RL-Course: Final Project Report

Sebastian Breit and Katharina Anderer

March 16, 2021

1 Introduction

Why should we be interested in studying games such that an algorithm is learning how to play it very well? David Silver [13] argues in his lecture on Classic Games that games kind of give an IQ test for RL algorithms. They contain quite simple, limited rules, but in order to play well, an agent has to learn an often complex relationship between a large variety of experiences and a given reward.

For this project the goal was to train an algorithm for playing hockey against a single opponent.

The problem is quite complex, with a state space $S \in [-1, 1]^{18}$ and a continous action space $A \in [-1, 1]^8$.

One challenge of this problem lies in the fact that rewards are quite sparse. The agent, the instance of our algorithm, is only receiving a large reward if it scores a goal. By default, the sparseness of the reward was somewhat reduced as the reward for being close was active as well. It returned small positive and negative rewards for being close to the puck while it moves to the enemy goal or to the own goal respectively.

As we are not dealing with discrete actions but continuous ones, we cannot just pick the action that is maximizing the expectation of future reward, but we can use policy gradient methods to set the direction of the loss gradient in a higher dimensional space such that reward expectation is highest. The legitimization that we can use a policy gradient method for such a problem, was given by the policy gradient theorem [15] that shows how the policy gradient can be simplified to an expectation formula, which allows to form a sample-based estimate for this expectation [14].

$$\frac{\delta p}{\delta \theta} = \sum_s d^\pi(s) \sum_a \frac{\delta \pi(s, a)}{\delta \theta} Q^\pi(s, a)$$

, where π, the policy, is assumed to be differentiable, Q is the value of a state-action pair and d^π is defined as discounted weighting of states [15].

Subsequently, we will introduce the two algorithms that we were testing - DDPG, which is short for Deep Deterministic Policy Gradient and PPO, the Proximal Policy Optimization.

Whereas PPO is an on-policy algorithm that collects experience using the latest policy, DDPG is an off-policy algorithm where all new experiences collected from each policy are stored in a so called replay buffer.

2 Methods

2.1 PPO - Proximal Policy Optimization (Anderer)

PPO was proposed by [12] as a robust, data efficient algorithm, that works similar compared to trust region policy optimization (TRPO), but is much easier to implement and better compatible with different

architectures. PPO estimates the performance of a policy with a clipped probability ratio. The ratio r_t should tell if a new computed policy is expected to perform better or worse compared to the old one:

$$r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

A clipping of the ratio with a clipping parameter ϵ restricts the gradient change to lie within a certain range. Another important part is the advantage function that, if positive, expresses that an action is better than expected and, if negative, tells that it is worse than expected. The advantage function is multiplied with the probability ratio and in order to avoid a huge update step, one takes the minimum of the clipped and unclipped product of probability ratio and advantage.

The objective for the policy is as follows [12]:

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_{\pi_{\theta_{old}}}(s_t, a_t), clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_{\pi_{\theta_{old}}}(s_t, a_t)]$$

Instead of evaluating the advantage of a whole trajectory, as it was proposed by [12], we calculate the advantage by taking the difference between the actual rewards and the expected rewards which are estimated by the critic of our network:

$$\hat{A} = rewards - \hat{rewards}$$

The total loss is decomposed into the policy loss $L^{CLIP}(\theta)$ and the critic loss for which we take the mean squared error of the expected state values and the rewards.

$$L^{total} = -L^{CLIP}(\theta) + MSE(rewards, \hat{rewards})$$

Some drawbacks of PPO are that there is no convergence guarantee, even tough in practice this is usually no problem [5], and that it can have a high sample complexity because after each policy update, we need to collect new data in order to perform the next update.

Despite this, fine tuning of hyperparameters is not such a problem compared to other methods, which makes the method quite robust.

2.1.1 Multivariate Distribution for sampling actions

We adapted our implementation from [2] and inherited the idea of sampling the actions from a multivariate distribution from it. Therefore, the current state is given to a sequential network with Tanh activation functions in between the 3 linear layers. The sequential network gives an output of 8 values for the action layer and an output of 1 value for the critic layer. For sampling the actions, we only regard the action layer for the moment. Every value of the output is regarded as the mean of a normal distribution. So, we can sample from a multivariate distribution with a covariance matrix that assumes the same standard deviation for each variate. The covariance matrix has entries equal to zero, except at the diagonal where the entries are equal to the variance that is assumed for each action dimension.

