RL-Course 2024/25: Final Project Report

Bellman's Bandits: Raul Grau, Niclas Lietzow, Philipp Davydov

February 26, 2025

1 Introduction

The following report presents our solution for the Laser Hockey Challenge as part of the Reinforcement Learning course at the University of Tübingen.
The Laser Hockey environment is built using the Gymnasium API [18] in Python. At its core, the environment is a continuous control task that simulates a two-player hockey game. At each time step, the agents observe the environment and take simultaneous actions with the goal of scoring a goal against each other. The environment comes with a weak and strong version of a basic opponent, which can be used for training RL agents and also serves as a baseline for running experiments.
We present different methods to train an agent to play the game. We will observe that the trained agents can easily defeat the basic opponent provided by the environment, and present solutions on how to use self-play training to develop policies with a performance that goes far beyond the basic opponents.
Specifically, we have implemented the following standard algorithms with modifications

1. Dueling DQN (Philipp Davydov)
2. Twin Delayed DDPG (Raul Grau)
3. Soft Actor-Critic (Niclas Lietzow)

For each algorithm, we will give a brief overview of the theoretical foundations, followed by modifications and implementation details. Finally, we present studies on model architecture, hyperparameter choice, and training setup. The code for our implementations can be found in our Github repository.

2 Rainbow

Rainbow [10] explored various extensions that were previously independently proposed for the Deep Q-Network (DQN) algorithm [16], as well as their combinations. The authors showed that combining all methods led to a significantly better performance and higher data efficiency. Based on their findings, we implemented three of the proposed additions for the laserhockey challenge, namely Prioritized Experience Replay (PER) [17], Double DQN [9] and Dueling DQN [19]. We outline each of the methods in Section 2.1 and present our experimental findings in Section 2.2, highlighting the influence of individual components and hyperparameters used.

2.1 Methods

All following algorithms are based on the original Q-Learning algorithm [20]. We assume that our agent interacts with the environment for t timesteps. At each timestep, it observes the state s_t, performs the

action $a_t \in \mathcal{A}$ chosen from a discrete action space $\mathcal{A}$ using the policy $\pi(s) = P(a|s)$, after which the environment returns the reward r_{t+1} along with the next state s_{t+1}. Our goal is to maximize the discounted future return $G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$ at each time step t, where $\gamma \in [0, 1]$ is a hyperparameter. For each policy, we can define the value function and action-value function:

$$V^\pi(s) = \mathbb{E}_\pi\Big[G_t|s_t = s\Big] \qquad\qquad Q^\pi(s, a) = \mathbb{E}_\pi\Big[G_t|s_t = s, a_t = a\Big] \tag{1}$$

For the optimal Q-function $Q^*(s, a) = \max_\pi \ Q^\pi(s, a)$, Bellman's optimality equation holds: $Q^*(s, a) = \mathbb{E}\Big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}\Big]$. Based on that, Q-Learning defines an iterative method that approximates Q^*:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big([r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})] - Q(s_t, a_t)\Big) \tag{2}$$

$\Big([R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})] - Q(s_t, a_t)\Big)$ is also called the TD error, which is the difference between the value if Q was optimal, and the real value predicted by the current Q function.
Having an approximation of Q^*, the policy can be extracted by simply setting $\pi^*(s) = \max_a \ Q^*(s, a)$.

2.1.1 Deep Q-Learning

Deep Q-Learning [16] borrows the main ideas from Q-Learning, but models the Q-Function using a Deep Neural Network $Q(\theta)$ with input s and $|\mathcal{A}|$ output values, corresponding to each value of $Q(s, a; \theta)$ for the current input. Squared TD error is used as a loss function for gradient descent. As Deep Learning requires uncorrelated samples to converge well, the authors introduce a replay buffer D, which is a dataset containing the agent's past interactions with the environment and is used to sample uncorrelated minibatches during training. To further stabilize the algorithm and avoid a continually shifting target distribution, two Q-networks are used, leading to the loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1}) \sim \mathcal{U}(D)} \left[\left(\Big[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i^-)\Big] - Q(s_t, a_t; \theta_i) \right)^2 \right] \tag{3}$$

Here i denotes the current timestep, θ are the parameters of the Q-network and θ_i are the parameters of the target network. The target network is not trainable and periodically updated with the Q-network's state during training.
In our implementation we use smooth L1 loss instead of L2 loss, because it is less sensitive to outliers.

