (the output size is the number of available actions). Clearly this kind of architecture is incompatible with a continuous action space. We also trained a DQN agent on LunarLander-v2, which has a discrete action space. The agent mostly solved the environment after about 500 episodes, although the solution is not very stable, see Figure 1 (right). There are versions of DQN that work for continuous action spaces, but the unstable result on Lunar Lander shows that DQN might not be powerful enough to achieve good performance on the more challenging Hockey environment.

2.2 TD3 to the rescue

To solve the environments Pendulum-v0, LunarLanderContinuous-v2 and finally Hockey, we decided to implement the Twin Delayed DDPG (TD3) algorithm [3]. Like DQN, TD3 is an off-policy algorithm, which allows for an experience replay buffer which can contain experiences from previous, outdated policies. This improves its sample efficiency over on-policy algorithms. In our context it is much more efficient to do a policy update using N past experiences (vectorized using PyTorch) than it is to sample N new experiences (slow, using `env.step(a)`), so sample efficiency is important. On the other hand, sample efficiency is not everything, it is still not very costly sampling new experiences, so we decided against model-based methods. This discussion is valid also for the off-policy Soft Actor-Critic algorithm (Section 3). In Figure 10, the training of a PPO-agent (on-policy) is shown on the Hockey environment (mode `TRAIN_SHOOTING`). Compare this to Figures 4 and Figure 7a, of TD3 and SAC, respectively.[1]

[1]This discussion, as well as Figure 10, are a work of both authors

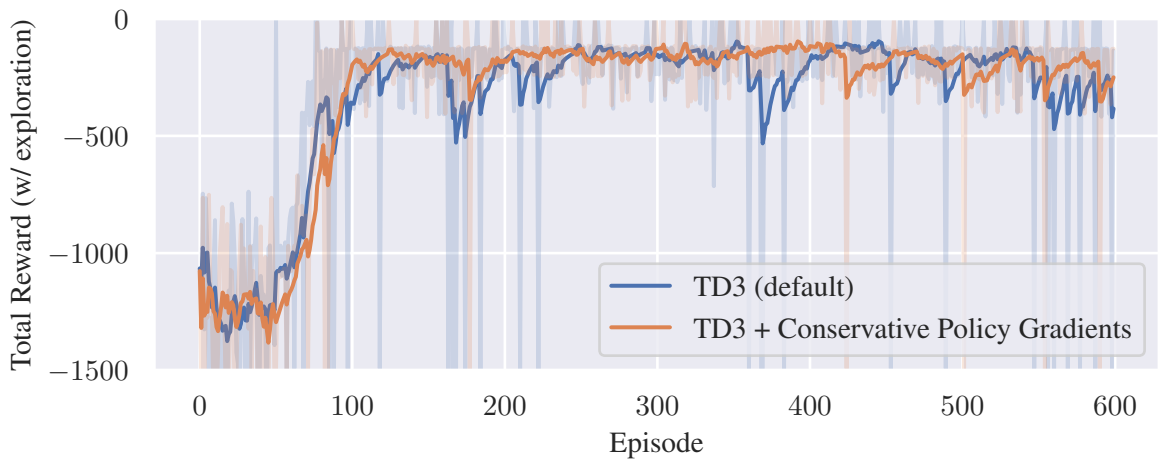


Figure 2: TD3 solves the Pendulum-v0 environment nicely.

2.2.1 Twin Delayed DDPG (TD3)

TD3 is a successor to the Deep Deterministic Policy Gradient (DDPG) algorithm [6]. It tries to solve some problems associated with DDPG. There are three key differences between TD3 and DDPG:

- **Clipped Double Q-Learning**. In DDPG, there is a problem of overestimation of the state-action value $Q(s, a)$. In TD3, this problem is addressed by learning two separate Q-functions: Q_{ϕ_1} and Q_{ϕ_2}. When calculating the target return, the minimum of the two Q-values is chosen, as in Equation (5).

$$y(r, s', d) = r + \gamma(1 - d) \min_{i \in \{1,2\}} Q_{\phi_{i,\text{target}}}(s', a') \tag{5}$$

- **Target Policy Smoothing** is a method used in TD3 to prevent the exploitation of errors in the Q-function. Instead of passing the target action directly from the target policy into the target Q-function, some Gaussian noise is added, as shown in Eq. (6). This results in a more robust policy.

$$a'(s') = \text{clip}(\mu_{\theta_{\text{target}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{min}}, a_{\text{max}}), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{6}$$

- **Delayed Policy Updates**. Instead of updating Q-function and policy at the same time, in TD3, the Q-functions are updated more frequently. Normally, the Q-functions are updated twice as often as the policy.

