RL-Course 2024/25: Final Project Report

Jungi Hong, Till Köpff

February 26, 2025

1 Introduction

In this report, we outline our approach for solving the Laser Hockey challenge, which is part of the final project for the Reinforcement Learning course at the University of Tuebingen.

The *Laser Hockey* OpenAI Gym environment simulates the popular air hockey game as computer game. The objective is similar too many two player fields game: Score goals but defend your own goal. The underlying mechanics and the general big state space makes the environment a non-trival environment to learn and master. Rewards are sparse in general e.g. score goals. In the base environment an additional reward for 'puck closeness' is already added. Using gym wrappers the state space, action space and the rewards can further be modified.

By leveraging reinforcement learning, we develop agents which successfully learn the underlying dynamics and consistently defeat built-in bot opponents.

The algorithms developed by the authors are:

- Deep Deterministic Policy Gradient (Jungi Hong)
- Soft Actor-Critic (Till Koepff)

In the following sections, we analyze the results of each approach in the order listed above. The code for our implementation is attached as zip-directory to the hand-in assignment.

2 Deep Deterministic Policy Gradient (DDPG)

Deep reinforcement learning has achieved remarkable performance by combining deep learning with reinforcement learning. In continuous control domains, Deep Deterministic Policy Gradient (DDPG) is widely used. However, it typically relies on uniform experience replay, treating all transitions equally. In practice, not all experiences are equally informative. Some transitions provide stronger learning signals than others, requiring a better sampling strategy. To address this, we integrate Prioritized Experience Replay (PER) into DDPG. With PER, each transition is assigned a priority based on its temporal difference (TD) error—higher errors indicate greater potential for learning and result in more frequent sampling.

2.1 Prioritized Experience Replay (PER)

PER improves learning efficiency by sampling transitions according to their TD error. This ensures that significant transitions are replayed more often.

2.1.1 Implementation and Integration with DDPG

Our approach integrates Prioritized Experience Replay (PER) into DDPG by modifying the replay buffer mechanism as follows:

1. **Storing and Updating Priorities:** Each transition is initially assigned a maximum priority (e.g., 1.0). After each training step, the priority is updated based on the absolute TD error:

$$priority_i = \left(|TD\_error_i| + \epsilon\right)^{\alpha},$$

   where ϵ is a small constant to prevent zero priorities and α controls the degree of prioritization.

2. **Sampling Transitions:** Transitions are sampled with probability:

$$P(i) = \frac{priority_i^{\alpha}}{\sum_j priority_j^{\alpha}}.$$

   To counteract the bias from non-uniform sampling, importance sampling weights are computed as:

$$w_i = \left(NP(i)\right)^{-\beta},$$

   where N is the total number of transitions and β is gradually annealed to 1.

To integrate PER with DDPG, we replace the standard ReplayBuffer with a PrioritizedReplayBuffer during agent initialization and modify the sampling process to use the above probabilities. After each learning step, the buffer updates the priorities using the new TD errors, ensuring that more informative transitions are replayed more frequently.

2.2 Experiments

The DDPG agent was trained by relying on the the DDPG PyTorch implementation from stable-baselines-3. Thus for this section we explored the effects that certain hyperparameters choices such as the learning rate have on performance. The metric of interest is the percentage of games won across 50 evaluation episodes. Each evaluation was against the weak opponent, and we report the average win percentage. We perform an evaluation rollout after every 100,000 training timesteps to track the progress of each algorithm.

For this task, we train a feed-forward architecture using experience replay with the Huber loss function and the Adam optimizer, initialized with a learning rate of 0.0001. Additionally, we use a network architecture with two layers of 256 units each. The agent is trained with a batch size of 32 experiences from the replay buffer, which has a capacity of 10^5 transitions.

Firstly we investigated the effect of varying the learning rate. We tested learning rates of 0.0001, 0.0003, and 0.0005 to determine which one allowed for faster convergence. From our results in Figure1 we found that both 0.0001 and 0.0003 resulted in smoother convergence than 0.0005 at almost every time interval. The figure also showed at the agent trained with a learning rate of 0.0001 had a higher win rate than the others at almost every time steps, so we decided to stick with 0.0001 for the rest of our experiments. There are many other hyperparameters that could have been tuned, but due to time constraints, we were only able to look at the learning rate.