2.1.2 Prioritized Experience Replay

Instead of uniformly sampling experience from the replay buffer, Prioritized Experience Replay (PER) [17] proposes to sample experiences that are more useful for learning with higher probability, considering that samples with high TD error δ must me beneficial. For that, each data point in the replay buffer is assigned a priority value $p_i = |\delta_i| + \epsilon$, with $\epsilon > 0$ ensuring all samples have at least a small probability of getting sampled, even if they have low loss.
We can now assign each data point i a sampling probability $P(i)$ and importance sampling weight w_i:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \qquad\qquad w_i = \Big(\frac{1}{P(i)} \cdot \frac{1}{N}\Big)^\beta \tag{4}$$

The hyperparameter α determines the strength of prioritization, $\alpha = 0$ yields the uniform case. However, since prioritized sampling introduces a bias in the training process, stochastic gradient descent may converge to a different solution than previously. PER corrects the bias by adding importance sampling weights w_i, which are further normalized by setting $w_i = \frac{w_i}{\max_i w_i}$. The weights are used during loss calculation and fully compensate for the priority sampling if $\beta = 1$. For that reason, β is typically gradually increased to 1, annealing the effect of prioritized sampling towards the end of training.

2.1.3 Double Deep Q-Learning

The max-operation in the normal DQN formulation introduces a bias by systematically overestimating its target, because it uses the same parameters for selecting a and evaluating Q:

$$r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^-) = r_{t+1} + \gamma Q(s_{t+1}, \underset{a_{t+1}}{argmax}\, Q(s_{t+1}, a_{t+1}; \theta^-); \theta^-) \tag{5}$$

Double Q-Learning addresses this by changing the target to:

$$r_{t+1} + \gamma Q(s_{t+1}, \underset{a_{t+1}}{argmax}\, Q(s_{t+1}, a_{t+1}; \theta); \theta^-) \tag{6}$$

Now, the selection of a is done according to the current state of Q, but the evaluation is done with respect to the target Q-network, which reduces the bias.

2.1.4 Dueling Networks

The idea of Dueling Networks [19] is to decompose the Q-function into the value function V^π and advantage function $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$, which are both learned explicitly during training, using a network with a shared backbone and two heads.
Since it's impossible to uniquely recover V^π and A^π from Q^π using the formulation above, the following decomposition is used for training instead:

$$Q^\pi(s, a) = V^\pi(s) + \Big(A^\pi(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} A^\pi(s, a)\Big) \tag{7}$$

2.2 Experiments

We performed various experiments for Rainbow to determine good hyperparameters and decide which extensions are helpful for the hockey environment. Each experiment was done three times with the random seeds $\{42, 43, 44\}$, and we report the median value for each timestep. Training was performed for 8 million timesteps while periodically calculating the percentage of wins against the provided weak basic opponent.
Unfortunately, the space of hyperparameters is very large and prohibits extensive hyperparameter search, such as grid search. Instead, we tested hyperparameters independently of each other and combined the insights to train the final model for the tournament.
Our first ablation study, shown in Fig. 1 **(a)**, highlights the effect of removing either of the DQN extensions: PER and Dueling Networks are the main contributors to the convergence rate of Rainbow. Interestingly, disabling Double DQN had no strong effect, or even led to slightly more stable training.

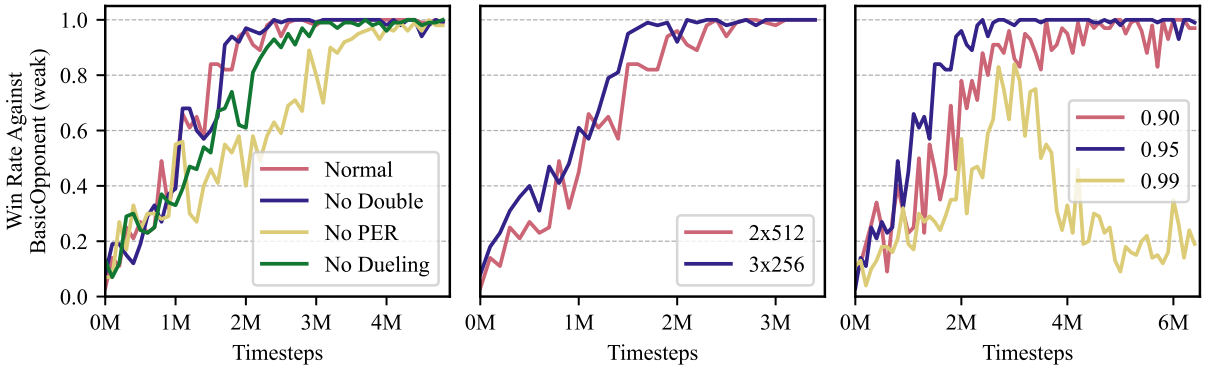


Figure 1: Experiments for architecture and hyperparameter choice. **(a)**: Effect of removing DQN extensions. PER and Dueling Networks have the largest impact. **(b)**: Convergence of network with 2x512 vs. 3x256 hidden units. The deeper network converges faster and is more stable. **(c)**: Effect of the discount factor γ. Convergence behavior is best for $\gamma = 0.9$

