Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Byzantine Reinforcement Learning

Master's Thesis

Lazar Rakic

lrakic@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Darya Melnyk, Oliver Richter
Prof. Dr. Roger Wattenhofer

October 29, 2020

Acknowledgements

First of all, I would like to thank my supervisors Darya Melnyk and Oliver Richter for their guidance throughout the course of this Master's thesis. I was really fortunate to have two supervisors with slightly different backgrounds, which provided me with the ability to learn a lot and improve my skills in diverse disciplines. They have invested a lot of time in our weekly meetings, and have always provided effective comments and interesting ideas. Furthermore, I am really grateful that Prof. Dr. Roger Wattenhofer hosted me in the Distributed Computing Group at ETH Zurich and enabled me to write my Master's thesis. I would also like to thank all the people that supported me throughout this time. This goes specifically to my family, that has always been there for me when I needed them. Thank you all.

Abstract

The byzantine agreement problem has been extensively studied in the literature since the 1980s, and gained even more attention with the emergence of blockchain structures. Of particular interest is the byzantine agreement in the asynchronous fully-connected network, without cryptographic assumptions. The proposed algorithm for solving this problem by Ben-Or [1] in 1983 has expected exponential communication time, and this result has not yet been significantly improved. On the other hand, reinforcement learning has proven very successful in scenarios where agents have to learn complex strategies through self-play [2], [3]. In this project, we present the novel approach of combining these two exciting fields – we are using reinforcement learning to solve the asynchronous byzantine agreement problem through self-play. In our setup, byzantine adversary has access to all the messages in transit, and can hide the values of some nodes from some of the correct nodes, while revealing them to the other correct nodes. The goal of the correct nodes is to reach agreement in the smallest number of rounds, and the goal of the byzantine adversary is to disturb this process. We have modelled this setup as a zero-sum game, and created an environment in which the correct and byzantine agents interact and receive rewards. We have then employed reinforcement learning algorithms and analysed the learned strategies. Finally, we argue that reinforcement learning can be successfully used in the asynchronous byzantine agreement problem and might lead to a better algorithm in the future.

Contents

Chapter 1

Introduction

1.1 Motivation and background

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing. When the malicious participants are also present besides the correct ones, the main objective of distributed computing is to develop a system that is resistant to malicious behavior. The goal of the malicious nodes is to prevent the correct nodes from reaching agreement. These malicious participants are called "Byzantine". They possess a wide array of actions where they can arbitrarily manipulate the messages they send. If the communication between the nodes is maintained through private channels, it is easier to solve asynchronous byzantine agreement efficiently [4]. However, we are interested in the worst-case scenario in the system, where the byzantine parties can see all the messages flowing through the communication channels. Therefore, we will be examining the case with public channels, and the byzantine adversary can access all messages. We should also note that the channels are authenticated, meaning that each node sees exactly where the message comes from and that the byzantine parties cannot impersonate other nodes. That means that every node knows at every point in time which node sent which message. In the case when the byzantine could also forge node addresses, agreement would not be possible. The first algorithm to solve Byzantine agreement within the asynchronous communication model was proposed by Ben-Or [1] in 1983. This algorithm relies on randomness to accomplish agreement, and it has expected exponential running time. Since then, there exist no algorithms solving asynchronous byzantine agreement without cryptographic assumptions more efficiently. In the recent work of King and Saia [5] from 2016, it was shown that the expected polynomial communication time is achievable. However, there are some doubts regarding the correctness of this result [6].

An algorithm that needs expected exponential running time is not fast enough for solving the asynchronous byzantine agreement problem with many participating parties. Since the Ben-Or algorithm cannot be easily extended to be faster, a different approach is necessary. Most probably, an algorithm that we would

be looking for would have to either ask the nodes to locally give weights of how much they trust other nodes, or it would ask the nodes to remove the byzantine parties locally from their views. However, such strategies might be quite hard to be analysed theoretically. We have decided to use a self-play scenario where we would train correct and byzantine agents competing against each other using reinforcement learning algorithms. If the strategies prove successful, we would have hope that we can interpret the behavior theoretically and derive the algorithm for the case with more nodes.

1.2 Related literature

Byzantine agreement was first introduced by Pease, Shostak, and Lamport [7], 40 years ago. They considered a fully-connected network of participants and a synchronous communication model, which means that the nodes are communicating in discrete rounds. They showed that byzantine agreement is impossible to solve if at least one third of the parties are byzantine. Shortly after, Ben-Or in [1] considered an asynchronous communication model, meaning that the nodes cannot differentiate between slow and crashed nodes in the system. Consequently, the participant that does not receive a message cannot determine whether the message was not sent or takes too long to arrive in the particular channel. Ben-Or has shown if each node has randomness manifested through a random coin, then byzantine agreement would be possible in such a system. Following this work, Paterson, Lynch, and Fischer [8] showed that byzantine agreement is actually not possible in an asynchronous communication system if the nodes do not possess some form of randomness. It would already be enough to have only one byzantine party present in the system and the agreement could never be accomplished. Many results for the synchronous setup exist in the literature, but the asynchronous setup solutions are scarce. After the result of Ben-Or, whose algorithm works in the case $f < n/5$, Toueg and Bracha [9], Bracha [10], and Mostefaoui et al. [11] use the improved tolerance of $f < n/3$, where f is the tolerated number of byzantine nodes in the system with n participating nodes. Some time later, King and Saia [5] have proposed a better solution. Nodes can exchange many random coins, and using reliable broadcast they assimilate the local views, and byzantine actions would be revealed through a statistical deviation following the central limit theorem. However, in their results the number of tolerated byzantine nodes compared to the number of the correct nodes is too small, and therefore cannot be used in the simulations. It is also noted in [6] that their solution might not hold in the general case, and that byzantine values and byzantine scheduling strategies form together necessary conditions to prevent the asynchronous byzantine agreement from terminating within expected polynomial running time.

In the previous work in the semester project [12] in the distributed comput-

ing group, reinforcement learning was used to enable the byzantine party to learn the best strategy against already existing algorithms in synchronous and asynchronous setups. It was also shown in recent works of DeepMind and OpenAI in [2], [3] and [13], that the reinforcement learning agents through self-play sometimes learn sophisticated strategies. Through learning their strategies by simply playing against each other, in our scenario, the correct nodes could also learn the counter strategy to the byzantine strategy. Therefore, it would be intriguing to construct this setup and observe what strategies do the opposed parties develop.

1.3 Contributions

Our goal in this Master's thesis was to find out whether there exists an efficient asynchronous byzantine agreement algorithm that could be established through self-play. So far, the result has not yet been found theoretically, since the analysis for this setup is particularly complicated. The only advantage of correct nodes in algorithms was to choose randomly and surprise the adversary, but that was not time-efficient. In this work, we have constructed a novel setup suited for reinforcement learning, where we were able to learn algorithms for solving the asynchronous byzantine agreement problem. In Chapter 3, we explain how we have modeled the problem as a zero-sum game and created an environment in which the correct and byzantine agents interact and receive rewards. We have analysed the learned strategies and tested them versus fixed policies, and used exploiter and cached strategies to improve the results, and to expose any flaws. Our findings bring us closer to finding an efficient algorithm that would solve this agreement problem completely, and we present all our results in Chapter 4. Furthermore, our results show that it is possible to use reinforcement learning as a tool to teach the correct nodes the optimal asynchronous byzantine agreement algorithm, and could prove useful in the future, combining its forces with the theory.

Chapter 2

Preliminaries

2.1 Byzantine agreement

Following the theory in [14], [15], and [16], a system with n nodes, among which $f < n/3$ nodes are byzantine, where the goal of the remaining $n_c = n - f$ correct nodes is to agree on a common value, is a byzantine agreement problem. Furthermore, each pair of nodes is connected with an authenticated channel. That means, for every message sent through this channel, the receiver always knows who sent the message. We further assume that channels are public, meaning all the messages that are passing through them can be read by the byzantine party. The communication model is asynchronous, meaning that the nodes cannot differentiate between slow and crashed nodes in the system. Every correct node starts with a local input value, and correct nodes have to decide on one of the input values, satisfying:

- Agreement: all correct nodes have to decide on the same value;
- Termination: all correct nodes must terminate in a finite number of steps;
- All-same validity: if all correct nodes start with the same input value, they must decide on that value.

Byzantine nodes are malicious nodes whose objective is to prevent the correct nodes from reaching an agreement. Byzantine nodes can either not send any messages, send different messages to different nodes, or communicate false input values. The adversary can also control the message scheduling. If it is controlled, then the byzantine scheduler has the power to delay all messages passing through channels for an arbitrarily long time. The behavior of the byzantine party can be restricted by forcing them to send the same message to all the nodes or no message at all, by implementing reliable broadcast [10]. This is a considerable limitation for the byzantine node since it cannot try to deceive the correct nodes by sending different values to different nodes.

2.2 Reinforcement learning

Reinforcement learning [17] is a powerful tool used to analyse environments that are described by the Markov Decision Processes (MDP). The agent observes state $s_t \in \mathrm{S}$, and then takes an action $a_t \in \mathrm{A}$, an receives a reward $r_t \in \mathrm{R}$. We will consider discrete time steps $t = 0, 1, 2, 3, ...$ The agent acts according to its policy $\pi(a|s_t)$ which is a conditional probability distribution over the actions, given the current state s_t. The goal of an agent is to maximise the expected reward given by $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1}$, where γ is the discount factor and $0 \leq \gamma \leq 1$.

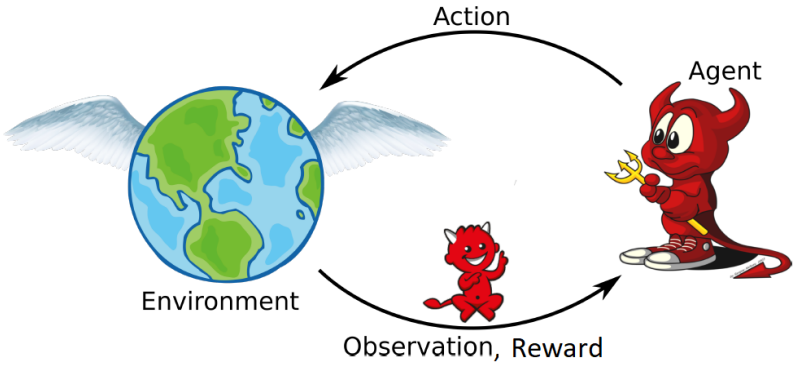


Figure 2.1: Agent interacts with the environment.

The approach we propose in our work is independent of the used reinforcement learning method. We have decided to use Proximal Policy Optimisation (PPO) algorithm [18] because it worked well in our experiments.

Chapter 3

Algorithm Setup

3.1 Our Setup

Our algorithm setup depicts a distributed system that consists of four nodes, which asynchronously communicate with each other through authenticated public channels. There are three correct nodes, which we will call *corr_ 0, corr_ 1, corr_ 2*, and whose goal is to reach agreement in the smallest number of rounds. However, there is also a byzantine adversary, *byz_ 0*, who should prevent them. Their node IDs are respectively $0, 1, 2$, and 3. In our algorithm, every node can broadcast a message *(node ID, round, input value)*.The input value takes binary values of $+1$ or -1. Motivated by the work of [5], the nodes in our setup write their messages to the matrix that we will also call the blackboard matrix. We are encoding nodes' messages in this structure, with rows corresponding to the current round, columns corresponding to the node IDs, and matrix entries corresponding to the input values.

$$
\begin{array}{c}
 \\
\textit{round\_0} \\
\textit{round\_1} \\
\textit{round\_2} \\
\textit{round\_3}
\end{array}
\begin{array}{c}
\begin{array}{cccc}
\textit{corr\_0} & \textit{corr\_1} & \textit{corr\_2} & \textit{byz\_0}
\end{array} \\
\begin{pmatrix}
-1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 \\
-1 & 1 & -1 & 1 \\
\textbf{-1} & \textbf{-1} & \textbf{-1} & 0
\end{pmatrix}
\end{array}
$$

Since some of the values in the blackboard might be hidden by the adversary, as explained in the next section, our motivation was to enable nodes to access the past messages. That way, the difference in local views between different nodes would get smaller and hopefully, this can make them stronger and able to reach agreement earlier.

