School of Human Sciences

Multi-Agent Reinforcement Learning with Atari's Pong

Implementing Artificial Neural Networks with TensorFlow

Final Project

Authors Nico Burgstaler, nburgstaler@uni-osnabrueck.de
Lisa Dollmann, ldollmann@uni-osnabrueck.de
Hendrik Warnecke, hwarnecke@uni-osnabrueck.de

Instructor Leon Schmid
Institute of Cognitive Science

Date Winterterm 2022/ 2023
Osnabrück, 1^{st} April 2023

Contents

1 Context and Motivation

Our goal is to train both Agents in Pong with Reinforcement Learning. Pong is a two-player game imitating "tennis" (See figure 1). There are two paddles and a ball. Each player controls a paddle which they can move up and down to catch the ball. If the ball hits the paddle it moves back to the opponent. A player receives a point if the opponent misses a ball. The game is played until one player reaches 21 points (in the original version from Atari it was 10 points[1]). We aim to reach our goal by training both paddles in the classic pong game with reinforcement learning. As we are training both paddles, it is more specifically Multi-Agent Reinforcement Learning.

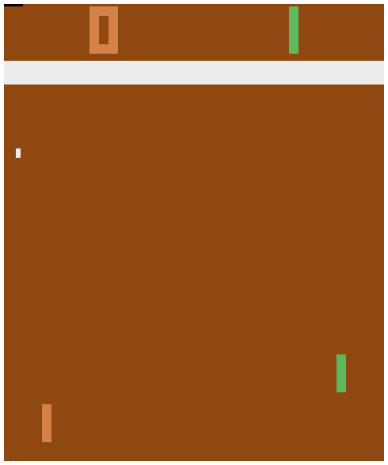

Figure 1: Atari's Pong played with the Gymnasium API

1.1 Theoretical Background

1.1.1 Value-Based Deep Reinforcement Learning

Reinforcement Learning (RL) deals with an agent learning to choose the right actions in a given state to receive a predefined reward in an environment. By obtaining the reward, the behavior of the agent is reinforced. Thereby, behavior or more commonly *policy* means choosing this action in a given state. The reward is mostly not obtained right after taking an action. A good policy, therefore, needs to choose the actions which eventually lead to a reward. To evaluate which actions will probably lead to a reward, the expected reward for taking action a in state s is inscribed in the Q-value. This Q-value can be calculated for every state and every action by the state-action-value-function $q_\pi(s, a)$ whereas the next actions are chosen according to policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \tag{1}$$

The optimal q-function q^* then obeys the Bellman-optimality equation

$$q^*(s, a) = \mathbb{E}[r + \gamma \max_{s', a'} q^*(s', a')]. \tag{2}$$

In typical Q-Learning this is done with a Q-table storing a value for every possible combination of state and action. In real-world applications, this is often not feasible as such a table can only deal with discrete values. And even discrete observation spaces can lead to unpractical high dimensions very easily. In our example of Pong using the

[1] `http://www.ponggame.org/`

PettingZoo environment the observation space is simply the colored video output with the shape (210, 160, 3). The action space contains 6 actions. This means a potential Q-table has to include and learn $210 * 160 * 3 * 6 = 604800$ entries. This problem is part of the curse of dimensionality.

One possible solution to this problem is the use of an artificial neural network as a function approximator that maps the state to one of the possible actions. This approach is called Deep Q-Network (DQN) and is originally proposed by Mnih et al [10]. The DQN makes it possible to deal with high dimensional or continuous observation spaces as long as the action space is discrete and, at best, rather small. Figure 2 depicts Deep Reinforcement Learning, the generic of DQN. Like in usual Reinforcement Learning, an agent takes actions in an environment and obtains a reward. The difference is, the agent's policy comes from a Deep Neural Network.

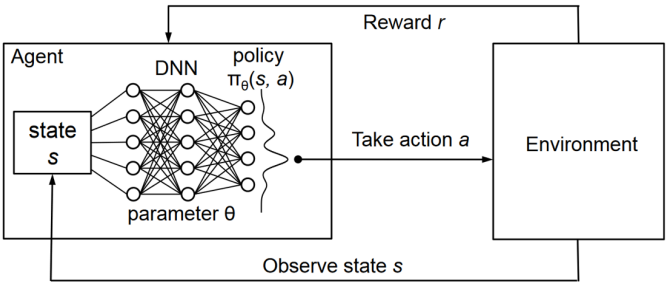

Figure 2: Deep Reinforcement Learning [8]

The targets to train the network are an iterative update following the Bellman-optimality equation 2. The targets are a better approximation of the expected reward in the investigated state, as they are the sum of the reward for the action taken and the maximal q-value of the next state.

$$y_t = r_t + \gamma \max_{a'} q_\pi(s_{t+1}, a') \tag{3}$$

As the targets are calculated after the original action was selected, they are also called temporal difference targets. Furthermore, the targets are always selected after a greedy policy, where the agent also explores the environment. Because of this difference in the agent's behavior policy and its learning policy, DQN is an off-policy algorithm. The training loss in DQN is calculated with the Mean Squared Error.

1.1.2 Experience Replay Buffer

In Reinforcement Learning the agent creates its own training data while interacting with the world. When only using the immediate data to train, we run into two problems. The first one is that using a number of consecutive frames that all happen directly after each other does not contain enough distinct information between the frames, as their content is highly correlated. This can lead to potential overfit to this specific sequence. The second problem is that to properly learn the correct action to a certain state, the agent needs to experience this state multiple times. This means that we have to hope that the agent explores the environment enough to land in the same situation again and again. Both problems can be mitigated by using an Experience Replay Buffer (ERP). The ERP simply stores the experience the agent made in a circular buffer. This usually includes the current state, the action taken, the received reward, and the next state. When training the network, the agent chooses its training samples randomly from this buffer which makes sure that the data is non-continuous and allows for re-using prior made experiences to fully exploit their potential, increasing the efficiency of the training process. The circular

nature of the buffer makes sure that the old data points are eventually replaced by newer experiences. When working with an ERP, there is usually a filling phase at the beginning where no training occurs. Either this is done with completely random samples or the agent navigates the environment and fills the ERP up to a minimum size before starting the training. This ensures that the data points at the beginning are not sampled overly often.

1.1.3 Delay Target Network

Another commonly used method that helps with training stability is to use two different networks for choosing an action and for generating the targets for the Q-function. The second network, often called target or delay target network gets its weights updated to the weights of the main network after a certain number of parameter updates in the main network. This ensures that the Q-values and the targets used for calculating the network updates are not affected by changes in the same way at the same time, bringing more stability into the learning process.

1.1.4 Exploration vs. Exploitation paradigm

In Reinforcement Learning, we are using the current policy to gain new knowledge that we can use to update the policy. This also means that if we always follow what is currently the optimal policy we are potentially stuck to local optima since the policy will always choose the same sequence of steps. To avoid this problem, we use a standard epsilon-greedy approach. This simply means that with a probability of epsilon, we will choose a random action instead of using the policy. In order to find a good tradeoff between exploration (using random actions) and exploitation (using the optimal policy) we start with a very large epsilon and slowly decay it over time to a minimum value. This process is also known as simulated annealing.

1.1.5 Multi-Agent Reinforcement Learning

A lot of real-world applications as well as (computer) games feature more than one agent that acts and influences the observation space. Multi-Agent Reinforcement Learning (MARL) deals with the topic of having two or more agents that simultaneously act and learn in the same environment. While many concepts from Single-Agent Reinforcement Learning can be used, there are a couple of distinguishments to classify a situation to choose successful strategies which suit the situation. These are well summarized in [17]. First, like in Single Agent Reinforcement Learning we distinguish between Value-Based and Policy-Based methods. As we have a finite action space in Pong it suffices to use Value-Based methods. In general, they deal with finding a good estimate of the optimal state-action-value-function (q^*-function). In our case of Deep-Q-learning a neural network is used to approximate the q^*-function. Furthermore, in MARL two frameworks are distinguished: Markov-Games and Extensive-Form-Games. In Pong we deal with a Markov Game: Each agent receives the same input at the same time and acts on it (see figure 3).

Lastly, there are cooperative, competitive, and mixed types of games. Usually, Pong is a competitive game, as each player receives a reward for scoring a point and a negative reward for losing one. Because scoring a point for one agent means losing this point for the other agent, the total reward is always zero. Therefore we deal with a zero-sum game.

