RL-Course: Final Project Report

YUGO (JYГO): Zafir Stojanovski, Dimitrije Antic, Jovan Cicvaric

March 17, 2021

1 Introduction

In the following report we describe our solution for the Laser Hockey challenge as part of the final project of the lecture course Reinforcement Learning offered at the University of Tuebingen.

Laser Hockey is a custom environment built using the Open AI gym (https://gym.openai.com). The environment is essentially a two player hockey game, in which the agents compete to score a goal against each other. Although seemingly simple, the environment encapsulates a lot of complexities and hardships under the hood.

Is reinforcement learning truly needed to find an optimal policy for playing the game? The answer to this question is in fact revealed by our project solution. In short, yes. We demonstrate that our trained reinforcement learning agents easily manage to defeat the algorithmic basic opponent provided by the environment.

We present both discrete and continuous action-space solutions for this problem. In particular, these are the algorithms that each of the authors have implemented:

1. **Dueling DQN** (Zafir Stojanovski)
2. **Soft Actor-Critic** (Dimitrije Antic)
3. **Deep Deterministic Policy Gradient** (Jovan Cicvaric)

In the rest of the report, we investigate the results of each solution in the order presented above. The code implementation is available at our Github repository. Note that we are using a customized version of the environment, where we colorize some outputs and suppress others, which can be installed as a pip package: pip install git+https://github.com/antic11d/laser-hockey-env.git.

2 Dueling DQN

In this section we overview the theory and performance results of the Dueling DQN [Wang et al., 2016] architecture.

2.1 Methods

We consider a sequential decision making setup, in which an agent interacts with an environment $\mathcal{E}$ over discrete time steps by choosing an action a_t from a discrete set $\mathcal{A} = \{1, ..., |\mathcal{A}\}|$, for which it observes a reward signal r_t. The goal of the agent is to maximize the expected discounted return $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards. For an agent behaving according to a stochastic policy π, the values of the state-action pair (s, a) and the state s are defined as follows:

$$Q^\pi(s,a) = \mathbb{E}[R_t|\mathbf{s}_t = s, a_t = a, \pi] \tag{1}$$
$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)] \tag{2}$$

The preceding state-action value function can be computed recursively with dynamic programming:

$$Q^\pi(s,a) = \mathbb{E}_{s'}[r + \gamma\mathbb{E}_{a'\sim\pi(s')}[Q^\pi(s',a')|s,a,\pi]] \tag{3}$$

Another important quantity relating the value and Q function is the advantage function:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{4}$$

The goal of the advantage function is to obtain a relative measure of importance for each action.

To translate these ideas into the world of continuous state spaces, we can use a deep Q-network: $Q(s, a; \theta)$ with parameters θ, leading us to the **Deep Q-Learning** [Mnih et al., 2015] algorithm. Generally, we optimize the following sequence of loss functions at iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y_i^{DQN} - Q(s,a;\theta_i)^2], \text{ with} \tag{5}$$
$$y_i^{DQN} = r + \gamma max_{a'} Q(s',a';\theta^-) \tag{6}$$

where θ^- represents the parameters of a fixed and separate target network. We could attempt to use standard Q-learning to learn the parameters of the network $Q(s, a; \theta)$ online, but this estimator usually performs poorly in practice. A key innovation was to freeze the parameters of the target network $Q(s', a'; \theta^-)$ for a fixed number of iterations while updating the online netwrok $Q(s, a; \theta_i)$ by gradient descent.

This approach is model free in the sense that the states and rewards are produced by the environment. It is also off-policy because these states and rewards are obtained with a behavior policy (epsilon greedy in DQN) different from the online policy that is being learned.

Another key ingredient behind the success of DQN is experience replay. During learning, the agent accumulates a dataset $\mathcal{D}_t = \{e_1, e_2, ..., e_t\}$ of experiences $e_t(s_t, a_t, r_t, s_{t+1})$ from which we sample minibatches uniformly to train the agent. Experience replay vastly reduces the variance as sampling from the buffer decreases the correlation among the samples used in the update. However, this approach simply replays transitions at the same frequency that they were originally experience, regardless of their significance. To overcome this, [Schaul et al., 2015] introduced **Prioritized Experience Replay**, where samples are sampled with probability proportional to the respective TD-error. In particular, the probability of sampling transition i is defined as:

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

with priority $p_i = |\delta_i| + \epsilon > 0$. The estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed). We can correct this bias by using importance-sampling (IS) weights

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \tag{7}$$

In a typical reinforcement learning scenario, the unbiased nature of the updates is most important near convergence at the end of training, as the process is highly non-stationary due to changing policies, state distributions and bootstrap targets. For this reason, it is common to anneal the amount of importance-sampling correction over time, by defining a (linear) schedule on the exponent β.

The **Double DQN** [van Hasselt et al., 2015] algorithm was introduced in order to mitigate the overoptimistic value estimates that the vanilla DQN makes. In particular, DDQN uses the following target:

$$y_i^{DDQN} = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta_i); \theta^-) \tag{8}$$

Finally, we arrive at the architecture of interest for this chapter - **Dueling DQN** [Wang et al., 2016]. The key insight behind this algorithm is that for many states, it is unnecessary to estimate the value of each action choice. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common backbone. Then, the two streams are combined via an aggregating layer to produce an estimate for Q.

Let us consider a dueling network, where we make one stream of fully-connected layers output a scalar $V(s;\theta,\beta)$, and the other stream output an $|\mathcal{A}|$-dimensional vector $A(s,a;\theta,\alpha)$. Here, θ denotes the parameters of the backbone layers, while α and β are the parameters of the two streams. Using this, we might be tempted to construct the following aggregating module:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{9}$$

However, this equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely, which results in poor practical performance. To address this issue, we can force the advantage function estimator to have 0 advantage at the chosen action. Then, the last module of the network implements the forward mapping

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - max_{a'} A(s,a';\theta,\alpha)) \tag{10}$$

An alternative module replaces the max operator with an average:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)) \tag{11}$$

On one hand, this loses the original semantics of V and A because they are now off-target by a constant, but on the other hand it increases the stability of the optimization: here the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage.