3.1.1 Our adversary – Byzantine agent

The byzantine adversary controls (less than) one-third of all the participating nodes. Through adversarial scheduling, it has the power to delay a message sent from some node, so we can assume that it hides the value of one node in a round. Furthermore, we can say that it chooses which correct nodes may see the value, and which may not. In the following round, the byzantine node has two options. It can choose to hide the same node again, and that node's input value then remains hidden from all other nodes – it is delaying this node's messages. It can choose to hide another node, and the previously hidden node's values get revealed – now it is delaying the other node's message. In this case, it chooses again which correct nodes see the value, and which not, since it chooses when the message gets delayed for each of them. Such a configuration is reasonable because, with the reliable broadcast, the byzantine node cannot send different input values to different nodes as these values would not get accepted. A hidden value is represented in the nodes' local blackboard view with a value 0. Therefore, values that are seen by nodes in the blackboard matrix are $\{-1, 0, 1\}$, where -1 and 1 are input values, and 0 is a hidden value.

3.1.2 Our adversary – Dealer agent

We have decided to add an additional agent to our setup – a "Dealer" that chooses the initial values for the correct nodes, instead of them being randomly chosen within the environment. The main idea is that the dealer agent can choose the distribution of the initial values such that the correct nodes have more difficulty agreeing. Therefore, the dealer agent can be considered as a part of the adversary team, meaning it shares the common objective with the byzantine node.

3.1.3 Caching policies and exploiters

To avoid the agents overfitting their policies to just counter over-specific opponents' strategies, and make the learned strategies more general, we can use "cached" policies [2], [13]. These are the policies from earlier in the training process, which are saved in the policy pool and sampled with some probability. Then, the agent is effectively playing against numerous agent instances and is not converging to a narrow strategy.

Another question that arose during the course of our training was the following: How could we possibly know whether the learned policies are optimal and whether there exist some better policies that might be maybe also easier to learn? An interesting approach would be to utilise the "exploiter agents", whose objective is to expose the flaws of the main agent, and thus make it stronger [3]. Therefore, we have decided to implement exploiter policies in our setup, where we improve the learned policies by fixing one of them for one agent and letting the

other agent learn the new policy from scratch, seeing if the new policy achieves better results than the old one. If the new policy is better, then it can also be added to the policy pool and be sampled with some probability, like cached policies.

This architecture can be implemented like in Algorithm 1. For a pre-determined number of phases, or until the policies are no longer exploitable, we perform learning through self-play for N iterations, saving cached policies at certain iteration steps. Each iteration contains a certain number of timesteps, where stochastic gradient descent is performed on given mini-batches to train the network. Policies are sampled from the policy pool, with given probabilities of choosing cached or exploiter policies. Then, through M iterations byzantine and correct policy try to exploit each other. If they succeed, then their policies deserve to be added to the policy pool, and the learning continues.

Algorithm 1: Caching policies and exploiters

```
phase = 0
repeat
    for iter = 1, 2, ..., N do
        train ppo_agent with SGD
        if iter = caching_iter then
            add cached_policy to policy_pool
        end
    end
    for iter = 1, 2, ..., M do
        train ppo_exploiter with SGD
    end
    if exploiter_policy better than latest_ppo_agent then
        add exploiter_policy to policy_pool
    end
    phase = phase +1
until agent is not exploitable anymore or the maximum number of
  phases is reached;
```

3.2 Environment interface

3.2.1 Action spaces

1. **Action space of the correct nodes** has 4 actions. Two of them are for sending input value of -1 and 1, respectively. The other two actions – locking actions are used when they decide to terminate on -1 or 1. After they decide to terminate, they cannot change their input value in further rounds until the environment has terminated. However, this information is

not explicitly communicated to the other nodes; the only way for the node that has not terminated to learn that the other node has terminated is to observe the values in the given column; our implementation is such that the node that terminated always repeatedly writes the same decided value in the following rounds to the blackboard matrix. Complete correct nodes' action mapping can be found in Appendix B.

2. **Action space of the byzantine node** is composed of:

   (a) an input value of -1 or 1,

   (b) choosing which out of n nodes is hidden,

   (c) and which of the n_c correct nodes see the hidden node in the current round ("hiding mask").

   The *hiding mask* determines which of the correct nodes would see the hidden value in the round the value is hidden, and there are 2^{n_c} possibilities because each of the correct nodes may or may not see the value, which are permutation with repetitions (choosing n_c times 2 values). This gives us $2 \times n \times 2^{n_c}$ possible actions and for $n = 4$, $2 \times 4 \times 2^3 = 64$ discrete actions for the byzantine agent. Complete byzantine agent's action mapping can be found in Appendix C.

3. **Action space of the dealer agent** The action space of the dealer agent consists of all the possible combinations of the initial values of the correct nodes. Since the starting initial values are the binary values of -1 or $+1$, the dealer agent's action space size is therefore 2^{n_c}, and it is a map to all the possible permutations with repetition of -1s and 1s.
   Another question that arose with the addition of a new agent was how to implement the shuffling of the node IDs, i.e., how to choose which columns correspond to which of the nodes, and where is the byzantine node column located. In our problem, it is crucial to have a varying column ID for the byzantine agent, so that the correct nodes cannot learn where the malicious values are coming from. The columns of the correct nodes do not have to be shuffled because correct nodes have their own local views of the blackboard, and there is no difference if, for example, *corr_0* sees what *corr_1* would see, and vice versa. Consequently, what we needed to somehow implement, was the adaptive choice of the byzantine node column in the blackboard.
   We have decided to experiment with giving the dealer node the power to choose the byzantine node column, and with choosing the byzantine node column randomly within the environment.
   This gave us two possibilities for the dealer agent's action space:

   - **Dealer agent does not place the byzantine column**
     The action space in this case is just 2^{n_c}, and the placement of the

byzantine column is handled randomly within the environment, on one of n available blackboard columns;

- **Dealer agent places the byzantine column**
  This increases the action space of the dealer agent to $n \times 2^{n_c}$, where 2^{n_c} are all the possible permutations with repetition of the initial values for the correct nodes, and n stands for choosing one out of n columns that would belong to the byzantine node in the blackboard.

Complete dealer agent's action mapping can be found in Appendix D.

3.2.2 Observation spaces

1. **Observation space of the correct nodes** is a matrix with the number of rows equal to the maximum number of rounds, and the number of columns equal to the number of nodes, representing a local view of the blackboard matrix for the correct nodes. In this local view, some of the values might have been hidden by the byzantine adversary.

2. **Observation space of the byzantine node** is a matrix with the number of rows equal to the maximum number of rounds times the number of nodes, and the number of columns equal to the number of nodes. The byzantine node is given the observation of a true blackboard matrix, as well as all the correct nodes' views of the blackboard matrix from the previous round, thus possibly helping it to track correct nodes' statistics.

3. **Observation space of the dealer agent** Since the dealer agent does not receive any input, i.e., it needs to converge to the distribution of the initial values to maximize the byzantine node's reward, we have implemented the dealer agent as a multi-armed bandit, meaning that it is just learning a bias that would maximize the reward.

3.2.3 Reward system

For each elapsed round in the environment, i.e. for each action they execute, byzantine nodes receive a reward of +1 and correct nodes receive a penalty of -1. However, correct nodes may not all terminate at the same time, meaning not all the correct nodes spend the same number of rounds in one environment instance. Their time is calculated and saved, and then after the environment termination, they are assigned the remainder of the success reward/penalty. If they are successful and are agreeing on the same value, without violating the all-same validity condition, their penalty will be the number of played rounds of the last correct node to decide (correct node that spent the most time in the given episode).
If they are not successful, then they receive a fixed penalty which is equal to the

maximum number of playable rounds, i.e. we are implementing the same penalty for all the failures:

- violating all-same validity condition,
- not agreeing,
- reaching the maximum number of rounds.

The dealer agent receives the same reward as the byzantine node; in our implementation, they are on the same "team". It is the first agent that chooses the action after the environment is reset and then waits until the environment terminates, and then receives the same reward as the byzantine node, meaning high values for the failure cases and higher values for longer episodes.

3.2.4 Environment rollout and termination

At the beginning of an environment episode, each of the correct nodes gets assigned an initial input value, which is either chosen by the dealer agent or chosen randomly. Afterwards, the byzantine agent receives these initial values in the form of observation and chooses its action. From this point on, correct and byzantine nodes act alternately receiving observations and choosing actions.

The environment is terminated when either the maximum number of rounds is reached, or when all of the correct nodes decide to terminate. A penalty equal to the number of remaining steps in the episode is given if nodes terminate without agreement.

3.3 Additional actions

Following the work of [10], we were motivated to experiment even further and see how the learned strategy-counter-strategy pairs would change with additional communication between the nodes, namely, by having additional values that inform the nodes whether another node is getting ready to terminate on some value.

3.3.1 Fixed strategy of the correct nodes

We use the Ben-Or as a baseline to test the byzantine agent. The algorithm is from [10], which tolerates $f < n/3$ byzantine nodes.

3.3.2 Byzantine and correct agents with additional actions in self-play

Following the ideas from the literature, we have decided to also encode these actions as either -2 or 2, written to the blackboard matrix; we justify this idea by stating that these actions might create a larger bias in the views and majority value statistics, thus sending the signal to other nodes that they might be also better switching to either more positive or more negative values, in order to successfully approach the termination. The action spaces were therefore modified in the following way:

- correct nodes now have 6 instead of 4 actions;
- byzantine nodes now also choose two additional input values of -2 and $+2$, meaning that their action space now doubled:

  $$4 \times n \times 2^{n_c},$$

  which in our example with $n = 4$ nodes would give a total of 128 instead of 64 possible actions.

More details about the actions are given in Appendix B and C.

3.4 Byzantine Adversary Controls Multiple Nodes

So far we have only observed the case with the byzantine adversary controlling only one node since the maximal number of nodes was less than 7. Our idea during this thesis was to first start with the smallest example, and then move to the higher dimension space with more nodes, where we would also find more of the correct, as well as byzantine nodes.

3.4.1 Action space of the Byzantine Adversary

The biggest problem arises with the dimension explosion of the action space of the byzantine adversary when it controls more than one node. With more byzantine nodes, the action space is given by:

$$2 \times n \times 2^{n_c f},$$

or

$$4 \times n \times 2^{n_c f},$$

where as before n is the total number of nodes, n_c is the number of correct nodes and f is the number of byzantine nodes. Here the first term is either 2 (input

values -1 and 1) or 4 (input values -1, 1, -2, and 2). and stands for the possible input values of the byzantine adversary. The second term n stands for the number of possibilities to choose a node to hide, and $2^{n_c f}$ stands for the possible values of the hiding masks (f hiding masks of 2^{n_c} for each byzantine node).
We can already calculate the action space for the minimal example with two controlled nodes, $n = 7$, $nc = 5$ and $f = 2$:

$$2^2 \times \binom{7}{2} \times (2^5)^2 = 86016,$$

which is already difficult to use in our setup and exceeds the used architecture capacity.

3.4.2 Implementation of two byzantine nodes

Our implementation that we show in the results section was the following:

- there is one byzantine adversary, i.e. one agent in the reinforcement learning setup is controlling two byzantine nodes, and hiding two nodes per round;
- no hiding masks were possible – the byzantine adversary only chooses which node to hide; it is being hidden from all the other nodes immediately, losing the local view difference;
- no dealer node is used, due to the issue that all-same validity has low sample probability.

Chapter 4

Results

We are interested to see whether the correct nodes learn a policy in a deep reinforcement learning framework to reach agreement. Furthermore, we are curious to see whether through self-play in a multi-agent environment setup, the byzantine adversary also learns a counter-strategy to successfully prevent them. In some of the results, we were not using cached policies, since we have achieved more stable and robust learning without deploying them, for the given configuration. However, we have tested how good the learned strategy-counter-strategy pairs were, and if the learned strategies could be exploited. We would stop the training once the learned strategies could not be exploited anymore, finalising our results. We would also like to note that due to the randomness and initial conditions with the used architecture, simulations with the same parameters might slightly differ. We have tried to find the most stable learning setup for each case. The chosen simulation parameters for every experiment are given in Appendix E.

4.1 Without dealer agent

Learning process

In this section, we will present the results from one of the successful hyperparameter combinations. To address the possible randomness issue with having randomly-assigned initial values for the correct nodes, we have repeated the simulation multiple times and we got similar results. In the following Figure 4.1 we can see the evolution of episode length. In Figure 4.2 we see the evolution of the mean policy reward for the nodes, over the course of 200 training iterations, where each iteration has 8192 timesteps. We have observed in this experiment that the convergence was pretty smooth.