The version of TD3 we implemented is based on the pseudocode from the OpenAI Spinning Up documentation [1] as well as the original algorithm from the paper [3]. The algorithm works unaltered on the Pendulum-v0 environment (see Figure 2).

2.2.2 Solving Lunar Lander

When training a TD3 agent on the LunarLanderContinuous-v2 environment however, training goes very badly, as can be seen in Figure 3 (in blue). It seems like the training is rather unstable, sometimes performance drops by large amounts and then stays like that. To try and resolve this issue, we implemented a method called *Conservative Policy Gradients* [11]. This method tries to prevent issues with instability during training for DDPG-like methods by changing how the target policy is updated. In standard TD3, the target policy is updated exactly like the target Q-networks, by Polyak averaging. With Conservative

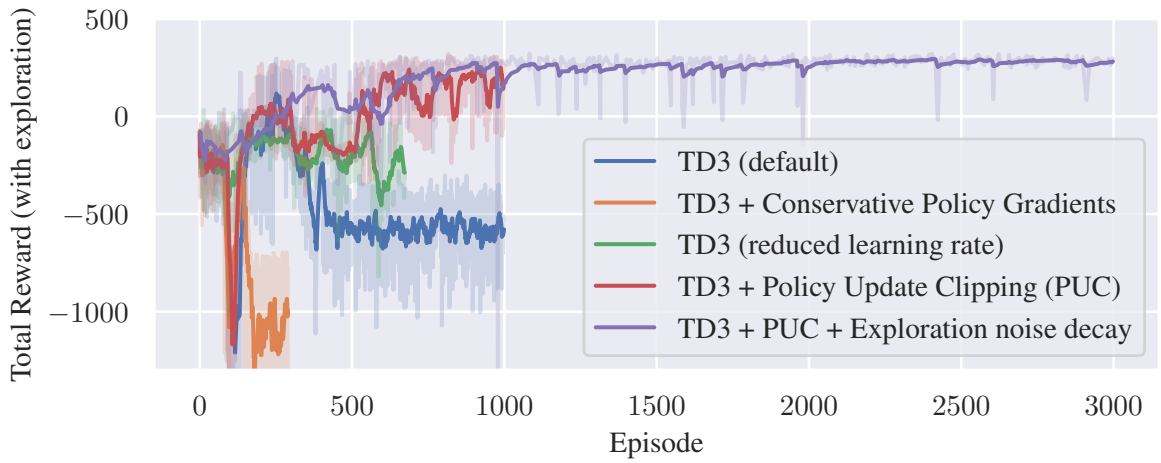


Figure 3: TD3 has trouble solving LunarLanderContinuous-v2 without Policy Update Clipping

Policy Gradients, the target policy weights are updated all at once to the online policy weights; however, this update only occurs if the online policy is better than the target policy. Whether this is the case is tested by simulating a few episodes on both policies and comparing their performances. Additionally, the target policy won't be updated every time the online policy is updated, but much less frequently. Because of the simulation step, this method is quite a bit slower than standard TD3. The results of using this method on the Lunar Lander environment can be seen in 3 (in orange). Clearly, this does not solve our problem. In Figure 2, the Conservative Policy Gradients method is used on the Pendulum environment (in orange), it does not make a big difference here.

When looking at how our agent actually acts, we saw some strange behaviour. After some training, the agent often learned to make loopings (see Figure 11a), or sometimes it learned to shut of all engines at all time (see Figure 11b). Upon further inspection, the reason for this strange behaviour was NaN entries in the policy network's weights. When looking at the loss curves for the policy, it also became apparent that something was wrong, because here, too, NaNs appeared, or simply losses orders of magnitude too large (see Figure 12). The first idea to solve this problem was simply to decrease the learning rate (in this case to 0.0001). This helped to get rid of the NaNs, but also hindered learning to the point where the total reward did not improve over the course of training (see Figure 3, in green).

