RL-Course 2024/25: Final Project Report

The Best Team EveRL: Cyprian Bohojlo, 6633499

1 Introduction

Can Reinforcement Learning agents find a optimal solution for hockey games? In this report, I introduce the answer for the questions and present my solution for the Hockey game challenge, which was developed as part of the final project for the Reinforcement Learning course at the University of Tübingen.

The project was completed by one person. The Hockey environment is implemented using the Gymnasium API (formerly Open AI Gym, https://gymnasium.farama. org/, which features a custom environment developed by the Martius Lab. The environment simulates a two-player hockey game in which agents compete to score goals against each other. It presents numerous challenges, including continuous state spaces, complex dynamics, and the need for both defensive and offensive strategies.

2 Soft Actor-Critic

In this section, the algorithm of Soft-Actor Critic (SAC) will be discussed along with its modifications. As one of powerful off-policy methods, Actor-Critic method performs well for continuous control that maximizes both the expected return and policy entropy, which is the key novel component of the SAC algorithm. The joint effort of the actor who learns the optimal policy and of the critic who learns the value of Q-function ensures that the agent is able to maximize its expected reward. In the following subsections we will discuss basic equations that build up the algorithm and then discuss the modifications that were introduced in this final project.

By combining these elements, SAC achieves robust exploration, stable learning via a replay buffer, and reduced overestimation bias using double Q-networks.

The full code can be found in this Github repository: Cyprian Bohojlo. *Reinforcement Learning Final Project*. Available at: GitHub Repository link (please click).

2.1 The Algorithm

Reinforcement Learning methods introduced in the lecture rest on the idea of Markov Decision Processes (MDPs) that were introduced in the early stages of the semester. In the case of this algorithm Markov Decision Process can be defined by the 5-tuple

$$(\mathcal{S},\, \mathcal{A},\, p,\, r,\, \gamma),$$

where $\mathcal{S}$ is the **state space**, $\mathcal{A}$ is the **action space**, $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$ is the **transition probability function** , $r(\mathbf{s}_t, \mathbf{a}_t)$ is the **reward function**, $\gamma \in [0, 1]$ is the **discount factor**.

In the standard Reinforcement Learning setting, an agent interacts with an environment over discrete time steps. At **time** t, the agent observes a **state** $\mathbf{s}_t \in \mathcal{S}$, chooses an **action** $\mathbf{a}_t \in \mathcal{A}$ based on **policy** $\pi(\mathbf{a}_t \mid \mathbf{s}_t)$, and receives a **reward** for a given **action** taken by my agent $r(\mathbf{s}_t, \mathbf{a}_t)$. The whole process then transitions to $\mathbf{s}_{t+1}$, and the agent's objective is to maximize the long-term, discounted sum of rewards. In that way we try to optimize the behavior of the agent in the certain enviiroment. The goal of the agent is to learn a policy that maximizes the discounted cumulative return.

2.1.1 Maximum-Entropy Objective

Many Reinforcement Learning methods in continuous action spaces face challenges, such as exploration-exploitation trade-off for example. Other methods discussed in our lecture, such as Deterministic Policy Gradient (DPG) can either suffer from high variance estimates or fail to explore sufficient portion of the action space. In order to address these issues, Soft Actor-Critic (SAC) introduced by Haarnoja et al. (2018)(1) incorporates two critical ideas: The first one: **Maximum-Entropy Objective:** The policy is optimized not only for reward but also for higher entropy, encouraging exploration. The second one: **Off-Policy Learning:** By reusing past transitions from a replay buffer.

Reinforcement Learning algorithms such as Soft-Actor Critic try to maximize the expected return,

$$\mathbb{E}\Big[\sum_{t=0}^{T-1} \gamma^t \, r(\mathbf{s}_t, \mathbf{a}_t)\Big],$$

where T can be a finite or infinite number of time steps. However, compared to other algorithms as DPG for example, Soft Actor-Critic provides an extension via adding an entropy term to encourage exploration and prevent premature convergence to overly deterministic strategies. To be more precise, we define the maximum-entropy objective:

$$J(\theta) \;=\; \sum_{t=0}^{T} \mathbb{E}_{s,a\sim\pi_\theta}\big[r(s_t, a_t) \;+\; \alpha\, H\big(\pi_\theta(\cdot \mid s_t)\big)\big]. \tag{1}$$

where $\alpha\, H\big(\pi_\theta(\cdot \mid s_t)\big)$ is the **entropy of the policy** at a given state, and α is a **temperature parameter** that controls for trades off between exploitation (reward maximization) and exploration. A higher α encourages more stochastic policies. Including an entropy term ensures that the agent remains uncertain about its action choices when many actions appear similarly good. This promotes broader exploration of the state-action space, leading to improved learning.