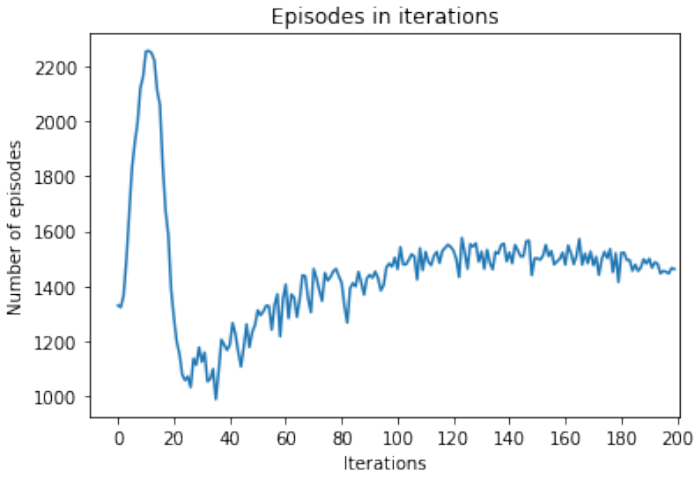


Figure 4.1: Episodes in iterations.

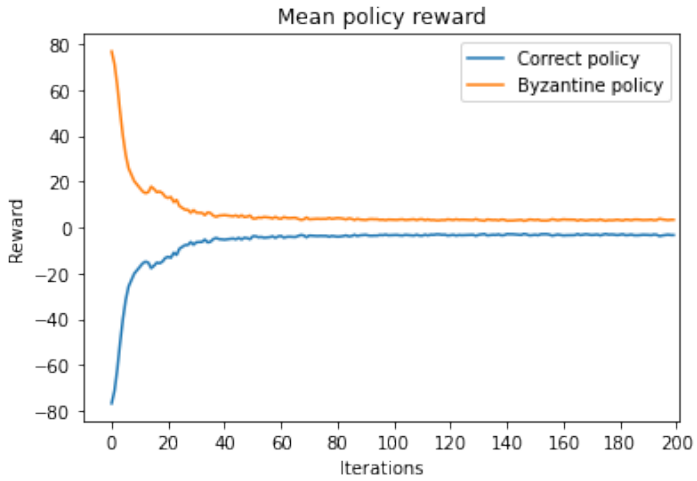


Figure 4.2: Mean policy reward.

Statistics overview

For our simulation, we have also calculated the extended statistical information linked to the learned strategy-counter-strategy pairs of the byzantine and correct nodes in the self-play configuration. The statistic could be divided into four groups, based on whether the actions are dependant on each of the eight initial starts, or on the environment round, i.e. whether the action was chosen in $1st$, $2nd$, $3rd$, or $4th$ round (and onwards), giving us:

- initials-dependent and round-dependent,

- initials-dependent, summed over rounds,
- round-dependent, summed over initials,
- summed over rounds and over initials.

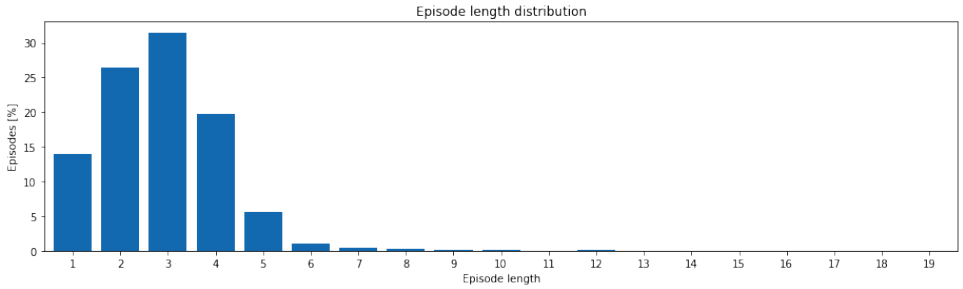


Figure 4.3: Episode length distribution.

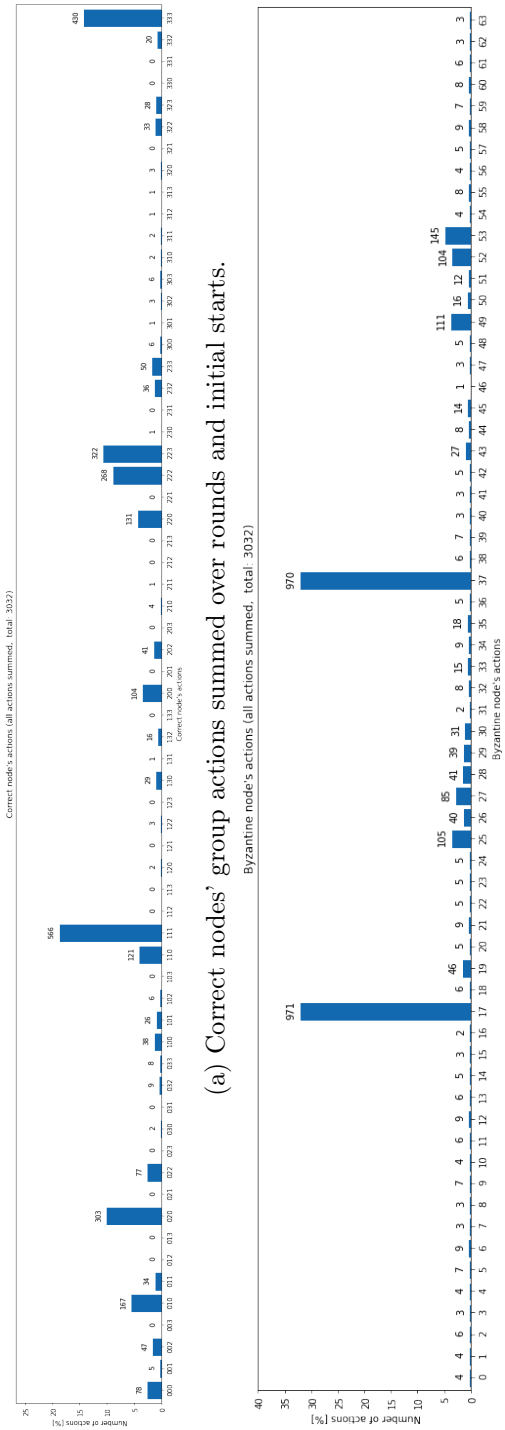
The episode length distribution in 1000 environment test rollouts is given in Figure 4.3, with the average episode length being around three rounds. We can see in Figure 4.4 that the correct nodes go through a lot of uncertainty before deciding. There are quite a few peaks with the frequent group actions of the nodes, besides the peaks for deciding action groups – $[1, 1, 1]$ and $[3, 3, 3]$ (two highest peaks). Furthermore, we can see that byzantine adversary has mainly converged to choosing two actions, where it chooses input values of -1 and 1, respectively. The failure report in this case is provided in 4.1. It is also interesting

Failure report	% **of games**
All-same validity breach of $(1, 1, 1)$	0
All-same validity breach of $(-1, -1, -1)$	0
Not agreeing	0.4
Max-round reached	0
Success	99.6
Average number of rounds	3.03

Table 4.1: Failure report for 1000 environment rollouts.

to observe initials-dependent, summed over round statistics from the simulation results. Here we can analyse how the nodes change their strategies for different initial values. In Figure 4.5, we can observe how the byzantine chooses the action that is actually opposed to the majority value seen for the correct nodes.

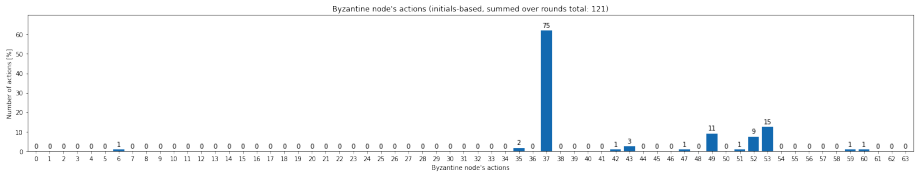
We can observe that the nodes switch strategies and that there are some majority views, for example, the byzantine adversary is clearly trying to steer the local views of the correct nodes towards similar numbers of -1s or 1s, as seen previously. Correct nodes almost approach the success rate of 100%, but there were always some environments that terminated with nodes not agreeing.



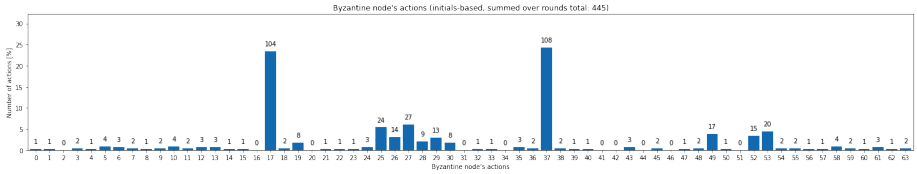
(a) Correct nodes' group actions summed over rounds and initial starts.

(b) Byzantine nodes' actions summed over rounds and initial starts.

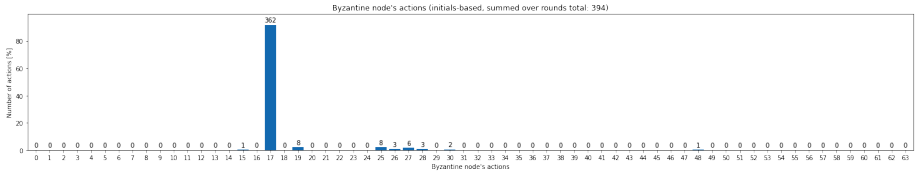
Figure 4.4: Total distribution of nodes' actions in the setup without the dealer agent.



(a) Byzantine nodes' actions when the correct nodes start with $[-1, -1, -1]$.



(b) Byzantine nodes' actions when the correct nodes start with $[1, -1, 1]$.



(c) Byzantine nodes' actions when the correct nodes start with $[1, 1, 1]$.

Figure 4.5: Byzantine nodes' actions for different initial input values.

4.1.1 Discussion

We have compiled more simulations with the same setup, to test the achieved robustness with the chosen hyperparameter configuration and algorithm structure. We have observed that the results in different instances were very comparable and that the strategies did not differ much one from another.

However, we would also like to steer your attention to yet another obstacle in our current setup of this project. Since the correct nodes get their initial values randomly, and that randomness is handled within the environment, with an increasing number of the correct nodes, the probability of randomly starting an environment rollout where the all-same validity condition has to be satisfied, i.e. the probability of randomly starting with all the same values for the correct nodes gets exponentially smaller:

$$p = 2 \cdot \frac{1}{2^{nc}} = \frac{1}{2^{nc-1}},$$

and already for the minimal case where the byzantine adversary controls two nodes, we have $nc = 5$ correct nodes, and therefore:

$$p = \frac{1}{2^4} = 0.0625,$$

which would mean that on average we could expect only around 6% of the episodes to be started with all the same values for the correct nodes. There-

fore, we had to decide on what additional features we should implement, such that the all-same value starts might be more favored in the learning process. That is also one of the reasons why we have added an additional agent to our setup – "Dealer" agent. Every time the environment is reset, dealer decides which values initially go to which of the correct nodes, instead of these values being random.

4.2 With dealer agent

4.2.1 Results

When the dealer agent is placing the byzantine node, it has to converge to the distribution of actions with different places for the byzantine column. Had it just converged to putting the byzantine node in one place, the correct nodes could easily know then in every round where the byzantine column was, and safely ignore that column. We will present and compare results with the dealer placing the byzantine node, and with the byzantine node being randomly chosen within the environment. We have chosen a few pairs of simulation parameters to test the dealer agent in both scenarios.

4.2.2 Dealer agent does not place the byzantine column

Learning process

Learning curves for this setup can be seen in Figures 4.6, 4.7, where we depict the episode length and mean policy reward evolution through network training, over the course of 100 training iterations. We have observed that the addition of the dealer agent in this case does not significantly alter the learning phase. In this setup the dealer agent has only $2^{n_c} = 8$ actions, and has converged to the action distribution shown in Figure 4.8. In this distribution, all-same values of 1 are chosen rarely (action 7), and actions 3 ($[-1, 1, 1]$) and 5 ($[1, -1, 1]$) are chosen with highest frequency.