The next step was then to find out where the NaNs were coming from. They were found in the policy loss, which is defined in Equation (7), using the notation from [1].

$$\mathcal{L} = -\frac{1}{|B|}\sum_{s\in B} Q_{\phi_1}(s, \mu_\theta(s)) \tag{7}$$

The NaNs could emerge either in the Q-network Q_{ϕ_1} or in the policy network μ_θ. We thought it might be possible that the NaNs arise in the Q-network as a result of plugging in actions that are impossible in the given environment. That is, actions $\mathbf{a}$ for which $a_i > a_{\max}$ or $a_i < a_{\min}$ for any component i. This is especially likely, because actions are clipped to this valid range everywhere else in the algorithm, except at this place. A reasonable change to the update might thus be to adapt the loss function, such that the action is also clipped here, as in Equation (8).

$$\tilde{\mathcal{L}} = -\frac{1}{|B|}\sum_{s\in B} Q_{\phi_1}(s, \mathrm{clip}(\mu_\theta(s), a_{\min}, a_{\max})) \tag{8}$$

We call this modification of the algorithm Policy Update Clipping. With the algorithm otherwise unchanged, the result of using this modified loss function when training an agent on Lunar Lander can

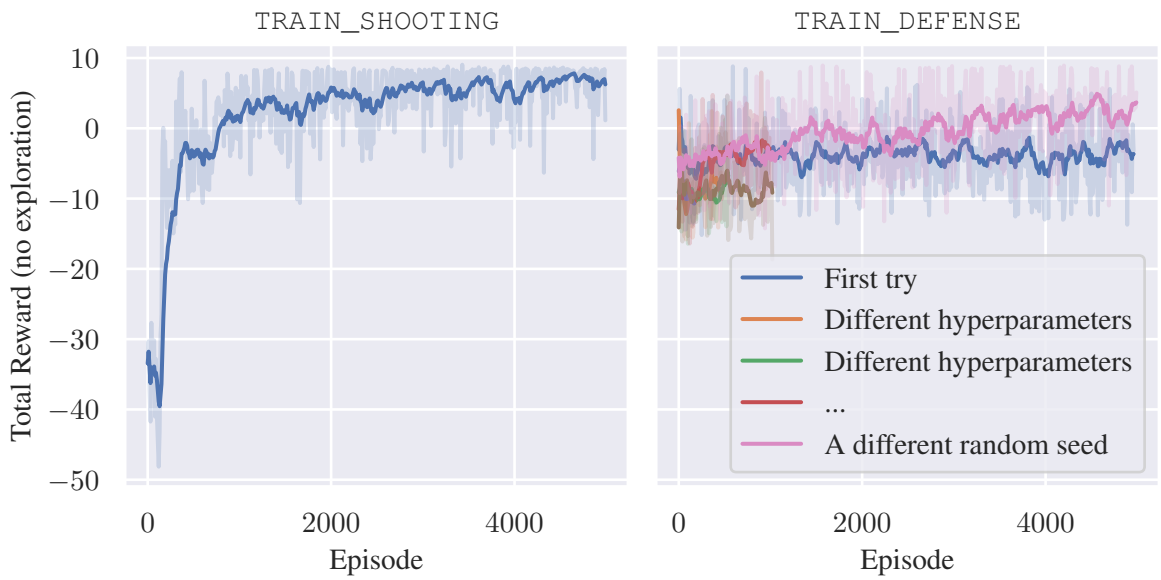


Figure 4: The algorithm goes through the training camp

be seen in red in Figure 3. There is clearly a huge difference. Finally, to increase performance further and to ensure convergence, we increased the network to 3 hidden layers with 256 units each and introduced an exploration decay parameter, such that the exploration noise decreases exponentially during training. The final training curves for Lunar Lander are shown in Figure 3 in purple. After training for 5000 episodes, the agent achieves an average total reward of 283 per episode (solved is 200, the Stable Baselines bechmark [5] for TD3 on LunarLanderContinuous-v2 lists 228 points on average as their result).

2.2.3 Solving Hockey

Now that TD3 has proven itself, it is time to train some agents to play hockey. We started with the training camp, the training curves for `TRAIN_SHOOTING` and `TRAIN_DEFENSE` are shown in blue in Figure 4 on the left and right, respectively. The shooting training seems to go quite well, however the training for defense does not work at all. This time however, there are no NaN values to blame. Something else must be going wrong. After trying many different combinations of hyperparameters to get the training to work (see Figure 4, right), as an act of desperation, we tried to change the random seed with which the random number generators of Python, NumPy and PyTorch are initialized. Surprisingly, this gets the training going (see Figure 4, pink line). Now that the training camp works, the next step is to train the agent to play the provided basic opponent. In Figure 5, the futile attempt to do this is visualized on the left; we tried many different hyperparameters (see colorful lines, except $s = 6$), but nothing seemed to work. Finally the only thing left to try was to fiddle with the random seed again. This time, we did a full grid search for a good random seed. We trained an agent against the weak basic opponent for 300 episodes 10 times, each time setting the random seed to a different number ($s \in \{0, 1, \dots, 9\}$). The result of the search is shown in Figure 5 on the right. When choosing an appropriate random seed, training against the weak basic opponent works very well, shown in Figure 5 on the left in orange ($s = 6$).
For each stage of the training (`TRAIN_SHOOTING` $\rightarrow$ `TRAIN_DEFENSE` $\rightarrow$ weak basic opponent $\rightarrow$ strong basic opponent), a new agent with newly initialized weights was trained. This was done because reusing weights learned by earlier stages did not make any difference in performance or time of training.