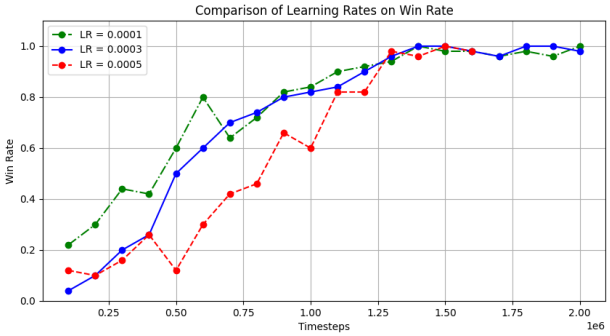


Figure 1: Performance of the DDPG with different learning rates during training.

2.3 Experimental Results with PER in DDPG

To assess the impact of Prioritized Experience Replay (PER), we compare DDPG with and without PER. The evaluation is performed over 50 episodes, measuring the win rate. The results, shown in Figure 2, indicate that PER slightly improves sample efficiency by prioritizing informative transitions, leading to a faster learning process.

As depicted in the win rate curves, the agent trained with PER initially learns at a similar pace as the one without PER but surpasses the baseline (DDPG without PER) as training progresses. This suggests that the prioritization mechanism helps stabilize training by focusing on transitions that contribute more to the learning process. By contrast, the standard experience replay samples transitions uniformly, leading to slower improvements in policy performance.

Furthermore, we observe that the PER-enabled agent achieves a win rate of 1.0 at approximately 1.2 million timesteps, whereas the baseline takes closer to 1.4 million timesteps to reach similar performance. While this demonstrates some of the benefit of PER in accelerating convergence, the improvement is relatively modest. Fine-tuning certain parameters could potentially further emphasize the advantages of PER.

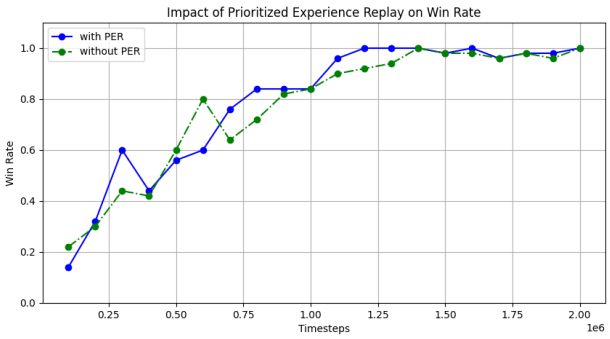


Figure 2: Comparison of DDPG with and without Prioritized Experience Replay (PER) during training.

In summary, these results demonstrate that PER enhances training efficiency, allowing the agent to learn more effectively from limited experience. Future work could explore tuning PER-specific parameters to further optimize performance.

3 SAC

The **Soft-Actor-Critic** (SAC) algorithm consists of two key neural networks: one for estimating the Q-value function and another for learning the optimal policy. Unlike the DDPG algorithm, which selects the action that maximizes the Q-value, SAC employs a stochastic policy that enhances exploration by introducing entropy regularization. This encourages the agent to explore diverse actions rather than greedily exploiting the current estimate of the best action. The policy π in SAC optimizes a trade-off between expected rewards and entropy, given by:

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{T} \mathbb{E}\left[r(s_t, a_t) + \alpha H(\pi(\cdot|s_t))\right],$$

where α is the temperature parameter controlling the balance between exploration (entropy) and exploitation (reward maximization).
To efficiently learn the state-action value function, SAC uses the soft Bellman equation, an entropy-regularized variant of the standard Bellman equation:

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

where the soft value function is defined as:

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi}\left[Q^{\pi}(s_t, a_t) - \alpha \log \pi(a_t|s_t)\right].$$