1.1.6 Challenges in Multi-Agent Reinforcement Learning

Besides the additional required distinguishments Multi-Agent Reinforcement Learning faces four main challenges: The non-unique learning goal, the non-stationarity, the scalability issue, and the various information structures. We will briefly address them here to point out how they influenced our work. The non-unique learning goal sums up the

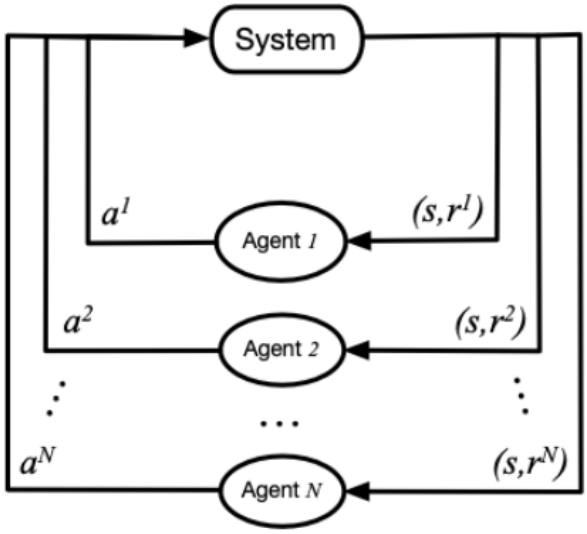

Figure 3: Markov Game [17]

vagueness of some MARL tasks which might be due to the multi-dimensionality of the learning goals. Since we only have two agents in Pong and the goal for both agents is simply to score a point, this challenge is not relevant to our work. The non-stationary describes the changing optimal policy and leaves a possible need to change one action when facing the same state again. This is due to both agents trying to maximize their reward and adjusting their behavior towards this. However, this means, if they face the same state again, the opponent might behave differently and thus a former good action might not be successful again. Furthermore, this is a violation of the Markov property, one general assumption in Single-Agent RL that the reward only depends on the current state. Hence, it is formally not correct to apply Single-Agent Reinforcement Learning algorithms to Multi-Agent systems. Because we update our ERP frequently, the effect of the Markov property is less important and we decided to use DQN anyway. Other solutions may be found in [5]. The scalability issue addresses that each agent needs to account for the joint action space which increases exponentially with the number of agents and thus makes the convergence analysis complicated. Because we only consider the two-agent version of Pong, this is neglected. Lastly, the various information structures, which stem from partially observable environments, are not a problem as Multi-Agent Pong is a Markov-Game and therefore every agent has the same information structure.

1.2 Relevancy of the Project

Reinforcement Learning focuses on goal-directed learning from interaction, i. e. an agent needs to discover which action leads to the goal. In Deep Reinforcement Learning this implies that no data needs to be gathered and prepared in advance of the learning process like in supervised learning and not even provided like in unsupervised learning. The training data is simply incrementally acquired in the course of the learning process, saving time and money to obtain the required data otherwise. Besides this, Reinforcement Learning is a very human-like form of learning to gain knowledge and become more intelligent. For example, if a child does something good, the mother will probably praise the child or reward it e.g. with a piece of chocolate to reinforce their behavior. Therefore, DRL bares the potential for intelligent machines.

One might wonder why teaching an AI to play Pong is relevant. Yet, small problems like this are a preparation for general AI. Atari's Pong is a relatively controlled environment to try out DRL algorithms. It offers the chance to study the algorithm, repair minor errors, and generate a solid implementation that can be further used in more complicated

environments.

Especially because most scenarios include multi-agents and these scenarios are in general more challenging, it is important to have a good understanding of the basics. As the basics are already applied in a small problem like Pong, solving Pong is a good opportunity to begin with, and the knowledge may then be expanded to more complex problems.

2 Related Literature

The idea of combining Reinforcement Learning in form of Q-learning with deep neural networks to successfully train an agent to play Atari games is approximately ten years old now. It was first proposed by Mnih et al. in 2013 for Single-Agent Reinforcement Learning [9]. Two years later, this approach was expanded to Multi-Agent Reinforcement Learning by Tampuu et al. [11]. In the following chapter, we will look at the literature our project is based on and how they successfully solved the problem at hand.

2.1 Single-Agent

Playing Atari games with reinforcement learning is based on the Arcade Learning Environment (ALE) which was introduced by Bellemare et al. in 2013 [2]. ALE is an interface that is built on the open-source Atari 2600 emulator Stella. It provides a game-handling layer that returns the observation, action, and score transformed into a reward as well as a status of whether the game has ended or not. The ALE interface is integrated into the Gymnasium API[2] which can be used for Single-Agent Reinforcement Learning. Gymnasium is a fork from OpenAI Gym that was introduced in 2015 [3] and aims at providing a standard open-source API for connecting game environments like the ALE interface with learning algorithms. Later, the maintenance of the Gym library was transferred to the Farama Foundation which released Gymnasium.

The first deep learning model using Reinforcement Learning to play Pong was published in [9], [10]. The model is based on a convolutional neural network that has the Atari game frame as its input and estimates future rewards via a value function. It was tested on various Atari games without adjusting the architecture or hyperparameters to confirm its model-free approach to Reinforcement Learning. The results showed that agents trained with this deep Q-network can surpass human skills in most games.

In 2018, ALE was being reviewed and some key additions were made that will help us use the Atari environments. Without any adjusting measures performed on the Gymnasium environment, the Stella emulator as well as ALE are completely deterministic. This behavior can lead to problems where an agent just memorizes and executes an optimal set of actions from the initial state while ignoring the perceived observation. Therefore, [7] introduces several methods to include stochasticity in the environment, with sticky actions being the most prominent one. Some of these methods can be used in a preprocessing function to add stochasticity to the environment.

2.2 Multi-Agent

As the Gymnasium API, as well as the ALE interface, are only made for Single-Agent Reinforcement Learning, they needed to be adjusted for using multiple agents in the same environment. Therefore, [12] and [14] introduced multiplayer support for ALE and a Multi-Agent Reinforcement Learning API called PettingZoo, respectively.

The Multi-Agent ALE is an extension with minor changes to the standard ALE to allow each agent within the environment to receive its own reward given an action. PettingZoo is a Python library that provides useful in-build methods and pre-made environments specifically for multi-agent reinforcement learning. The library is maintained by the Farama Foundation which also maintains Gymnasium.

[2] https://gymnasium.farama.org/

In [11], the DQN architecture for Single-Agent Reinforcement Learning was extended to control both agents in a game of Pong. This was done by using two independent DQNs that control one agent each in the same environment. Additionally, a difference between competitive and collaborative behavior, where the two agents either try to score efficiently or keep the ball in the game as long as possible, can be achieved by changing the reward each agent receives. The classic competitive game of Pong is a zero-sum game where one agent gains a positive reward for scoring while the other gains a negative reward for conceding a goal. In a collaborative environment, both agents get negative rewards when the ball is out of play regardless of the side and cannot get a positive reward for scoring. The paper proves that adjusting the rewards leads to the expected outcome of competitive and collaborative behavior.

Instead of using several independent networks, [1] used self-play, a method where all agents in a Multi-Agent environment are controlled by the same network. Self-play has the key advantage of a natural curriculum which means that an environment full of agents with the same skill level - as they are all using the same network - will always have the right challenging level of opponents for optimal learning.

3 Methods

To solve Multi-Agent Pong with Reinforcement Learning we first attempted to solve Single-Agent Pong and then reused some parts of our implementation. We used the Pong environment from Gymnasium and PettingZoo, respectively. Hence, we could focus on the implementation of the actual DQN algorithm and could just use the reward function and environment dynamic implemented in the gyms.

For the Single-Agent case, we mostly followed the pseudocode in Algorithm 4 from [10]. As our procedure will be explained in 4 we only give a basic overview here. Basically, the agent plays one game of Pong during each episode. Thereby, it always takes one step in the environment and will be trained on random samples from the ERP afterward.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 4: Algorithm for DQN [10]

3.1 Architecture DQN

We use an architecture like in [10] (see figure 5) with minor changes. We have a separate output unit for each action and take a batch of observations as input. Our model then consists of three Conv2D layers, followed by two Dense layers. The first Conv2D layer has 32 filters, with a kernel size of 8 and stride of 4. The second Conv2D layer has 64 filters,

with kernel size 8 and stride 2. The last one has 64 filters, kernel size of 3, and stride 1. Each layer uses ReLU for the activation function and padding is same. We use a flatten layer to bring the 2D images into a 1D shape before the activation is passed to the Dense layers. The first Dense layer has 512 units and no activation function. The last layer has 6 units that correspond to the six actions the agent can choose from in Pong.

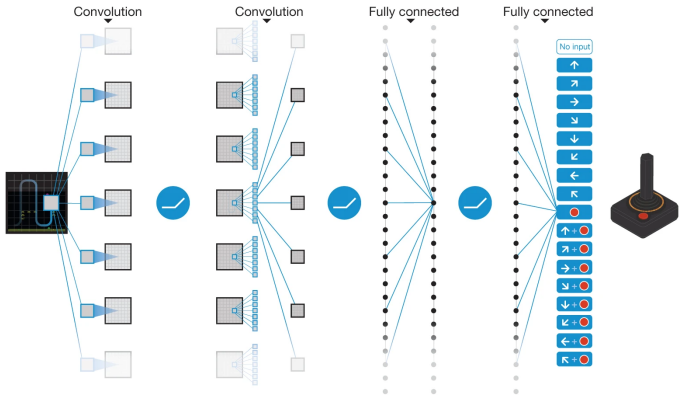

Figure 5: Architecture of DQN [10]

3.2 Multi-Agent Pong

To train the Agent in the Multi-Agent scenario we used self-play. This means the Agent is playing against itself to gather experience. This has two big advantages. First, the Agent plays against an opponent of the same strength. Second, we could reuse many implementations from the Single-Agent case and only adjust the gym to PettingZoo.

4 Single-Agent Implementation

Our implementation for Single Agent Pong mostly follows algorithm 1 from [10]. We implemented three python modules each containing a class for the agent 4.1, the DQN 4.2, and the ERP 4.3, respectively. These modules are imported and used in the main training loop. Besides our own modules, we import and use NumPy, TensorFlow, and Gymnasium.