Figure 5: The only way to get the training to work is a seed search

For the TD3 agent, self-training did not yield great results, neither did an approach where every second episode was self-training against the previous best policy and every other second episode was against the strong basic opponent. The reason why these attempts were not appropriate is because the agent was not very good against the strong basic opponent in the first place.

3 Approach 2: Sebastian

3.1 Method

3.1.1 Soft-Actor-Critc (SAC)

Soft Actor-Critic (SAC) [4] is an algorithm that optimizes a stochastic policy in an off-policy way. It is very similar to TD3 as it also incorporates the clipped double-Q trick and also is using target policy smoothing.
A central feature of SAC is entropy regularization. The policy is trained to maximize not only the expected return but also the entropy H of the policy. The entropy is a measure of how random the policy is. This is closely correlated with the exploration-exploitation trade-off. A higher entropy results in more exploration while a lower entropy results in more exploitation. The optimal policy can be formulated as

$$\pi^* = \arg\max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \right] \tag{9}$$

with the entropy H being

$$H(\pi(\cdot|s_t)) = \mathop{\mathbb{E}}_{a\sim\pi(\cdot|s_t)} [-\log P(\pi(a|s_t))] \tag{10}$$

There exist two major version of SAC, one where the entropy regularization coefficient α is fixed and one where α is learned during training. (In the experiments sections we look at both a fixed α and a learnable α).

3.1.2 Extension 1: Prioritized Experience Replay (PRE)

As an off-policy algorithm SAC can use a replay buffer to train on past events. In standard SAC the sampling from the replay buffer is random. This works well, but it could be beneficial to sample certain events more frequently than others. For example situations where the agent is already performing very well aren't that beneficial for training. On the other hand situations where the agent performs quite bad, are very beneficial for training.
Therefore instead of sampling random from the replay buffer [9] proposes to sample a transition i with the probability

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{11}$$

where p_i is the priority of transition i. The priorities will be derived from the TD-Error and α is a parameter which regulates how much priority sampling should be done, with $\alpha = 0$ no priority sampling. But priority sampling introduces a bias while training, as the agent will not see all samples uniformly. To mitigate this a weight factor

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right) \tag{12}$$

is needed when calculating the policy loss. With N being the buffer size and β being a linearly annealed exponent which regulates the correction.

3.1.3 Extension 2: Emphasizing Recent Experience (ERE)

Emphasizing Recent Experience without forgetting the past [12] is another method to improve the sample efficiency from the replay buffer and can also be used in combination with PRE.

For this a new term is introduced

$$c_k = \max(N \cdot \eta^{k*\frac{1000}{K}}, c_{min}) \tag{13}$$

which determines how far in the past should be sampled. With c_{min} being a lower bound for the minimal distance to the past and η being the coefficient of how much emphasis should be put on recent data. K being the number of minibatch updates with $1 < k < K$.
This can be used seamlessly with PER together.

3.2 Experiments

3.2.1 Checking the Implementation

To verify the SAC implementation and its extensions the LunarLanderContinuous-v2 environment was used. The environment is well known from past exercises and it is also quite similar to the hockey environment, as it has a continuous action space and it is also using the Box2d style.
For the initial training the hyperparameters from the SAC paper were used. Hyperparameter tuning will only be done for the Hockey environment.

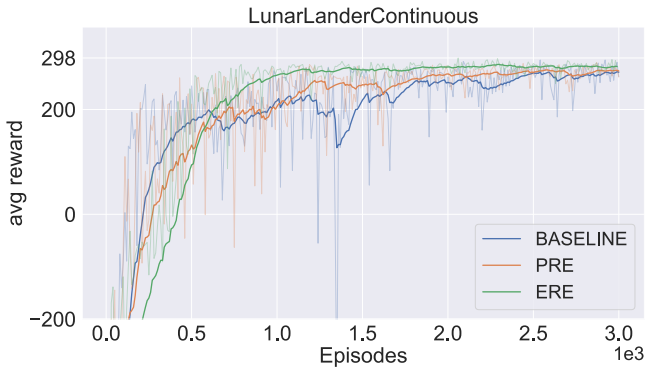


Figure 6: SAC BASELINE vs. PRE vs. PRE+ERE

Parameter	Value
learning rate	$3 \cdot 10^{-4}$
optimizer	Adam
discount	0.99
number of hidden layers	2
number of hidden units	256
nonlinearity	ReLU
target smoothing coefficient	0.005
target update interval	1
number of gradient steps	1
replay buffer size	10^6
automatic entropy tuning	True
samples per minibatch	8