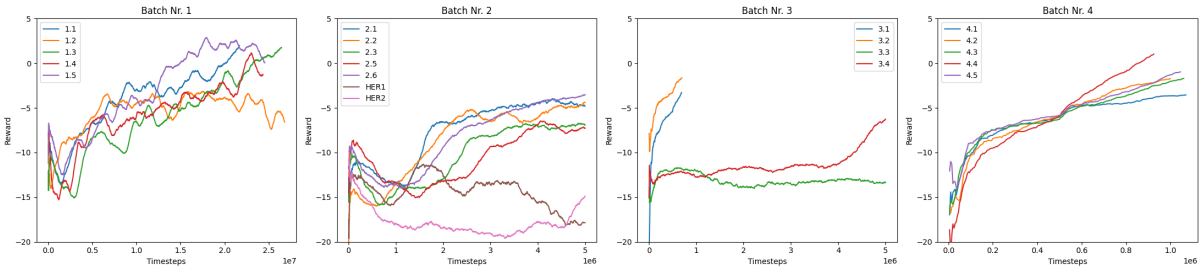
In addition to SAC, **Hindsight Experience Replay** (HER) is particularly useful in environments with sparse rewards. HER improves learning efficiency by relabeling failed trajectories with a new goal, often set as the final state of the trajectory. This allows the agent to extract meaningful training signals even from unsuccessful episodes. The idea behind HER is that, although the agent may not reach the intended goal, the trajectory still provides valuable insights about the underlying environment.
However, incorporating HER increases the dimensionality of the state space, as the reward function now depends on both the achieved goal and the desired goal. The advantage of HER lies in its ability to generalize across goals by enabling the state-action space to interpolate between different goal states.

3.1 Experiments

The SAC agent was trained by relying on the existing PyTorch implementation of the stable-baselines-3 library. Thus the main work of the project lied within finding good hyperparameters for fast convergence of the training algorithm. The most important hyperparameters of the SAC algorithm are amongst others: network architecture, learning rate, replay buffer size, discount factor and training frequency. In the experiments two versions of HER were implemented but did not improve convergence and were thus rejected. The first version had 8 additional discrete goals (puck behind agent, puck behind opponent, puck touching borders, ball possession), the second tried to map current puck position and velocity to the desired goal continously.

3.1.1 Training

In total I employed four batches of manually hyperparameter searches. Each batch consists of multiple training runs with different hyperparameters and a post-evaluation of those. Table 2 shows the hyperparameters of each run. The corresponding convergence of every run is plotted in figure 3 sorted by batch number. Note that the scale of the x-axis is differently within different batch plots. We can clearly see that the hyperparameters have a huge influence on the convergence rate and that we manage to constantly



Label names correspond to {batch_number}.{run_number}. Hyperparameters can be looked up in Table 2.

Figure 3: Reward Evolution over different batches and training runs (100 point moving average).

improve this rate in the training batches carried out. In the following I will explain considerations made before and after each training batch.

First Batch

In the first batch, hyperparameters are set to vanilla values and in doubt are chosen optimistically. Overall the convergence rate of the first batch is very low, thus hyperparameters are chosen very bad.
Figure 3 shows that an MLP of two inner layers of 64 neurons is not expressive enough and hems learning. Also, a low learning rate of 5e-4 clearly shows better convergence properties than a learning rate of 5e-3. A learning rate of 1e-3 performs even a bit better then the learning rate of 5e-4. Still I decided to continue with a learning rate of 5e-4 as a low learning rate can become beneficial around the optimum which is not yet reached in the given plot.
The performance of 3 inner layers of 64 neurons could not be clearly accessed and was carried out in a next batch with adapted learning rate and replay buffer.

Second Batch

Before trying to start training with a second batch of hyperparameters, I reassessed the hyperparameters chosen in the first batch altogether. This was necessary because of the very bad convergence in the first training batch: There has been no convergence of the reward nor winning rate within 25 million timesteps. In the following I will describe the considerations made when changing the replay buffer size and the discount factor.
The replay buffer of 1e4 is likely chosen too small. It only reflects about 50 episodes (200 timesteps per episode). A replay buffer of 1e5 can save around 500 episodes and can therefore reflect more data variance. However, the replay buffer must be chosen in accordance with the training frequency. In case that the training frequency is low, the buffer is changing fast an the buffer size will be second-tier as the overall variance of the data used for training will still be high.
The discount factor might also influence training convergence massively. The game is played at 50 fps, thus one second will correspond to 50 timesteps (states). The hockey game is very fast and it can be said that two states which are one second apart will almost be independent of each other, whereas states being half a second apart might indeed be dependent substantially. As a consequence we propose $\gamma = 0.95$ as lower bound for the next hyperparameter search ($0.95^{50} \approx 7.7\%$, $0.95^{25} \approx 27.7\%$). The results of batch 2 confirmed the considerations made. Overall better convergence properties can be found. A discount factor of 0.95 shows the best convergence within the different discount factor choices.
Also a network architecture of an MLP with two inner layers of 255 neurons outperforms the MLP networks with different sizes.