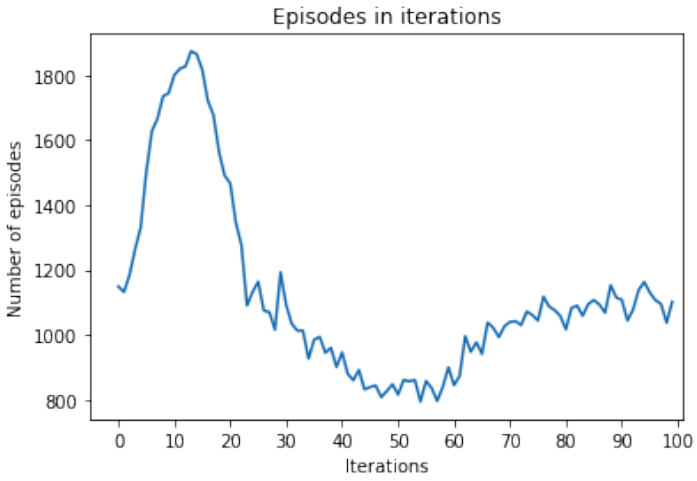


Figure 4.6: Episodes in iterations.

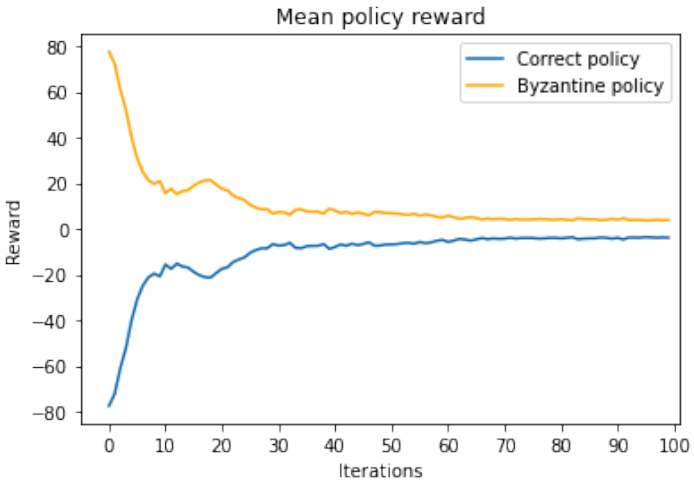


Figure 4.7: Mean policy reward.

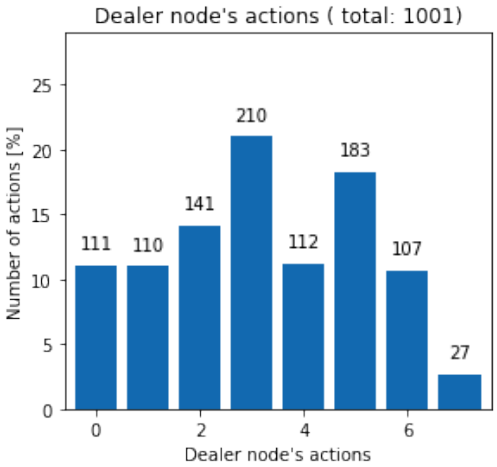
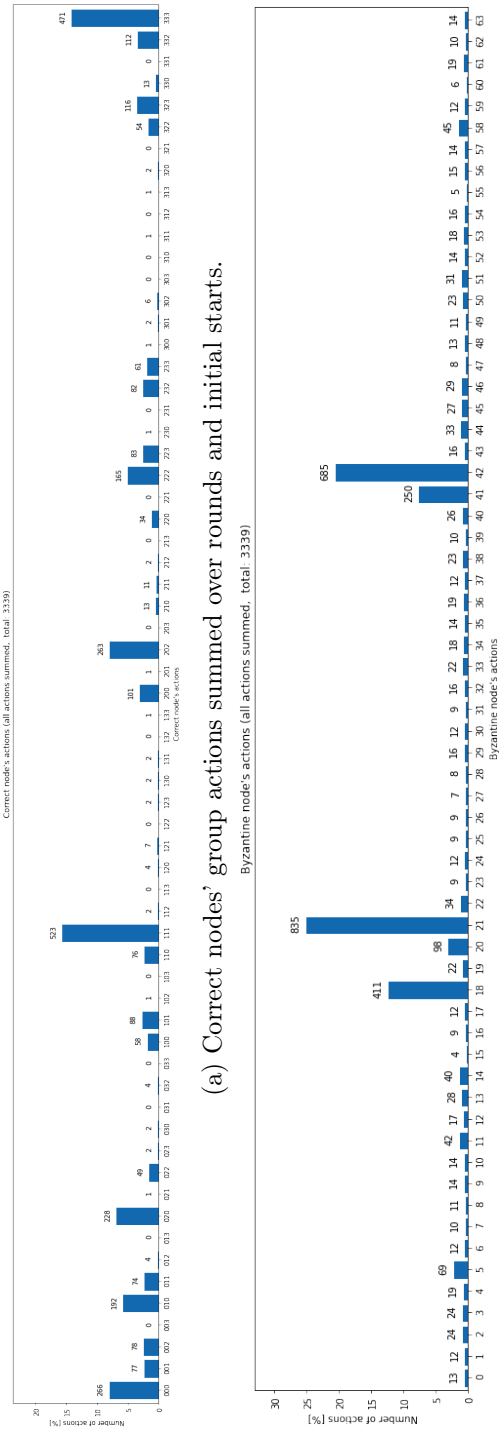


Figure 4.8: Dealer's node action distribution.

Statistics overview

We can again see in Figure 4.9 that the correct nodes go through uncertainty. This is similar to the case without initial values being placed by the dealer agent, presented in the previous section. The byzantine adversary has converged to choosing mainly four actions. These actions are 18 and 21, where the input value chosen is -1, 41 and 42, where the input value chosen is $+1$. The failure report in this case is provided in 4.2.



(a) Correct nodes' group actions summed over rounds and initial starts.

(b) Byzantine nodes' actions summed over rounds and initial starts.

Figure 4.9: Total distribution of nodes' actions in the setup with the dealer agent and randomly placed byzantine column.

Failure report	% of games
All-same validity breach of $(1, 1, 1)$	0
All-same validity breach of $(-1, -1, -1)$	0
Not agreeing	0.5
Max-round reached	0.1
Success	99.4
Average number of rounds	3.34

Table 4.2: Failure report for 1000 environment rollouts.

4.2.3 Dealer agent also places the byzantine column

Learning process

We can again see the evolution of episode length and mean policy reward during training in Figures 4.10 and 4.11. The network was trained in 100 training iterations, where each iteration has 8192 timesteps. In this setup, the dealer agent has more actions, since it also places the byzantine node on one of $n = 4$ columns. The dealer agent has $n \times 2^{n_c} = 32$ actions, and has converged to the action distribution shown in Figure 4.12. From this distribution, we also see that again all-same values of 1 are chosen rarely (last four actions, $\{28, 29, 30, 31\}$). However, this is not a rule, as we will also show in results that all-same values of -1 are sometimes sampled the least.

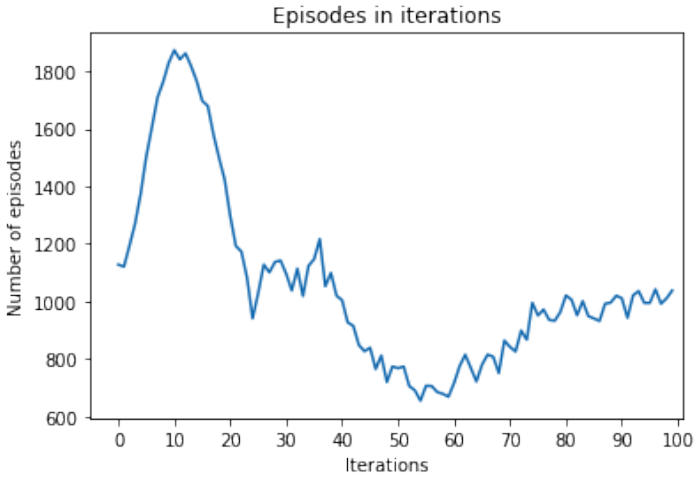


Figure 4.10: Episodes in iterations.

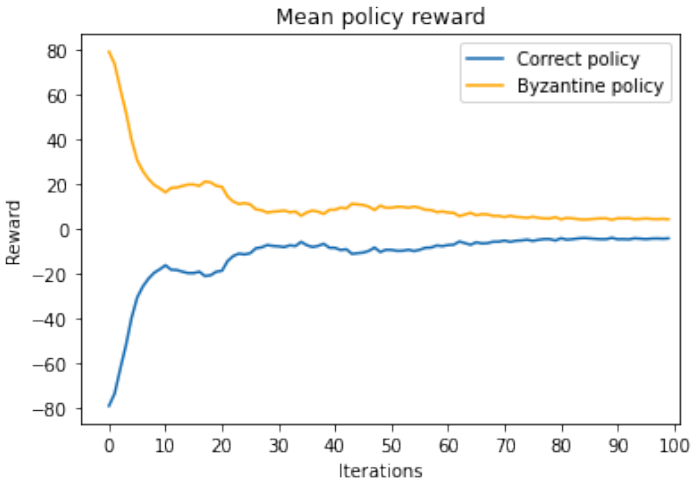


Figure 4.11: Mean policy reward.

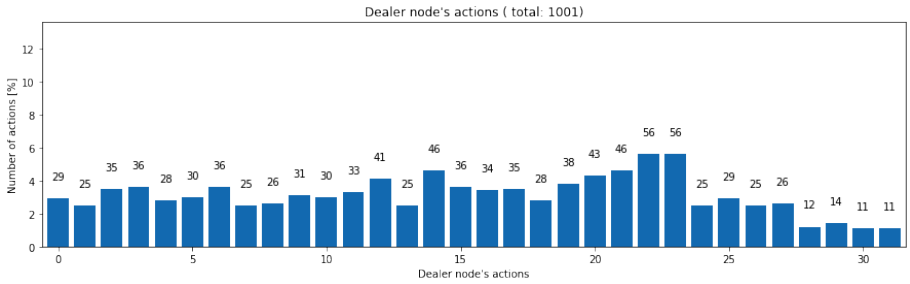
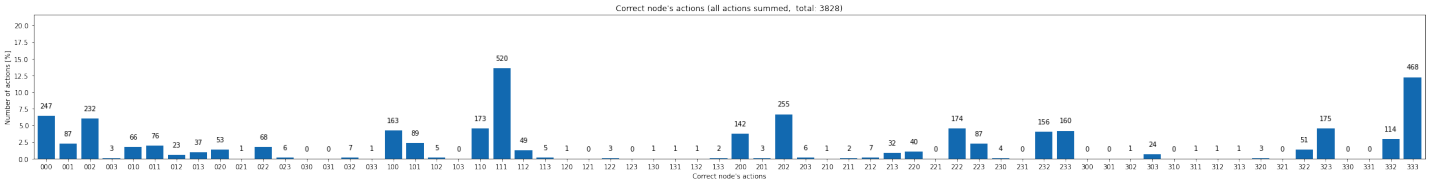


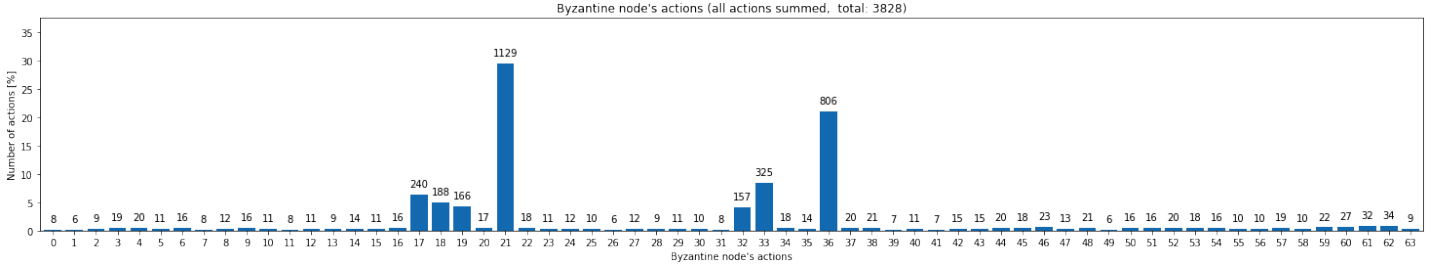
Figure 4.12: Dealer's node action distribution.