2.2 Experiments

In this section we will perform an ablation/sensitivity study, where we explore what effect certain hyperparameters and architecture choices have on the performance. The metric of interest is the percentage of won games on 1000 evaluation episodes. We perform an evaluation rollout after every 2000 training episodes in order to track the progress of each algorithm.

For the purpose of the task, we train a feed forward dueling architecture using prioritized experience replay, with the Huber loss function and the Adam optimizer with an initial learning rate of 0.0001. Moreover, we use three schedulers: a step scheduler that halves the learning rate at predefined checkpoints; a linear scheduler that anneals beta (for Prioritized Experience Replay) to 1; and a linear scheduler that decreases epsilon (for epsilon-greedy) to 0.1 in an attempt to mitigate the exploration-exploitation dilemma. We train the agent every 4 environment steps with a batch of 32 experiences from the replay buffer with a capacity for $5 * 10^5$ transitions. In total, there are well over 20 hyperparameters that can be tuned, which was not a trivial task. In the rest of this chapter, due to space constraints, we only cover a small portion of the intuitions gained in training the agent to beat the Weak Opponent.

In the first experiment we observe the importance of how often we update the target network (Figure 1). Updating too often (every 500 gradient updates) or too rarely (every 5000 gradient updates) causes the model to yield lower performance. The best solution updates the target network every 1000 gradient updates.

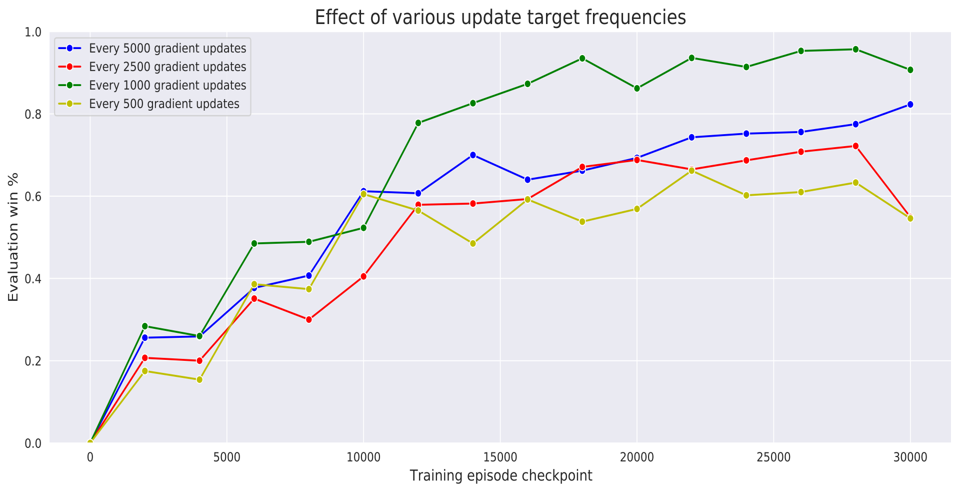

Figure 1: Performance difference caused by how often we update the target network

In the second experiment we observe how augmenting the reward from the environment affects the learning progress (Figure 2). In particular, we use the following reward in each step:

$$step\_reward = \underbrace{env\_reward}_{A} + \underbrace{5 * reward\_closeness}_{B} -$$
$$\underbrace{(1 - touched) * 0.01}_{C} + \underbrace{touched * first\_time\_touched * episode\_step\_number * 0.1}_{D}$$

The idea is apart from the base reward (A), we want to penalize the agent for not following the puck (B). Also, we penalize the agent if it doesn't touch the puck at all (C), with a chance to redeem itself if it does (D). Using the augmented reward, we observe faster convergence.

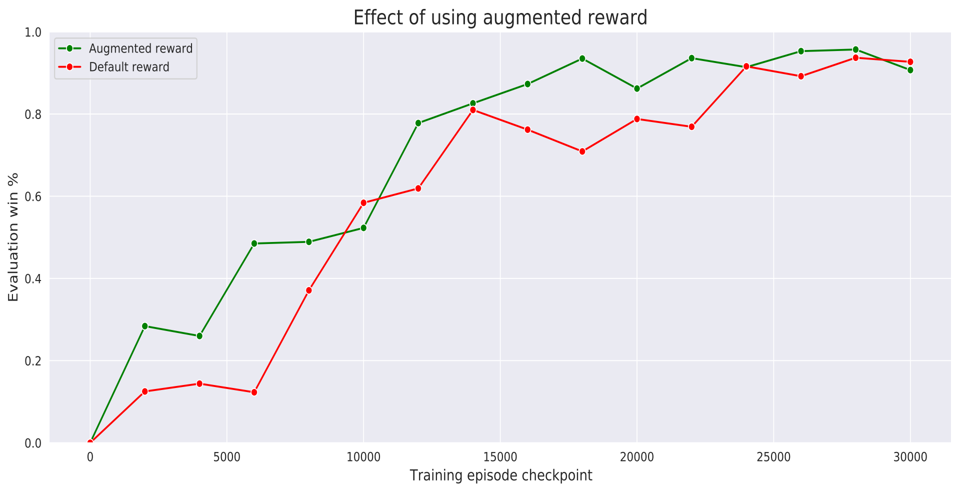

Figure 2: Performance difference caused by using the augmented reward

In the third experiment we look into how halving the learning rate affects the overall performance (Figure 3). It becomes obvious that by taking smaller steps towards the end, it is possible to converge towards the optimal solution. For this experiment, the first halving is performed after the 10000-th training episode, and the second halving is performed after the 18000-th training episode.