In the second batch I also implemented a training algorithm employing Hindsight Experience Replay. However, this implementation lacked good convergence properties. More details are found below.

Third Batch

The third batch confirmed the assumption that a larger MLP with two inner layers of 256 outperforms the MLP with two inner layers of 128.
Furthermore a learning rate of 1e-4 showed beneficial convergence properties especially when coming closer to the optimum. This confirms the considerations made after batch one.

Fourth Batch

In the final batch of experiments, I analyzed the impact of learning rate, replay buffer size, gradient step frequency and batch size on training performance. The results indicate that using 10 training steps per rollout episode and a replay buffer size of 1×10^5 is sufficiently large, while a batch size of 128 provides a good balance between variance and convergence stability.
This means that while at most 250 new entries ($\approx 0.25\%\cdot$Buffer Size) are added to the buffer per episode, training still utilizes 1,280 entries ($\approx 1.28\% \cdot$ Buffer Size), leading to significant variance and favorable convergence properties.
Assuming an episode consists of 100 steps on average, a sample remains in the buffer for approximately:

$$N = \frac{10^5}{100} = 1000 \text{ episodes.}$$

The probability of a specific sample being selected during training is:

$$p = \frac{1280}{10^5} = 1.28\%.$$

Given that a sample stays in the buffer for 1,000 episodes, it is expected to be used in training about 13 times on average. This level of data reuse appears reasonable. However, there might be potential to exploit buffer data even further.

Feature: Hindsight Experience Replay

In Batch 2, I attempted to accelerate convergence using a Hindsight Experience Replay (HER) Buffer. However, this approach failed and led to significantly worse training performance. Upon further reflection, HER does not seem well-suited to this particular scenario. Since we are dealing with a two-player game, the state variance is inherently high due to the opponent's influence. As a result, achieving specific states does not necessarily provide meaningful insights toward the ultimate objective—scoring a goal. In this context, HER behaves more like a poorly designed reward-shaping mechanism, making it difficult to implement effectively. Instead of aiding learning, it unnecessarily increases the state space without offering a well-defined structure for goal selection. The goals defined within this framework turn out to be ineffective and uninformative.

Feature: Online Learning during Tournament

An additional feature for the tournament was an agent which was trained online. For this a child class for the comparl agent was written (see source code *koepff/run_client_and_train.py*). In the client function *get_step* and *on_end_game* the observations where added to an online replay buffer and training was

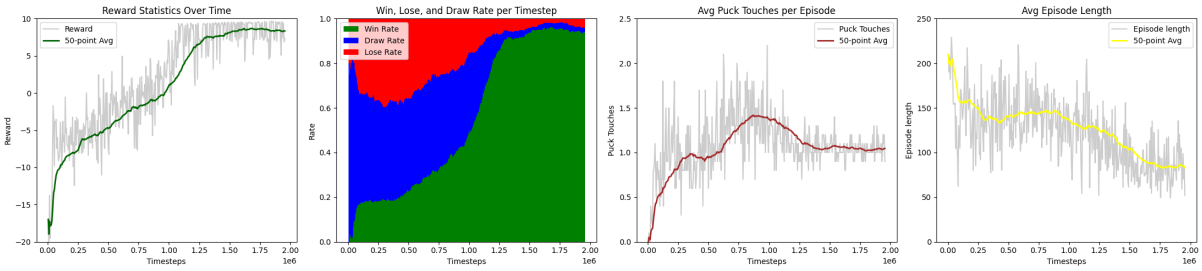


Figure 4: Statistics of the final SAC tournament agent

performed, respectively. After each game the agent was trained with 320 samples from the online buffer while 250 steps where additionally added from the environment given a strong bot.
Unfortunately. due to a lack of time the online agent could not be tested before the tournament but was directly deployed. The agent diverged most probably due to bad training hyperparameters. In the appendix a heatmap of the agents positions can be found (see figure 9). The divergence could be caused by multiple reasons. The most probable reason is that the reward function was reshaped, namely only the goal reward was considered while disregarding the puck closeness reward. Furthermore, the replay buffer was reset at the beginning of the tournament, so that it could have happened that the agent overfitted to the first few games. What indeed caused the divergence is up for investigation.