Statistics overview

We have obtained results that are similar to the case when the byzantine node is randomly placed, with correct nodes' actions depicted in Figure 4.13. The byzantine agent has also more actions, but mainly chooses again 21 with input value -1, and 36, with input value $+1$. The failure report in this case is provided in 4.3.



(a) Correct nodes' group actions summed over rounds and initial starts.



(b) Byzantine nodes' actions summed over rounds and initial starts.

Figure 4.13: Total distribution of nodes' actions in the setup with the dealer agent that also places the byzantine column.

Failure report	**% of games**
All-same validity breach of $(1, 1, 1)$	0.1
All-same validity breach of $(-1, -1, -1)$	0
Not agreeing	1.0
Max-round reached	0.2
Success	98.7
Average number of rounds	3.83

Table 4.3: Failure report for 1000 environment rollouts.

4.2.4 Discussion

In both simulations, we have observed that the dealer node converges to the distribution where it favors some of the actions, but those actions are never all-same values. In the case when the dealer also places the byzantine column, it does not converge to only showing an action with one column choice. What it does instead, it keeps a distribution where it places byzantine column on each of 4 possible places. This way, the task for the correct nodes is harder because they cannot learn where the byzantine column location. In Figures 4.14 and 4.15, we show a brief preview of multiple simulation executions, for slightly different hyperparameter configurations. We can easily deduce the previously described trend in the dealer agent's action from the presented results.

We can therefore conclude that the dealer agent could successfully team up with the byzantine agent in the common goal of disrupting the correct nodes' agreement. However, there exists another problem. We wanted to gain more insight into how efficient byzantine learning is, and how much it is influenced in the self-play setup by the adopted strategy of the correct nodes. Therefore, we have decided to test how good the byzantine agent performs by letting it play against fixed policies. The motivation for the fixed policy came from [10], where an additional action representing "tagged value" is used.

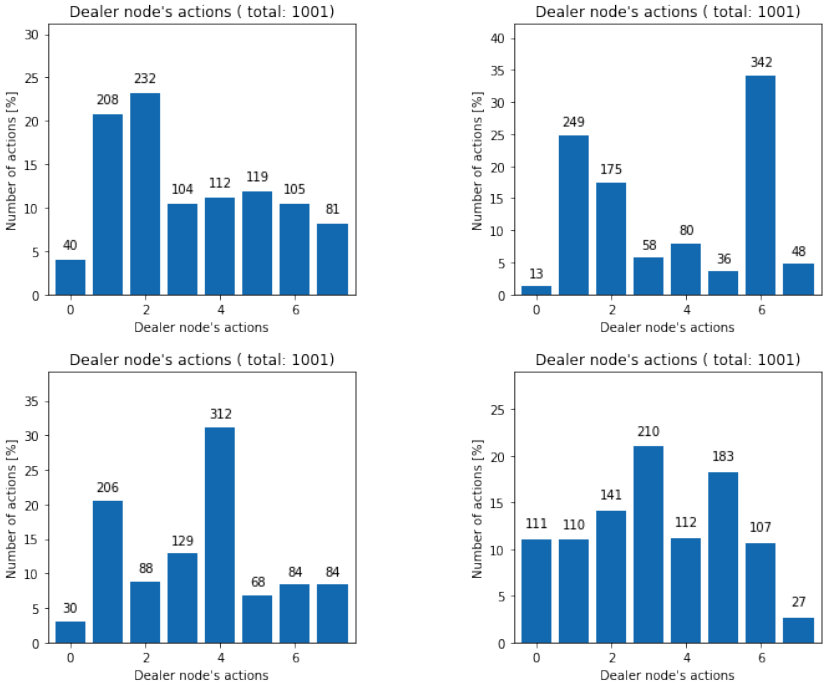


Figure 4.14: Dealer agent's actions in 4 different experiments when not placing byzantine node.

4.3 Fixed strategy of the correct nodes

The correct nodes are using fixed strategy in this setup and only the byzantine agent is being trained. We will analyse the results with and without dealer agent's action.

4.3.1 Without dealer agent

Learning process

The evolution of episode length during training is shown in Figure 4.16. In Figure 4.17 we see the evolution of the mean policy reward for the nodes, over the course of 200 training iterations, with 8192 timesteps within each iteration.

Statistics overview

All actions of correct nodes with fixed policy are depicted in Figure 4.18. Furthermore, we also note that byzantine policy has converged to a policy with more actions, what was not observed before. Correct nodes also achieve 100% success

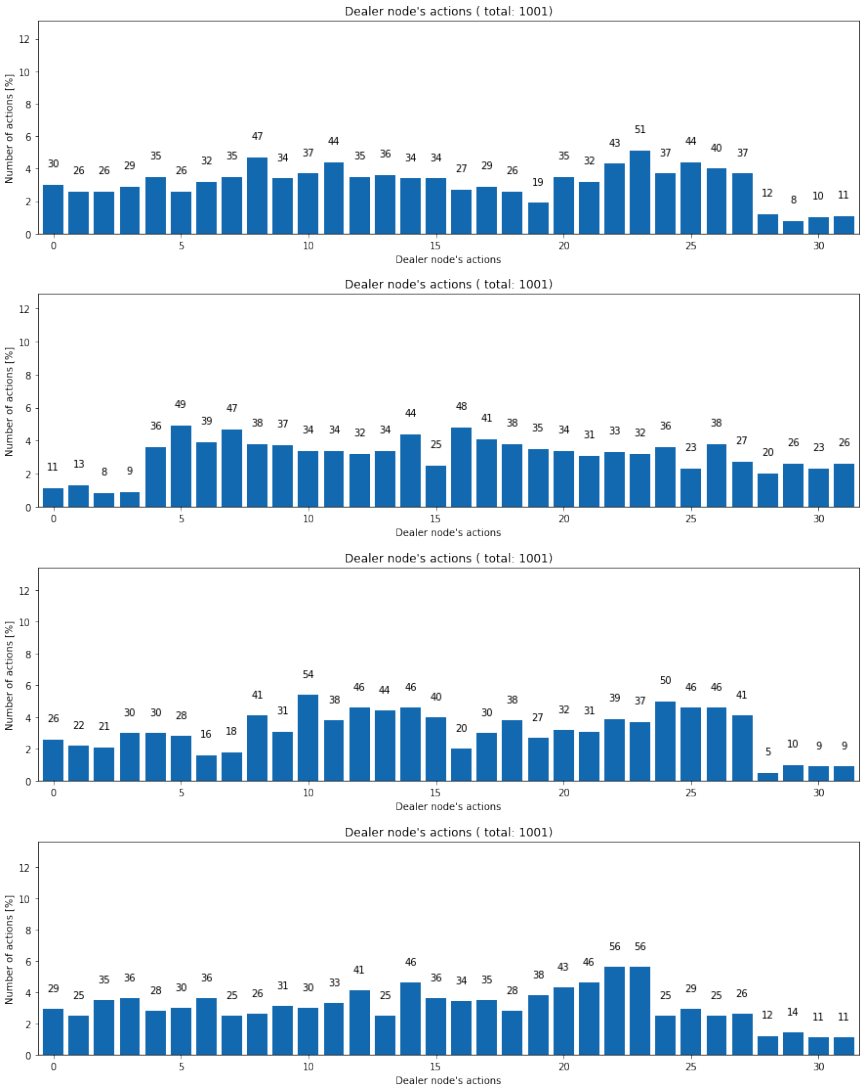


Figure 4.15: Dealer agent's actions in 4 different experiments when placing byzantine node.

rate, since they are using the consensus protocol, and are never deciding before seeing enough of the majority, as shown in Table 4.4.

We see again that the byzantine chooses the action that is actually opposed to the majority value seen for the correct nodes in Figure 4.19. This is an optimal tactic against the given protocol, which is to manipulate the local views of the correct nodes towards equal numbers of -1s or 1s. However, in the implementation in the protocol, even if the correct nodes see no majority, they still toss a coin in phase 1, and can thus terminate by chance.

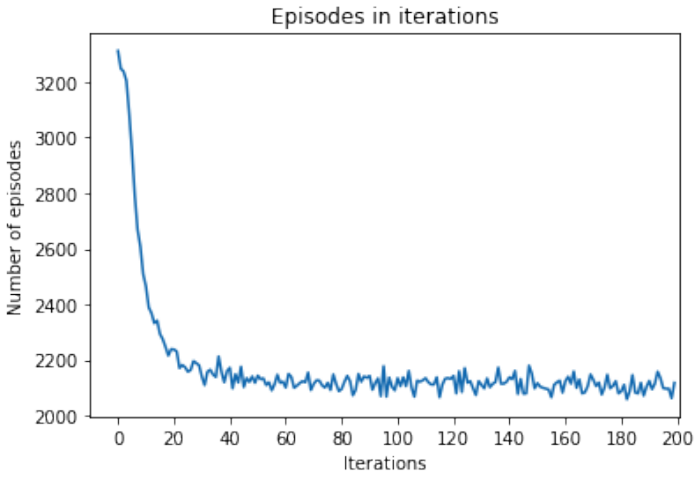


Figure 4.16: Episodes in iterations.

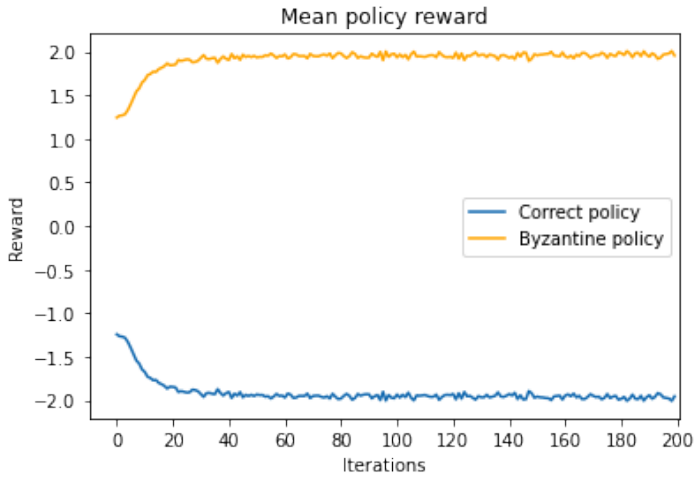


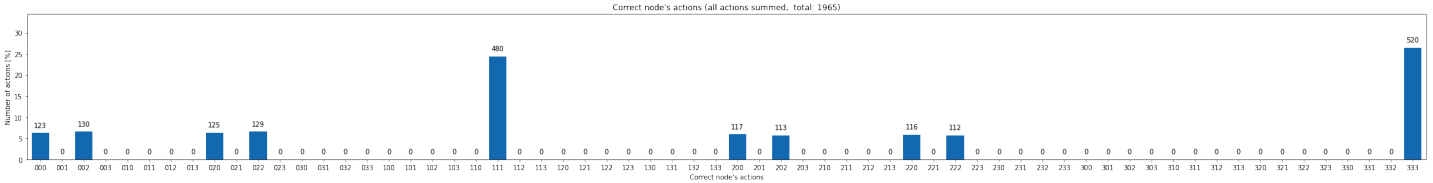
Figure 4.17: Mean policy reward.

Success	100%
Average number of rounds	1.96

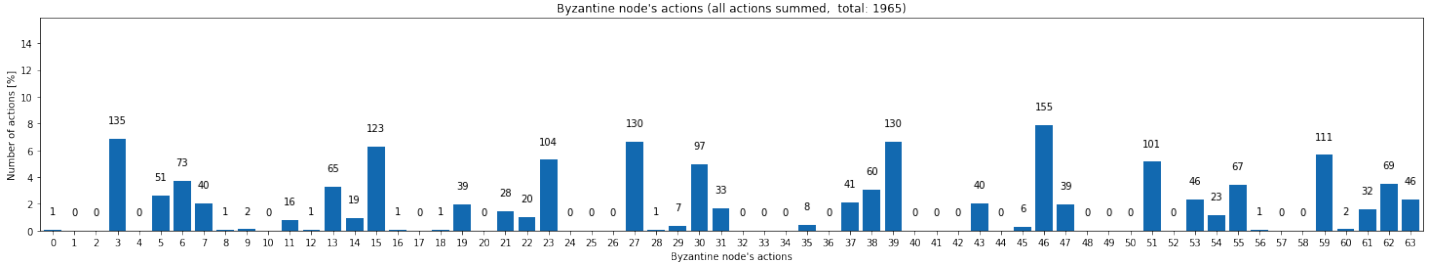
Table 4.4: Statistics report for 1000 environment rollouts.