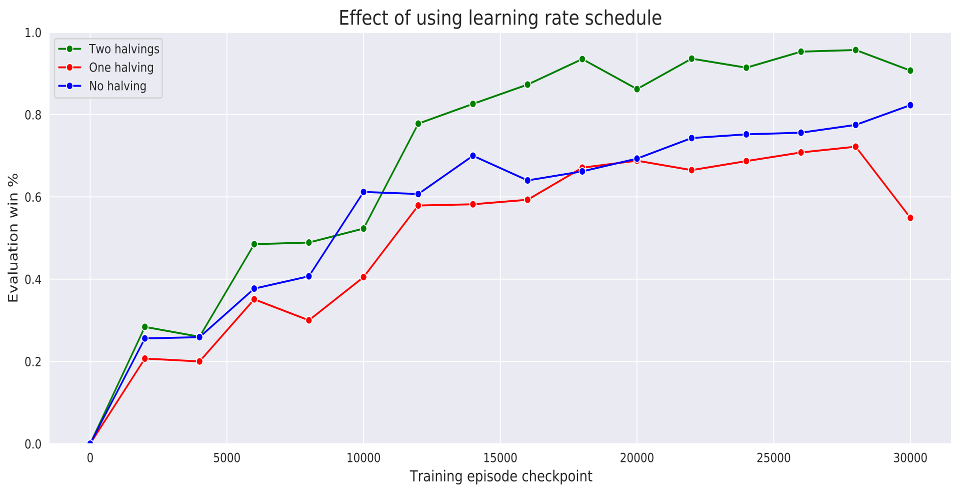

Figure 3: Performance difference caused by halving the learning rate

For the fourth experiment (Figure 4), we look into how augmenting the default action space affects performance. In particular, from the 8 base actions (left, right, up, down, ...), we create in total 20 combinations of these actions (ex. right-up, right-down, ...). From the figure, it becomes obvious that this algorithm converges to a better solution.

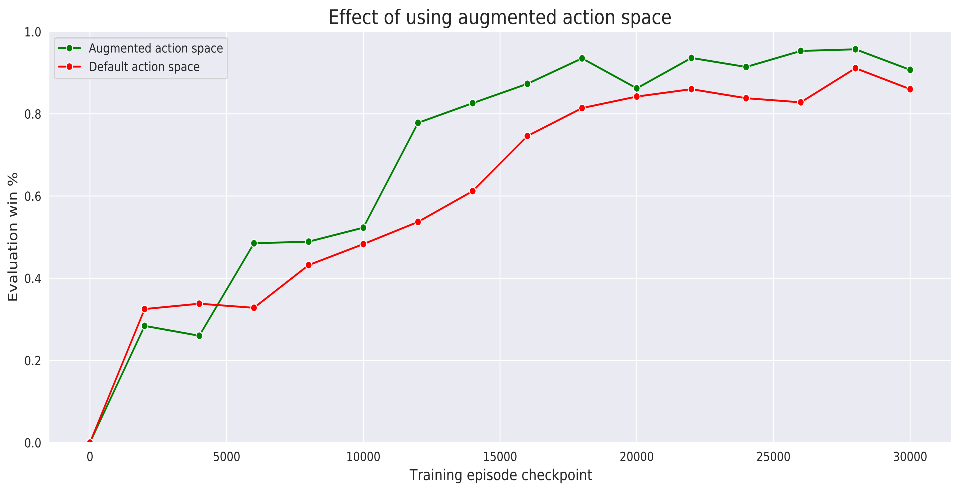

Figure 4: Performance difference caused by augmenting the action space

For the fifth experiment we look into how using the Double DQN target affects performance (Figure 5). Surprisingly, we did not notice any significant improvement. Following the Occam's razor principle, we omit using this particular configuration.

For the tournament mode, we trained the DQN agent with self-play. In particular, we train the agent in total for 90,000 episodes, out of which we train it on the weak and strong opponent in the first 50,000, and then for the next 40,000 we continuously add a copy of ourself to the list of opponents every 100,000 gradient steps. This resulted in **95%** performance on both the weak and strong opponent (not considering the most overfitted solution).

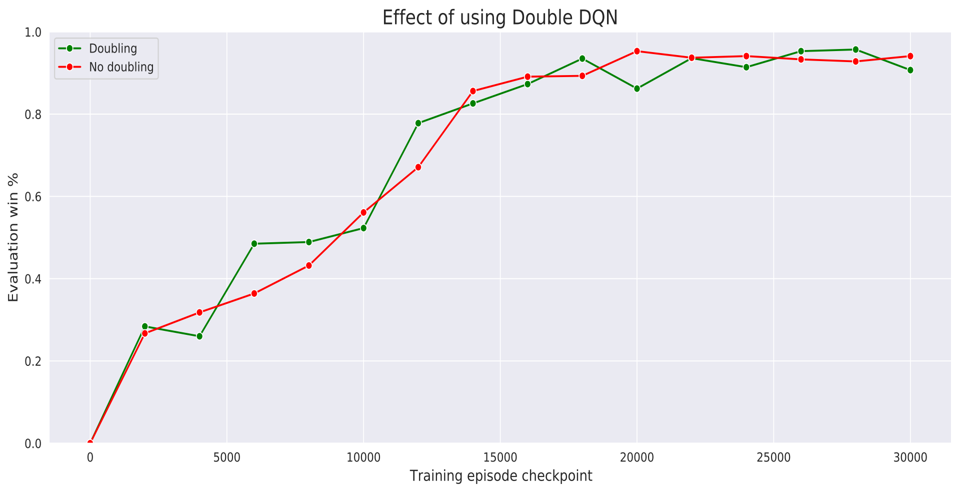

Figure 5: Performance difference caused by using the Double DQN target

3 Soft Actor-Critic

In the following subsections 3.1 the method Soft Actor-Critic [Haarnoja et al., 2018] and the performance results 3.2.2 versus environment's basic opponent.