3.1.2 Final Agent Performance

In the final training run with hyperparameters equivalent to the hyperparameters of batch 4 run 4 from table 2, but with strong opponent training, we reach a convergence plateau at around 1.6 million timesteps. The winning rate converges to 1 (maximum winning rate) and the reward to around +10 (reward amount for a goal). Figure 5 visualizes the reward evolution and the winning rate evolution over the timesteps. Furthermore, it shows that the episode length is steadily declining which obviously makes sense. Interestingly the average amount of timesteps in one episode in which the agent has the puck is first rising and at around 0.9 million timesteps declining again. It converges to one touch per game. This essentially means that the agent always shoots a goal if he tries to. This behavior might indicate overfitting.
To investigate the issue of overfitting in more detail we compare the agents winning rate against the weak opponent while training on the strong opponent. If our agent begins to overfit to the strong agent, the performance against the weak opponent will drop.
From figure 5 we can see that the winning rate against the weak opponent is decreasing from 1.5 million timesteps on. The loosing rate is rising sharply afterwards. Overfitting will take place the latest at this timepoint. However as the behavior of both bots are similar due to their implementation overfitting might already take place earlier. After 0.9 million timesteps we can see that the winning rate is rising double as fast as in the time before. I suspect overfitting is already taking place from this point in time onward.
Given this assumption I used the agent trained until 0.9 million timesteps for the tournament. To be more sure about when overfitting to the strong agent starts, some more research should have been done but this was not further pursued due to the lack of time. Further investigations could have been a broad comparison with other agents e.g. the agents from the tournament test mode.
The submitted agent reached place 80 in the final tournament and does showed general environment knowledge and exploitation.

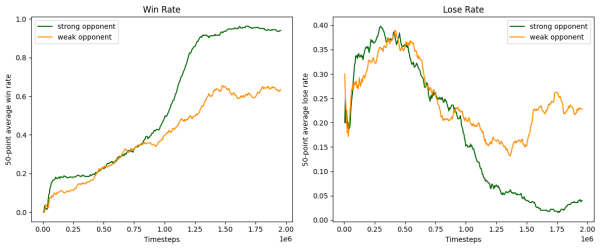


Figure 5: Win Rate and Lose Rate of SAC Agent against strong and weak Bot

4 Final Overview

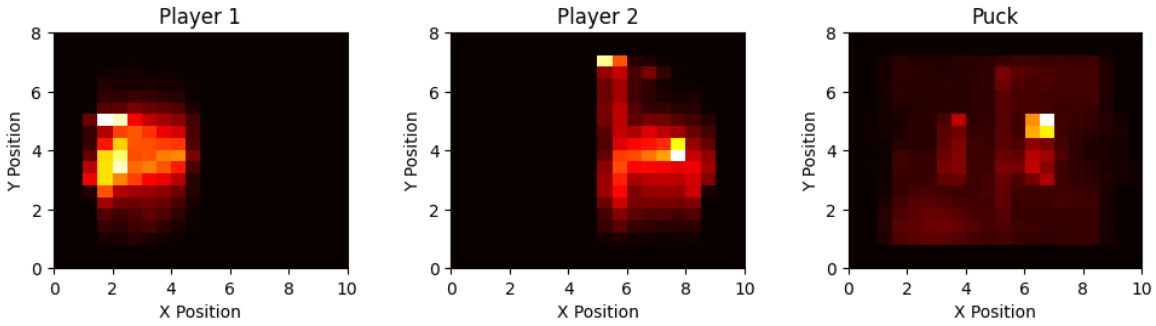
In the final overview we would like to give a more detail comparison of both agents. Therefor we let them play against each other and both bots (strong and weak). Table 1 shows the final stats playing 1000 games in each combination. Our DDPG algorithm clearly outperforms the SAC algorithm and confirms the result from the tournament. However the SAC agent has a significantly higher winning rate on both weak and strong players. This might be because of a higher variance in the players states and actions. The DDPG algorithm seems to have better defence properties as its loosing rate is constantly at around 10%. We conclude that as the DDPG algorithm is trained for more timesteps (around 1.6 million) it probably has seen more states and can thus better defend itself. Moreover the DDPG algorithm was partly trained with self-play. As agents trained on the same bot might exploit same weaknesses of the bots game play, self-play itself helps to learn a defense against these weaknesses and may explain the better performance of DDPG against SAC.