2.1.2 Soft Policy Evaluation

SAC modifies the standard Actor-Critic method by introducing a soft Q-function and a corresponding soft value function. The policy learned by the agent is off-policy, meaning it can reuse past transitions from a replay buffer defined in replay.py, which stores $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ tuples.

Following the lecture slides, we define the soft policy evaluation $q^\pi(s_t, a_t)$:

$$q^\pi(s_t, a_t) \;=\; r(s_t, a_t) \;+\; \gamma\, \mathbb{E}_{s'\sim\pi}\Big[\tilde{v}(s')\Big], \tag{2}$$

$$\tilde{v}(s) \;=\; \mathbb{E}_{a\sim\pi(s)}\Big[q^\pi(s_t, a_t) \;-\; \alpha\, \log \pi(a \mid s)\Big]. \tag{3}$$

Where $q^\pi(s_t, a_t)$ approximates the **expected return** plus an entropy correction for taking action $\mathbf{a}_t$ in state $\mathbf{s}_t$ and $\tilde{v}(s)$ is the **soft state value**, which includes an entropy penalty term $\alpha\, \log \pi(a|s)$.

In my implementation, two critics are learned in parallel in order to avoid value overestimation referred to by Haarnoja et al. (2018)(1) as Double Q-learning). The critic loss can be written as:

$$L_v(\theta) \;=\; \mathbb{E}_{s_t\sim D}\left[\frac{1}{2}\left(v_\psi(s_t) - \mathbb{E}_{a\sim\pi_\theta}\, \underbrace{\big[\hat{q}(s_t, a, w) - \alpha \log \pi_\theta(a \mid s_t)\big]}_{\text{off-policy}}\right)^2\right], \tag{4}$$

$$L_q(w) \;=\; \mathbb{E}_{(s_t, a_t, s_{t+1})\sim D}\left[\frac{1}{2}\left(\hat{q}(s_t, a_t, w) - \big(r(s_t, a_t) + \gamma v_\psi(s_{t+1})\big)\right)^2\right]. \tag{5}$$

where ψ is the **target network** parameters.

2.1.3 Soft Policy Improvement

The next crucial element of the SAC algorithm is policy improvement step in which we aim to update the policy in a way that maximizes both expected return and entropy. Unlike in other reinforcement learning methods discussed in our lecture, which rely on deterministic policy updates, SAC performs stochastic policy improvement.

The policy update is performed with minimizing the Kullback-Leibler (KL) divergence between the current policy π_θ and an exponential of the Q-function, ensuring that the policy favors actions with high Q-values:

$$\pi' \;=\; \arg\min_{\theta} D_{\mathrm{KL}}\left(\pi_\theta(\cdot \mid s) \,\middle\|\, \frac{\exp(q^\pi(s,\cdot))}{Z^\pi(s)}\right), \tag{6}$$

where $Z^\pi(s)$ is a **normalization factor** that tries ensuring that the right-hand side of our formula is a valid probability distribution. This way it encourages the policy to assign higher probabilities to actions that get better Q-values while preserving sufficient randomness for exploration of the state-action space.

In order for the algorithm to efficiently update the policy, SAC minimizes the following objective:

$$L_\pi(\theta) \;=\; \mathbb{E}_{s_t \sim D, \epsilon_t \sim \mathcal{N}}\left[\log \pi_\theta(s_t;\epsilon_t) - \underbrace{\hat{q}(s_t, \pi_\theta(s_t;\epsilon_t), w)}_{\text{DDPG loss}}\right]. \tag{7}$$

This specific loss function consists of two terms. The first one is **Entropy Maximization** $\log \pi_\theta(s_t;\epsilon_t)$,which encourages the policy to remain stochastic, ensuring enough level of exploration. The second term is the **Expected Q-Value Maximization** $\hat{q}(s_t, \pi_\theta(s_t;\epsilon_t), w)$, which guides the policy toward selecting high-reward actions, learned by the critic.

Another important component is the **Reparameterization Trick**, which allows direct optimization through stochastic gradient descent, SAC applies the reparameterization trick, where actions are sampled as:

$$\pi_\theta(s; \epsilon \sim \mathcal{N}(0,1)) = \mu_\theta(s) + \epsilon\sigma_\theta(s)$$

This should allow our gradients to flow directly into the policy parameters θ, improving stability and convergence speed of the algorithm.

By combining off-policy updates, entropy regularization, and stochastic gradient optimization, the algorithm ensures that the learned policy is both effective in maximizing long-term cumulative rewards and more efficient in dealing with exploration.

