RL-Course 2024/25: Final Project Report

fast_and_FOURIERous: Lennart Goldscheider, Felix Schlechte

March 5, 2025

1 Introduction

Reinforcement Learning (RL) describes the problem of learning how to act in a certain *environment* by maximizing a numerical *reward* [6, p. 1]. The decision maker is called *agent* and it is a function that maps a *state* $s_t \in \mathcal{S}$ to an *action* $a_t \in \mathcal{A}(s_t)$, or to a distribution of probabilities for each action [6, p. 6]. Here, $\mathcal{S}$ refers to the state space of the environment and $\mathcal{A}(s_t)$ to the set of all possible actions in the state s_t. The way in which an actor behaves is also referred to as the actor's *policy* π [6, p. 6]. When an action is executed, the environment returns the next state s_{t+1} and a reward r_t that indicates how good the performed action was. The probability density for the next state given the current state and an action is denoted by $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. The transition probabilities induced by a policy π are denoted by ρ_π. Using the *discount factor* γ, the *discounted return* [6, p. 55] is defined as $G_t := \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. To specify how good the given state is in the long run if the policy π is followed [6, p. 6] the *value function* is defined as $V_\pi(S_t) := \mathbb{E}_\pi(G_t|S_t)$. Furthermore, the value of taking an action A_t in a state S_t and then following policy π is denoted by the *q-function* $Q_\pi(S_t, A_t) := \mathbb{E}_\pi(G_t|S_t, A_t)$ [6, p. 58]. Since the ground truth of these functions is almost never known, they are often approximated by artificial neural networks. This field is known as *deep reinforcement learning* [6, p. 475].

In this report, we present how we used deep reinforcement algorithms to train an agent on the *Hockey environment*. This environment is a two-player game with the goal of shooting a puck into the opponent's goal. The state space $\mathcal{S}$ consists of 18-dimensional arrays with continuous values between -1 and 1, representing values like, e.g. the position of the players and the puck. The action space is $\mathcal{A}(s) = [-1, 1]^4 \quad \forall s \in \mathcal{S}$ to control the x- and y-coordinate of the player, to rotate their shooting paddle, and whether to shoot.

We use Soft-Actor-Critic (SAC) [4] and variations of it such as SAC with a recurrent neural network (SAC-RNN), Quantile-Regression Soft-Actor-Critic (QR-SAC) [7], Hindsight Experience Replay (SAC-HER) [1] and Simplicity Bias (SAC-SimBa) [5]. Our code can be viewed here.

2 Methods

2.1 SAC and SAC-RNN

SAC is an off-policy algorithm, which seeks to maximize the reward while also tries to act "as random as possible", i.e., it tries to maximize the *entropy* as good as possible [4]. Thus, the standard objective,

that RL algorithms attempt to maximize $\sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)]$ changes to [4, eq. 1]

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))],$$

where $\mathcal{H}$ is the entropy term and α is a *temperature* parameter that controls the randomness of the policy (also *actor*). Note that the standard objective can be obtained if $\alpha \to 0$ [4]. By using the entropy term, the algorithm has more exploration that allows for faster learning, while still recognizing which states are not beneficial. Furthermore, it also captures a broader spectrum of near-optimal behaviour [4, sec. 3.2].

Let now $Q_\theta(s,a)$ be a parametrized q-function (functions as the *critic*) with parameters θ and $V_\psi(s)$ be the parametrized value function with parameters ψ. Also, let the target network parameters be denoted by $\bar{\theta}$ and $\bar{\psi}$. To use the reparametrization trick, let $a_t = f_\phi(\epsilon_t; s_t)$ be a neural network that returns an action for a given state s_t and noise ϵ_t and let $\pi_\phi(f_\phi(\epsilon_t; s_t)|s_t)$ be a policy similar to [4]. Then the parameters of the q-function can be optimized by minimizing the Bellman residual [4, eq. 7]