4.3.2 With dealer agent

Learning curves can be seen in Figures 4.20 and 4.21. Strategies with dealer agent are very similar to the ones analysed previously, so the same observations hold. However, it is very interesting to show the dealer agent's adopted strategy – Figure 4.22. We can clearly see that the dealer never shows both all-same

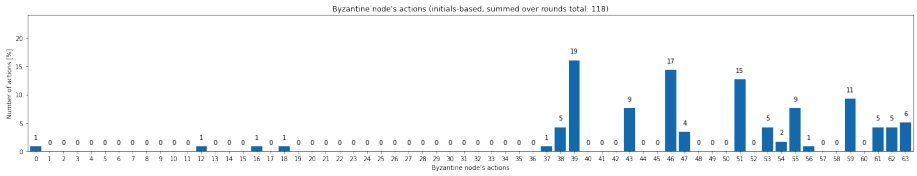


(a) Correct nodes' group actions summed over rounds and initial starts.

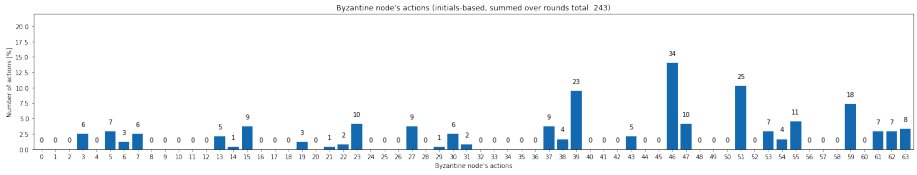


(b) Byzantine nodes' actions summed over rounds and initial starts.

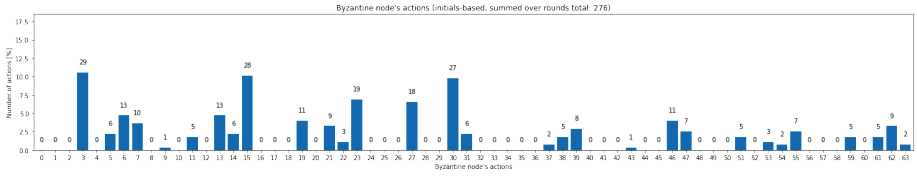
Figure 4.18: Total distribution of nodes' actions in the setup with the fixed correct nodes' policy and without the dealer agent.



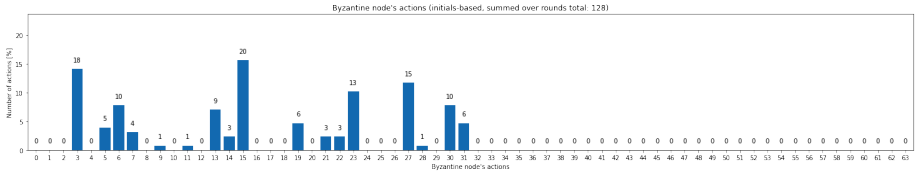
(a) Byzantine nodes' actions when the correct nodes start with $[-1, -1, -1]$.



(b) Byzantine nodes' actions when the correct nodes start with $[-1, 1, -1]$.



(c) Byzantine nodes' actions when the correct nodes start with $[1, -1, 1]$.



(d) Byzantine nodes' actions when the correct nodes start with $[1, 1, 1]$.

Figure 4.19: Byzantine nodes' actions for different initial input values.

values because they guarantee the termination of correct nodes already in the first round. Other initial value combinations are almost equally distributed. In this example, the correct nodes needed a little more time to reach agreement, as shown in Table 4.5.

Success	100%
Average number of rounds	2.22

Table 4.5: Statistics report for 1000 environment rollouts.

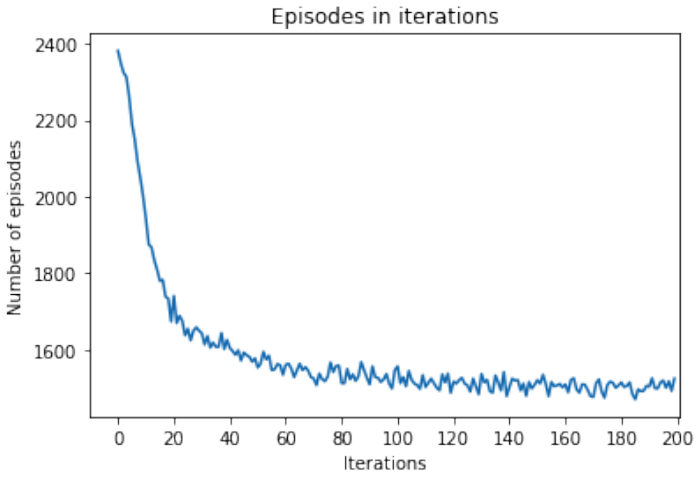


Figure 4.20: Episodes in iterations.

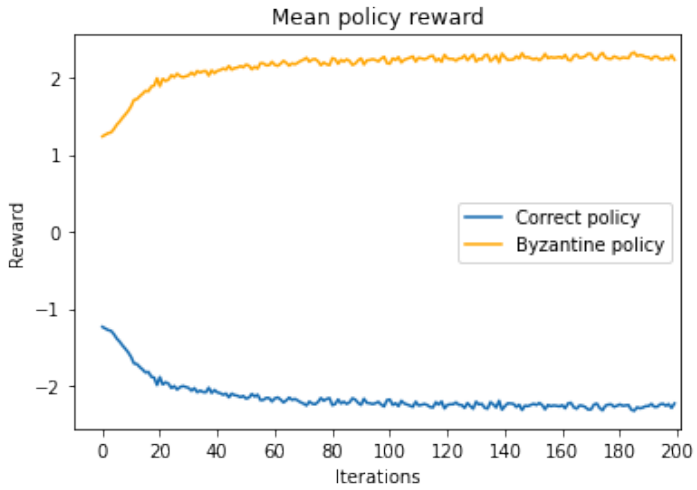


Figure 4.21: Mean policy reward.

4.4 Self-play with additional actions

Previously only the byzantine agent was trained against a fixed correct nodes' policy. Now we are presenting results for the self-play scenario with two additional actions, mentioned in Chapter 3. Our motivation was to see whether the correct nodes can perhaps make use of this additional action and reach agreement in a fewer expected number of rounds.

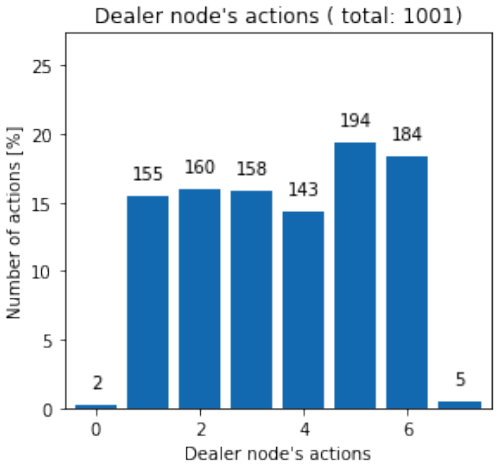


Figure 4.22: Mean policy reward.

4.4.1 With dealer agent

Learning process

In this setup, we are analysing results with the dealer agent who has $2^{n_c} = 8$ actions. An interesting observation in this setup is that the dealer agent has not converged to any particular action distribution. In this scenario, multiple simulations showed that it was mostly optimal for the malicious party to just preserve the initial randomness. The dealer agent's distribution is depicted in 4.23. Learning curves are shown in Figures 4.24 and 4.25.

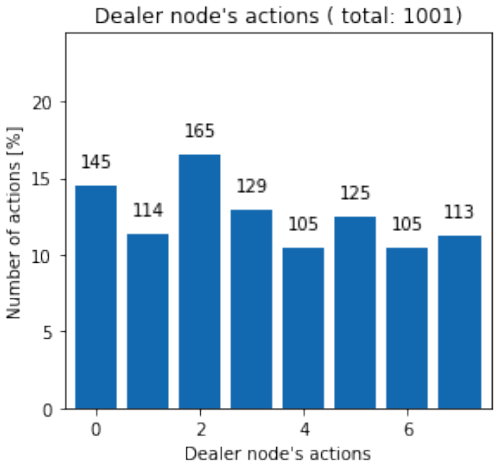


Figure 4.23: Mean policy reward.

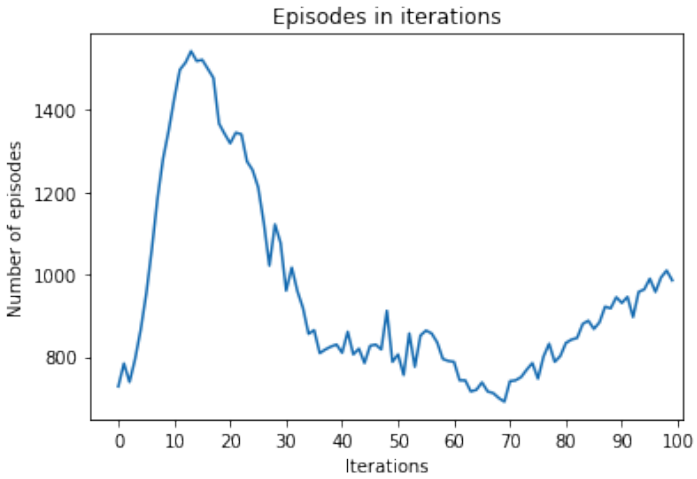


Figure 4.24: Episodes in iterations.

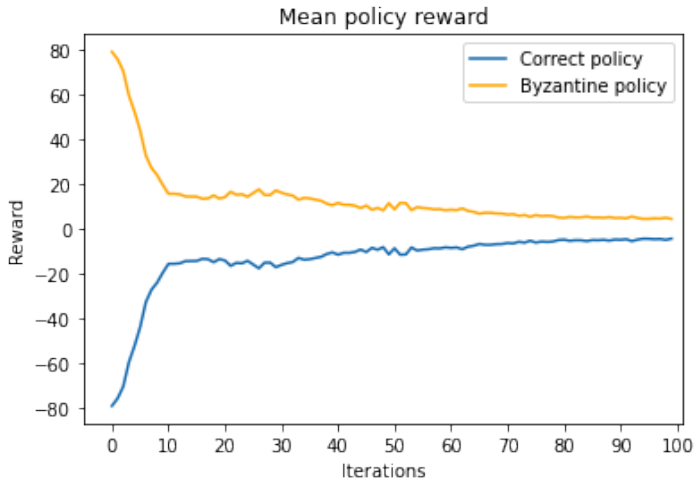
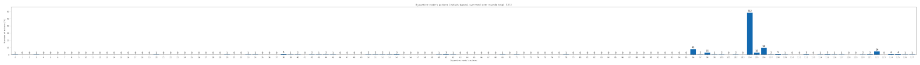


Figure 4.25: Mean policy reward.

Statistics overview

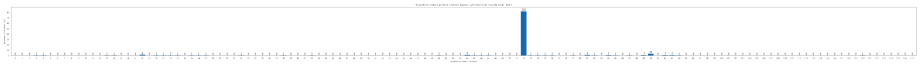
This example also shows a particularly interesting action distribution of byzantine and correct nodes. Same as before, we note that the byzantine chooses the action that is opposed to the majority value seen for the correct nodes, which might be the best it can do. However, in this setup the byzantine agent almost always converges to an action distribution where it chooses only -2 and $+2$ as its input values. Most probably, this happens because it is trying to again flip the majority, and since correct nodes converged to choosing preferentially -2 and 2, it works better. The byzantine agent's action distribution can be seen in Figure 4.26.

We also analyse the correct nodes' actions for different input value combinations. Here we can observe that the nodes choose mostly actions 1 (terminate on -1) and 4 (input value -2) if they start with all-same values of -1. If they start with all-same values of 1, then they opt for actions 3 (terminate on 1) and 5 (input value 2). This means correct nodes are not using previous "uncertain" actions of 0 and 2 anymore. The correct nodes' action distribution is shown in Figure 4.27. This would mean that the addition of two input actions does not help correct nodes to make a better decision. What happened, was that the previous uncertain actions 0 and 2 got exchanged for new actions 4 and 5. However, it might be interesting in future work to further investigate why this phenomenon occurs. A failure report is very similar to the case without additional actions, shown in Table 4.6.