As described in the pseudocode, we first initialize an ERP, a DQN along with its delay target network, and, in addition, an agent. We decided to write an agent class because it is handier to assign methods like epsilon-greedy-sampling. Unlike the pseudocode, we fill up the ERP with random samples before we start training. This is necessary to have samples available to start training during the first episode and faster than the agent choosing the random samples. At the beginning of each episode, we reset the environment and store this first normalized observation of the environment in the variable observation. We also reset the variable terminated and truncated as well as the agents' attribute $reward\_of\_game$. Within the next loop, the agent plays Pong. We decided to use a while-loop instead of another for-loop like in the algorithm. That is due to the boolean variables terminated and truncated which are returned from the environment during each step. The first Boolean is set to True if one player has scored 21 points and the second Boolean is set to True if a time limit is reached. Both are intuitive criteria to end a game and justify our choice. To play Pong, the agent uses its *play*-method 4.1.2, which accounts for the next five lines of pseudocode. Afterward, we call the agents *training*-method 4.1.2 accounting for the next three lines. Lastly, we log some metrics, update the delay target network 4.1.2 and print the reward the Agent received for the recently played game.

4.1 Agent Class

4.1.1 Attributes

The agent needs to receive an *environment*, an ERP, and a *model_name* for logging. One may also set the *num_actions* which we defaulted to 6 the number of possible actions in Pong and *epsilon* which we defaulted to 1, to ensure a lot of exploration at the beginning of training. Besides these parameters, the agent has a *network* and a *delay_target_network*, both are instances of our DQN class 4.2 with *num_actions* outputs. Furthermore, the agent keeps track of the $reward\_of\_game$, which is 0 in the beginning. For logging, the agent has a *metrics_list* for the *loss*. The tracking of the reward is done outside of the agent class.

4.1.2 Methods

call **method**

The *call* method calls the agents' DQN to get the q-values for an observation x.

epsilon_greedy_sampling **method**

The function *epsilon_greedy_sampling* balances exploration and exploitation. It receives an *observation* and an *epsilon* determining the probability for exploration. The decision on how to choose the action is made by an if-else query. If a random number between 0 and 1 is bigger than epsilon, the Agent takes the action with the largest q-value from the DQN. Because *epsilon_greedy_sampling* only receives a single observation, this needs to be preprocessed before it is given to the DQN. Numpys' *argmax* then returns the index with the largest q-value. Else, it takes a random integer between 0 and *num_actions* representing the actions from the action space. To create the random numbers we use NumPy's random functions. The chosen *action* is returned by the function.

play **method**

To play Pong, the Agent uses its *play* method which receives a normalized observation as input. Inside this method, the agent first chooses an action with epsilon-greedy-sampling 4.1.2. Then the step-function of the environment is called with this action. The step function returns the next observation, the reward for this action, the booleans terminated and truncated, and info. The latter we never use, and thus do not save it to a variable. Then, the next observation is normalized with the ERPs' *normalizing* method. Next, a tuple consisting of observation, reward as $tf.float32$, action, and next observation overwrites the oldest sample in the ERP. Therefore, we track the index of the ERP and set it to the oldest sample again after each step 4.3.2. Afterward, epsilon decay 4.1.2 is called, to secure more exploration in the beginning and more exploitation later on, and the reward is added to the agents' attribute $reward\_of\_game$. Lastly, the current observation and the two booleans are returned. The former is now the observation for the next step, which follows after training.

training **method**

To train the network, we sample experiences from the ERP with the ERPs' method *sample*. Afterward, the ERP method *preprocessing* is called transforming the list to a tf.Dataset to speed up training. With the newly created dataset, the DQN is now trained. Therefore, the targets are created with the agents' method *q_target* 4.1.2. These q-targets give a better estimation of the expected reward and may now be used to train the DQN. Thus we call the *train* function of the DQN of the agent via $self.network.train$ 4.2.2 and provide it with the observation, the action the agent chose before, and the just calculated q-targets.

q_target **method**

The *q_target* function creates the q-targets with the delay target network. Its inputs are the batched rewards and the batched next observations. Based on the target delay network the q-values of the next observations are calculated. The function $tf.math.reduce\_max$ allows to get the maximum q-value for each batch element, and thus the highest expected reward for each batch element. This is now discounted by the discount factor with a default 0.99 and added to the former reward. The so-created variable *q_targets* is then returned by the function.

update_delay_target_network **method**

The function *update_delay_target_network* sets the weights of the delay target network to the current weights of the DQN by using the two keras methods *set_weights* and *get_weights*.

epsilon_decay **method**

The function *epsilon_decay* decays the agents' attribute *epsilon* to ensure a good balance between exploration and exploitation. At the beginning of training, the agent is supposed to explore a lot, meaning it should choose many random actions. As its performance improves in the course of training it should exploit its knowledge and choose the highest q-value. Therefore, as long as epsilon is bigger than the threshold $MIN\_EPSILON$, epsilon is decayed by the factor $EPSILON\_DECAY$.

4.2 DQN Class

4.2.1 Attributes

To initialize an instance of the DQN, it only needs the *num_actions*. Besides this, the DQN has the attributes *loss*, namely keras' Mean Squared Error, *optimizer* which is set to keras' Adam, and a *metrics_list* keeping track of the loss. The last attribute of the DQN is the *network* a list of keras layers according to the architecture described in section 3.1. The last layer outputs a q-value for each action. It is initialized with kernel- as well as bias-initializers like in TensorFlows tutorial for DQN[3].

4.2.2 Methods

call **method**

The *call* function is used to predict the q-values for each of the *num_actions* actions. It passes the input, which is an observation, through the layers of the *network* via a for-loop. It then returns the q-values.

train **method**

The *train* function trains the network via backpropagation. Its inputs are a batch of observations, the according actions the agent chose for this observation, and the q-targets. With $tf.GradientTape$ the q-values are predicted and stored in the variable *predictions*. For each prediction the q-value is selected which corresponds to the action the Agent chose during the game. This is done with $tf.gather$. Next, the *loss* is calculated from the just calculated predictions and the *targets* using the Mean Squared Error. Lastly, the *metrics* are updated and returned.

[3] `https://github.com/tensorflow/agents/blob/master/docs/tutorials/1_dqn_tutorial.ipynb`

4.3 Experience Replay Buffer Class

4.3.1 Attributes

An instance of the Experience Replay Buffer (ERP) is initialized with the *size* of the ERP. Additionally, it has the attributes $experience\_replay\_buffer$, a list of zeros of length *size* representing the whole ERP, $experience\_replay\_buffer\_samples$, an empty list to store the samples on which we want to train during each step, and the index which is 0 in the beginning.

4.3.2 Methods

sample **method**

Inside the *sample* function *batch_size* many experiences with a default set to 32 are chosen randomly with numpys' random method and stored in the ERPs' attribute list $experience\_replay\_buffer\_samples$.

$fill$ **method**

The $fill$ function fills up the whole ERP with random samples from the environment. The environment is thereby the only parameter. The $fill$ method works like the *play* method (4.1.2) of the agent, except that it "plays" until the ERP is filled up, the actions are always chosen randomly, and the environment is reset during this procedure if a game has finished.

preprocessing **method**

The function *preprocessing* transforms the ERPs' attribute list $experience\_replay\_buffer\_new\_samples$ to a $tf.data.Dataset$ and preprocesses the entries as usual. That is, the data is cached, shuffled, batched, and prefetched. Before the data is returned, the attribute $experience\_replay\_buffer\_samples$ is reset to an empty list.

set_index **method**

The function *set_index* sets the index of the ERP to enable the overwriting of the oldest sample. Therefore it usually increases the index by 1, except if it is at the end of the ERP. In this case, it sets the index to 0.

normalizing **method**

The function *normalizing* normalizes its input observation, namely dividing it by 255. and casting it to a $tf.float32$. This function is applied to any observation before it is stored in the ERP, so each observation is normalized only once.

4.4 Hyperparameter Setting

We use a function to set the hyperparameters. The defaults are set to the values from the nature paper [10]. The first parameter *num_actions* is set to 6, the number of possible actions for a Pong player. The next ERP_size determines the size of the ERP and is set to 75000. The $EPISODES$ determines the number of games played and is set to 20000 to keep the agent playing. Next, the parameter *epsilon* is for exploration and set to 1 as it will be decayed in the course of the games. The last four parameters are used for logging. The $MODEL\_NAME$ is needed to save the model. $AGGREGATE\_STATS\_EVERY$ enables saving the model only every n episodes, yet the default is 1. Also, MIN_REWARD enables saving the model only if a certain threshold is reached, yet its default is also -21.

$UPDATE\_TARGET\_EVERY$ determines how often the weights of the delay target network are updated. The default 10 approximates the update rate 10000 from [10].

4.5 Preprocessing via Environment Wrappers

Naturally, the Atari environments from Gymnasium and PettingZoo come in an observation space shape of 210x160x3 that includes the two-dimensional resolution of the displayed game and a third dimension for the three color channels. This can result in high computational effort and required memory for training the model [10]. Additionally, Atari games are completely deterministic, resulting in a possibility where the Agent is just memorizing an optimal set of actions to gain the highest reward without reacting to the observation made [7]. Therefore, the environment needs to be prepared in form of a preprocessing function for smoother and faster use in the DQN.