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}(Q_\theta(s_t,a_t) - r(s_t,a_t) - \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_{t+1})]\right]. \tag{1}$$

Since we have $V(s_t) = \mathbb{E}_{a_t\sim\pi}(Q(s_t,a_t) - \alpha\log\pi(a_t|s_t))$ by [4, eq. 3] there is no need for a parametrized value function. Lastly, the parameters of the policy can be updated by minimizing the objective [4, eq. 12]

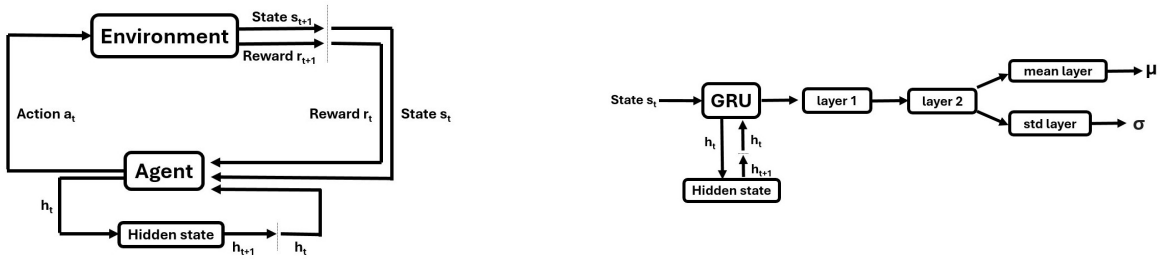
$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D},\epsilon_t\sim\mathcal{N}}[\log\pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t))], \tag{2}$$

which is equivalent to minimizing the expected KL-divergence [4].
Like it was proposed in [3], we also use two q-networks with parameters θ_i and take the minimum q-value to mitigate positive bias in the policy improvement step [4]. Also, for every training step, we perform a soft update of target q-network and q-network via $\theta \leftarrow (1-\tau)\theta_{target} + \tau\theta$, where τ is the *smoothing constant* and a hyperparameter.

After trying SAC in the Pendulum environment and the Hockey environment and getting promising results, we started to modify the policy network architecture by adding a RNN layer (see fig. 1b). We call this attempt **SAC-RNN**. The idea is that the policy can store information from previous time steps in a *hidden state* $h_t \in \mathbb{R}^{18}$ to be able to react to what happened earlier and draw better conclusions. An illustration of the interaction can be viewed in fig. 1a. We initialize h_0 with 0. For the RNN layer, we chose a GRU. With this adaptation, the state space expands to $\widehat{\mathcal{S}} = \mathbb{R}^{36}$ with one state $\widehat{s}_t \in \widehat{\mathcal{S}}$ being a concatenation of the actual state that the environment yields at time step t, $s_t \in \mathcal{S}$ and the hidden memory state h_t: $\widehat{s}_t = concatenate[s_t, h_t] := s_t \| h_t$.

2.2 QR-SAC

QR-SAC is an extension to SAC and the main adjustment is that it can handle N-step returns [7, p. 225]. Also, instead of predicting the q-value, the q-network returns a probability distribution of the expected value of the future rewards [7, p. 225]. The M single outputs of the q-network are referred to as the estimated quantile value Z_{τ_i} for the i-th *quantile* τ_i. In conclusion, the q-value can be computed as the mean of all estimated quantile values. M and N are two additional hyperparameters. While the objective for the policy network of SAC (eq. (2)) is the same for QR-SAC [7, eq. 3], the update for the q-networks



(a) agent-environment-hidden state interaction

(b) SAC-RNN architecture with a GRU layer before the linear layers.