Next, we compared two network architectures: a shallow, but large network with two hidden layers of size 512, against a deep, but small network with three hidden layers of size 256. Fig. 1 **(b)** shows that using the deeper network led to faster and more stable convergence, while requiring less computation time due to the smaller number of parameters, hence we used this network architecture for the tournament.
Finally, we tuned the discount factor γ in Fig. 1 **(c)**. While $\gamma = 0.9$ made the training slightly instable and took longer to converge, the training progress completely broke down for $\gamma = 0.99$ after $30 \cdot 10^5$ timesteps. The best-behaved value was $\gamma = 0.95$.
Since DQN operates on a discrete action space, but the provided hockey environment had a continuous one, we had to manually define actions. We compared a minimal set of actions, only changing one action space dimension at a time, to a full set of actions, which contained all combinations of $\{-1,\ 0,\ 1\}$ for the first three dimensions of the action space ("left/right", "top/bottom", "spin"), as well as the shooting action. We also included a compromise between the two, in which moving left/right and top/bottom at the same time was prohibited. We found that there was no significant difference on the weak basic opponent, but decided to use the full action space for the tournament in hope of generating more refined behavior that would generalize better to strong opponents.
Our final hyperparameter choices are reported in Appendix A. The parameters for PER were inspired by the hyperparameters reported in [10]
To train for the tournament, we implemented self play: During training, an opponent pool was maintained, from which a new model was uniformly sampled every two episodes. Every 50,000 timesteps, the agent was evaluated against the current opponent pool by playing 250 games each and computing the average win rate per opponent. Every time the agent beat his previous best average win rate, a copy of itself was added to the opponent pool. Initially, the pool contained only the basic opponents and the win rate threshold was set at 90%. The self play process is displayed in Fig. 2.

3 TD3

Twin Delayed Deep Deterministic policy gradient (TD3) [6] is an actor-critic based reinforcement learning algorithm that improves upon the earlier DDPG [14] approach by adding three modifications, stabilizing the learning. TD3 has one neural network as actor π_θ. Given an observation s during training, TD3 uses actor π_θ with exploration noise to choose the next action $a = \pi_\theta + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$. After training,

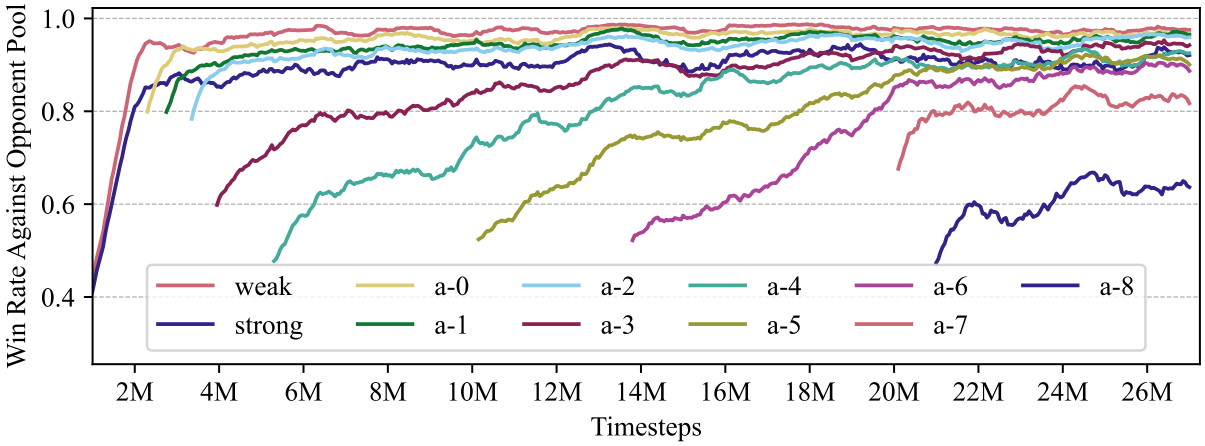


Figure 2: Win rates of Rainbow against opponents from the opponent pool during self play, out of 250 games per evaluated point. For better clarity, we apply a moving average filter with kernel size 20. The first two opponents are the provided basic opponents, while "a-[idx]" are checkpoints from the agent's training.

the actor can be used in a deterministic way without exploration noise. In each training step, a batch of N transitions (s, a, r, s') is sampled from replay buffer $\mathcal{D}$, and the target network of the actor predicts the action for the next step while using **target policy smoothing regularization** to compute the current Q-values.

$$\tilde{a} = \pi_{\theta'}\left(s'\right) + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$$

Adding noise ensures that similar actions have similar Q-values. This is then used to compute the smoothed temporal-difference (TD) errors, where TD3 uses as an improvement **two critics** ("twin"), Q_{ϕ_1} and Q_{ϕ_2}, and takes the minimum of both, reducing the overestimation when predicting the Q-values for state-action pairs:

$$\delta\left(s, a, \tilde{a}, r, s'\right) = \underbrace{\left(r + \gamma \min_{i=1,2} Q_{\phi_i'}(s', \tilde{a})\right)}_{\text{TD target} = y(r,\tilde{a},s')} - Q_{\phi_i}(s, a)$$

whose mean squared is minimized to optimize the critics Q_{ϕ_i}:

$$L\left(\phi_i, \mathcal{D}\right) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_{\phi_i}(s, a) - y\left(r, \tilde{a}, s'\right)\right)^2\right]$$