Table 1: SAC Hyperparameters

The standard SAC, with the standard hyperparameters, solves the landing task well. The SAC with PRE learns the tasks marginally faster but in the end the performance is very similar. The SAC implementation with ERE and PRE reaches it's peek performance much faster but after longer training of the standard SAC the performances are similar again.
(Although PRE with ERE yields a small performance boost, only the PRE is used for further training as there seems to be a bug in my ERE implementation which makes training take longer per sample.)

3.2.2 Building a strong agent step by step

Training the agent only with self-play might result in a strong agent but this would take very long and hyperparameter tuning would become very tedious as the agent has to learn complex continuous behavior. Therefore instead of training the agent only with self-play, we propose the following training regime:

1. Train shooting

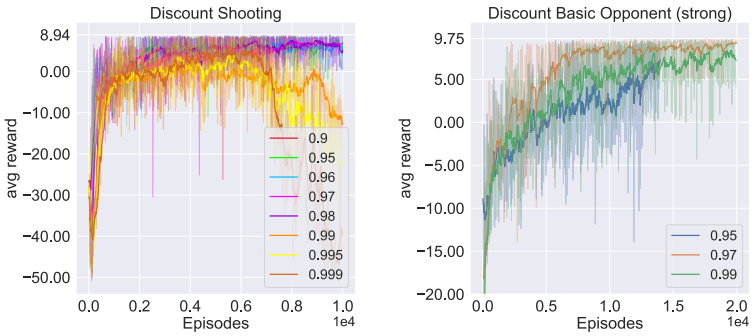
2. Train defending
3. Train against the basic strong opponent
4. Self-play

No real opponent was used for the shooting and defending training, as the agent will be trained against the basic opponent anyways in a later step. Instead the opponent for these simple tasks was an agent which only moved forward all the time until it reached the middle line. This brings two advantages:

1. Whenever the puck hits the opponent, the opponent will shoot it directly back into the direction of the agent's goal, which results in additional defence training
2. By being as close to the agent as possible direct shots in the direction of the goal become more difficult, this incentives the agent to use the banks to score a goal

3.2.3 Hyperparameter Search

Although SAC is suppose to be more stable regarding hyperparameters compared to other off-policy algorithms, a hyperparameter search is still needed to maximize the performance.
As the time and compute for the tasks was limited, only a subset of hyperparameters with a limited range was tuned. Also instead of trying all combination of hyperparameters we instead assumed the hyperparameters were more or less independent of each other (this is of course not correct but doing a full grid search would be too time consuming).
The tuning was done while training against the basic opponent. This gives the most representative results, as the performance and convergence speed can be clearly measured and the task is very similar to self-play which will ultimatively used to create the strongest agent.
A very essential hyperparameter is the discount factor. This became apparent when we did training for the shooting task. The discount factor from the paper of 0.99 was not stable and did not converge. We tried different discount factors from 0.9 to 0.999, a discount factor between 0.95-0.98 seems to give stable results.

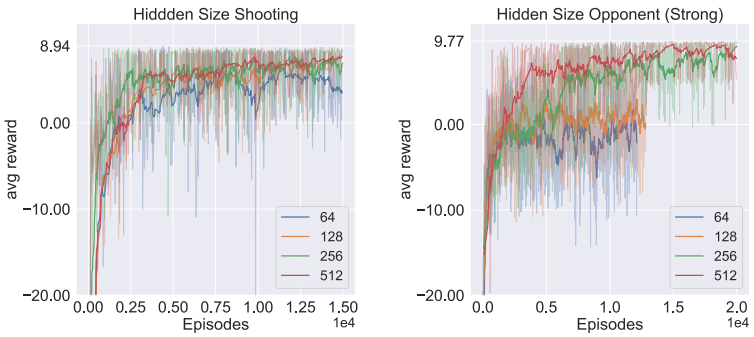


(a) (b)

Figure 7: Effect of the discount while training shooting (a) and against the basic strong opponent (b)

For the full play against the basic opponent we again tried a wide range of discount factors and although 0.99 was not stable for training shooting, for training against the basic opponent 0.99 was stable. The best discount discount reward for this environment was 0.97 though.
The hidden size of the actor, critic and target critic networks is also an important hyperparameter. If the

capacity of the network is to small the agent can't learn a complex policy. During training shooting it did not become apparent that one hidden size is better than the other, but when training against the basic opponent it became clear that a larger hidden size is preferred. We choose a hidden size of 512 to make sure we would not unterfit the optimal policy.