3.1 Method

Soft Actor-Critic method is used to address learning of maximum entropy policies in continuous action spaces. The reinforcement learning problem can thus be defined as policy search in a Markov decision process (MDP). MDP is defined by a tuple $\mathcal{S}, \mathcal{A}, p, r$. The state space $\mathcal{S}$ and action space $\mathcal{A}$ are assumed to be continuous, and the state transition probability $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, \infty)$ represents the probability density of the next state $s_{t+1} \in \mathcal{S}$ given the current state $\mathbf{s}_t \in \mathcal{S}$ and action $\mathbf{a}_t \in \mathcal{A}$. The environment emits a reward $r : \mathcal{S} \times \mathcal{A} \rightarrow [r_{min}, r_{max}]$ on each transition. Also $\rho_\pi(\mathbf{s}_t)$ and $\rho_\pi(\mathbf{s}_t, \mathbf{a}_t)$ to denote the state and state-action marginals of the trajectory distribution induced by a policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$.

3.1.1 Maximum Entropy Reinforcement Learning

Maximum Entropy Reinforcement Learning aims to learn a policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$ that maximizes the objective:

$$\sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot|\mathbf{s}_t))] \tag{12}$$

Thus, desired policy is given by:

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot|\mathbf{s}_t))] \tag{13}$$

where $\mathcal{H}(\pi(\cdot|\mathbf{s}_t))$ is the entropy at each state, α is the temperature parameter that determines the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy. By augmenting the objective with an entropy term, the optimal policy aims to maximize its entropy at each visited state. Note that conventional objective can be recovered in the limit as $\alpha \rightarrow 0$. During the experiments, it was seen all the agents that were trained with temperature parameter dif-ferent than 0, learned policies were more "creative" and "adaptable" to different opponents strategies.

The maximum entropy objective gives us at least the following two advantages:

- the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues;
- the policy can capture multiple modes of near-optimal behavior. In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions.

3.1.2 Soft Policy Iteration

Soft Policy Iteration is a general algorithm for learning optimal maximum entropy policies that alternates between policy evaluation and policy improvement in the maximum entropy framework, and enables the deriving of the off-policy soft actor-critic algorithm. Thus, it can be verified that the corresponding algorithm converges to the optimal policy from its density class. Authors of the paper have proven the statements in [Haarnoja et al., 2018]. In the policy evaluation step of soft policy iteration, we wish to compute the value of a policy π according to the maximum entropy objective. For a fixed policy, the soft Q-value can be computed iteratively, starting from any function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and repeatedly applying a modified Bellman backup operator $\mathcal{T}^{\pi}$:

$$\mathcal{T}^{\pi} Q(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t_1} \sim p}[V(\mathbf{s}_{t+1})] \tag{14}$$

where

$$V(s_{t+1}) = \mathbb{E}_{\mathbf{a}_t \sim \pi}[Q(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log \pi(\mathbf{a}_t|\mathbf{s}_t)] \tag{15}$$

In the policy improvement step, we update the policy towards the exponential of the new soft Q-function. This update can be guaranteed to result in an improved policy in terms of its soft value. We restrict the policy to some set of policies Π, which can correspond, to a parameterized family of distributions such as Gaussians. To account for the constraint that $\pi \in \Pi$, we project the improved policy into the desired set of policies. For the sake of convenience, we use the information projection i.e. the Kullback-Leibler divergence. In the other words, in the policy improvement step, for each state, we update the policy according to:

$$\pi_{new} = \arg \min_{\pi' \in \Pi} D_{KL} \left(\pi'(\cdot|\mathbf{s}_t) || \frac{\exp(\frac{1}{\alpha} Q^{\pi_{old}}(\mathbf{s}_t, \cdot))}{Z^{\pi_{old}}(\mathbf{s}_t)} \right) \tag{16}$$

It is proven that this kind of soft policy improvement, and soft policy impropment leads to a policy π^* sucht that $Q^{\pi^*}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi}(\mathbf{s}_t, \mathbf{a}_t)$ for all $\pi \in \Pi$.

3.1.3 Soft Actor-Critic algorithm

For both the soft Q-function and the policy, function approximators will be used. We will consider a parameterized soft Q-function $Q_\theta(\mathbf{s}_t, \mathbf{a}_t)$ and a tractable policy $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$. The parameters of these networks are θ and ϕ. The soft Q-function can be modeled as neural network, and the policy as a Gaussian with mean and covariance given by neural network. We will next derive update rules for these parameter vectors. The soft Q-function parameters are trained to minimize:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - (r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p}[V_{\overline{\theta}}(\mathbf{s}_{t+1})]) \right] \tag{17}$$

where the value function is also parametrized through the soft Q-function via equation 15, and can be optimized by stochastic gradient algorithms:

$$\nabla_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - (r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p}[Q_{\overline{\theta}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha \log \pi(\mathbf{a}_t|\mathbf{s}_t)]) \right) \tag{18}$$

During the update, the target soft Q-function with params $\overline{\theta}$ is used. Parameters are obtained as an exponentially moving average of the soft Q-function weights, thich has been shown to stabilize training [Mnih et al., 2015]. Finally, policy parameters can be learned by directly minimizing the expected KL divergence in equation 16. Here, the target density is the Q-function, which is represented by a neural network, thus can be differentiated, and therefore the reparametrization trick is convenient to be applied. We reparametrize the policy using a neural net transformation: $\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t)$, where ϵ_t is an input noise vector sampled from a spherical Gaussian. Now the objective can be written as:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[\alpha \log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t) | \mathbf{s}_t) - Q_\theta(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t)) \right] \tag{19}$$

Therefore, the gradient can be approximated with:

$$\nabla_\phi \hat{J}_\pi(\phi) = \nabla_\phi \alpha \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) + (\nabla_{\mathbf{a}_t} \alpha \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)) \nabla_\phi(\epsilon_t; \mathbf{s}_t) \tag{20}$$

where $\mathbf{a}_t$ is evaluated at $f_\phi(\epsilon_t; \mathbf{s}_t)$

3.1.4 Automatic Entropy Adjustment

It can easily be seen that choosing the optimal temperature is non-trivial, and the temperature needs to be tuned for each task. Instead of requiring the fixed temperature value manually, we can automate this process by formulating a different maximum entropy reinforcement learning objective, where the entropy is treated as a constraint. Authors showed the derivation by soling the dual problem, but the proof and derivation will be omitted here due to space constraints, and only final solution will be presented. The dual variable α_t^* can be obtained, after solving the maximum entropy objective for Q_t^* and π_t^*:

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{\mathbf{a}_t \sim \pi_t^*} \left[-\alpha_t \log \pi_t^*(\mathbf{a}_t | \mathbf{s}_t; \alpha_t) - \alpha_t \mathcal{H} \right] \tag{21}$$

3.2 Experiments

In this section we will present the results of the ablation study where the effect of some of the parameters are explored. The main metric of interest is the percentage of won games out of 1000 evaluation episodes played against the weak basic opponent. The evaluation rollouts are performed after every 1000 training episodes.
Note that all the experiments were conducted before **07.03.2021.** and therefore used the version of the environment that was available at that particular moment.