Next, the actor is updated using the deterministic policy gradient in a **delayed** way every d'th critic update:

$$\nabla_\theta L(\theta) = \frac{1}{N}\sum \nabla_a Q_{\phi_1}(s, a)\Big|_{a=\pi_\theta}(s)\nabla_\theta \pi_\theta(s)$$

The delayed policy update reduces policy degradation caused by bad value estimates. The parameters from all three target networks $\pi_{\phi'}$, $Q_{\theta_1'}$, $Q_{\theta_2'}$ are updated with Polyak averaging using parameter $\tau \in (0, 1)$.

$$\phi_i' \leftarrow (1 - \tau)\phi_i' + \tau\phi_i$$
$$\theta' \leftarrow (1 - \tau)\theta' + \tau\theta$$

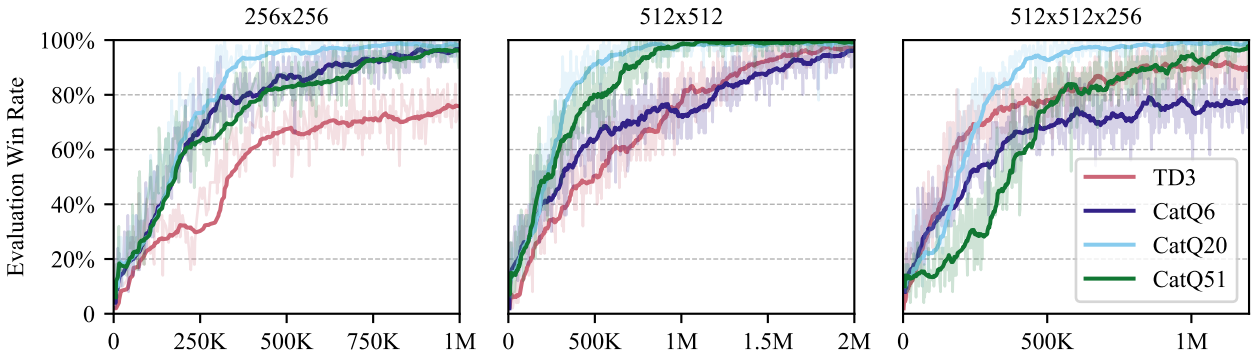


Figure 3: TD3 and TD3-CategoricalQ: comparing both approaches on different critic network sizes with hidden layer sizes of (256,256), (512,512) and (512, 512, 256) respectively. The x-axis shows timesteps and the y-axis shows the win rate at evaluation against the Basic Opponent. The lines for TD3-categoricalQ with 6, 20 and 51 bins are shown in purple, blue and green.

3.1 Categorical Q

As suggested in [5], using cross-entropy for training value functions increases scalability-related performance, as it focuses on categorical rather than numerical relationships between input and target. Since the operation for calculating the targets usually involves scalars, there are mainly two ways to deal with this: to define a distributional Bellman operator [2] or to perform two transformations: for the calculation of TD-target, categorical to scalar, and for the cross-entropy calculation, scalar back to categorical [11]. We tried the latter. To achieve this, the Q-function approximators output a distribution rather than a scalar; the last layer of the respective multilayer-perceptron, instead of having one neuron, has m neurons, the number of bins. Then we take the expectation of it to get a scalar for target computation:

$$y\left(r, s'\right) = r + \gamma \min_{i=1,2} \mathbb{E}\left[Q_{\phi_{i,\,\text{targ}}}\left(s', \tilde{a}\right)\right]$$

Then we use the HL-Gauss method: we project the scalar target to onto the categorical distribution supported on $\{z_i\}_{i=1}^{m}$ with bins of width $\delta = (v_{max} - v_{min})/m$ by integrating over:

$$p_i(S, A; \tilde{\theta}) = \int_{z_i-\delta/2}^{z_i+\delta/2} f_{Y|S,A}(y \mid S, A)dy = F_{Y|S,A}\left(z_i + \delta/2 \mid S, A\right) - F_{Y|S,A}\left(z_i - \delta/2 \mid S, A\right)$$

where $Y \mid S, A \sim \mathcal{N}(\mu = y\left(r, s', d\right), \sigma^2)$. We used a fixed sigma ratio to calculate the respective sigma value: $\sigma = (v_{max} - v_{min})/m \cdot \frac{\sigma}{\delta}$. We choose 0.75, because it is a recommended value. The CrossEntropy Loss is then calculated which is used to optimize the critics.

3.2 Pink Noise

Since TD3 is an off-policy algorithm, it allows for the replacement of the commonly used Gaussian noise with temporally correlated alternatives, such as Ornstein-Uhlenbeck noise, which can enhance state space exploration. Recent research has demonstrated that temporally correlated Pink Noise serves as an effective default choice, often outperforming other noise distributions without significant drawbacks in standard reinforcement learning environments [4]. It is characterized by a power spectral density that follows a $\frac{1}{f}$ distribution.