Figure 1: SAC-RNN illustrations

needs to be adjusted. Analogous to SAC, we take the N-th state quantiles from the q-network that returns the smaller target q-value. So let $k = \arg\min_{m=1,2} Q(s_{t+N}, a'|\bar{\theta}_m)$. For τ_i we then get

$$y_i = R_t + Z_{\tau_i}(s_{t+N}, a'|\theta_k) - \alpha \log \pi(a'|s_{t+N}, \phi)$$

as the target function, where $R_t = \sum_{i=1}^{N} \gamma^{i-1} r_{t+1}$ is the N-step discounted return, $Z_{\tau_i}(s_{t+N}, a'|\theta_k)$ is the estimated quantile value at the N-th future state and $\log \pi(a'|s_{t+N}, \phi)$ is the SAC entropy term [7] similar to [4, eq. 3]. The update for the q-networks can be computed by minimizing the loss [7, eq. 2]

$$L(\theta) = \frac{1}{M^2} \sum_i \sum_j \mathbb{E}_{(s_t, a_t, R_t, s_{t+1}) \sim \mathcal{D}, a' \sim \pi} \rho(y_i - Z_{\tau_j}(s_t, a_t|\theta)),$$

where $\rho(u)$ is the quantile Huber loss function ρ^κ [2, eq. 10] with $\kappa = 1$.
In addition to the adjustments on the loss functions, we had to modify the replay buffer. Instead of saving every transition independently, it stores the whole trajectory of one epoch. When sampling a batch for training, we divide the sampled epochs into N-step trajectories. That is why in contrast to our implementation of SAC the epoch rollout is now decoupled from the update steps. This adds another hyperparameter: the number of update steps per rollout.

2.3 SAC-HER

At the beginning, we noticed that the agent avoided the puck due to a lack of exploration. This led to less exploration and seldom shooting a goal. Since the rewards at the start were "closeness to the puck" and the rewards at the end of an epoch, i.e. -10 for a loss, 0 for a draw and 10 for a win, a lot of the transitions in the memory had rewards, which caused suboptimal training.

To counteract this problem, we implemented Hindsight Experience Replay (HER) [1]. With this method, the agent could learn more from epochs where he did not shoot a goal. The approach is based on Universal Value Function Approximators [1, sec 3.2], which introduce goals to the agent that influence the policy and the q-function. Concretely, we define goals $g \in \mathcal{G}$ and predicates $f_g(s) = [s = g]$. In our case, $\mathcal{S} = \mathcal{G}$ and we want the agent to find $s \in \mathcal{S}$ with $f_g(s) = 1$. The reward at time t is then defined as $r(s_t, a_t) := r_g(s_t, a_t) = -[f_g(s_t) = 0]$. The problem is that the rewards are still sparse, since they are either 0 if $s = g$ or 1 if $s \neq g$.
HER solves that problem by first storing the original transition with the goal and then the transition with alternative goals and adjusted rewards. The agent is thus rewarded for achieving the alternative goals and learns faster. The hyperparameter k controls the ratio of real transitions to the artificially created

ones. There are 3 strategies to choose k random states as alternative goals: The future strategy takes states after the current state, the episode strategy takes any states from the same epoch, and the random strategy takes any states from any epoch. The last strategy is called the final strategy, where the last state that was reached in that epoch is the goal.

We implemented **SAC-HER** based on this code. We chose $\mathcal{G} = \{[3.7, 0]\}$, so the agent's goal should be to shoot a goal. The input of the actor and critic changes from s to $s||g$ which changes the observation space dimension of SAC from 18 to 20. Additionally, we changed the SAC code to not store single transitions, but the whole trajectory from one epoch. In the SAC update function, we then sample from the memory. For this, random epochs are sampled from the memory, and random time steps from each epoch are chosen to adjust the goal at that time step with a certain probability.
In this environment, the future strategy worked best. Thus, we take into consideration all future states that are beneficial for us, i.e. near the opponent's goal. We randomly select one of those states as the alternative goal if we reach a good state; otherwise, we don't adjust the goal. The adjusted reward is defined as 10 if the Chebyshev distance of the state to its goal is less than 1 and $-$Distance else.
Although [1, sec 4.4] states that shaping the reward function hinders the improvement of using HER, we actually saw a better performance. We included the rewards "touch puck" and "puck direction" and changed the end of epoch rewards to -1 for a loss, -1 for a draw and 1 for a win. Additionally, we scaled all rewards with 100 for better numerical stability. This resulted in more stable training and faster learning, likely because these changes encourage exploration rather than discouraging it.