2.1.2 LSTM component

We also tried a variant of PPO with an LSTM architecture that has the capacity to process previous hidden states and therefore account for long-term dependencies [7]. The architecture has a linear layer with the state dimension as input (18) and an output of 64 that is then fed into an LSTM with an output of 32. The LSTM not only takes the current input of the state, but takes also the previous hidden states as input. [6] mention that LSTMs might be used in contexts were global motion and local action dynamics

are important to predict. We did not find any reference tough where LSTMs were used in single agent games.

[10] is describing a combination of PPO and LSTM and is discussing that the refreshing of hidden states can be important in sparse environments.

Intuitively, we would also argue that in every dynamical game, a LSTM component in the architecture could help as it should be an advantage to know from which direction the puck or the opponent is coming from.

2.2 TD3 - Twin Delayed Deep Deterministic Policy Gradient (Breit)

2.2.1 Basic Deep Deterministic Policy Gradient (DDPG)

The second implemented algorithm is based on the DDPG algorithm introduced by [8]. It combines the actor critic approach of Deterministic Policy Gradient algorithms for model-free control on continuous action spaces [14] with the advantages of Deep Q-Networks (DQN), that lead to stable and robust function approximations for value functions. [9]

It is based on the Bellman equation for a stochastic policy π [9]

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1}, a_{t+1})]]$$

With a deterministic policy μ, the inner expectation can be omitted. The new expectation then depends only on the environment *E*, which means that the new value function Q^{μ} can now be learned offline from transitions generated by a different stochastic policy β:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))]$$

The Q-function is then approximated with parameters θ^Q and optimized by minimizing the loss

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^{\beta}, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t|\theta^Q) - y_t)^2]$$

with

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \, .$$

With the actor-critic approach from the DPG algorithm [14], the actor function $\mu(s|\theta^{\mu})$ is optimized with the following gradient

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

To enable offline learning, we used a replay buffer as proposed in the paper [9], which stores transitions as a tuple of (s_t, a_t, r_t, s_t, d) with the additional parameter d being a boolean that stores whether this transitions finishes an episode by scoring, or not. We also added an alternative implementation for experience replay, which we describe in subsection 2.2.3.

2.2.2 TD3

Additionally, we then augmented this basic implementation with the enhancements introduced in the Twin Delayed Deep Deterministic policy gradient algorithm (TD3). [3] The authors propose three modifications, which all lead to a decrease of overestimation bias, which in turn leads to a preference for states with lower variance in their value-estimates, leading to more stable targets for policy updates. [3]

We implemented all three of their improvements. The Clipped Double Q-learning, based on Double Q-learning, makes use of two critics, each producing an estimate of the value function, over which the minimum is taken for the target update:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi_1}(s'))$$

Further, we also make use of delayed policy update, updating only every 2 value updates, using the parameter value suggested by the authors. [3] Lastly, we implemented their target policy smoothing, which adds a normally distributed clipped noise ϵ to the target policy:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon)$$

Finally, we also exchanged the exploration policy of the DDPG algorithm, which was based on an Ornstein-Uhlenbeck-Process [9] and used a Gaussian Noise that is added to each transition before it is stored instead. Here we also started with the suggested parameters from the paper $\mathcal{N}(0, 0.1)$. [3]

2.2.3 Prioritized Experience Replay

As a further improvement, we implemented Prioritized Experience Replay. It improves normal experience replay by prioritizing transitions with a higher TD-error compared to the previous uniform sampling. This reflects the intuition that there is more to learn from experiences that resulted in the biggest surprises. [11]

The new sampling probability *P(i)* of a specific transition *i* is generated by weighting its priority p_i^a by the sum of all priorities in the memory, with the exponent α determining how much prioritization is used, $\alpha = 0$ being the uniform case:

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

The priorities are generated using the proportional prioritization proposed in the original paper [11], $p_i = |\delta_i| + \epsilon$, taking the absolute value of the TD-error. New transitions, that have not yet been assigned a TD-error always get the current maximal priority within the buffer, to ensure that new transitions always have a very high probability of getting sampled at least once. Further, we tested the authors proposal of annealing the prioritization exponent α from an initial value of 0.5 to 0, multiplying α each episode with 0.99 and compared it to a flat α of 0.5. Due to a time constraint, however we did not use the proposed importance-sampling weights[11], to reduce the sampling bias introduced by sampling some transitions more frequently.