3.2.1 Architecture

The architecture of the trained actor critic agent was as follows: two feedforward neural networks (actor and critic) with two hidden layers (ie. input_size, 256, 256, output_size) per network, and L2 loss (as suggested in [Haarnoja et al., 2018]. The parameters of both neural networks are optimized by the Adam optimizer [Kingma and Ba, 2014] with initial learning rate 0.0001. Also, for tuning the temperature parameter Adam is used, with initial learning rate 0.00001.
After each training rollout, 32 gradient update steps are performed, with the batch size of 128 transitions sampled from the experience replay buffer of the size 100000.
As mentioned in section 3.1 target critic network is used as a exponentially moving average, and for that averaging, constant $\tau = 0.005$.
All mentioned parameters were chosen by "manual" hyperparameter search.
Even if 10000 episodes were enough for convergence, each agent is trained for 20000 episodes.

3.2.2 Results

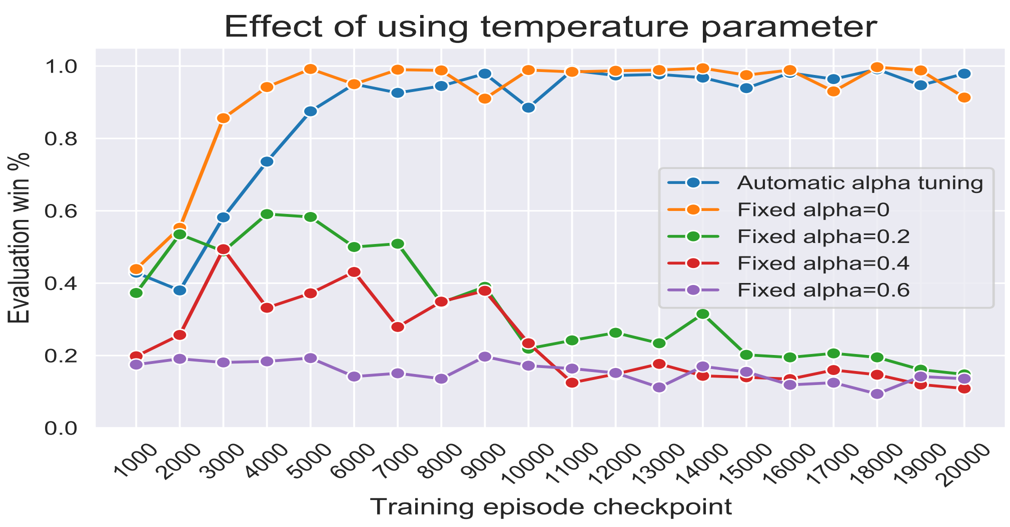

Figure 6: Performance difference caused by using the different values for temperature parameter

Fig. 6 shows the different results obtained with the different values for temperature parameter. Even though the agent with fixed alpha value 0 showed the fastest convergence, its strategy didn't perform well on strong version of the basic opponent. When the automatic alpha tuning is utilized, our agent converges a bit slower, but ends up winning all the games played versus both versions of the basic opponent, and shows interesting strategies in playing.

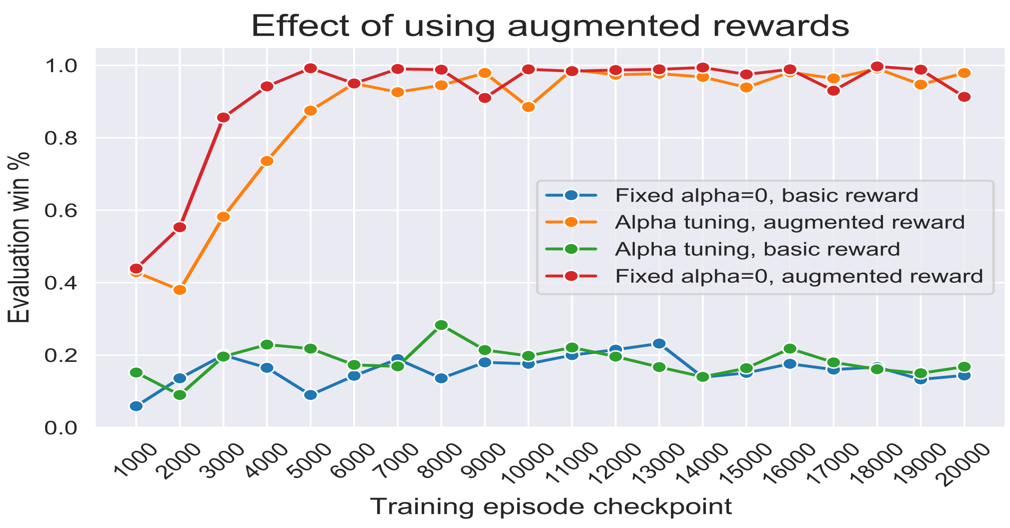

Figure 7: Performance difference caused by using the augmented reward

As the most important part in the fast convergence of the agents (see Fig. 7), I would single out the augmented reward that was used. It motivated agent to learn to pick up the ball in the early episodes of the training. The reward is defined in the following way:

```
step_reward = (
    reward                                    # Reward given by the environment
    + 5 * _info['reward_closeness_to_puck']   # Motivating agent to be close to puck
    - (1 - touched) * 0.1                     # Penalizing agent if he hasn't touched
    + touched * first_time_touch * 0.1 * step # Rewarding agent if he touched the puck
                                              # at any point during the rollout
)
```

4 Deep Deterministic Policy Gradient

In this section we will overview the theory and the performance of the DDPG architecture [Lillicrap et al., 2015] .