2.4 SAC-SimBa

Another problem we ran into was that the agent overfitted against the opponents. To counter this, we implemented SimBa [5]. This architecture allows to scale up parameters in a neural network and also induce a simplicity bias, which enables neural networks to converge to a simple but often good, generalizable solution. This is achieved by using ReLU activation functions, the RSNorm (Running Statistics Normalization), a Residual Feedforward Block and Post-Layer Normalization [5, sec. 4].
The RSNorm uses a running mean μ_t and variance σ_t to normalize the state $\text{RSNorm}(s_t) = \frac{s_t - \mu_t}{\sqrt{\sigma_t^2 + \epsilon}}$

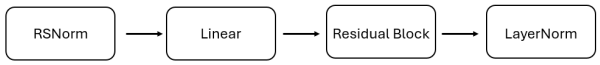


Figure 2: SimBa architecture

We first started by converting the original code from using Jax to PyTorch. We also did minor adjustments to match the SACActor methods to the plain SAC policy methods.

Surprisingly, the RSNorm and Post-Layer normalization didn't improve the performance of the agent. Because of this, we experimented whether the residual block is better than a normal MLP Block (see sec. 3). Because the MLP Block architecture performed best we continued with this architecture. This means, that the actor and critic have the same architecture as the actor and critic of plain SAC without the Layer Normalization (see app.A).
That means that **SAC-SimBa** just uses the SAC architecture (see app. A) and the initialization of the weights according to the SimBa architecture.

3 Evaluation

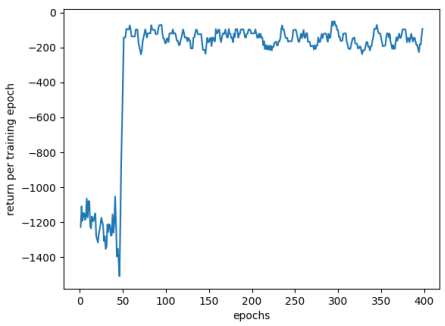
First, we trained SAC on the *Pendulum* environment, where the goal is to balance a pendulum on the tip. The agent gets penalized, i.e. getting bigger negative rewards, the further away the agent is from the terminal state. If it achieves the terminal state, the reward is 0. We used the hyperparameters from app. A and observed, that SAC solved the task fast (fig. 3a) since in the end the return of one training epoch is close to 0.

Then we trained SAC with the same hyperparameters and architectures on the hockey environment, but the agent did not learn that well. Since the original rewards from the hockey environment are in the $[-10, 10]$ range and the rewards from the Pendulum environment have the scale of $[-1000, 0]$, we improved the learning process by multiplying the hockey rewards by a factor of 100 and adding a layer normalization before the ReLU activation function. Also, we observed that directly training against the `weak_opponent` or the `strong_opponent` leads to no or poor training, since the agent looses almost every time. As a consequence, the agent has very few transitions to learn from. This is the reason why we almost always use curriculum learning in the following. So our first successful attempts on training SAC on the hockey environment were achieved by playing first against the random agent and after that against a pool of agents to avoid overfitting against one opponent: alternatingly against the `strong_opponent` and against itself. Every 50 epochs, the current agent plays 10 matches against the `strong_opponent` in inference mode as a test. The average return of that 10 testing epochs can be viewed in fig. 3b. You can also see here that the agent gets consistently a reward close to 1000 except for some outliers which are likely caused by the alternating of the opponents. A reward of 1000 indicates that the agent scores a goal since we multiply the default rewards by 100. For the sake of clarity, we smoothed our results (x_t) by using a moving average $s_t = \frac{1}{w}\sum_{i=0}^{w-1} x_{t-i}$ with window size w. Important to mention is that we just trained SAC as an early attempt and after it learned to maximize the reward we proceeded with the modifications. However, we used the resulting SAC agents as benchmarks and as opponents for training.