3 Evaluation

3.1 Evaluation of PPO

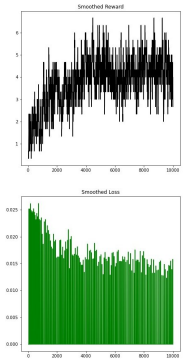
In the following, we will describe our training process for the PPO algorithm and how we did some of hyperparameter tuning. We also tested some variations of network architectures as it was mentioned in [4] that the number of hidden neurons or the choice of activation function can make a large difference.

After some hyperparamter tuning were we tested epsilon clipping parameters between 0.1 and 0.3, different learning rates between 0.003 and 0.0001, and discount factors between 0.9 and 0.99, we set these hyperparameters to the configuration that can be found in Table 1.

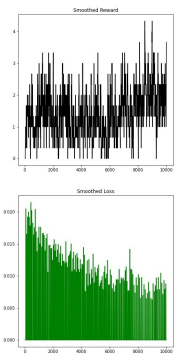
We were comparing network architectures with a set of (64, 64), (64, 32) and (200, 200) hidden neurons. The (64, 64) -architecture seemed to work best as we first evaluated from eye-balling the training curves (Figure 1) and then the average winning rate where we let the trained agent play 500

Param	Value
Clipping ϵ	0.2-0.3
learning rate	1e-4
discount factor γ	0.99

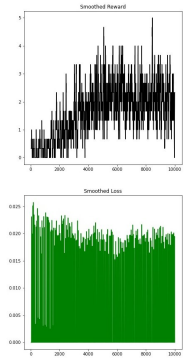
Table 1: Hyperparameters PPO



(a) Training curve for (64,64) architecture



(b) Training curve for (200, 200)-architecture



(c) Training curve for (64, 32)-architecture

Figure 1: Different network architectures

games and evaluated the winning percentage as the fraction of won games and the sum of won and lost games: $win\% = \frac{won}{won+lost}$. The (64, 64)-architecture had a $win\% = 0.57$ when averaging over 500 games, the (64, 32) a $win\% = 0.47$, and the (200, 200)-architecture a $win\% = 0.22$. It surprises that the performance differs that much between the different architectures. We suppose that overfitting can occur faster in deeper architectures, which might be reflected by the poor performance of the (200, 200)-architecture. It can further be the case that the (200, 200)-architecture would have simply needed more epochs. Either way, the (64, 64) seems to learn fastest and was therefore most appealing for further evaluation and training.

3.1.1 LSTM architecture

As the update computation got much more slower and learning was less stable, we did not pursue this further. Training around 3000 epochs took us around 8 hours on a local machine and the training curve was increased only a bit with still a high variance (see Figure 2). We provide the code in our submission.

3.1.2 Training modes and self-play

The training was quite sensitive to whether we included the reward information of closeness to the puck, puck direction or even the information of the opponent scoring. The latter was an idea that was mentioned by another student in the recitation class for making the agent more aggressive and have courage to play in a way that is not only avoiding opponent goals. We also made the experience that this is working quite well for our player such that, for training our tournament player, we used only the reward of a goal and excluded closeness of puck and the negative reward of an opponent goal. Further, we used a combination between training against the opponent and self-play. For self-play, we loaded the weights of a previous version of the agent and selected the actions for the opponents as if the agent was playing against its old self. We trained first against only the basic, weak opponent and achieved a winning rate

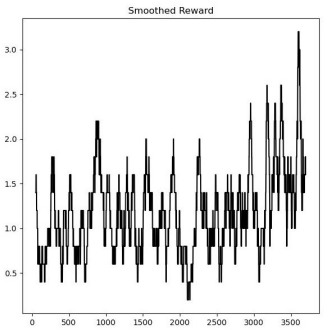
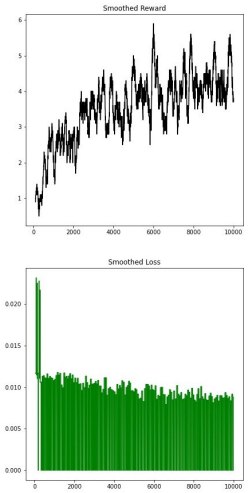
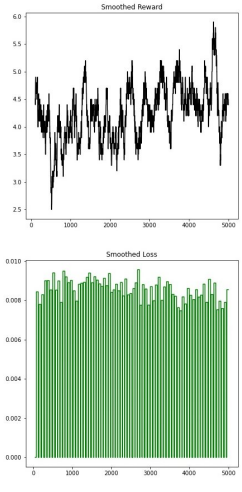