4.1 Methods

We consider a standard RL setup consisting of an agent interacting with an environment E in discrete timesteps. At each timestep t the agent receives an observation x_t, takes an action a_t and receives a scalar reward r_t. An agents behaviour is defined by policy π, which maps states to a probability distribution over actions. The return from the state is defined as the sum of discounted future rewards

$$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) \tag{22}$$

where $\gamma \in [0, 1]$. The return depends on the actions and therefore on our policy. The goal is to learn policy that maximizes the expected return from the start distribution:

$$J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1] \tag{23}$$

The action-value function describes the expected return after taking an action a_t in state s_t and then following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i \geq t} \sim E, a_{i \geq t} \sim \pi}[R_t | s_t, a_t] \tag{24}$$

In RL Bellman equation is widely used in many approaches:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_t \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi} Q^{\pi}(s_{t+1}, a_{t+1})] \tag{25}$$

If the target policy is deterministic we can describe it as a function $\mu : \mathbf{S} \rightarrow \mathbf{A}$ and avoid inner expectation:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_t \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))] \tag{26}$$

Now the expectation depends only on the environment and we can learn Q^{μ} off-policy, using transitions which are generated from a different stochastic behaviour policy β. Q-learning uses the greedy policy $\mu(s) = argmax_a Q(s, a)$. We consider function approximations parametrized by θ^Q, which we optimize by minimizing the loss:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^{\beta}, a_t \sim \beta, r)t \sim E}[(Q(s_t, a_t | \theta^Q) - y_t)^2] \tag{27}$$

where $y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1} | \theta^Q))$. In order to scale Q-learning algorithm we will use two major changes: replay buffer and a separate target network for calculating y_t. It's not possible to apply Q-learning straightforward to continuous action spaces, because greedy optimization for a_t is too slow. Instead, we use an actor-critic approach based on the DPG algorithm.
The DPG algorithm maintains a parameterized actor function $\mu(s | \theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s, a)$ is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to be the expected return from the start distribution J with the respect to the actor parameters:

$$\begin{aligned} \nabla_{\theta^{\mu}} J \approx & \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_{\theta^{\mu}} Q(s, a | \theta^Q)|_{s=s_t, a=\mu(s_t | \theta^{\mu})}] \\ = & \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_a Q(s, a | \theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^{\mu}} \mu(s_t | \theta^{\mu})|_{s=s_t}] \end{aligned} \tag{28}$$

Challenge when using neural networks for RL is that most optimization algorithms assume that the samples are i.i.d. When the samples are generated from exploring sequentially in an environment this assumption no longer holds. As in DQN we are using replay buffer to fight with this problem.

The replay buffer is a finite sized cache $\mathcal{R}$. Transitions are sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. When the replay buffer is full the oldest samples are deleted. At each timestep actor and critic are updated by sampling a minibatch uniformly from the buffer. For stability we create a copy of the actor and critic networks $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, that are used for calculating the target values. The weights of this networks are updated using the soft update:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{29}$$

with $\tau \ll 1$.
Because of this target values are constrained to update slowly and therefore improving the stability of learning.
One of the main challenges of learning in continuous spaces is exploration. In our implementation we have used decaying probability of an action being randomly sampled with uniform distribution. This approach proved to work well and because of declining probability agent is able to learn and at the same time continue exploration, but with much lesser probability than in the beginning of the learning process.
But DDPG as many other actor critic approaches has two major issues, which were addressed in [Fujimoto et al., 2018], those are - overestimation bias and variance. In the paper it was shown that value estimate will be overestimated and although this overestimation might be minimal, it may develop into more significant bias over updates and also may lead to a poor policy updates. To oppose this problem it was proposed using clipped Double Q-learning algorithm. By using the pair of critics and taking the minimum over the two value estimates. This may result in underestimation bias, but that's preferable since the value of underestimated actions will not be explicitly propagated through the policy update.
The second issue is variance. High variance estimates provide a noisy gradient for the policy update. This is known to reduce learning speed and worsen the performance. To fight with this problem delayed target update is proposed. By sufficiently delaying the policy updates we limit the likelihood of repeating updates with respect to an unchanged critic. The less frequent policy updates will use a value estimate with lower variance. A big concern with deterministic policies is that they can overfit to narrow peaks in the value estimate. The variance induced because a learning target using a deterministic policy is highly susceptible to inaccuracies can be reduced through regularization. The proposition is that fitting the value of a small area around the target function $y_t = r(s_t, a_t) + \mathbb{E}_\epsilon Q(s', \pi_{\theta\mu'}(s') + \epsilon|\theta^{Q'})$ would benefit from smoothing the value estimate by bootstrapping off of similar state-action value estimates. In practice we can modified target update that approximates expectation over actions by adding a small random noise to a target policy:

$$y_t = r(s_t, a_t) + \gamma Q(s', \mu_{\theta\mu'}(s') + \epsilon|\theta^{Q'}) \tag{30}$$

where $\epsilon \sim clip(\mathcal{N}(0,\sigma), -c, c)$. The main idea is that similar actions should have similar value.
This three modification combined were applied to DDPG to receive the TD3 algorithm, which was also implemented. So now our target policy is:

$$y = r + \gamma min_{i=1,2} Q(s', \mu_{\theta\mu'}(s') + \epsilon|\theta^{Q'_i}) \tag{31}$$

4.2 Experiments

In this section we will provide results of the ablation test that explore the effect of the certain hyper-parameters on the agents performance. The metric used in this section is the percentage of wins in 1000 evaluation games, so the draws are considered as losses. All the training and evaluation was done against weak opponent.
The architecture used for both actor and critic is a feedforward neural network with two hidden layers of size 256 (this architecture proved to be the best solution both for performance and speed). We are

using ReLU activation functions, in actor as a readout tanh was used, because possible actions lie between -1 and 1. Adam was used as optimizer with learning rate 0.0001. During every episode we are performing 30 gradient steps with the batch size 128.
First we're going to look at the effect of delayed target update. That's one of the principles that TD3 is using to improve its performance, so we should see improvement with the DDPG. We have also added a hard update every 1800 gradient steps to this plot.