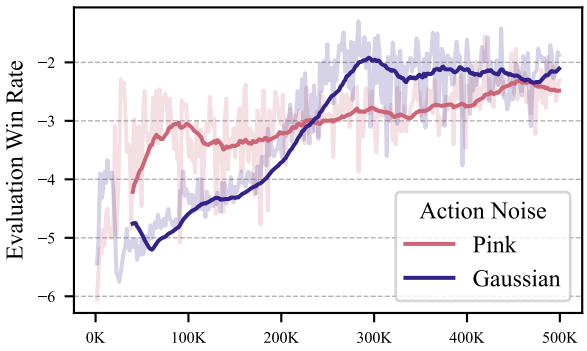


Figure 4: Comparison of normal action noise and pink action noise.

3.3 Experiments

We compared the standard implementation of TD3 with the one with categorical critics across different critic network architectures with 4 different seeds. The policy network remains with two hidden layers of size 256. At each episode during learning a BasicOpponent was uniformly sampled from [weak, strong]. The hyperparameters for the experiment can be found in the Appendix B. The results in Fig. 3 confirm that setting up the correct number of bins is crucial, especially in larger networks. We can see that the model with 20 bins shows a good performance during learning against the basic opponents.
We setup an experiment to test the impact of using coloured action noise, specifically pink noise, for our algorithms, and for that we trained against the BasicOpponent with the same hyperparamters. It didn't yield better results as Fig. 4, we hypothesize that the broad state space exploration facilitated by pink noise may that much of an advantage in the hockey environment

3.4 Selfplay

For the competition we trained the TD3-categoricalQ algorithm, with (256, 256) parameters and hyperparameters as in the appendix table. We pretrained the algorithm against the weak and strong opponent by sampling uniformly from them. Then after a certain evaluation threshold was reached (0.9), we started adding frozen copies to a opponent pool of fixed size every 1M steps and let it run for a total of 12M steps.

4 Soft Actor-Critic

In this section, we introduce the Soft Actor-Critic algorithm [7] and its modifications before performing ablation studies on hyperparameters and architectural design. The algorithm has changed slightly over the years. For the derivation and implementation of the modern version of the algorithm we follow [1].

4.1 Methods

Soft Actor-Critic (SAC) [7] is an algorithm that builds on previous actor-critic approaches, particularly the Deep Deterministic Policy Gradient (DDPG) [15], which uses off-policy data from a replay buffer to learn the Q-function and then leverages the Q-function to learn the policy. The algorithm also incorporates clipped double Q learning introduced in Twin Delayed DDPG (TD3) [6].
The key feature of SAC is the use of a stochastic policy that allows entropy regularization [21] to be added to the training. The policy is optimized with respect to a trade-off between the expected return and

the entropy of the policy. Increasing entropy enhances exploration without requiring additional action noise while still avoiding clearly unpromising trajectories.
In the following sections, we first introduce the framework of entropy-regularized reinforcement learning [21]. We then derive the update steps for the Q-functions and the policy to reflect the regularization term. Finally, we discuss the automatic tuning of the entropy coefficient [8] and CrossQ [3], two popular modifications of the standard SAC algorithm.

4.1.1 Entropy-Regularized Reinforcement Learning

The standard objective function in reinforcement learning is given by the discounted expected return when acting according to a policy π. When using a stochastic policy, it becomes possible to add a regularization term to the objective function that reflects the entropy of the policy. Specifically, the objective function is modified to $J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right)\right]$, where H is the entropy, $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ is a trajectory whose distribution is derived from the policy, and $\alpha > 0$ is the entropy coefficient, also called the temperature parameter. The value functions in the maximum entropy framework consequently change to

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right)\middle| s_0 = s\right],$$
$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\middle| s_0 = s, a_0 = a\right].$$

The connection between the two value functions is given by $V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$.

4.1.2 Soft Actor-Critic

Similar to TD3 [6], the algorithm simultaneously learns a policy π_θ and two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}. The Q-functions are learned via stochastic gradient descent with respect to the mean squared Bellman error by regressing to a shared target using the clipped double-Q method. The targets are computed using target Q-functions, which are obtained by iteratively applying Polyak averaging during training.
Using the definition of entropy and the action-value function from above, we get

$$Q^\pi(s,a) = \mathbb{E}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha\log\pi(a'|s')\right)\right].$$