2.1.4 Adaptive temperature α

In some implementation, the key parameter to tune is **temperature** α, which normally is fixed. However, for the purposes of our course my implementation of SAC (in `sac.py` and `train.py`) will use **adaptive temperature tuning** described in Haarnoja et al. (2018)(1) meaning that α is learned automatically and is not fixed across episodes. The Haarnoja et al. (2018)(1) writes the objective in the folllowing way:

$$J(\alpha) \;=\; \mathbb{E}_{\mathbf{s} \sim d_\pi,\, \mathbf{a} \sim \pi_\theta(\cdot|\mathbf{s})}\Big[-\alpha\big(\log \pi_\theta(\mathbf{a}|\mathbf{s}) \;+\; \bar{\mathcal{H}}\big)\Big].$$

$$\nabla_\alpha J(\alpha) \;=\; \mathbb{E}_{(s,a)\,\sim\, d_\pi}\Big[-\big(\log \pi_\theta(a \mid s) + \bar{\mathcal{H}}\big)\Big].$$

Here, $\bar{\mathcal{H}}$ represents a **target entropy** that decides how stochastic the policy should be. By taking the gradient with respect to α, we push α to increase when the policy entropy is below $\bar{\mathcal{H}}$ and decrease otherwise. This should encourage the algorithm to automatically balance between more exploration and reducing unnecessary randomness.

When looking at implementation details, a key component is the `target_entropy` hyperparameter in `train.py` defined when creating an agent, it indicates whether automatic temperature tuning is enabled. If `target_entropy` = `-action_dim` then it's enabled and we create an `alpha_optimizer` and a trainable parameter `log_alpha`. If `target_entropy` = `None` then it's disabled. In case of using adaptive α tuning, at each update step, we compute a gradient for $\log \alpha$ based on how far the policy's entropy deviates from $\bar{\mathcal{H}}$. In practice, if the policy is too deterministic (entropy too low), α is increased, and if the opposite then decreased. The temperature at any point is $\alpha = \exp(\texttt{log\_alpha})$. This value is then used to assess the weight of the entropy term $\alpha\ \log \pi_\phi(a \mid s)$ in both the actor and critic updates.

By adapting α, SAC can automatically trade off exploration versus exploitation, removing the need for tedious and very difficult temperature tuning by hand.

2.1.5 Adaptive Learning Rate

In addition to adaptive α tuning, my code implements per-episode **adaptive learning rate** schedule via a simple reinforcement learning scheme inspired by Xu et al. (2019) (2). At the start of each episode, the algorithm constructs a controller that returns a state and passes it into a policy network that rescales the current learning rate. In the paper the authors do not specify the update rule formula, instead they describe it, however, this could be written as :

$$\eta_{t+1} \;=\; \max\Big(\eta_t \times a_t,\, 10^{-8}\Big).$$

where a_t is the **scaling factor** as described in the paper. We also clip the result to avoid small or negative values of η_{t+1}. This is necessary as in early versions of the code the adaptive learning rate would reach values that would lead to no learning over time.

Furthermore, we can see that in `train.py`, in each episode in training We define a 5-dimensional vector: $(\eta_{t+1},\, L_{\text{critic1}},\, L_{\text{critic2}},\, L_{\text{value}},\, L_{\text{actor}})$, representing the old learning rate and the average losses from the previous episode. Also, the controller network outputs a scaling factor that multiplies the current learning rate as indicated in the equation above. This is sampled from a small Gaussian around the network's output, allowing log-prob computation for REINFORCE. After the episode ends, we compute the sum of average losses, so that *minimizing* these losses gives us a positive reward signal.

One significant point to be made is that unlike in the Xu et al. (2019) (2), here I implemented the REINFORCE update instead of PPO. This is a simplification that should still lead to increased performance of the model without increasing the complexity to much. To train the controller, we accumulate the log-probabilities of each action and the rewards. In code, this appears as `loss += -log prob * r` , which is then backpropagated to update the controller's parameters via Adam.

2.1.6 Neural Network Architectures

The implementation of SAC includes few neural network architectures: the Actor, Critic networks, Value networks, and the Adaptive Learning Rate Controller network.

Each of these networks follows a similar structure with two hidden layers of 256 neurons (for example, input size, 256, 256, output size), using L2 loss (weight decay) as indicated in n Haarnoja et al. (2018)(1), and ReLU activation functions, which are the most popular in Deep Learning. Also, the most popular optimizer Adam is used for all networks. In order to improve numerical stability, I introduced several modifications such as clipping the the log standard deviation in Actor network, and using two Critic networks to avoid overestimation.