(a) (b)

Figure 8: Effect of the hidden size while training shooting (a), defending (b) and against the basic strong opponent (c)

We also did other experiments to tune the rest of the hyperparameters of SAC. But these experiments basically only confirmed the hyperparameters from the SAC paper, therefore we won't discuss them here any further. The rest of the experiments can be found in appendix D.

3.2.4 Self-play

Now that good hyperparameters have been found for the hockey environment and the agent can win against the basic strong opponent, the agent can start training self-play. To prevent the agent only learning to exploit one of it's weakness the following training regime will be proposed:

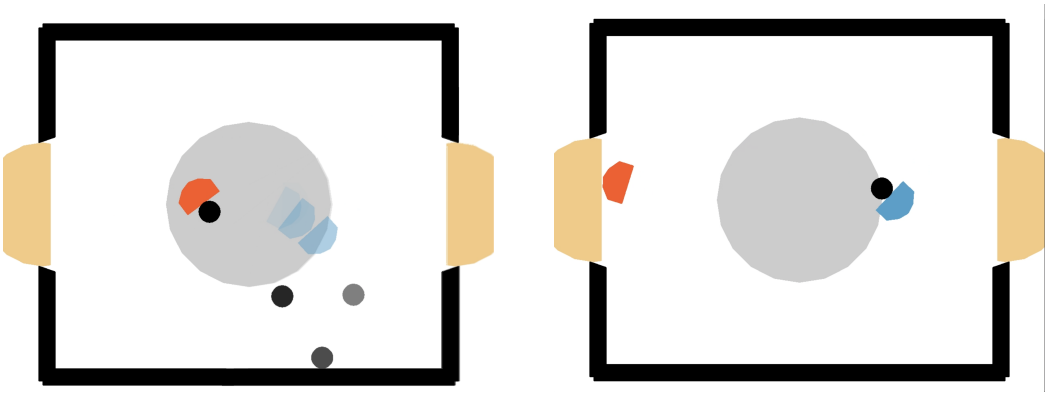
1. Start with the agent that can beat the basic strong opponent
2. Make one agent trainable, the other exploits it's policy
3. Train for at least x epochs
4. Update the opponent if the agent reaches an average reward of y or above over the last x episodes
5. Repeat until the agent cannot improve anymore
6. With a low probability switch the opponent with an old version of the agent or the basic opponent

The self-play training was repeated multiple times with different minimum epochs x and different thresholds y. In the end the agent which was trained with x=500 minimum epochs and a reward threshold y=4 was the best. Doing self-play among the self-play agents did not result in a stronger agent.

4 Discussion

4.1 SAC behavior

After training SAC with self-play the agent shows reasonable behavior. The agents prefers to attack over the right bank. This maximizes the chance of scoring as a direct shot can be easily saved by the opponent. Although the agent shows a clear strategy of how to score goals, defending is where the agent is really performing excellently. When the opponent has the puck the SAC agents retreats to its own goal-line and only leaves it when it is confident to catch the puck. This is an excellent defence strategy as the agent is maximizing its possible response time for direct shots and avoids indirect goal attempts over the banks.



(a) Attacking (b) Defending

Figure 9: The agent used in the tournament prefers to attack over the right bank (a), when not having the ball it retreats to its own goal-line to have as much time as possible to reacts to incoming shots

4.2 TD3 Behavior

The agent learns to win consistently against the weak basic opponent. After a longer amount of training time against the strong basic opponent it learns to play approximately at the level of the strong opponent. However, if the training lasts for a longer time, the performance will drop again. No extensive hyperparameter search was conducted, for lack of computing resources (no GPU) and a general concern about our environment. The training against both the weak and strong opponents is highly dependent on the initial random seed.

4.3 Mini Tournament, first to reach 100 goals

To evaluate which of our algorithms is the better one, we chose to conduct a mini tournament between the best of our agents. To get a true strength rating we decided to play until one agent reaches 100 goals: The SAC agent managed to beat the TD3 agent comfortably with a final score of 100:9. Unfortunately as the skill levels of the agents were so different, it made no sense for either agent to use the other agent for training.

References

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

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[3] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[4] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[5] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[6] 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, 2019.