Figure 2: Training curve for the LSTM architecture

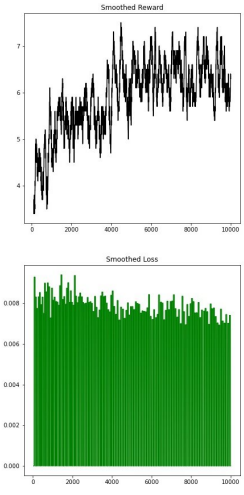
of 68.5%, with regard to this fraction again: $win\% = \frac{won}{won+lost}$. After having achieved a 68.5% winning rate against the basic opponent, we trained on self-play and sampled the opponent's actions from the now fixed network that had achieved these 68.5% and trained our player against it. After 5000 epochs the winning rate against the basic opponent got lower (around 58%). After 15000 epochs, it had gotten even lower (48%). We suppose that it failed to learn a good new policy, having already adapted a low learning rate and lower explorative behavior. So another idea was to train a 'new agent' without weights and optimizer parameters already adapted, but now randomized the opponents such that the agent was playing either against the weak basic opponent, the strong basic opponent or the agent that had received a winning rate of 68.5% earlier on. The agent was trained 'from scratch' again, but now had to learn how to play well against three different playing styles. This should prevent overfitting to some degree and give an incentive for the agent to generalize its behavior. With this training, we achieved a winning rate around 47.7% for the weak opponent and around 47.6% for the strong opponent.



(a) First 10000 epochs against basic opponent, about 52% winning prob.



(b) Additional 5000 epochs, about 56% winning prob.



(c) Additional 10000 epochs, about 65% winning prob.

Figure 3: Training against basic opponent (weak = True)

3.2 Evaluation of TD3

3.2.1 Different training configurations

For evaluating and optimizing the performance of the agent trained with the TD3 algorithm, we tried optimizing hyperparameters, usage of prioritized experience replay and modification of the rewards given by the environment. For most hyperparameters of TD3, we used the preconfigured ones recommended in the original papers. We modified the discount, replay memory size, priority weighting and introduced a parameter for updating the sampling probabilities of transitions, the last being simply due to performance reasons for running on a slow local laptop.

Param	Value
Lr Critic	1e-3 [8]
Lr Actor	1e-4 [8]
Optimizer	Adam [8]
Target update rate τ	5e-3 [3]
Policy update rate	every 2nd Q-update [3]
Batch Size	100 [3]
Exploration Policy	$\mathcal{N}(0, 0.1)$ [3]
Neurons layer 1	400 [3]
Neurons layer 2	300 [3]
Discount	0.95
Buffer size	1e5 / 5e5
Priority weight α	0.5
Decay α	0.99
Probability update every n eps	100

Table 2: Hyperparameters TD3 and Prioritized Experience Replay

With these hyperparameters, we then evaluated multiple different scenarios, described in the following:

1. Prio replay annealing - size 1e5, strong opponent, normal reward, 2k eps
2. Prio replay annealing - size 1e5, strong opponent, reward without punishment if opponent scored, 2k eps
3. Prio replay annealing - size 1e5, weak opponent, normal reward, 5k eps
4. Normal replay - size 1e5, strong opponent, normal reward, 5k eps
5. Prio replay annealing - size 5e5, uniformly random opponent from {weak, strong, self}, normal reward, 4k eps
6. Prio replay flat α - size 5e5, uniformly random opponent from {weak, strong, self}, normal reward, 2k eps
7. Prio replay flat α - size 5e5, random opponent from {weak=0.4, strong=0.4, self=0.2}, normal reward, 2k eps
8. Normal replay, random opponent from {weak=0.4, strong=0.4, self=0.2}, normal reward, 5k eps