There are several ways to handle the preprocessing for Atari's Pong with Gymnasium. First of all, we could have written this completely from scratch by adjusting the observation output of the environment before inserting it into the network. Luckily, there are already existing functions called wrappers that can be used for this. As mentioned in [10], the preprocessing includes re-scaling the observation space from 210x160 to 84x84 as well as reducing the third dimension by gray scaling. Further, to prevent frame flickering[4] and reduce the necessary computations per game, a certain number of frames always get the same action without the network predicting it. In the end, four preprocessed frames are stacked together into one state and are inserted into the network at once. This frame stacking also allows the agent to detect the direction of motion of the ball in Pong.

On top, Gymnasium introduces two additional wrappers for stochasticity to tackle the deterministic character of Atari games. Sticky actions are used to repeat the previous action with a small probability instead of executing the action passed to the environment. Noop reset is forcing the agent to not execute an action for a random amount of frames at the start of each game.

Preprocessing in a Gymnasium environment is done via the included *gymnasium.wrappers* class. There are special wrappers for every step mentioned above. For Atari games, there is a specific wrapper called AtariPreprocessing which combines all the necessary wrappers into one function and can be adapted by the given arguments. While doing Single-Agent Reinforcement Learning we used $gymnasium.wrappers.AtariPreprocessing$ followed by $gymnasium.wrappers.FrameStack$ which are included in the *create_env* function.

4.6 Metric logging

In order to track the development of our network we logged the sum of the reward the agent got over one period. Instead of logging every single episode, we decided to only log after a certain number of episodes are passed. This was done because the single values do not contain important information compared to logging over a bigger number of trials and this allowed us to reduce the amount of read/ write necessary actions. The metrics logged are the average reward over the last episodes and the maximum and minimum reward that was returned, effectively showing the upper and lower bound of our model during that part of the training process. Additionally, after the metrics are logged, it is decided whether the model is saved or not. If the minimum reward the model obtained reaches a certain threshold, it will be saved. Since we are creating new save files and do not overwrite previous saves we could later check the logged metrics and pick the best-performing model, as it is possible that the performance could start to decline again after too many episodes of training.

[4]Frame flickering is an artifact of Atari games where some entities are only displayed on even or odd frames caused by limited RAM in early gaming consoles.

5 Multi-Agent Implementation

Our implementation of the multi-agent setup follows the Single-Agent one described above very closely. The major changes made to adapt the code are listed below.

5.1 PettingZoo API

The key difference lies in using the PettingZoo API instead of the Gymnasium API. The general usage is very similar to Gymnasium with a few changes made to the way each agent is collecting its reward. PettingZoo introduces the additional *env.last* function that returns the current observation and the rewards that the agent collected since the last time it acted. For taking an action in the environment, the *env.step* function is used, just that it does not return anything like it does in the gymnasium API, as these values will be returned by the *env.last* function the next time the agent acts. The *env.step* function also automatically hands over the controls to the next agent, making it possible to just iterate through the agents in a loop. The pong environment provided by PettingZoo has the same structure as the Gymnasium implementation in terms of observation space and action space, making it easy to reuse the network structure that we implemented for the Single-Agent network.

One change in the way PettingZoo is used compared to Gymnasium led to a slight problem that we had to overcome. As described above, when using Gymnasium you get the initial observation by calling *env.reset* and the next observation by calling *env.step*. Both, the observation and the next observation are then stored in the ERP and subsequently used for the training step. In PettingZoo, neither the *env.reset* nor the *env.step* method returns anything. The only way to get an observation is by calling *env.last*. This means that we only get the current but not the next observation that occurs after the agent takes a step. Our way of getting around this inconvenience is to call *env.last* twice. Once before the current agent takes a step to gather the current observation and the rewards and once after a step was taken. When calling *env.last* the second time only the observation is saved, as the reward and done flags are collected in the next iteration of the loop when *env.last* is called the 'first' time. It should also be possible to invoke the environment as a parallel version, where both agents can act at the same time. This would get rid of the *env.last* method and changes a step in the environment to look more Gymnasium-style just with a dictionary of actions and rewards keyed for the individual agents instead of single values but as our method seems to run as well we did not try to implement a parallel version.

5.2 Preprocessing via SuperSuit

For Multi-Agent Reinforcement Learning in a PettingZoo environment, we use the SuperSuit library [13] as there are no embedded wrappers included in the PettingZoo Atari environment. SuperSuit implements the same preprocessing functions known from the gymnasium wrappers. We used $max\_observation\_v0$, $sticky\_actions\_v0$, $frame\_skip\_v0$, $resize\_v1$, $color\_reduction\_v0$, and $frame\_stack\_v1$ for Multi-Agent Pong which are also included in the respective *create_env* function.

5.3 Self Play

Instead of creating and training a new network for each agent, we use a method called parameter sharing or self-play. This means that both agents are using and training the same network. The core idea behind this is to ensure that the agent always has an opponent with a similar skill level and thus ensure an optimal learning environment, making a conversion more likely. Additionally, for us, this means that the network trains twice as often during one episode compared to the Single-Agent but also takes about twice as long for one episode to complete. You can see this in the result section, as both models trained

for about 9 hours but the single-agent has over twice the amount of episodes completed. [1] and [15] showed that complex behavior can emerge when using self-play. We achieved self-play by simply feeding the information from both agents into the ERP from which the training data is sampled. The action for each agent is also drawn from the same network accordingly.

5.4 Metric logging

In the Single-Agent case, it was enough, to sum up the rewards the agent received in every step. In the Multi-Agent environment, this can lead to ambiguous results. Whether an agent performed well does not only depend on its actions but also the actions of the opponent. Consider the following scenario: Agent one and agent two are both very good at defending their side, meaning at the end of one episode very few points were scored or lost. The sum of all rewards during that episode for one agent will be close to zero in that case. Now, if both agents are equally bad at defending they will both score a lot of points but also lose a lot of points. In this case, the sum of the rewards will also be close to zero. So by looking at the cumulative reward alone we cannot safely infer if an agent played well or not. In theory, the same applies to the Single-Agent case but in practice, this is not an issue since the algorithm that controls the second panel plays rather consistently. To get around this issue we are logging the positive rewards and the negative rewards of each agent separately, effectively counting how many points were scored and how many points were lost. Additionally, we included the loss and the epsilon value in the logged metrics. Since the time it takes to go through one episode is longer in the Multi-Agent case, we decided to log the data every 10th episode.

6 Results

In the following chapter, we will showcase the results of training the Single-Agent and the Multi-Agent. The figures referenced can be found in the appendix of the paper. The full set of screenshots from the Tensorboard, including those that we did not present here, as well as the log files from our training can be found in our GitHub repository[5] where the code can be found as well.

6.1 Single-Agent Results

While we had the option to only log every n episode, in the final run of the Single-Agent we decided to log the metrics every episode. This means that the graph for the average, the maximum, and the minimum reward will show the same graph and we will only show one of them here. Starting with the loss (figure 6), after an initial drop to a minimum at around 30 episodes, there is a clear upwards trend with convergence towards 0.025 at the end of the 1000 trials. But since the loss is not as important in a Reinforcement Learning setting, we will focus on the rewards of the agent next.

Since we logged every episode the graph (figure 7) itself looks a bit messy, as it jumps up and down a lot. But when focusing on the maximum and minimum values it is apparent that the jumps only occur in the narrow corridor of -21 and -19. As the reward stays consistently between these boundaries it gets clear that our agent did not learn anything and shows the same poor performance at the end of our 9-hour training phase as it did at the beginning. See figure 8 for a heavily smoothed version of the reward that reduces the number of jumps in the data and makes the flat-line trend a bit more obvious.

We came up with a set of possible reasons for the poor performance of our agent, including the hyperparameter settings, the network architecture, and our ERP. These will be discussed in the next chapter.

[5] `https://github.com/doelfi/MAPP`

6.2 Multi-Agent Results

In our Multi-Agent approach, we are logging the positive and negative scores of each agent separately. Pong is a zero-sum game, this means that if one agent has a low negative score, the other agent will have a high positive score in that episode. This means that it is easier to infer the performance of one agent from his negative scores, as the positive score is correlated with the performance of the other agent. First, we take a look at the average negative reward over 10 episodes for each agent, found in figure 9 and figure 10. Here we can see that both agents performed rather poorly, always staying below the -10 points mark, often even lower than that. Agent 2 performs worse than Agent 1 in this case. The fact that the curve stays rather consistent over the whole training period shows that the agents did not significantly learn anything but performed the same at the end.

Next, we take a look at the maximum of the negative rewards, which can be found in figure 11 and figure 12. The graph effectively shows the best-case run over the last 10 episodes. An optimal result would be a convergence towards 0, as this indicates that the agent learns to properly defend its side. Again, we see no clear trend toward convergence. Agent one reaches zero in some cases but does not show a change over time. Agent two performs worse than agent one again, as it never reaches a maximum reward of 0 during training.

When looking at the maximum positive reward (figure 13 and figure 14) we find a flat line at 21 points for both agents. This shows that all games played ended with one agent winning and not because they managed to play long enough for the game to run out of time. It also shows that while agent two performs worse in all other metrics, it at least won one game during a 10-game period.

One anomaly can be found when looking at the lowest negative rewards collected by the agents. You can find the results in figure 15 and figure 16. The pong environment should only run until one agent reaches 21 points. This also means that the lowest possible negative reward should be -21. Looking at the graph we can see that this is not always the case. Here we can see that routinely, the agents scored negative rewards that go below -21, in an extreme case even down to -29. This is interesting as it breaks with the concept of a zero-sum game as the maximum positive reward did not exceed 21 points. It is unclear whether this behavior stems from our way of implementing the PettingZoo API or if this is an intended behavior by the API. Besides being interesting, it is unlikely that this anomaly caused poor training results.