[7] G. Martius. Reinforcement learning lecture notes, February 2021.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[9] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2016.

[10] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[11] C. Tessler, N. Merlis, and S. Mannor. Stabilizing off-policy reinforcement learning with conservative policy gradients, 2020.

[12] C. Wang and K. Ross. Boosting soft actor-critic: Emphasizing recent experience without forgetting the past, 2019.

A PPO Shooting test run

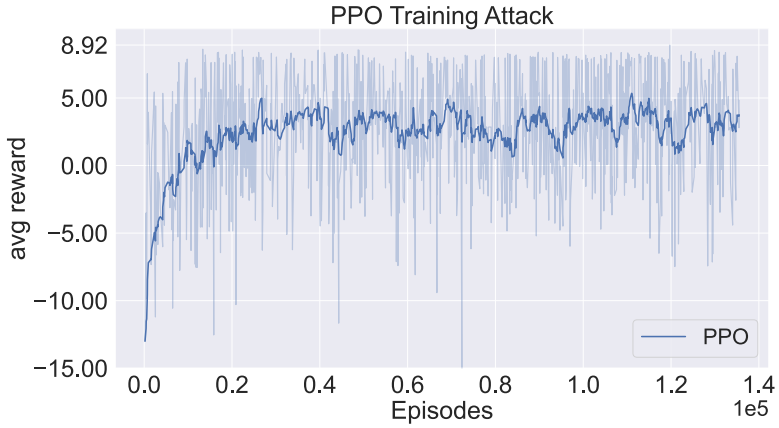
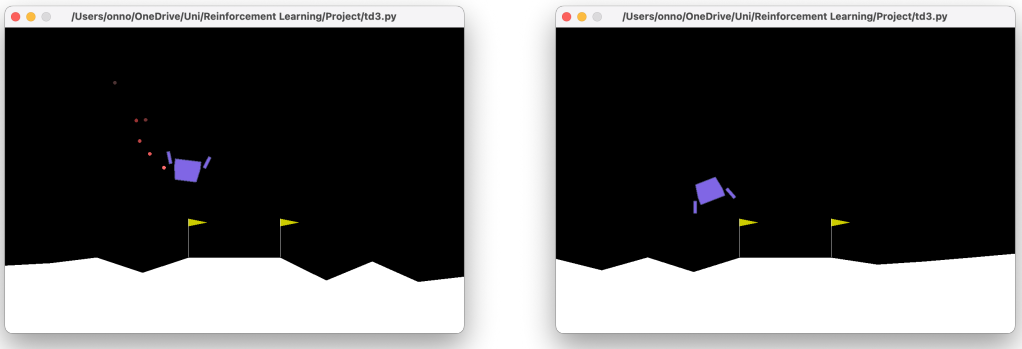


Figure 10

B Weird behaviour of Lunar Lander



(a) (b)