Players		**Win Rates (%)**	
Player 1	Player 2	Player 1	Player 2
DDPG	SAC	46.2	12.3
DDPG	weak Bot	28.6	10.9
DDPG	strong Bot	28.1	10.3
SAC	weak Bot	51.4	19.1
SAC	strong Bot	52.5	18.4

Table 1: Win Rates of Players Competing Against Each Other

Figure 6 gives another nice overview over the different states visited for all agents and pucks. This gives some information about the variance over the different agents and the states visited in a game. We can see that the SAC agent moves wider across the field why the DDPG agent mostly stays in front of the goal. This also might explain better defense properties of the DDPG algorithm. Furthermore, the SAC agent has a bias towards upper field positions. It visits the upper field positions more frequent. This indicates some underfitting and could be tackled by longer training time or different training scenarios e.g. self-play.

Compared to the heatmap of the weak and strong bot (see Appendix fig. 8, fig. 7) both agents have a clearly higher variance in the positions visited. Nevertheless it must be said that the position only partially reflects the variance of a players state as the state of a player also include the orientation and the velocity.



Player one: DDPG, Player two: SAC

Figure 6: Heatmap of positions visited by players and puck.

The final take-aways of the project are as follows:

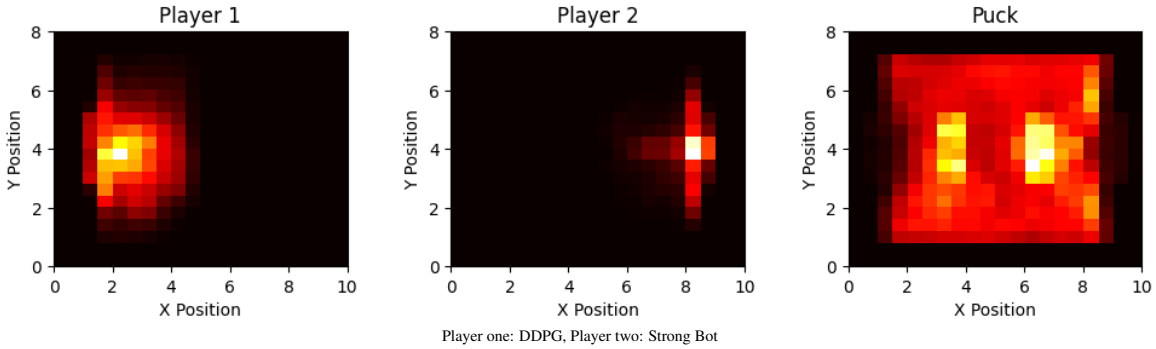
1. training might be frustrating and takes more time as you would expect
2. hyperparameters are crucial for efficient learning and dependent on the environment

If we would have had more time we would have tried to implement a working version of the online learning agent with tuned hyperparameters. Furthermore, we would have used more time for self play and training testing against other agents.