This enables us to derive the loss function

$$L(\phi_i,\mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2\right],$$

where $\mathcal{D}$ represents the replay buffer. The target is given by

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',\tilde{a}') - \alpha\log\pi_\theta(\tilde{a}'|s')\right),$$

where $\tilde{a}' \sim \pi_\theta(\cdot|s')$ is sampled from the current policy.
For the policy, the authors suggest using a squashed Gaussian policy to ensure that actions remain within a finite range. Additionally, we use the reparametrization trick, which enables us to compute the gradient with respect to θ. For a given state s, samples from the policy can be obtained according to

$$\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s)\odot\xi\right), \quad \xi\sim\mathcal{N}(0,I),$$

where $\mu_\theta(s)$ and $\sigma_\theta(s)$ are deterministic functions parameterized by neural networks.
In each state s, the policy should act to maximize the value function $V^\pi(s)$. Using the reparametrization trick, we can rewrite the value function as follows:

$$V^{\pi_\theta}(s) = \mathbb{E}_{\xi\sim\mathcal{N}(0,I)}\left[Q^{\pi_\theta}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right].$$

We use the minimum of the two Q-functions to approximate Q^{π_θ}, which yields the optimization problem

$$\max_\theta \mathbb{E}_{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}(0,I)}}\left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right].$$

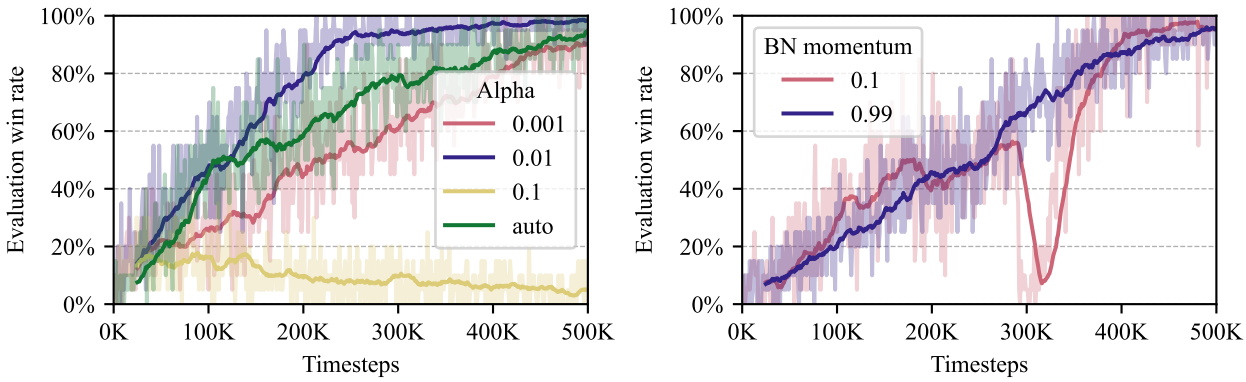
Finally, the derived loss functions can be optimized using stochastic gradient descent, alternating the update steps of the Q-functions and the policy.

4.1.3 Automatic entropy tuning

In the original paper, the authors introduced the entropy coefficient $\alpha > 0$ as a fixed hyperparameter that has a strong influence on the performance of the algorithm but needs to be tuned for the specific task at hand. However, manual tuning can be difficult because the optimal value varies greatly between environments and may also change as the policy improves. Thus, an automatic tuning mechanism has been proposed to dynamically adjust this parameter [8]. We set a target entropy $\mathcal{H}$, typically defined as $\mathcal{H} = -\dim(\mathcal{A})$, and adjust α based on the current policy to match the target entropy. To achieve this, we use the loss function $J(\alpha) = \alpha\mathbb{E}_{s_t\sim\mathcal{D}, a_t\sim\pi}\left[-\log\pi(a_t|s_t) - \mathcal{H}\right]$ and update α using stochastic gradient descent. It can be verified that α decreases when the policy entropy exceeds the target entropy and increases when it falls below the target. As a result, automatic entropy tuning dynamically adjusts α during training to prevent excessive or insufficient exploration.

4.1.4 CrossQ

SAC and many other approaches rely on target networks to stabilize training, even though Polyak averaging arguably slows down the training process. The authors of CrossQ [3] propose removing the target networks altogether and achieving training stability using variants of Batch Normalization (BN) [13], specifically Batch Renormalization (BRN) [12]. As a result, CrossQ accelerates training by improving both sample and computational efficiency. In the following, we explain why removing target networks is necessary when using BN and how to adjust the forward pass to avoid distribution shifts.
Recall the loss function of the Q-functions, $L(\phi_i, \mathcal{D})$, which depends on $Q_{\phi_i}(s,a)$ and $Q_{\phi_{\text{targ},j}}(s', \tilde{a}')$. The batch $(s', \tilde{a}')$ is sampled from the current policy, while (s,a) comes from the replay buffer and was therefore generated by older policies. The BN running statistics of the live network are estimated from replay buffer batches (s,a) and thus have not seen samples from the distribution of $(s', \tilde{a}')$, causing a mismatch in their statistics. In particular, the state-action values evaluated by the target network will be out of distribution. To avoid issues caused by these test-time distribution shifts, the target networks are removed.
This enables a joint forward pass $Q_\theta\left(\begin{bmatrix}s\\ s'\end{bmatrix}, \begin{bmatrix}a\\ \tilde{a}'\end{bmatrix}\right) = \begin{bmatrix}q\\ q'\end{bmatrix}$ and ensures that the same BN normalization moments are used for both batches, contributing equally to the mixture with their respective distributions. The authors emphasize that this trick is necessary for safely applying BN in critic networks.
Beyond the use of BRN and the removal of target networks, the authors suggest widening the critic layers, adjusting optimizer hyperparameters, and applying a policy delay analogous to TD3. We incorporate these modifications and compare the resulting algorithm with standard SAC in the following experiments.