We also tracked the loss during training (figure 17). You can detect a quick drop toward a minimum after around 50 episodes and a steady climb that goes over the starting values afterward. Besides this behavior the loss itself stays rather small, never exceeding a value of 0.0244. But the loss is typically not of the same importance in Reinforcement Learning as it is in other fields of deep learning, hence why we focus on the reward of the agents.

7 Discussion

To sum up our documentation we will put our results and overall project into context. First, we will address our successes and challenges and then continue with further improvements. Overall, our Single-Agent model, as well as our Multi-Agent model, are running but the agents are not improving.

7.1 What worked, what did not?

First of all, we would like to point out our successes. To begin with, our code is running and the model is compiling. Further, we implemented all elements from the DQN-pseudocode and made use of the wrappers from Gymnasium in the Single-Agent case and of SuperSuit for PettingZoo in the Multi-Agent case. Moreover, we logged our progress with TensorBoard and visualized the results. Yet, to continue with our failures, our Models did not

learn and we could not find the reason. We tried multiple versions of the DQN algorithm, various attempts on optimizing the data processing with the ERP, and different model architectures of the convolutional neural network. Each version had some effect on the pace of training but not on its performance or success. Noticeably, the ERP versions in literature were of size $1,000,000$. However, our RAM already ran out of memory with an ERP of size $100,000$ which indicates a possible mistake. Moreover, our Multi-Agent model did not work because our Single-Agent model did not work. As training multiple agents is usually more complex than training a single agent, we did not expect our MARL environment to perform better than our Single-Agent environment.

Hyperparameter-setting

We used most Hyperparameters from [10]. Due to memory constraints, we could not use an ERP of size 1000000. Yet, we found example code on GitHub e.g. [16] which also worked with a smaller ERP of size 10000. Especially, as the ERP is rather responsible to avoid seeing the samples in order and enabling seeing one sample more than once. This is also possible with a smaller ERP and in the Multi-Agent case even necessary to account for the non-stationary of the environment.

Network-Architecture

In addition to the architecture proposed in [10] we wanted to make use of pooling layers. They prevent overfitting and help to reduce computational cost (further details e.g. in [4]) and are commonly used with CNNs. Hence, we wanted to add them to our network architecture. In combination with the Gymnasium environment wrapper $frame\_stack$ this brought another challenge. This wrapper is responsible for the first dimension being 4 in each observation from the environment. To enable our network to handle this dimension in pooling layers we used keras' $TimeDistributed$ layers. However, we also tried to train without these layers which cause the same results both in performance and in training results. The final run, from which the results were shown above, was made without the use of pooling layers.

ERP

In the beginning, we tried to train on experience samples of type list or array instead of a tf.Dataset. This led to very poor performance. As we didn't know back then, that it was due to the data structures, we took out some fancy features we used to have like training with multiple environments. Due to time constraints and some changes in the data structure of the ERP, we could not reimplement these features. Yet, they may be found in our GitHub history. Furthermore, we found the training to be much faster if we train at the end of a game on a dataset consisting of all the samples in the ERP if we store them in arrays (see 7.1). However, as the pseudocode in [10] suggests training after each step, we adjusted our code on the cost of performance by using the $tf.experimental.from\_list$ function. By now it takes on average 0.348 seconds to train on a single batch of 32 samples. This average was taken over 855 times calling the training function 4.1.2. To get a better feeling for the duration, we also timed our play function 4.1.2 which lasted on average 0.002 seconds and thus is 174 times faster. We used time's time function to stop the time and only used a CPU.

Data Structure ERP

One crucial choice to minimize training time is the data structure of the ERP. In the course of our project, we experimented with different data structures and want to explain our results even though we did not use them in the end.

While the DQN samples from a tf.data.Dataset for training, a tf.Dataset is unhandy for the collection of experience as every experience consists of different data types and the replacement of old experiences. Each experience consists of an observation, an action, a reward, and the next observation. While the observation and the next observation are both images and thus *numpy.ndarrays* of shape $(4, 84, 84, 1)$ after preprocessing the environment, the action and the reward are *integers* and $floats$ respectively, and thus both scalar values. To collect multiple experiences with the common *python* data structures list and array, this leaves two options to create datasets later on: append a tuple of experience to a list or store each part of an experience at the same index in different arrays of the according shape. The list may be converted to a tf.data.Dataset via the command $tf.data.experimental.from\_list$. For the four arrays, each array is converted to a tensor via $tf.data.Dataset.from\_tensor\_slices$. Afterward, the four tensors are put together with the *zip* command. In both cases the datasets enable the usual preprocessing, namely chaching, shuffling, batching, and prefetching them. For all tests, we used the *time*-function from the *time*-module. First, we timed the duration to fill up the whole ERP with random samples. We set the ERP size to 5000 and averaged over 100 fill-ups. This first experiment revealed the list ERP to be filled 1.13 times faster than the ERP consisting of the four arrays (see table 1).

data structure	time (s)
list	1.969
arrays	2.228

Table 1: Average time in seconds over 100 trials to fill up an ERP of size 5000 with either the data structure list or array.

Next, we tested the duration to create and preprocess datasets out of both ERP types. Due to warnings on memory exceeding, we use an ERP size of 5000 here. As it was about 2.85 times faster to create and preprocess a dataset from the arrays (see table 2).

data structure	time (s)
list	1.676
arrays	0.589

Table 2: Average time in seconds over 100 trials to create and preprocess a dataset out of a list-ERP and an array-ERP respectively. Both ERPs are of size 5000.

Training Duration

Another possible reason was us not training on enough samples or not long enough. Thus we searched for other experiences with RL and found a detailed overview from [16]. Even though our code differs in some points, especially since it is not written in PyTorch, we still expected an improvement at least after 200 games, which never occurred.

7.2 What should further be improved?

Lastly, we would like to suggest some improvements to our project. Our GPU usage was only 40% during training. Thus, we could try to optimize the data flow of our code to reach a more effective utilization of the hardware. Despite that, the performance could be improved, e.g. by preparing some train datasets in every episode so this has not to be done after each step. When doing so, a good balance is required to include the new experiences. Moreover, the performance could be improved with ideas from [6], by setting priorities to challenging examples or by using multiple environments to gather many experiences at once.

References

[1] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, *Emergent complexity via multi-agent competition*, 2018. arXiv: 1710.03748 [cs.AI].

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, Jun. 2013.

[3] G. Brockman, V. Cheung, L. Pettersson, *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[4] H. Gholamalinezhad and H. Khosravi, *Pooling methods in deep neural networks, a review*, 2020. arXiv: 2009.07485 [cs.CV].

[5] P. Hernandez-Leal, M. Kaisers, T. Baarslag, and E. M. de Cote, *A survey of learning in multiagent environments: Dealing with non-stationarity*, 2019. arXiv: 1707.09183 [cs.MA].

[6] M. Hessel, J. Modayil, H. van Hasselt, *et al.*, *Rainbow: Combining improvements in deep reinforcement learning*, 2017. arXiv: 1710.02298 [cs.AI].

[7] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. J. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents," *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, 2018.

[8] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16, Atlanta, GA, USA: Association for Computing Machinery, 2016, pp. 50–56, ISBN: 9781450346610. DOI: 10.1145/3005745.3005750. [Online]. Available: https://doi.org/10.1145/3005745.3005750.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[10] V. Mnih, K. Kavukcuoglu, and D. e. a. Silver, "Human-level control through deep reinforcement learning," *Nature 518, 529–533 (2015)*, 2015. DOI: 10.1038/nature14236.

[11] A. Tampuu, T. Matiisen, D. Kodelja, *et al.*, *Multiagent cooperation and competition with deep reinforcement learning*, 2015. arXiv: 1511.08779 [cs.AI].

[12] J. K. Terry, B. Black, and L. Santos, "Multiplayer support for the arcade learning environment," *arXiv preprint arXiv:2009.09341*, 2020.

[13] J. K. Terry, B. Black, and A. Hari, "Supersuit: Simple microwrappers for reinforcement learning environments," *arXiv preprint arXiv:2008.08932*, 2020.

[14] J. K. Terry, B. Black, N. Grammel, *et al.*, *Pettingzoo: Gym for multi-agent reinforcement learning*, 2021. arXiv: 2009.14471 [cs.LG].

[15] J. K. Terry, N. Grammel, S. Son, and B. Black, "Parameter sharing for heterogeneous agents in multi-agent reinforcement learning," *CoRR*, vol. abs/2005.13625, 2020. arXiv: 2005.13625. [Online]. Available: https://arxiv.org/abs/2005.13625.

[16] J. Torres, *Deep reinforcement learning explained-15 - 16 - 17*, https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_15_16_17_DQN_Pong.ipynb, accessed: 26.03.23.

[17] K. Zhang, Z. Yang, and T. Başar, *Multi-agent reinforcement learning: A selective overview of theories and algorithms*, 2021. arXiv: 1911.10635 [cs.LG].