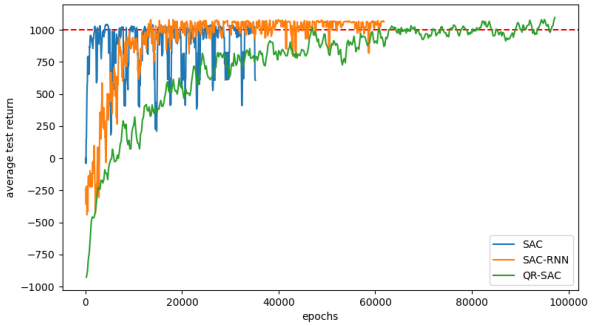
Similar to SAC, we first pretrained our SAC-RNN agent against the `weak_opponent` for 4000 epochs with the same hyperparameters as in app. A. After that we trained the agent against a pool of 4 opponents to have as much variance in the opponents as possible: Every 500 epochs we alternated between the `weak_opponent`, `stong_opponent`, a trained version of QR-SAC and an older version of the agent itself that gets updated every 5000 epochs. The resulting agent competed in the competition as our team agent. During the final training run we again recorded the average testing return against the `stong_opponent` which can be viewed in fig. 3b. Since the return is consistently close to 1000 and due to the absence of negative outliers, we concluded that using a greater mixture of opponents stabilizes training.

We also trained QR-SAC in various steps with parameters from app. A and the same architecture as for SAC. According to [7] we chose $N = 7$ and for the number of quantiles we chose $M = 16$. Since the rollout and the update steps are decoupled and since we chose a fixed number of update steps (70), the training manages to make more epochs in less time.
In the first run we trained against the random agent for 69000 epochs and in the following run against an old version of SAC and the `strong_opponent` alternating every 5000 epochs for 20000 epochs. For the final training run, we trained against the `strong_opponent`, two old versions of SAC and an older version of QR-SAC itself that is getting updated every 30000 epochs. These opponents alternated every 1000 epochs. After 90000 epochs, we got the agent that competed in the tournament. To monitor the ability to generalize, we adjusted the testing to be the average return of 20 matches against the 4 mentioned opponents that were used for training (5 matches each). The curve in fig. 3b shows that the agents

needs longer to beat all of the 4 opponents consistently, but given enough time, it also achieves this goal.



(a) Returns for every training epoch of SAC on Pendulum environment with no smoothing (w = 1)



(b) Average testing returns using smoothing with w = 5. SAC and SAC-RNN tested 10 epochs against the `strong_opponent` and QR-SAC 5 epochs each against a pool of 4 agents.

Figure 3: returns during training

Since a good reward just indicates that the agent can beat the chosen opponent, it doesn't tell anything about the real playing strength, or generalization to other agents. That's why we used a chess-like ELO rating system to compute, how good the agents really are. ELOs are initialized at 1500 and are updated after each match for player 1 and 2 using the expected points E_1 and E_2:

$$ELO_i^{new} = ELO_i^{old} + 20(points_i - E_i) \quad i \in \{1, 2\}, \tag{3}$$

with $E_1 = \left(1 + 10^{\frac{ELO_2 - ELO_1}{400}}\right)^{-1}$ and $E_2 = \left(1 + 10^{\frac{ELO_1 - ELO_2}{400}}\right)^{-1}$.