(a) Effect of α on performance and convergence speed.

(b) Effect of the BN momentum on training stability.

Figure 5: Hyperparameter tuning for SAC (left) and CrossQ (right).

4.1.5 Self-play training

Finally, to train an agent for the competition, we use the model architecture and hyperparameters as proposed in the CrossQ paper [3], with the only modification being the use of larger and deeper hidden layers of size $(512, 512, 256)$ in the actor network. The agent is trained using bootstrap self-play. We pre-train the agent for $500,000$ steps against the base opponent in strong mode. After that, we start building an opponent pool from the model's checkpoints. We gradually increase the frequency of checkpoint generation from 100 thousand steps to $500,000$ timesteps and train the agent for another 20 million timesteps. A comprehensive list of hyperparamerers used can be found in C.

4.2 Experiments

In the first experiment, we analyze the impact of the entropy coefficient in the standard SAC algorithm (Fig. 5a). It can be seen that the temperature coefficient greatly impacts the performance of the algorithm. Choosing the temperature value too high results in no convergence, while choosing it too low leads to very slow convergence. Carefully setting the temperature to a fixed value of $\alpha = 0.01$ achieves even faster convergence than automatic temperature tuning for this particular training scenario. However, to allow the temperature to adjust dynamically during training, we decided to proceed with automatic tuning instead of a fixed entropy coefficient.

In the second experiment, we tested the standard SAC algorithm against the CrossQ algorithm as suggested in the paper (Fig. 6). The fast convergence of SAC against the weak opponent might originate from the smaller critic networks, which are easier to optimize. However, CrossQ demonstrates better performance against the strong opponent. We therefore choose the CrossQ algorithm over standard SAC.

Finally, we demonstrate that using BN in reinforcement learning requires much higher BN momentum values than in other supervised learning applications (Fig. 5b). The authors of CrossQ suggest a BN momentum of 0.99. We compare this with a BN momentum of 0.1, which is commonly used in other applications. We observe that a low BN momentum value can accelerate training, but leads to significant instabilities, confirming the choice of a BN momentum of 0.99.

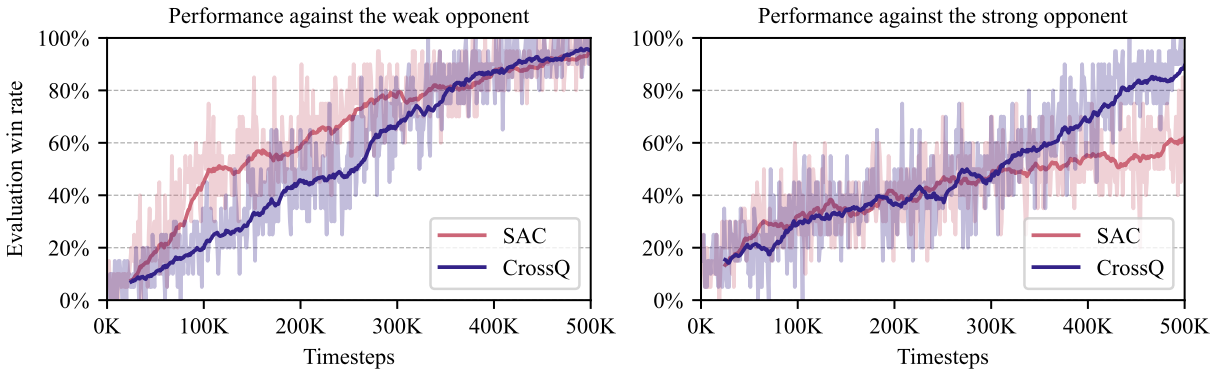


Figure 6: Comparison of standard SAC with the modifications introduced in CrossQ.

5 Discussion

In this paper, we implemented, trained and tested three off-policy algorithms: DQN/Rainbow, SAC/CrossQ and TD3. All of our agents were able to pass the provided requirement of consistently winning against the weak and strong basic opponents, but showed varying training behavior and competitive performance. To conclude, the following section provides a short summary and comparison of the methods.

It is most notable that contrary to our previous assumption, Rainbow performed best during the tournament, ranking third place overall, whereas SAC and TD3 performed second- and third-best, respectively. This can be seen as surprising, considering that only two extensions of DQN proved useful for Laser Hockey (namely PER and Dueling Networks), and furthermore comparing the constrained, discretized action space of Rainbow with the continuous action space of the other methods, that should be potentially able to learn more elaborate playing strategies.