Appendix

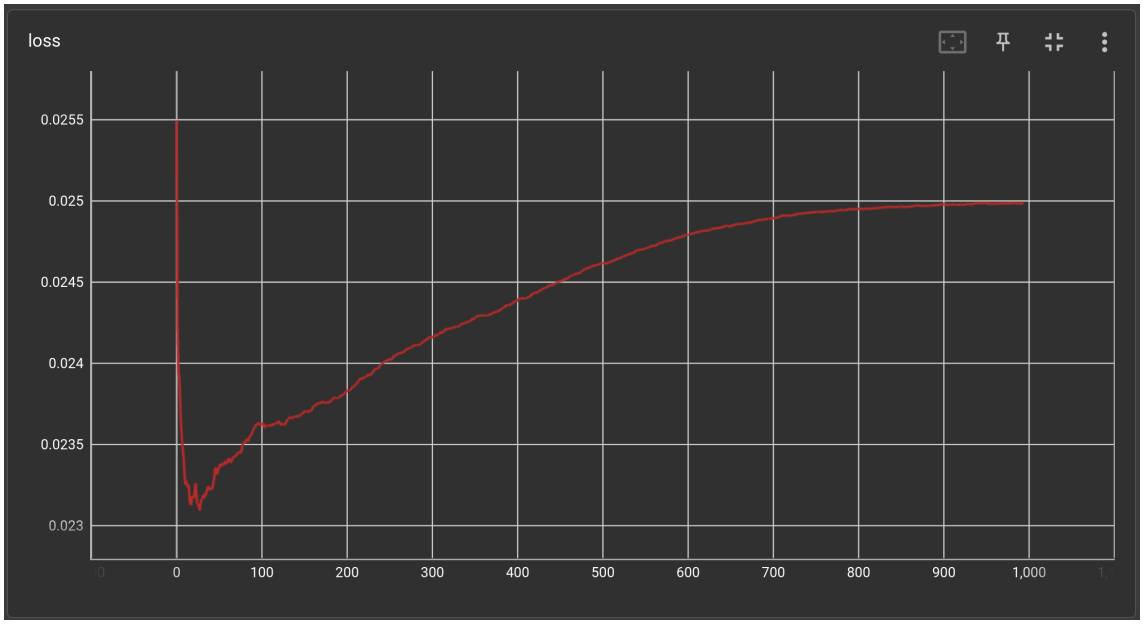

Figure 6: single-agent loss

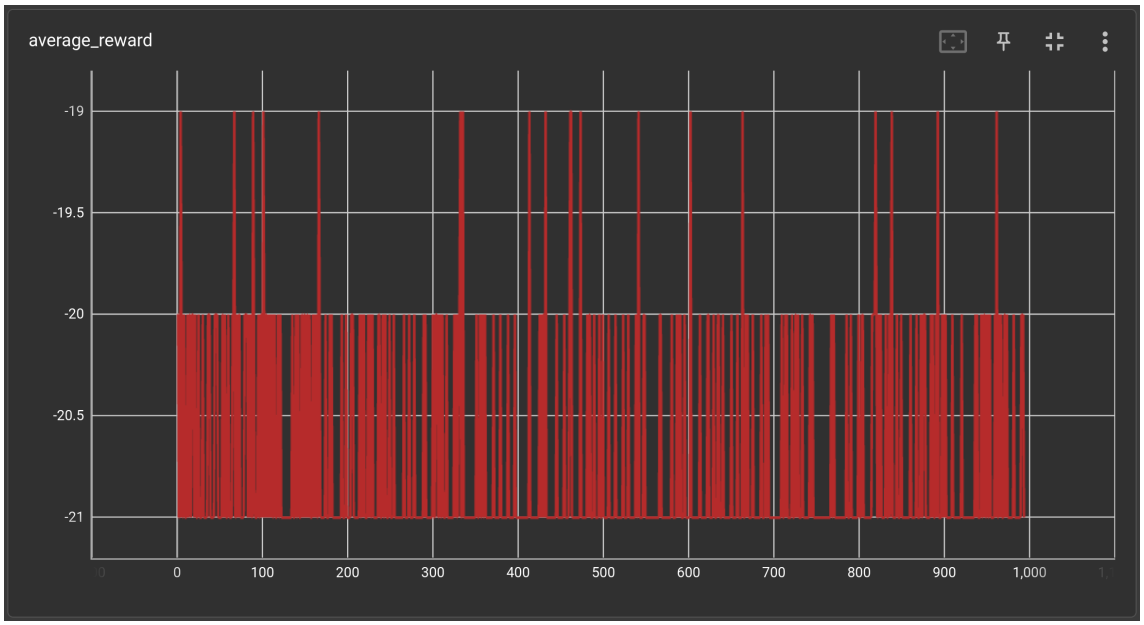

Figure 7: single-agent reward

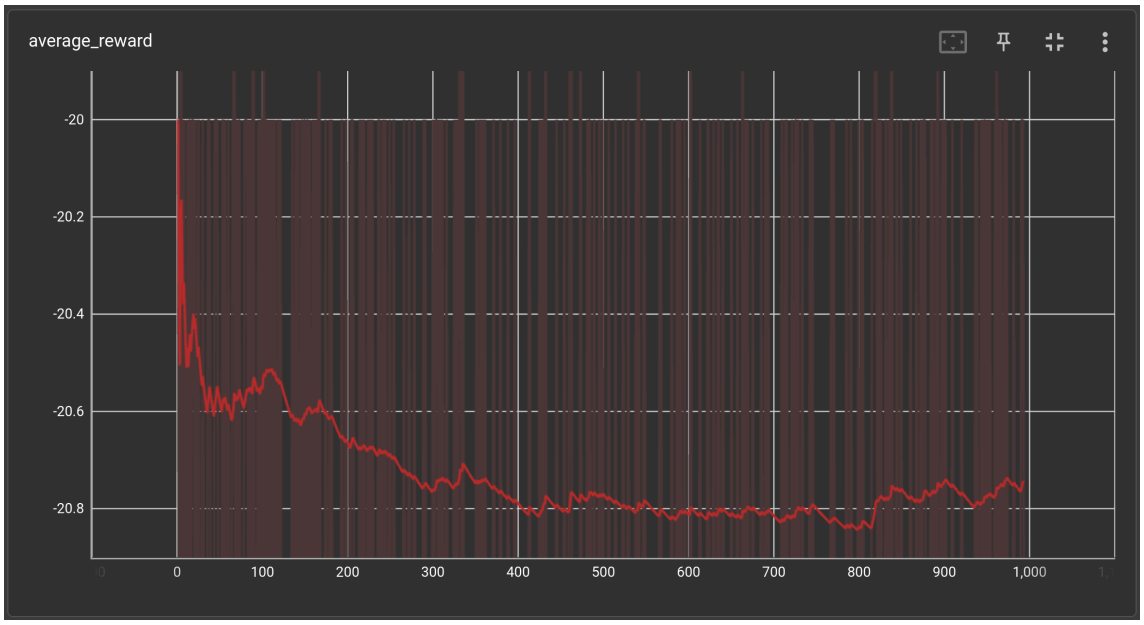

Figure 8: single-agent reward smoothed by the factor 0.99

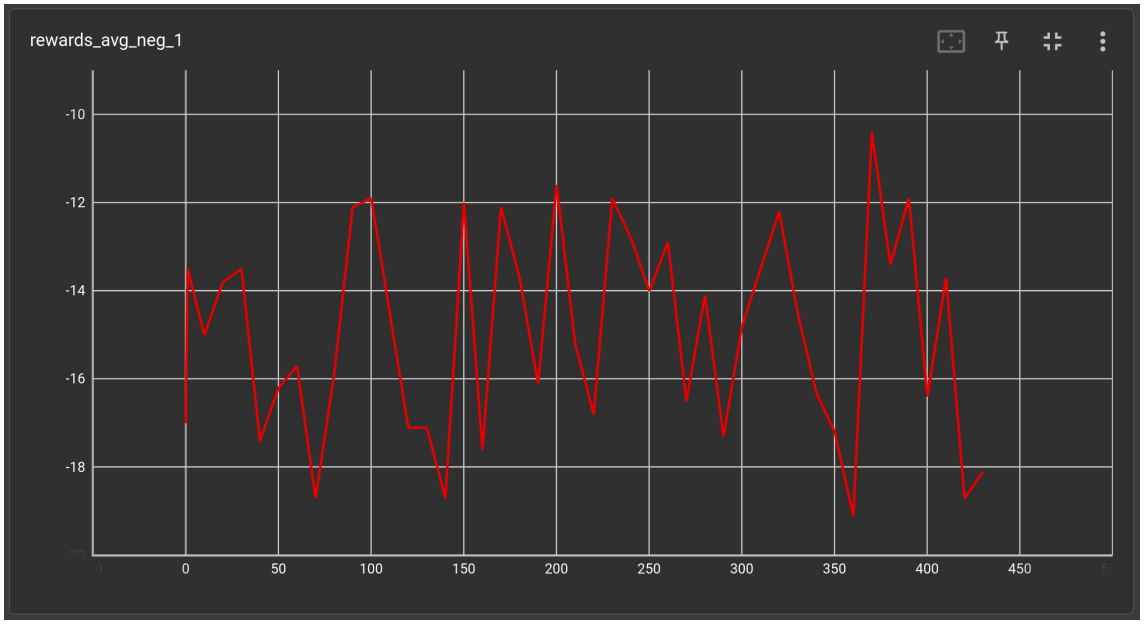

Figure 9: average negative reward agent one