C Unreasonable Policy Loss

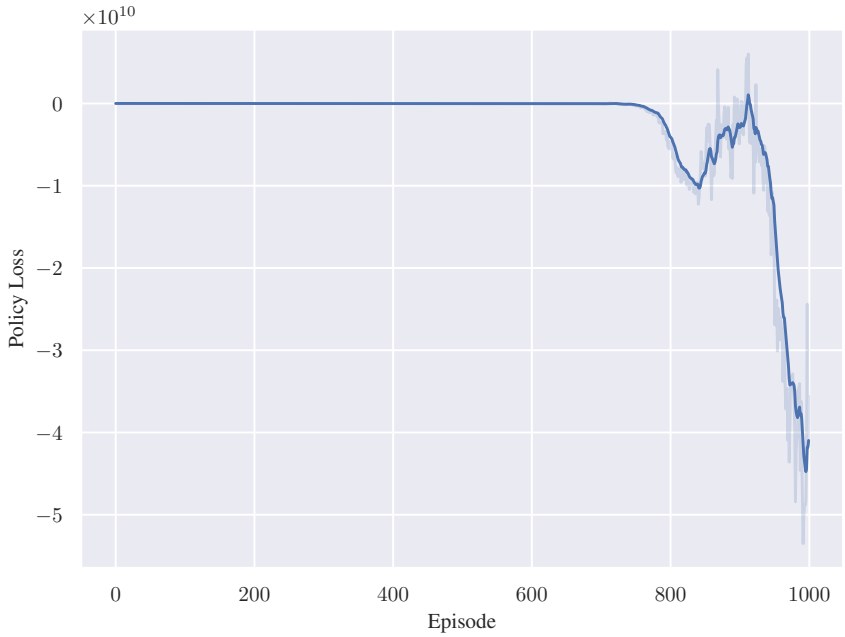


Figure 12

D SAC other hyperparameters

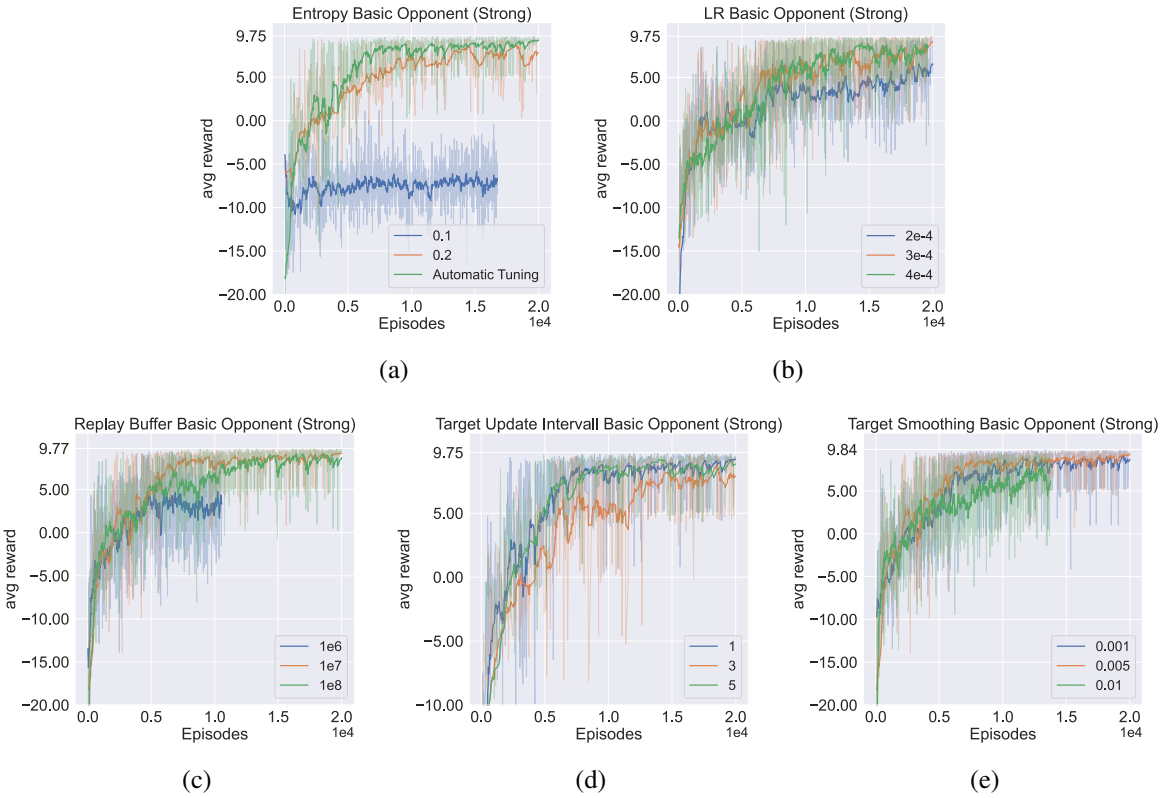


Figure 13: Automatic Entropy tuning (a), learning rate of size 3e-4 (b), replay buffer of size 1e7 (c), target update interval of 1 (d) and target smoothing of size 0.005 (e) seem to be the best hyperparamters

E Prioritized Replay Buffer Hockey Shooting

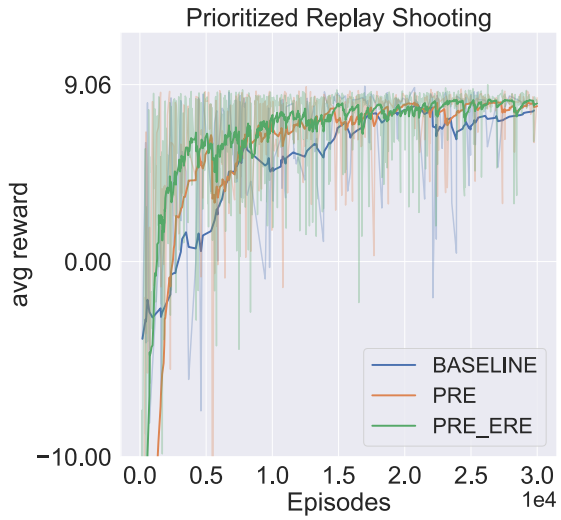


Figure 14: SAC vs. SAC with PRE vs. SAC with PRE and ERE for training shooting