3.2.2 Results

Our baseline for comparison will be the configuration used in the tournament, configuration 1. Here, we trained the agent using TD3 with prioritized replay and annealing of the priority weight α against the

strong basic opponent with the normal reward for 2000 episodes. Training the opponent in the defensive mode first, as suggested in the project description, did not provide any improvements and in fact even led to a worse overall performance compared to training directly against an opponent. The training curves can corresponding to the tournament configuration be seen in figure 4. As all algorithms seemed to

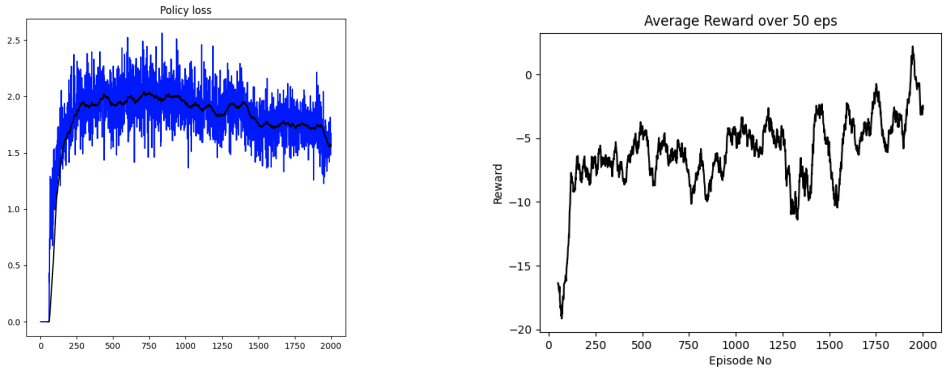


Figure 4: TD3 training curves from the tournament configuration

have problems with overfitting and had quite high variances between hundreds of episodes, we not only stored their final weights, but also at the point where they achieved the highest running reward across 50 episodes. For the final evaluation we used the latter weight configurations, as they tended to perform better across the board for all tested training configurations.
All configurations were evaluated over 1000 runs against both the weak and the strong basic opponent. The tournament configuration achieved 388 wins, 310 losses and 302 draws against the strong opponent, resulting in a win rate $\frac{\text{Wins}}{\text{Wins + Losses}}$ of 55.6%. Against the weak opponent, it achieved 249 wins, 488 losses and 263 draws, resulting in a much lower win rate of 33.8%. This indicates that our tournament configuration, playing against only the strong opponent, probably led to overfitting to its specific behaviour. Therefore, we decided to train on further configurations where the algorithm played against randomly sampled opponents, to reduce possible overfitting.
The best performance against the weak opponent was achieved by configuration 3 with a win rate of 70.0%, however its performance against the strong opponent was quite poor with a win rate of only 30.2%, again suggesting overfitting. Overall, configuration 6 performed best on both the weak opponent and the strong opponent, with win rates of 52.6% and 52.0% respectively. It consisted of prioritized experience replay without decaying α, uniformly choosing an opponent at random out of the weak opponent, the strong opponent and playing against itself, a memory of size 5e5 and training for 2000 episodes. Its training curves can be found in figure 5.

4 Discussion

With both algorithms, we could reach performances that were able to beat the weak and strong opponents with winning rates close to 50% or higher, dependent on whether we trained against only one opponent or both basic opponents. The winning rates were higher for a specific opponent when we also trained only against this specific opponent. In order to let our algorithms perform well in general, also with respect to the tournament, where we had to compete against a lot of different algorithms, we had to trade-off a high winning rate against a lower one that was more stable against different opponents.

In the end, both algorithms worked similarly well and did achieve performance rates that were alike.

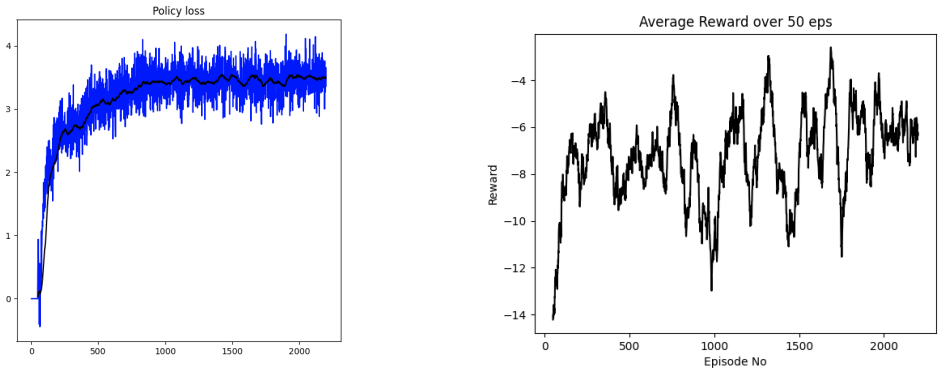


Figure 5: TD3 training curves from configuration 6 - best overall result

While TD3 needed usually less epochs, PPO had to train for more epochs, but could achieve this with a rather high velocity.