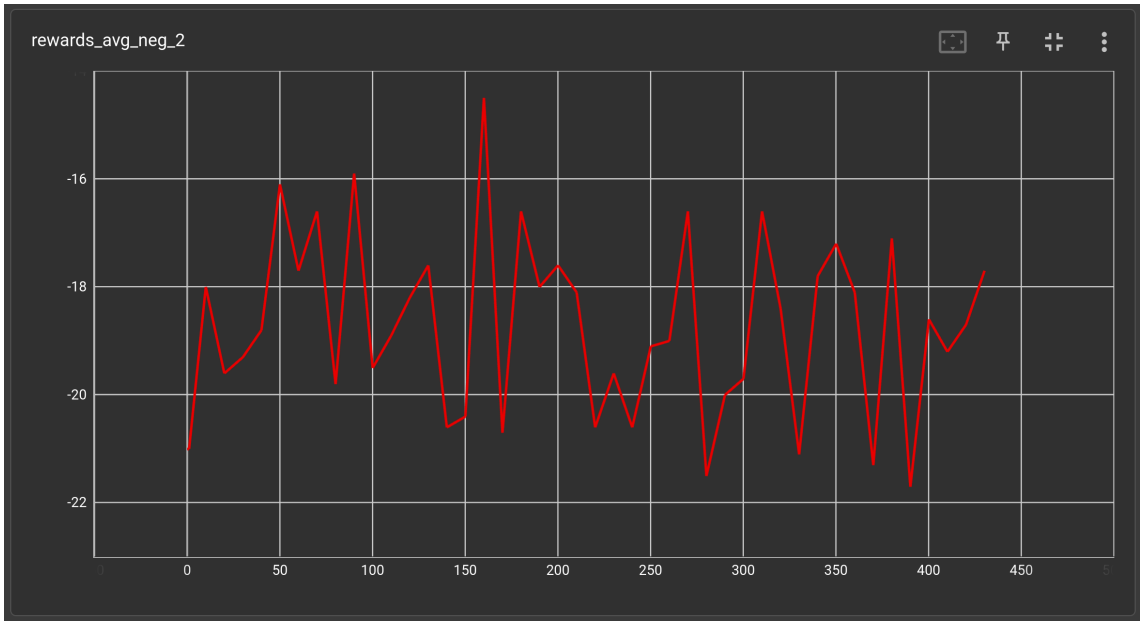

Figure 10: average negative reward agent two

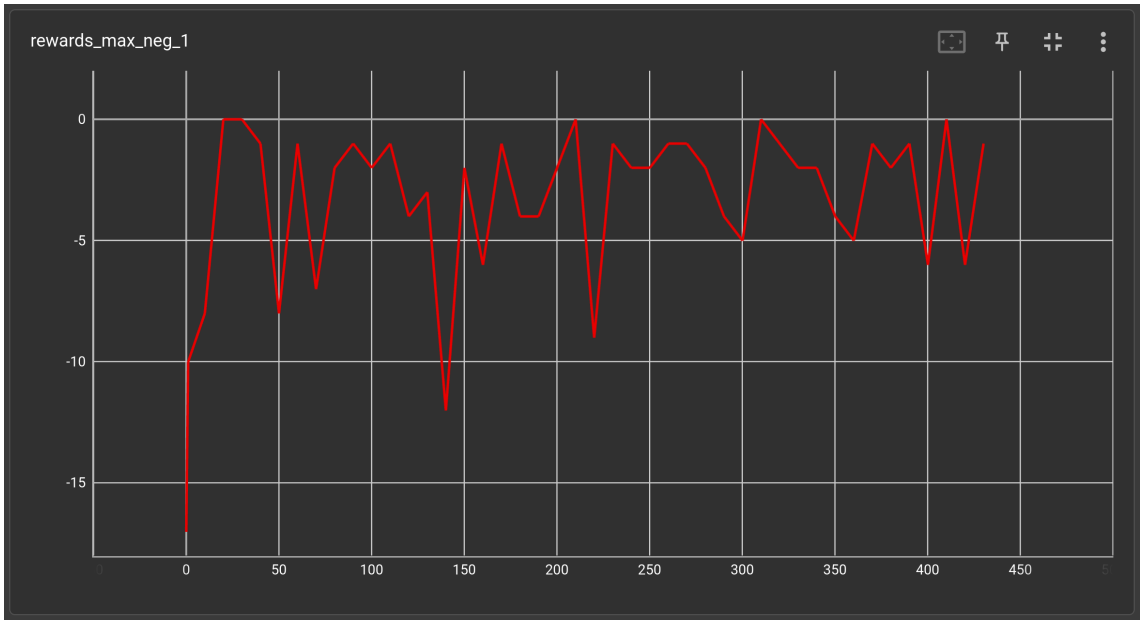

Figure 11: maximum negative reward agent one

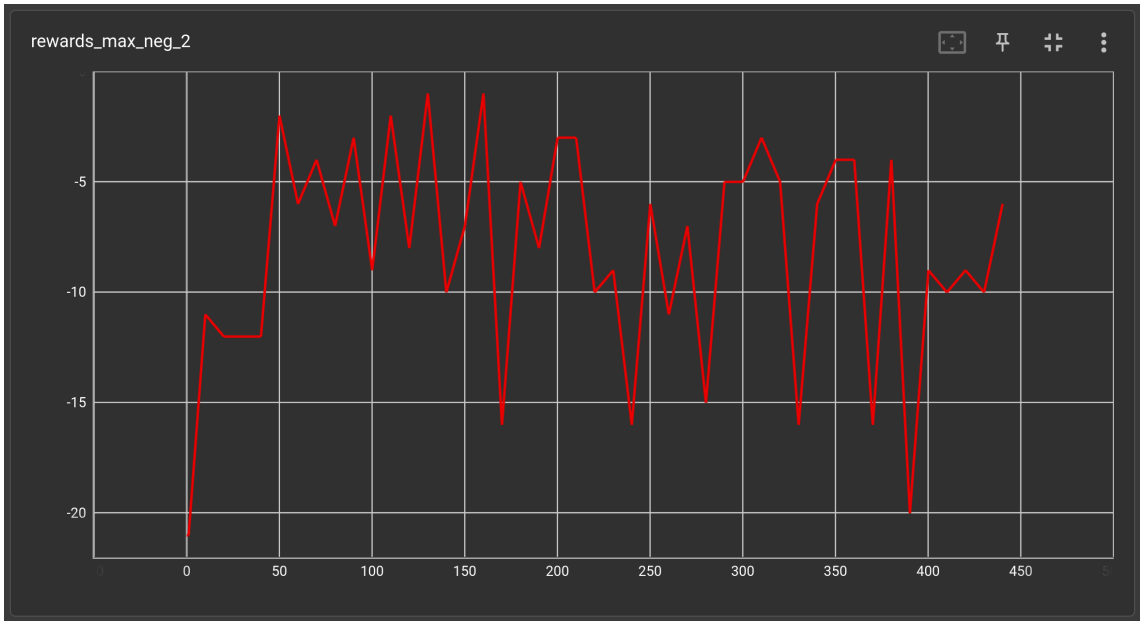

Figure 12: maximum negative reward agent two

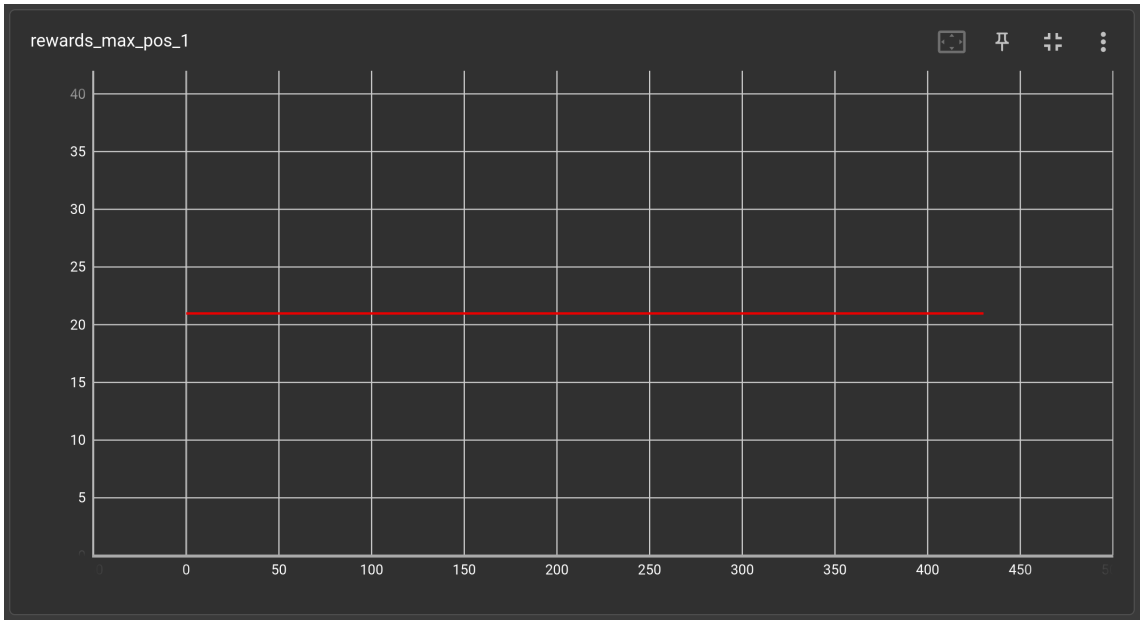

Figure 13: maximum positive reward agent one