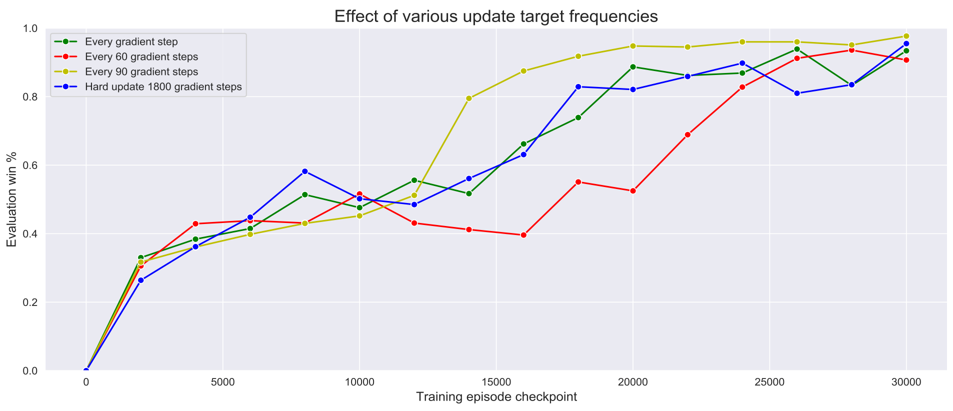

Figure 8: Performance difference caused by different frequency of a target network updates

From Fig. 8 we can see that all approaches to converge to approximately same level of wins, but one with target update every 90 steps does that much faster.
Next comes exploration of τ parameter used in a soft update.

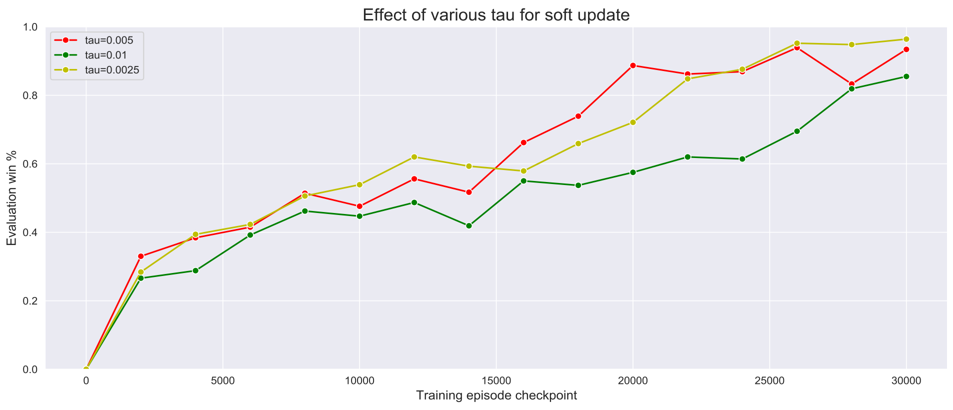

Figure 9: Performance difference caused by different taus in a soft update

From Fig. 9 we can see that there's no drastic difference in performance when using different taus, only the smaller ones show slightly better and more stable performance.
Next step was adding learning rate scheduler to our learning process. We will halve the learning rate first time after 10K episodes and the second time after 20K and look how that affects our results. From Fig. 10 we can see that halving learning rate doesn't improve results, quite contrary it slows convergence, the reason for that might be that we are using halving too early and thus don't allow agent to come close to the optimum. In our implementation we have been using a custom reward, from Fig. 11 it might look that there's no significant difference, but the basic reward is too sparse and it was much harder to get the good performance when using it, while augmented reward helped speed up the process of finding right hyperparameters.