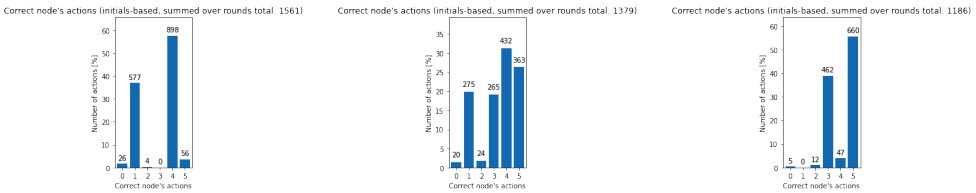
(a) Byzantine nodes' actions when the correct nodes start with $[-1, -1, -1]$.

(b) Byzantine nodes' actions when the correct nodes start with $[-1, 1, 1]$.



(c) Byzantine nodes' actions when the correct nodes start with $[1, 1, 1]$.

Figure 4.26: Byzantine nodes' actions for different initial input values.



(a) Correct nodes' actions when the correct nodes start with $[-1, -1, -1]$.

(b) Correct nodes' actions when the correct nodes start with $[-1, 1, 1]$.

(c) Correct nodes' actions when the correct nodes start with $[1, 1, 1]$.

Figure 4.27: Correct nodes' actions for different initial input values.

Failure report	**% of games**
All-same validity breach of $(1, 1, 1)$	0
All-same validity breach of $(-1, -1, -1)$	0
Not agreeing	0.4
Max-round reached	0
Success	99.6
Average number of rounds	3.75

Table 4.6: Failure report for 1000 environment rollouts.

4.4.2 Discussion

For the fixed strategy case, the correct nodes always terminate with a 100% success rate, and they do not need many rounds, on average about two for the majority of the successful simulations.
In the self-play setup, we have also managed to get a high success rate of close to 99.9%, but the correct nodes needed more rounds to terminate, most of the time between three and four rounds. Therefore, there might still be some space for improvement in the future and a possibility to teach the correct nodes to terminate in 100% of episodes. However, with numerous challenges that reinforcement learning carries, this might be hard to achieve. Another problem with the self-play setup is that most of the time one strategy can highly influence the other; making it, for example, weaker or suboptimal. The question then would be whether we need to make some additional changes in our algorithm setup in order to achieve higher accuracy.

4.5 Byzantine Adversary Controls Multiple Nodes

4.5.1 Two byzantine nodes without dealer agent

Learning process

In Figure 4.28 we can see the evolution of the episode length through the course of learning. In Figure 4.29 we see the evolution of the mean policy reward for the nodes, over the course of 100 training iterations, where each iteration has 8192 timesteps. We have observed that the agents have also converged in this setup, and went on to further analyse their strategies.

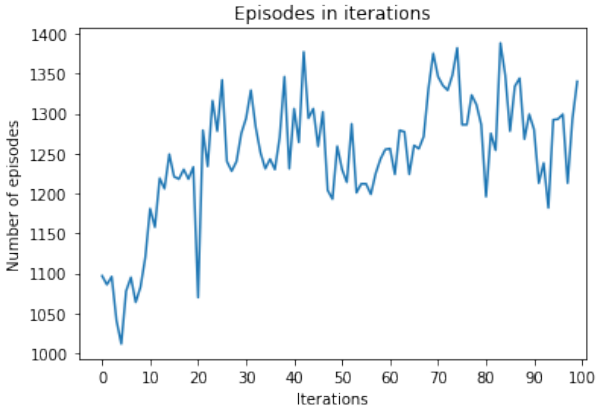


Figure 4.28: Episodes in iterations.

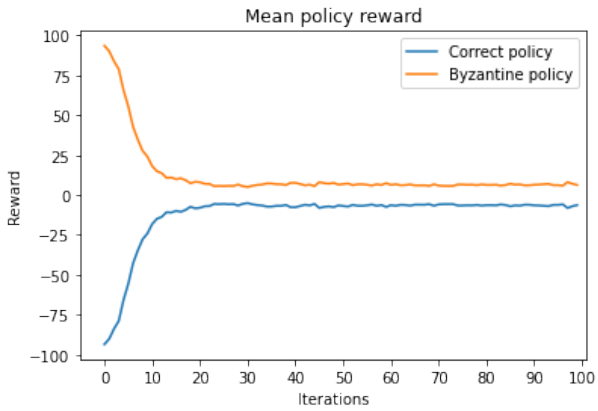


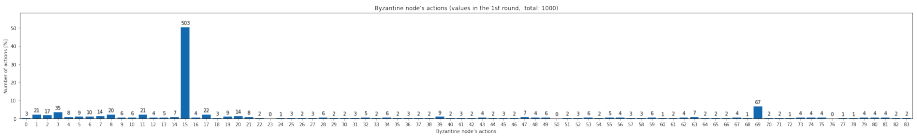
Figure 4.29: Mean policy reward.

Statistics overview

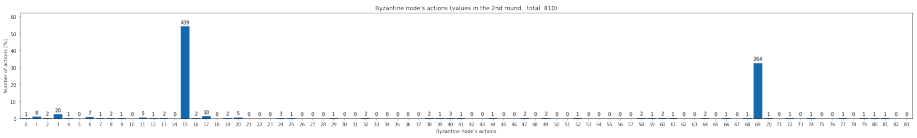
In the following Figure 4.30, we can observe how the byzantine adversary developed an interesting strategy to either send input values of $\{-1, -1\}$ (action 15), or to send input values of $\{1, 1\}$ (action 69). The failure report in this case is provided in 4.7.

Failure report	**% of games**
All-same validity breach of $(1, 1, 1)$	2.6
All-same validity breach of $(-1, -1, -1)$	0
Not agreeing	0.3
Max-round reached	0
Success	97.1
Average number of rounds	3.23

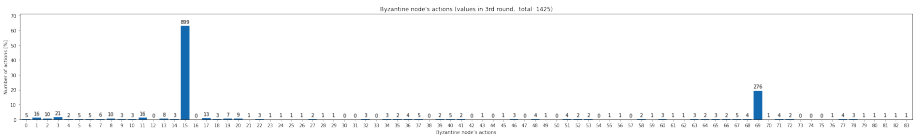
Table 4.7: Failure report for 1000 environment rollouts.



(a) Byzantine node's actions in 1*st* round, summed over initial starts.



(b) Byzantine node's actions in 2*nd* round, summed over initial starts.



(c) Byzantine node's actions in 3*rd* round onwards, summed over initial starts.

Figure 4.30: Byzantine node's action distribution.

Analysing these results, we can conclude that the byzantine agent is trying to trick correct nodes into thinking they are starting with the significant majority of some value, and go towards agreement. It seems that the correct nodes are sacrificing the condition of all-same validity since it appears with a very low probability of around 0.06 in the training. In this case, the dealer node would make the problem even worse, as the dealer wants the correct nodes to fail.

4.5.2 Discussion

Based on our results, the learning in the setup with two byzantine nodes is more unstable; the biggest issue now is that the chance that all the correct nodes start with the same value gets way smaller. This makes the correct nodes' adopted strategy fragile and sensitive to the randomness in the learning. We have observed that most of the time with the dealer node included, its strategy converges to showing only all-same values, because the correct nodes might have forgotten the strategy for that case before, and then fail to adapt. Future work might also include an interesting example with fixed probabilities, where the all-same values happen more often, and then we might see how the correct nodes adapt their strategy to this case.

Chapter 5

Conclusion

5.1 Summary

The main objective of this master's thesis was to investigate how reinforcement learning can be used to learn more about byzantine agreement. Our task was to simulate byzantine behavior and to derive protocols that are robust to this kind of malicious behavior. We have examined the asynchronous communication model with authenticated public channels. In our setup, the byzantine adversary has access to all the messages sent by nodes, and it has the power to hide the values of some nodes from some of the correct nodes while revealing them to the other correct nodes. The goal of the correct nodes is to reach agreement in as little rounds as possible, and the goal of the byzantine adversary is to disturb this process, preferably making the correct nodes fail in this task. We have modeled this setup as a zero-sum game, between the correct and malicious participants. We have created an environment in which the correct and byzantine agents interact. They receive observations from the environment, choose their actions accordingly, and finally, they receive a reward based on their chosen action. We have analysed the adopted strategy-counter-strategy pairs of the nodes. We have also tested the adversary's capabilities against a fixed correct nodes' policy. To sum up, we have shown that reinforcement learning could be used as a tool to make the participating nodes learn strategies in the asynchronous byzantine agreement setup through self-play. This might be valuable when the theory gets too complicated, and when testing already existing protocols and looking for counterexamples.

5.2 Future work

As mentioned previously in the introduction, our initial ideas were that the nodes would have to locally give weights of trust to other nodes, or they would have to remove the byzantine nodes, in order to successfully reach agreement. In the considered setup with just one byzantine node, there would be no significant difference in the implementation. However, we would be curious to find out if correct

nodes could also learn such a sophisticated algorithm at some point in the future. Therefore, it would be interesting to see how our results extend to two, three, or more byzantine parties; and whether they can develop strategies through self-play in that scenario as well. Unfortunately, there would possibly exist numerous implementation obstacles along the way, starting with the exponential growth of the action space of agents, to which we do not have a solution. An interesting approach in the case of more nodes might be to implement the neural fictitious self-play algorithm. This approach consists of attacking the problem of learning optimal policies in the self-play setup from a somewhat more game-theoretic angle. It is shown that this approach should converge to the Nash equilibria if some mild conditions are satisfied [19].

Furthermore, the probability of all-same initial values being sampled decreases exponentially, with the formula given by $p = 2^{1-nc}$. This means that already for the minimal case with three byzantine nodes, $f = 3$, $n = 10$ and $nc = 7$, the probability is

$$p = 2^{-6} = 0.015625.$$

Therefore, the dealer agent could not be used anymore and a new approach might be needed. We would also most probably need a deeper statistical view of the learned strategies. In the results with a smaller number of nodes, it was not too complicated to infer what the nodes might be doing. However, for the case with larger agent space, this might also become problematic.

In the end, this setup might also get some other interesting uses. One of the more intriguing ones might be the approach of using reinforcement learning to test already existing protocols. This would mean that in some cases where the theory gets too complicated, we could utilize reinforcement learning to help and lead us to the flaws in our results. Also, it could for example provide us with a counter-example of an algorithm that would have been otherwise not found.

Bibliography

[1] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '83, 1983, p. 27–30.

[2] "Openai five," June 2020. [Online]. Available: https://openai.com/blog/openai-five/

[3] "Alphastar: Grandmaster level in starcraft ii using multi-agent reinforcement learning," June 2020. [Online]. Available: https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning

[4] M. O. Rabin, "Randomized byzantine generals," in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, November 1983, pp. 403–409.

[5] V. King and J. Saia, "Byzantine agreement in expected polynomial time," in *J. ACM*, vol. 63 (2), March 2016.

[6] D. Melnyk, "Byzantine Agreement on Representative Input Values over Public Channels," in *PhD Thesis*, August 2020.

[7] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," in *ACM Transactions on Programming Languages and Systems*, vol. 4 (3), July 1982, p. 382–401.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," in *Journal of the ACM*, vol. 32 (2), April 1985, pp. 374–382.

[9] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," in *Journal of the ACM*, vol. 32 (4), 1985, pp. 824–840.

[10] G. Bracha, "Asynchronous byzantine agreement protocols," in *Journal of Information and Computation*, vol. 75 (2), 1987, pp. 130–143.

[11] A. Mostefaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14, vol. 63 (2), 2014, pp. 2–9.

[12] J. Moeller, "Deep reinforcement learning applied to byzantine agreement," Semester Project, April 2020. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2019-HS/SA-2019-22.pdf

[13] "Competitive self-play," June 2020. [Online]. Available: https://openai.com/blog/competitive-self-play/

[14] R. Wattenhofer, "Consensus," May 2020. [Online]. Available: https://disco.ethz.ch/courses/distsys/lnotes/chapter16.pdf

[15] R. Wattenhofer, "Byzantine agreement," May 2020. [Online]. Available: https://disco.ethz.ch/courses/distsys/lnotes/chapter17.pdf

[16] R. Wattenhofer, "Broadcast and shared coins," June 2020. [Online]. Available: https://disco.ethz.ch/courses/distsys/lnotes/chapter18.pdf

[17] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," October 2020, 2nd Edition. [Online]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[19] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1603.01121

[20] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2018.