In fact, no other top-ranked agent in the tournament was based on DQN, confirming our assumption that SAC and TD3 are generally more powerful. However, we point out that we managed to saturate the self-play process of Rainbow, meaning the last checkpoint added to it's opponent list was so powerful, that the trained agent never achieved a win rate larger than 70% against it. We conclude that the design and the duration of self-play was a crucial part of Rainbow's performance and made up for the algorithm's shortcomings. Moreover we assume that a larger architecture such as 2x512, which we discarded due to its slower convergence rate, could surpass the plateau of the used model and reach a top performance against a larger opponent pool, if we increase training time significantly.

While showcasing competitive performance, one of the biggest drawbacks of Rainbow compared to our other methods was its convergence rate: It took almost 2 million timesteps to consistently beat the weak basic opponent, meanwhile TD3 reached the same performance level in half the time. The fastest converging method proved to be SAC, beating both the weak and strong opponents after just 500,000 timesteps during training. In particular, the CrossQ extension was a key contribution the success and fast convergence rate of the method, whereas from the extensions tested for TD3, Categorical Q was most significant and Pink Noise turned out to be less useful.

In conclusion, all the implemented methods showcase strong performance against the basic opponents for most hyperparameter choices, but the generalization to other opponents was mainly determined by major architectural choices, as well as the duration of training within our self-play framework, which ultimately allowed our team to reach third place in the tournament.

References

[1] J. Achiam. Spinning up in deep reinforcement learning. `https://spinningup.openai.com`, 2018. Educational resource from OpenAI.

[2] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 449–458. JMLR.org, 2017.

[3] A. Bhatt, D. Palenicek, B. Belousov, M. Argus, A. Amiranashvili, T. Brox, and J. Peters. Crossq: Batch normalization in deep reinforcement learning for greater sample efficiency and simplicity, 2024.

[4] O. Eberhard, J. Hollenstein, C. Pinneri, and G. Martius. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR 2023)*, May 2023.

[5] J. Farebrother, J. Orbay, Q. Vuong, A. A. Taïga, Y. Chebotar, T. Xiao, A. Irpan, S. Levine, P. S. Castro, A. Faust, A. Kumar, and R. Agarwal. Stop regressing: training value functions via classification for scalable deep rl. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[6] 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.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 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 1861–1870, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[8] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications, 2019.

[9] H. v. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[10] M. Hessel, J. Modayil, H. V. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI*, 2018.

[11] E. Imani and M. White. Improving regression performance with distributional losses. 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 2157–2166. PMLR, 10–15 Jul 2018.

[12] S. Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models, 2017.

[13] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[14] 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.

[15] 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.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

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

[18] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. D. Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan, and O. G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[19] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, 2015.

[20] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[21] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, AAAI'08, pages 1433–1438. AAAI Press, 2008.

A Hyperparameters for Rainbow

Hyperparameter	**Value**
Batch size	32
Target network update frequency	1000
Epsilon decay for exploration	1×10^{-5}
Learning rate	1×10^{-4}
Optimizer	Adam
Discount factor γ	0.95
Hidden layers	[256, 256, 256]
Start training after n timesteps	50000
Training frequency	4
α (PER)	0.5
β (PER)	$0.4 \rightarrow 1$
PER buffer size	5×10^5

Table 1: Hyperparameters used in training. Training frequency refers to the frequency of one gradient descent step relative to the number of environment steps taken.

B Hyperparameters for TD3

Hyperparameter	**Value**
Training steps N	2,000,000
Learning starts	10,000
Policy Architecture	256×256
Critic Architecture	[(256, 256); (512, 512); (512, 512, 256)]
Discount factor γ	0.99
Polyak parameter τ	5×10^{-3}
Batch size B	256
Target policy noise σ	0.2
Target noise clip σ	0.5
Policy delay d	2
Learning rate critics	$0,0007$
Buffer size	1,000,000

Table 2: Hyperparameter settings in experiments.

Hyperparameter	Value
Batch size	512
Buffer size	1000000
Learning rate	1×10^{-3}
Optimizer	Adam
Polyak parameter τ	0.005
Discount factor γ	0.95
Hidden layers	[256, 256]
Learning starts	10000
Training frequency	4

Table 3: TD3 Hyperparameters - Pink Noise Test

C Hyperparameters for SAC

Hyperparameter	Value
Replay buffer size	10^6
Entropy coefficient	Automatic tuning
Initial entropy coefficient	1
Actor hidden sizes	$(512, 512, 256)$
Critic hidden sizes	$(2048, 2048)$
Batch size	256
Gamma	0.99
Optimizer	Adam
Adam optimizer β	$(0.5, 0.999)$
Learning rate	10^{-3}
Policy delay	3
Target networks	Removed
BN epsilon	10^{-5}
BN momentum	0.99

Table 4: Hyperparameters used for training the SAC algorithm.