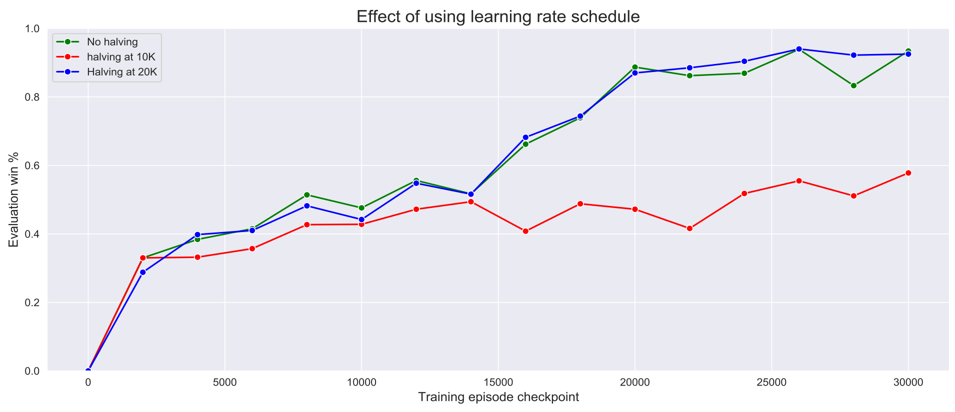

Figure 10: Performance difference caused by learning rate halving

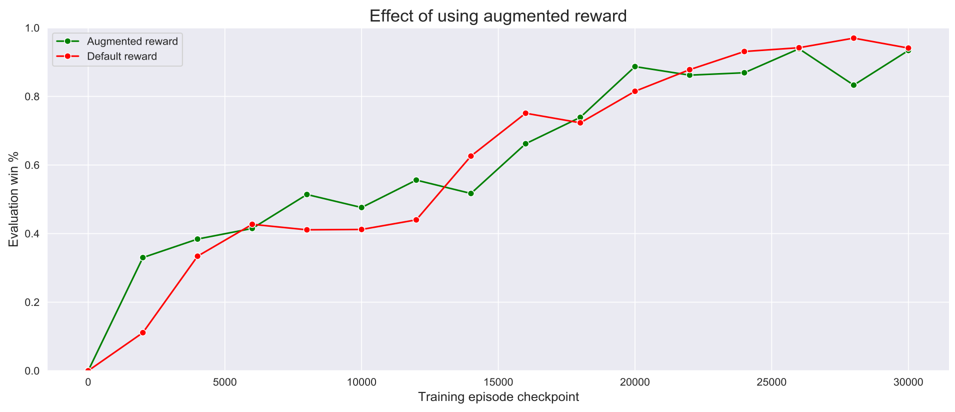

Figure 11: Default reward vs Augmented

Now lets compare the performance of DDPG versus TD3. It's important to note that the main subject of this research was DDPG algorithm, so possible under-performance of TD3 may be caused by not perfectly chosen hyperparameters. We can see that the performance of baseline TD3 is already slightly better than of the tuned DDPG and significantly better than the baseline DDPG.

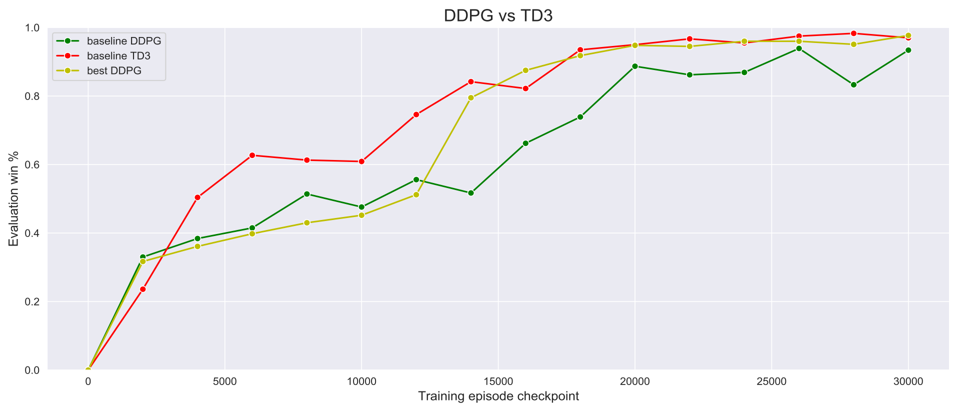

Figure 12: TD3 vs DDPG

5 Final overview

In this final section, we dive into a short overview of the performance results between the algorithms that we've implemented: Dueling DQN, Soft Actor-Critic, and Deep Deterministic Policy Gradient. It is important to note that these algorithms come with wide variety of traits, strengths, and weaknesses. The final results of the algorithms trained on the basic algorithmic opponent are summarized in Fig. 13.

First, let us note the most important achievement - all of the reinforcement learning algorithms managed to significantly win against the algorithmic rule-based basic opponent, which was the first goal of this team project.

Secondly, it is noticeable that the Soft Actor-Critic algorithm manages to converge to an optimal solution much faster than the other algorithms. It should also be said that this may be the consequence of the limited computation power, since having more resources may lead to finding better parameters for the algorithms.

However, we see that no matter whether it's a discrete or continuous, policy gradient or Q learning algorithm, at the end of the day all of them manage to converge to a relatively similar optimal solution.

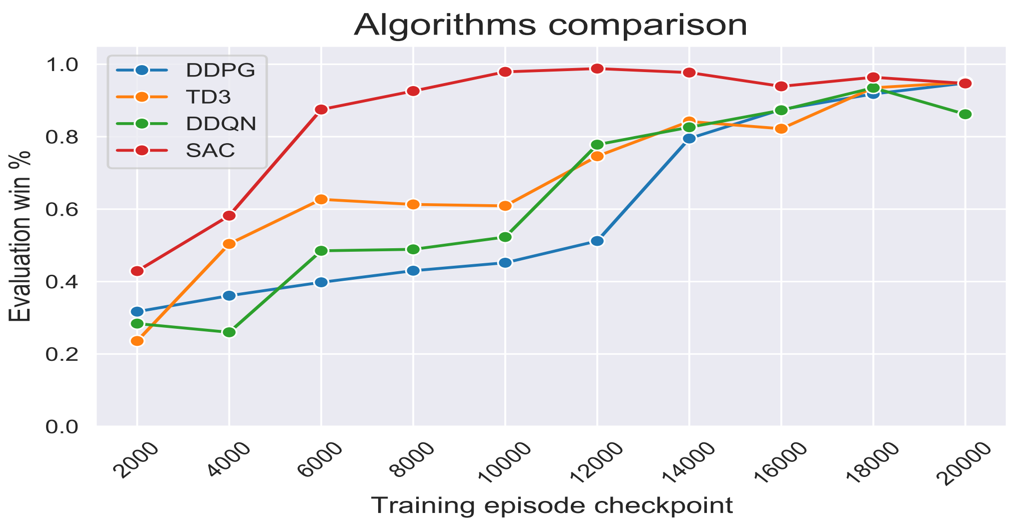

Figure 13: Comparison of all implemented algorithms

For each of the implemented algorithms we also trained their self-play versions. Every trained agent was better than corresponding version trained only on basic opponents. Different behaviours and polices were learned, but each one of them lead to an agent that was able to win in all games versus basic opponent, and also versus many agents in the tournament.

In particular, for the tournament we used the self-play mechanism in combination with the collected transitions from the rollouts during the evaluation phase. This turned out to be particularly effective against the other agents, which resulted in our team ranking at the number one spot.

That being said, this was a very fun, complex and engaging challenge that left each one of us with invaluable experience.

References

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

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *CoRR*, abs/1812.05905, 2018. URL http://arxiv.org/abs/1812.05905.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL http://arxiv.org/abs/1412.6980. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

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

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR. URL http://proceedings.mlr.press/v48/wangf16.html.