An interesting discovery was how we can influence the behavior of the agent by changing the reward parameters. This is something that seems crucial for the learning behavior of the agent. Interestingly tough, while the PPO agent seemed to profit best when only the positive reward was given as signal, the TD3 algorithms performed better with also the negative reward and the closeness to the puck signal.

In order to give an incentive for explorative behavior, we used two different strategies for the two algorithms. While for TD3s we simply added a Gaussian Noise to the actions before they are stored in the replay memory, we were using a multivariate sampling with a covariance matrix for the PPO algorithm. For both algorithms, the noise/variance stayed constant for the whole training process. Another idea to boost performance would be to lower the variance as training proceeds in order to find a more stable policy. This could maybe be done in a smooth way, similar like Adam is using the first and second moment of the gradient to adapt its learning rate.

Including a LSTM component in the architecture did not work well with our limited resources on a local machine, but still, the idea of using a LSTM for a dynamic action game seems appealing to us and we think that especially if the game has more players that need to interact with each other, this might be a promising approach.

Prioritized replay, as used with the TD3 algorithm, did improve the amount of episodes needed to reach good performance levels, however the actual training time for the episodes increased significantly. With running on a local machine, that meant training the algorithms over less episodes, as otherwise the training time became too excessive. It is possible that it is due to this fact, that the performance with prioritized replay of otherwise similar training configurations did not provide an improvement in the evaluation. Another possibility is that the sampling bias, which is corrected in the original paper using importance sampling weights [11], was responsible for the lack of performance difference.

Apart from prioritized experience replay, another possible improvement that similarly increases the efficiency of off-policy learning is hindsight experience replay (HER).[1] In their paper, the authors demonstrate that using hindsight experience replay, it is possible to significantly improve the learning efficiency, especially in environments with sparse and binary rewards. As this description fits very neatly

to our problem of an agent trying to learn how to play hockey, it might have provided another performance increase or at least an efficiency increase. In their paper, the authors even explicitly mention the situation of learning a hockey, providing an example where a player shoots slightly next to the goal. If the reward was sparse and only goals resulted in any returns, traditional reinforcement learning algorithms might learn very little from such a situation, whereas HER specifically addresses such situations by remodeling its goals. [1] Since our environment by default does not only return such sparse rewards however, returning many small rewards for being close to the puck, such a modification might not lead to the significant improvements suggested by the paper.

We would have expected that self-play leads to a large improvement, which we could not observe tough. If this is due to the fact that the agent was already overfitted to a certain opponent and failed to adapt to some situational changes or if it was a matter of the implementation of self-play, remains open.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

[2] Nikhil Barhate. Ppo-pytorch. `https://github.com/nikhilbarhate99/PPO-PyTorch/blob/master/PPO_continuous.py/`. [Online; accessed on 11 March, 2021].

[3] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[4] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[5] J. Janisch. Let's make a qqn-double learning and prioritized experience replay. `https://jaromiru.com/2016/11/07/lets-make-a-dqn-doublelearning-and-prioritized-experience-replay/`. [Online; published on 2016, November 7].

[6] A.J. Kulkarni and S.C. Satapathy. *Optimization in Machine Learning and Applications*. Algorithms for Intelligent Systems. Springer Singapore, 2019.

[7] Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent reinforcement learning: a hybrid approach. *arXiv preprint arXiv:1509.03044*, 2015.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[10] Kamal Ndousse. Stale hidden states in ppo-lstm. `https://kam.al/blog/ppo_stale_states/`. [Online; published on 2020, June 25].

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

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[13] Dabid Silver. David silver lecture on games. `https://www.youtube.com/watch?v=kZ_AUmFcZtk&list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB&index=11`. [Youtube, published 2015].

[14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[15] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999.