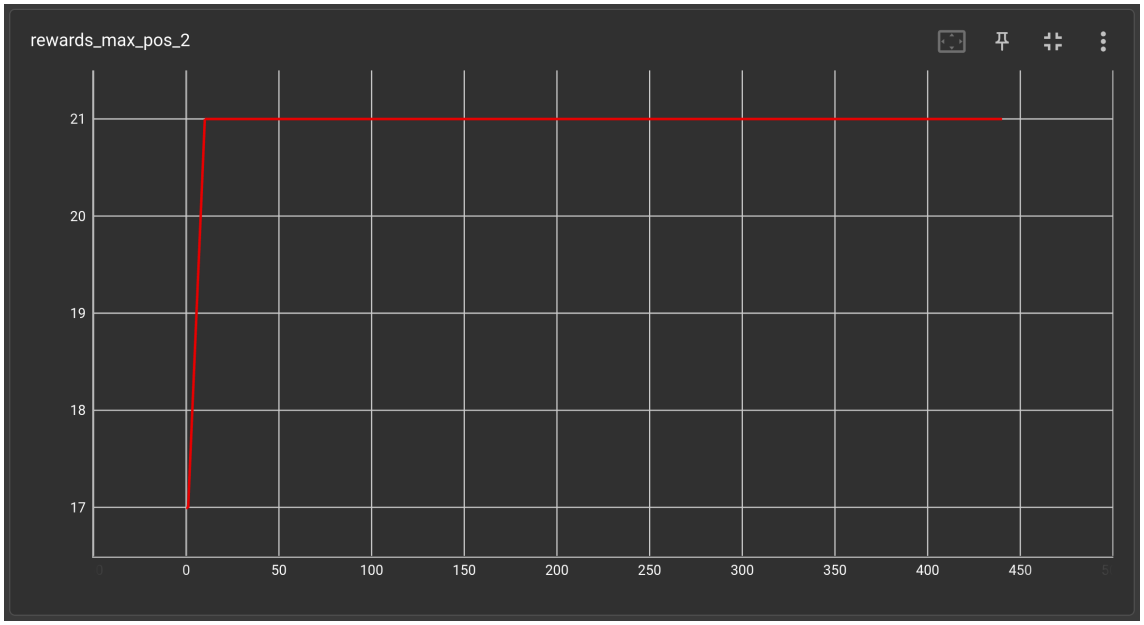

Figure 14: maximum positive reward agent two

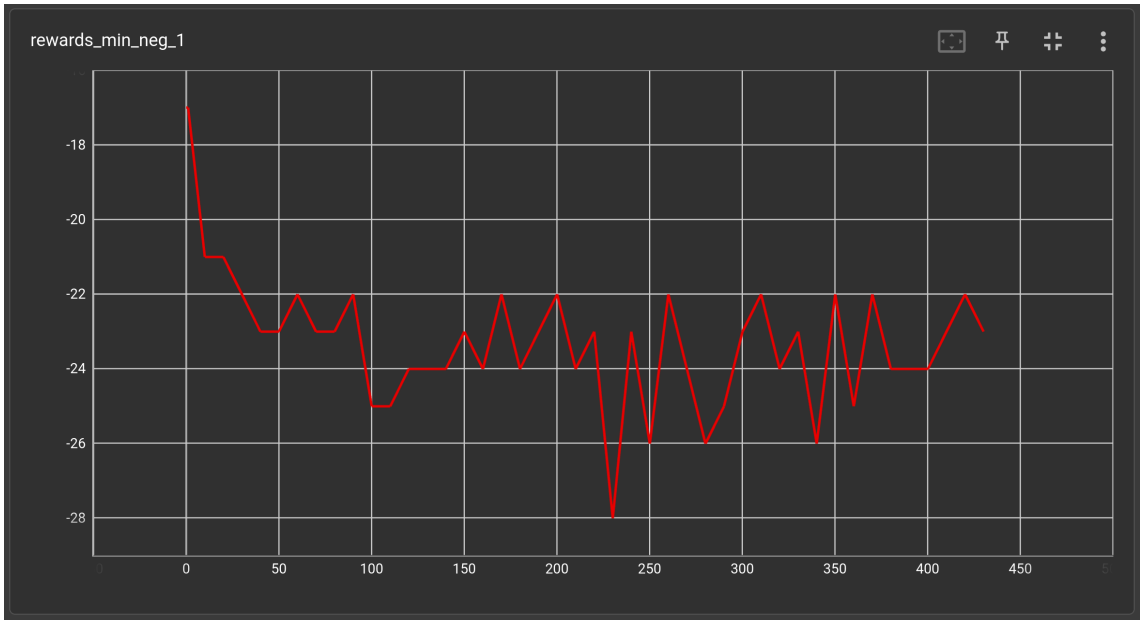

Figure 15: minimum negative reward agent one

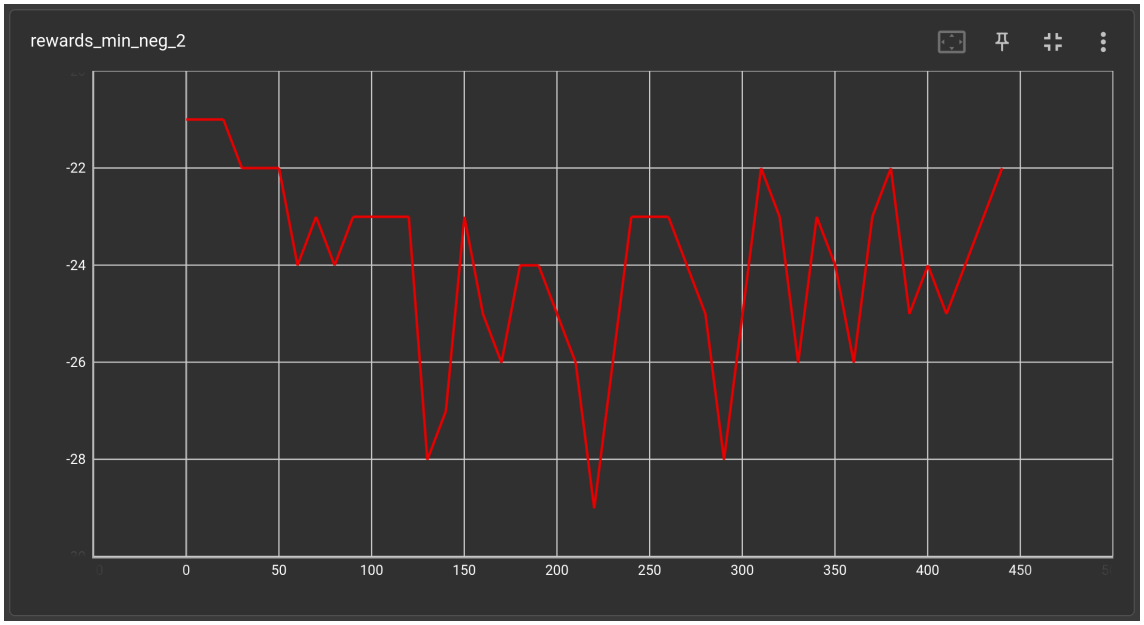

Figure 16: minimum negative reward agent two

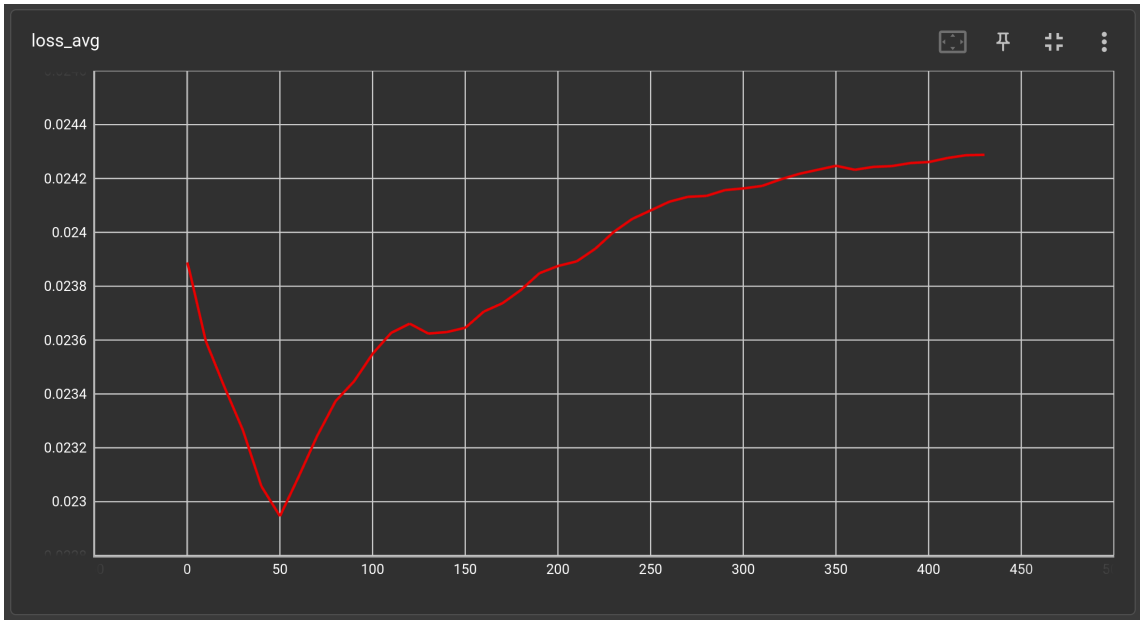

Figure 17: MARL loss