Appendix

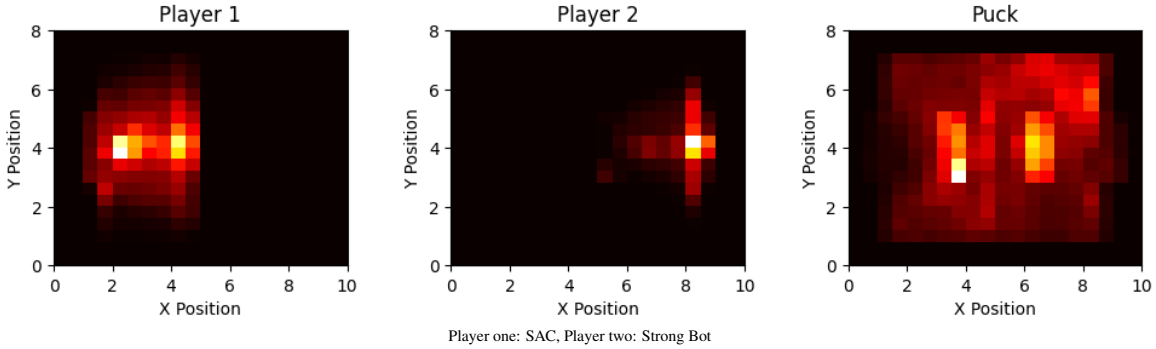
Batch	Run	Hyperparameters					
		Network Architecture	Learning Rate	Replay Buffer Size	Discount Factor	Training Frequency	Batch Size
1	1	2 x 128	1e-3	1e4	0.99	(2, episodes)	256
	2	2 x 128	5e-3	1e4	0.99	(2, episodes)	256
	3	2 x 128	5e-4	1e4	0.99	(2, episodes)	256
	4	2 x 64	1e-3	1e4	0.99	(2, episodes)	256
	5	3 x 64	1e-3	1e4	0.99	(2, episodes)	256
2	1	2 x 128	5e-4	1e5	0.95	(1, episodes)	512
	2	2 x 128	5e-4	1e5	0.97	(1, episodes)	512
	3	2 x 128	5e-4	1e5	0.99	(1, episodes)	512
	4	2 x 128	5e-4	1e5	0.99	(1, episodes)	512
	5	3 x 64	5e-4	1e5	0.99	(1, episodes)	512
	6	2 x 256	5e-4	1e5	0.99	(1, episodes)	512
	5^*	2 x 128	5e-4	1e5	0.99	(1, episodes)	512
	6^+	2 x 128	5e-4	1e5	0.99	(1, episodes)	512
3	1	2 x 128	1e-4	1e6	0.95	(1, step)	256
	2	2 x 256	1e-4	1e6	0.95	(1, step)	256
	3	2 x 128	1e-4	1e6	0.95	(5, episode)	4095
	4	2 x 128	5e-4	1e6	0.95	(5, episode)	4095
	5^r	2 x 128	1e-4	1e6	0.95	(1, step)	256
	5^w	2 x 128	1e-4	1e6	0.95	(1, step)	256
4	1	2 x 256	5e-4	5e5	0.95	(1/10, episodes)	128
	2	2 x 256	5e-4	5e5	0.95	(1/10, episodes)	32
	3	2 x 256	5e-4	5e5	0.95	(1/10, episodes)	256
	4	2 x 256	5e-4	1e5	0.95	(1/10, episodes)	128
	5	2 x 256	5e-4	1e6	0.95	(1/10, episodes)	128
	6	2 x 128	5e-4	5e5	0.95	(1/10, episodes)	128

Table 2: Batch-wise Hyperparameter Runs



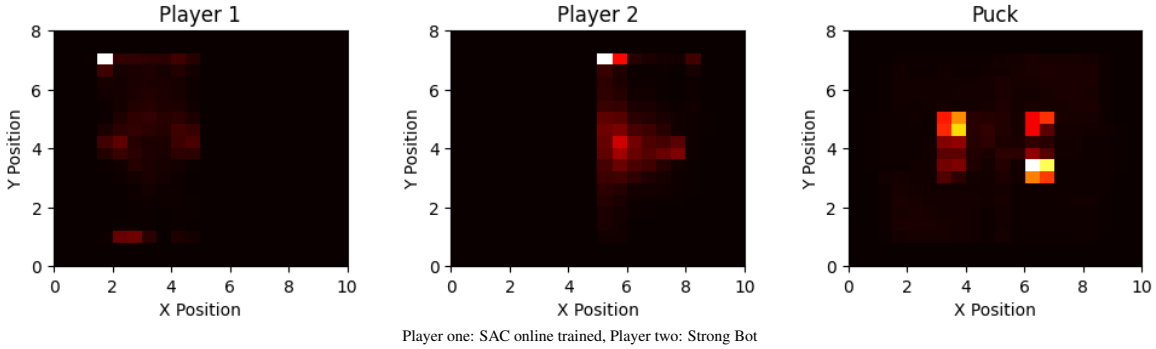
Player one: DDPG, Player two: Strong Bot

Figure 7: Heatmap of positions visited by players and puck.



Player one: SAC, Player two: Strong Bot

Figure 8: Heatmap of positions visited by players and puck.



Player one: SAC online trained, Player two: Strong Bot

Figure 9: Heatmap of positions visited by players and puck.