The training of SAC-HER and SAC-SimBa also used curriculum learning. SAC-HER played against the random agent, the `weak_agent`, the `strong_agent` and itself (self play). First, the agent played against The next opponent was chosen according to a win rate system. If they win at least 80% in a test with 100 epochs against the current opponent, they are moved up in the ranking and play against the next more difficult one. If they win less than 10% they are moved down in the ranking to play against easier opponents. To prevent overfitting, there is also a 10% chance to play against a randomly chosen agent if they won between 10% and 80% against the current opponent.
The training of SAC-SimBa was very similar, but it also had to play against 4 other checkpoints from other runs. Additionally, if the agent won at least 90% of the test games against one of these checkpoints, the win rate based system changed to an ELO ranking system. The next opponent was randomly chosen from opponents that were in a range of 300 ELO points around the current ELO points of the agent. If there is none, the next agent is chosen randomly. That is also the case in 10% of all cases. This system should encourage generalization by playing against different agents after each test.

Since the reward function for algorithms that use HER was changed in sec. 2.3, most of the rewards are negative. This is because the rewards "puck direction" and "closeness to puck" contribute the most to

the return at the end of an epoch. As a consequence, the agent tries to minimize the negative rewards, which means that he maximizes the return. This can be seen for SAC-HER as well as the 4 SAC-SimBa algorithms (see fig. 4). Important to note is that this reward function works well with the adjusted rewards by using HER. Thus, the reward function alone isn't the best way to measure the quality of the agents. Because of that, we also evaluate the performance based on an ELO rating system.

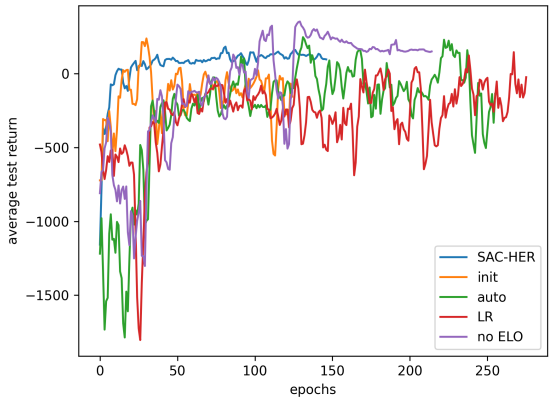


Figure 4: Average test returns using smoothing with w = 4 of SAC-HER and 4 different types of algorithms based on the SAC-SimBa architecture.

	SAC-HER	auto	LR	no ELO	init
ELO ratings	1530	1541	1515	1451	1461

Table 1: ELO ratings of different algorithm with different hyperparameters

Using the ELO rating system, the SimBa architecture without Residual Feedforward Blocks performs the best against the other possible architectures (see app. B). We then tried 3 different hyperparameter changes to see the effect of them and to see which algorithm performs best. These changes were an automatic entropy tuning (auto), a slower learning rate of 10^{-4} instead of 3×10^{-4} (LR) and a different initialization (init). Additionally, we tried one run without training based on the ELO rating system (no ELO). We found that the auto algorithm worked the best from among those, hinting that entropy tuning could be very effective in this environment (see tab. 1).We call this agent from now on SAC-SimBa.

We also conducted an evaluation of the best agents of every method against the `weak`− and `strong`−`opponent`. All agents are able to consistently beat the two base agents (see tab. 2).

	SAC-HER	SAC-SimBa	SAC-RNN	QR-SAC
`weak_opponent`	0.5 - 4.5 - 95	8 - 30 - 62	0 - 0.5 - 99.5	3.5 - 19.5 - 77
`strong_opponent`	1.5 - 0 - 98.5	7.5 - 32.5 - 60	0.5 - 0.5 - 99	1 - 4 - 95

Table 2: The win rates of our agents against the `weak_opponent` and `strong_opponent`. 200 games were played in each duel in total. The entry of one cell a - b - c means that the player in the respective column lost a % of the games and won c % of the games to the player in the respective row. b % of the games ended in a draw.