3 The Results and Discussion

In general all agents trained by my SAC implementation were set to 20,000 max episodes and 500 max steps in each episode. Also, after training the agent for each episode, we then test it on 10 evaluation episodes without learning where we collect the evaluation rewards and record the number of wins, draws, and losses against the weak opponent in order to assess how our agent is performs. As indicated in the instructions they were trained on the weak opponent mode that is defined in `training sac agent.py`.

The following variations of the SAC agent were ran:

- SAC with adaptive α parameter trained on normal mode
- SAC with adaptive α parameter trained on weak opponent mode
- SAC with fixed α parameter of 0.05 trained on normal mode
- SAC with fixed α parameter of 0.05 trained on weak opponent mode

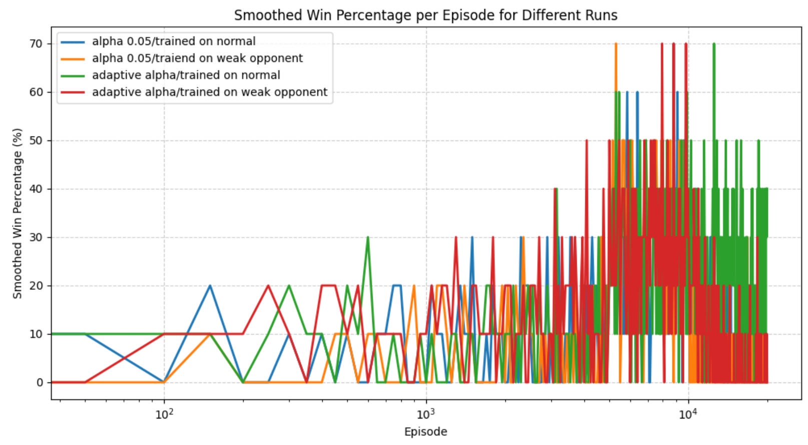

Figure 1: Smoothed Rewards Plot

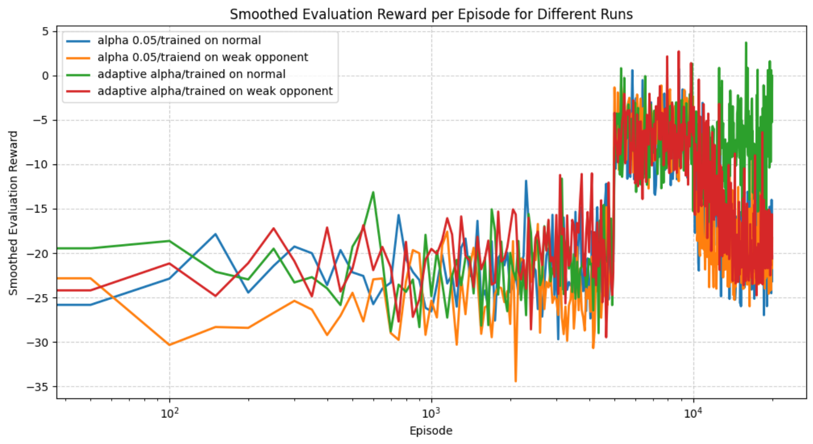

Figure 2: Smoothed Rewards Plot

In the plots above we can see that the learning algorithm for some reason is unstable, this could be caused by the fact that the implementation of adaptive learning rate approach using REINFORCE update instead of PPO makes the algorithm to update the learning rate in a way that causes divergence. Also, another reason could be the system of changing the playing modes between normal, shooting, and defense automatically introduces numerical instability during the training and confuses the agent in the learning process. However, it is worth noting that in some instances the SAC implementation with adaptive alpha tuning was able to achieve a 70% winning rate both trained on normal mode and weak opponent mode. An **important point** here is that the code is set to save both the final model in the last episode as well as the **best model** during the training. This allows to extract a specific model that was able to achieve around 70% win rate. Also, when looking at the Rewards plot one can notice that the models with adaptive α tuning tend to achieve higher rewards over time than models with fixed α.

References

[1] T. Haarnoja, A. Zhou, K. Hartikainen, et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.* In *ICML*, 2018.

[2] Z. Xu, A. M. Dai, J. Kemp, and L. Metz. *Learning an Adaptive Learning Rate Schedule.* Google Research, 2019. Available at: arXiv:1909.09712.

[3] Cyprian Bohojlo. *Reinforcement Learning Final Project*. Available at: https://github.com/CyprianBohojlo/Reinforcement-Learning-final-project/tree/main.

[4] G. Martius.*Reinforcement Learning - Lecture 8*.University of Tübingen, 2024.

[5] G. Martius.*Reinforcement Learning - Lecture 9*.University of Tübingen, 2024.