[21] "Ray rllib," May 2020. [Online]. Available: https://github.com/ray-project/ray

[22] "Ray rllib package reference," October 2020. [Online]. Available: https://docs.ray.io/en/latest/rllib-package-ref.html

Appendix A

Implementation Details

The work on this project was done using RLlib [20], [21] framework for deep reinforcement learning and for constructing the gym-like multi-agent environment we needed in our experimentation and algorithm development. The work was done in Python programming language, using Google Colaboratory resources. The package reference can be found in [22].

Appendix B

Correct Nodes' Action Mapping

This chapter describes the correct nodes' action mapping from Chapter 3.

Correct nodes' action space is a gym.spaces - spaces.Discrete(4) instance, with a map inside the environment B.1, through which we get the input values to be written to the blackboard matrix and these are later observed by the other agents.

Action Nr.	**Action description**
0	−1, continue
1	−1, terminate
2	1, continue
3	1, terminate

Table B.1: Correct nodes' action mapping for 4 actions.

If the correct nodes have also two additional inputs, action mapping can be found in Table B.2.

Action Nr.	**Action description**
0	−1, continue
1	−1, terminate
2	1, continue
3	1, terminate
4	−2, continue
5	2, continue

Table B.2: Correct nodes' action mapping for 6 actions.

Appendix C

Byzantine's Action Mapping

Byzantine node's action space is a gym.spaces - spaces.Discrete(64) instance, with a map inside the environment:

$$\{0, 1, \ldots, 63\} \longrightarrow [-1, 0, 0, 0, 0], [-1, 0, 0, 0, 1], \ldots, [1, 3, 1, 1, 1].$$

A function is implemented in python that does this action encoding, and each of the 64 possible actions is then represented in the list form which is then used within the code. The first value in the list can be one of the values from $\{-1, 1\}$ and it represents the chosen input value written to the blackboard matrix by the byzantine node. The second value can take the value of $\{0, 1, 2, 3\}$, and it determines which node byzantine agent decides to hide in the current round, referring to one of the four node IDs. The next three values are used as a "hide mask", and they take the values of $\{0, 1\}$, meaning the byzantine node hides the node from that correct node in the current round or not. This way we wanted to model adversary scheduling. So, for example, the action

$$62 \longrightarrow [1, 3, 1, 1, 0]$$

means that the byzantine node chose its input value to be 1, and chose to hide its input value (node ID 3) from the third correct node (*corr_2*), because the last three values are the hide mask $[1, 1, 0]$, and there is a zero in the third place of the mask, which refers to the third correct node (*corr_2*). Also, there is a difference compared to the results from before; we have decided not to restrict the byzantine adversary's actions yet in smaller examples and returned to the full number of masks. Before, there were 56 possible actions in our setup because we were not allowing the byzantine adversary to choose a node to hide, but then choose not to hide it from any of the nodes (mask $[1, 1, 1]$ was not allowed). Then, the possible number of actions was $2 \times 4 \times 2 \times 2 \times 2 - 2 \times 4 \times 1 \times 1 \times 1 = 64 - 8 = 56$; now it is simply $2 \times 4 \times 2 \times 2 \times 2 = 64$.

In the following Table C.1 we can see the byzantine adversary's action mapping for the case of $n = 4$ nodes from Chapter 3, which makes the statistics analysis easier. As we can deduce from the table, first 32 actions produce an input value of -1, and other 32 actions produce an input value of 1.

For the case with two additional inputs, when byzantine can also write input values of -2 and 2 to the blackboard, the action space is simply doubled. This means that first 32 actions have value -1, next 32 have value 1, next 32 have value -2 and finally last 32 actions have value 2. Hiding mask distribution and node selection stays the same within these 32 action-blocks.

Action Nr.	**Action description**	**Action Nr.**	**Action description**
0	$[-1, 0, [0, 0, 0]]$	32	$[1, 0, [0, 0, 0]]$
1	$[-1, 0, [0, 0, 1]]$	33	$[1, 0, [0, 0, 1]]$
2	$[-1, 0, [0, 1, 0]]$	34	$[1, 0, [0, 1, 0]]$
3	$[-1, 0, [0, 1, 1]]$	35	$[1, 0, [0, 1, 1]]$
4	$[-1, 0, [1, 0, 0]]$	36	$[1, 0, [1, 0, 0]]$
5	$[-1, 0, [1, 0, 1]]$	37	$[1, 0, [1, 0, 1]]$
6	$[-1, 0, [1, 1, 0]]$	38	$[1, 0, [1, 1, 0]]$
7	$[-1, 0, [1, 1, 1]]$	39	$[1, 0, [1, 1, 1]]$
8	$[-1, 1, [0, 0, 0]]$	40	$[1, 1, [0, 0, 0]]$
9	$[-1, 1, [0, 0, 1]]$	41	$[1, 1, [0, 0, 1]]$
10	$[-1, 1, [0, 1, 0]]$	42	$[1, 1, [0, 1, 0]]$
11	$[-1, 1, [0, 1, 1]]$	43	$[1, 1, [0, 1, 1]]$
12	$[-1, 1, [1, 0, 0]]$	44	$[1, 1, [1, 0, 0]]$
13	$[-1, 1, [1, 0, 1]]$	45	$[1, 1, [1, 0, 1]]$
14	$[-1, 1, [1, 1, 0]]$	46	$[1, 1, [1, 1, 0]]$
15	$[-1, 1, [1, 1, 1]]$	47	$[1, 1, [1, 1, 1]]$
16	$[-1, 2, [0, 0, 0]]$	48	$[1, 2, [0, 0, 0]]$
17	$[-1, 2, [0, 0, 1]]$	49	$[1, 2, [0, 0, 1]]$
18	$[-1, 2, [0, 1, 0]]$	50	$[1, 2, [0, 1, 0]]$
19	$[-1, 2, [0, 1, 1]]$	51	$[1, 2, [0, 1, 1]]$
20	$[-1, 2, [1, 0, 0]]$	52	$[1, 2, [1, 0, 0]]$
21	$[-1, 2, [1, 0, 1]]$	53	$[1, 2, [1, 0, 1]]$
22	$[-1, 2, [1, 1, 0]]$	54	$[1, 2, [1, 1, 0]]$
23	$[-1, 2, [1, 1, 1]]$	55	$[1, 2, [1, 1, 1]]$
24	$[-1, 3, [0, 0, 0]]$	56	$[1, 3, [0, 0, 0]]$
25	$[-1, 3, [0, 0, 1]]$	57	$[1, 3, [0, 0, 1]]$
26	$[-1, 3, [0, 1, 0]]$	58	$[1, 3, [0, 1, 0]]$
27	$[-1, 3, [0, 1, 1]]$	59	$[1, 3, [0, 1, 1]]$
28	$[-1, 3, [1, 0, 0]]$	60	$[1, 3, [1, 0, 0]]$
29	$[-1, 3, [1, 0, 1]]$	61	$[1, 3, [1, 0, 1]]$
30	$[-1, 3, [1, 1, 0]]$	62	$[1, 3, [1, 1, 0]]$
31	$[-1, 3, [1, 1, 1]]$	63	$[1, 3, [1, 1, 1]]$

Table C.1: Byzantine's action mapping.

Appendix D

Dealer's Action Mapping

This chapter describes the dealer agent's action mapping for the case of $n = 4$ nodes from Chapter 3.

D.1 Dealer node does not place the byzantine column

Action space is 2^{n_c}, and for $n_c = 3$ correct nodes we get eight possible actions, ranging from $[-1, -1, -1]$ to $[1, 1, 1]$, which can be seen in the following Table D.1. These three number denote the initial value distribution to corr_0, corr_1 and corr_2 nodes.

Action Nr.	Action description
0	$[-1, -1, -1]$
1	$[-1, -1, \ 1]$
2	$[-1, \ 1, -1]$
3	$[-1, \ 1, \ 1]$
4	$[\ 1, -1, -1]$
5	$[\ 1, -1, \ 1]$
6	$[\ 1, \ 1, -1]$
7	$[\ 1, \ 1, \ 1]$

Table D.1: Dealer's action mapping when not placing byzantine column.

D.2 Dealer node places the byzantine column

Action space is $n \times 2^{n_c}$, and for $n_c = 3$ correct nodes we get 32 possible actions, ranging from $[0, [-1, -1, -1]]$ to $[3, [1, 1, 1]]$, which can be seen in the following Table D.2. In the first place is selected byzantine column ID, and next three number denote the initial value distribution to corr_0, corr_1 and corr_2 nodes.

Action Nr.	**Action description**
0	$[0,\ [-1,-1,-1]]$
1	$[0,\ [-1,-1,\ \ 1]]$
2	$[0,\ [-1,\ \ 1,-1]]$
3	$[0,\ [-1,\ \ 1,\ \ 1]]$
4	$[0,\ [\ \ 1,-1,-1]]$
5	$[0,\ [\ \ 1,-1,\ \ 1]]$
6	$[0,\ [\ \ 1,\ \ 1,-1]]$
7	$[0,\ [\ \ 1,\ \ 1,\ \ 1]]$
8	$[1,\ [-1,-1,-1]]$
9	$[1,\ [-1,-1,\ \ 1]]$
10	$[1,\ [-1,\ \ 1,-1]]$
11	$[1,\ [-1,\ \ 1,\ \ 1]]$
12	$[1,\ [\ \ 1,-1,-1]]$
13	$[1,\ [\ \ 1,-1,\ \ 1]]$
14	$[1,\ [\ \ 1,\ \ 1,-1]]$
15	$[1,\ [\ \ 1,\ \ 1,\ \ 1]]$
16	$[2,\ [-1,-1,-1]]$
17	$[2,\ [-1,-1,\ \ 1]]$
18	$[2,\ [-1,\ \ 1,-1]]$
19	$[2,\ [-1,\ \ 1,\ \ 1]]$
20	$[2,\ [\ \ 1,-1,-1]]$
21	$[2,\ [\ \ 1,-1,\ \ 1]]$
22	$[2,\ [\ \ 1,\ \ 1,-1]]$
23	$[2,\ [\ \ 1,\ \ 1,\ \ 1]]$
24	$[3,\ [-1,-1,-1]]$
25	$[3,\ [-1,-1,\ \ 1]]$
26	$[3,\ [-1,\ \ 1,-1]]$
27	$[3,\ [-1,\ \ 1,\ \ 1]]$
28	$[3,\ [\ \ 1,-1,-1]]$
29	$[3,\ [\ \ 1,-1,\ \ 1]]$
30	$[3,\ [\ \ 1,\ \ 1,-1]]$
31	$[3,\ [\ \ 1,\ \ 1,\ \ 1]]$

Table D.2: Dealer's action mapping when placing byzantine column.

Appendix E

Simulation parameters

Simulation parameters for $n = 4$ nodes without the dealer agent are given in Table E.1.

Network size:	$[64, 64]$
Train batch size	8192
Mini-batch size	1024
Iterations	200
Max rounds	100
Learning rate Corr	$1.5e^{-5}$
Learning rate Byz	$1.2e^{-5}$
γ	1.0

Table E.1: Simulation parameters.

Simulation parameters for $n = 4$ nodes with the dealer agent are given in Table E.2.

Network size:	$[64, 64]$
Train batch size	8192
Mini-batch size	512
Iterations	100
Max rounds	100
Learning rate Corr	$1.5e^{-5}$
Learning rate Byz	$1.5e^{-5}$
γ	1.0

Table E.2: Simulation parameters.

The simulation parameters for the scenario with correct nodes' fixed policy are given in Table E.3.

The simulation parameters for the self-play with additional values of -2 and 2 are given in Table E.4.

Network size:	$[64, 64]$
Train batch size	8192
Mini-batch size	1024
Iterations	200
Max rounds	100
Learning rate Byz	$1.25e^{-5}$
γ	1.0

Table E.3: Simulation parameters.

Network size:	$[64, 64]$
Train batch size	8192
Mini-batch size	1024
Iterations	100
Max rounds	100
Learning rate Corr	$1.5e^{-5}$
Learning rate Byz	$1.25e^{-5}$
γ	1.0

Table E.4: Simulation parameters.

Simulation parameters in scenario with two byzantine nodes is given in Table E.5.

Network size:	$[128, 128]$
Train batch size	8192
Mini-batch size	1024
Iterations	100
Max rounds	100
Learning rate Corr	$2.5e^{-5}$
Learning rate Byz	$1.25e^{-5}$
γ	1.0

Table E.5: Simulation parameters.