4 Discussion

To compare the playing strength during training of all of our implemented algorithms, we played a tournament and calculated the ELO points at different stages. For each algorithm, we chose checkpoints at

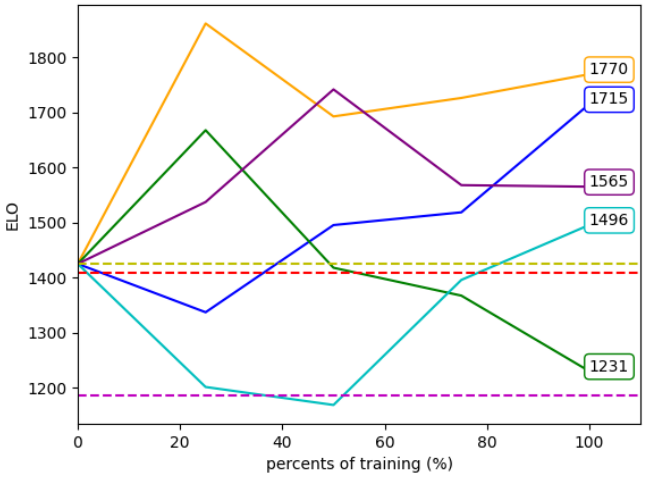


Figure 5: ELO evolution during training with SAC-HER (—), SAC-SimBa (—), SAC-RNN (—), QR-SAC (—), random agent (- - -), `weak_opponent` (- - -), SAC (—) and `strong_opponent` (- - -).

25, 50, 75 and 100 % of their training and played 100 epochs against every other agent, the `weak`−, `strong_opponent` and the random agent twice: one time on the left side and one time on the right side of the pitch. The evolution of ELO are shown in fig. 5 and the win rates of our agents against the `weak_opponent` and `strong_opponent` in tab. 2. The high ELO score of the random agent can be explained by the alternating start point of the puck, which leads to draws in almost 50% of the games. The lower ELO score of the `strong_opponent` in comparison to the `weak_opponent` is caused by more training against the `strong_opponent`. As an additional side node, the training of plain SAC is significantly shorter and less mature because it was only for early tests and to get benchmarks.

As we can see in fig. 5, our training didn't always improve the playing strength. We suspect that one reason for this could be our small pool of good opponents. With more opponents, better generalization may have been possible. Additionally, we also observed overfitting in some cases. A good example for a combination of both of the problems is QR-SAC. In tab. 2 you can see, that QR-SAC has a high win rate against the `strong_opponent`, but has a rather poor performance against the `weak_opponent`. The reason for this is that in the final training run, QR-SAC trained very long (about 3 days) against a pool of opponents that did not include the `weak_opponent`. Another example is SAC-SimBa, where the opposite is the case (see tab. 2 and fig. 5). However, if used right, the curriculum learning based on the win rate led to a better generalization, as could be seen with SAC-HER.

Further improvement would also have been possible with longer training against also stronger opponents. Due to these time constraints, we also didn't see any significant improvement when using SAC-SimBa, which is why we ended up using the original actor-critic architecture of plain SAC, but with the initialization of SAC-SimBa.

The last big opportunity for enhancement is hyperparameter finetuning. For QR-SAC, the number of steps (N) and the number of quantiles (M) and for SAC-HER the number of adjusted epochs in HER (k) could influence the training. Considering the tournament we participated with the agents after 100 % of our training, but as fig. 5 shows, it would have been better to take the real best agents.

All in all, with the given results from sec. 3 we can conclude that every algorithm was able to learn and maximize the rewards. It was interesting to see the different playing styles due to the different characteristics of the algorithms and the different opponents used in the training loops.

References

[1] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay, 2018.

[2] W. Dabney, M. Rowland, M. Bellemare, and R. Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

[4] 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.

[5] H. Lee, D. Hwang, D. Kim, H. Kim, J. J. Tai, K. Subramanian, P. R. Wurman, J. Choo, P. Stone, and T. Seno. Simba: Simplicity bias for scaling up parameters in deep reinforcement learning, 2024.

[6] S. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2020.

[7] P. R. Wurman, S. Barrett, K. Kawamoto, J. MacGlashan, K. Subramanian, T. J. Walsh, R. Capobianco, A. Devlic, F. Eckert, F. Fuchs, et al. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228, 2022.

Appendix

A Hyperparameters

	QR-SAC	SAC-RNN	SAC (hockey & pendulum)
policy	Gaussian	Gaussian	Gaussian
γ	0.99	0.99	0.99
τ	0.005	0.005	0.005
learning rate	0.0003	0.0003	0.0003
α	0.1	0.2	0.2
batch size	1024	265	265
number of update steps per epoch	70	length of epoch	length of epoch
target update interval	1	1	1
trajectory length (N)	7	-	-
number of quantiles (M)	16	-	-

Table 3: hyperparamters

QR-SAC architecture: (LN = layer normalization)
policy: $\mu = linear_\mu(ReLU(linear_2(ReLU(LN(linear(s_t)))))) \in \mathbb{R}^4$ with hidden dimension: 256
q-network: $Z = linear_3(ReLU(LN_2(linear_2(ReLU(LN_1(linear(s_t, a_t))))))) \in \mathbb{R}^M$ with hidden dimension 256.

SAC-RNN architecture:
policy: $(x, h_t) = GRU(s_t, h_t) \in \mathbb{R}^{18\times 18}$ and $\mu = linear_\mu(ReLU(linear_2(ReLU(LN(linear(x)))))) \in \mathbb{R}^4$ with hidden dimension: 256
q-network: $q = linear_3(ReLU(LN_2(linear_2(ReLU(LN_1(linear(s_t, h_t, a_t))))))) \in \mathbb{R}^1$ with hidden dimension 256.

SAC (hockey) architecture:
policy: $\mu = linear_\mu(ReLU(linear_2(ReLU(LN(linear(s_t)))))) \in \mathbb{R}^4$ with hidden dimension: 256
q-network: $q = linear_3(ReLU(LN_2(linear_2(ReLU(LN_1(linear(s_t, a_t))))))) \in \mathbb{R}^1$ with hidden dimension 256.

SAC (pendulum) architecture:
policy: $\mu = linear_\mu(ReLU(linear_2(ReLU(linear(s_t))))) \in \mathbb{R}^4$ with hidden dimension: 256
q-network: $q = linear_3(ReLU(linear_2(ReLU(linear(s_t, a_t))))) \in \mathbb{R}^1$ with hidden dimension 256.

B Architectures

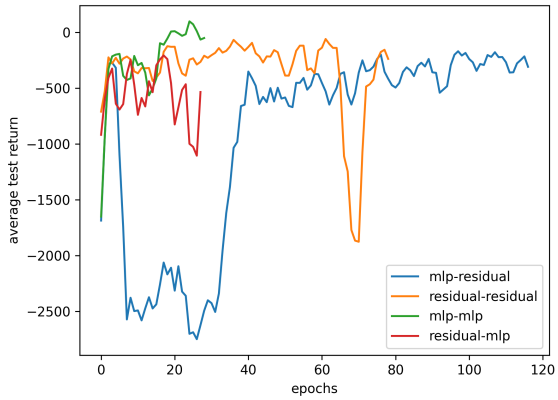


Figure 6: Average test returns of different SimBa architectures using smoothing with w = 3. Notation: Actor-Critic blocks in neural network.

This figure shows, that using no Residual Feedforward Block in the architecture, achieved the highest reward.

C Statement of Contributions

Felix was mainly responsible for the implementation of SAC, SAC-RNN and QR-SAC. Lennart was mainly responsible for the implementation of SAC-HER and SAC-SimBa. All members contributed equally to writing the rest of the report.