information

Article

Anticheat System Based on Reinforcement Learning Agents in Unity

Mihael Lukas [1,†], Igor Tomicic [2,*,†] and Andrija Bernik [3,†]

[1] Novena d.o.o., 10000 Zagreb, Croatia; mihael.lukas@novena.hr
[2] Faculty of Organization and Informatics, University of Zagreb, 10000 Zagreb, Croatia
[3] Department of Multimedia, University North, 42000 Varazdin, Croatia; abernik@unin.hr
* Correspondence: igor.tomicic@foi.unizg.hr
† These authors contributed equally to this work.

Abstract: Game cheating is a common occurrence that may degrade the experience of "honest" players. It can be hindered by using appropriate anticheat systems, which are being considered as a subset of security-related issues. In this paper, we implement and test an anticheat system whose main goal is to help differentiate human players from AI players. For this purpose, we first developed a multiplayer game inside game engine Unity that would serve as a framework for training the reinforcement learning agent. This agent would thus learn to differentiate human players from bots within the game. We implemented the Machine Learning Agents Toolkit library, which uses the proximal policy optimization algorithm. AI players are implemented using state machines, and perform certain actions depending on which condition is satisfied. Two experiments were carried out for testing the agent and showed promising results for identifying artificial players.

Keywords: security; artificial intelligence; infosec; reinforcement learning; agents; games; gaming; unity

Citation: Lukas, M.; Tomicic, I.; Bernik, A. Anticheat System Based on Reinforcement Learning Agents in Unity. *Information* **2022**, *13*, 173. https://doi.org/10.3390/info13040173

Academic Editor: Nelly Leligou

Received: 28 February 2022
Accepted: 25 March 2022
Published: 29 March 2022

1. Introduction

This research proposes a contribution to the domain of game anticheat systems through the implementation of reinforcement learning agents. The goal of the research is to implement and test a functional self-learning agent that is able to detect a nonhuman player based on artificial intelligence in real time within the game.

The anticheat system here was built inside the third-person perspective game. The game consists of two teams, where the player's goal is to transfer five boxes to the opposite side faster than the rival team in a limited time frame. Unity was used to develop the game, and the implementation of the reinforcement learning agent was accomplished using the Machine Learning Agents Toolkit (ML-Agents) library [1]. The main goal of reinforcement learning is to have an intelligent agent that acts in the environment by maximizing its reward. In order to accomplish that goal, the agent needs observational data and rewards. If the agent does what it is required to do, it receives a positive reward, and a negative reward if it fails to do so.

Reinforcement learning is advancing quickly as a field, and is used in many real-world examples. The computer learns to drive a car on its own, helps with trading and finances, processes natural language, and it is used in healthcare, games, robotics, and other application domains [2].

2. Related Work

There are several papers related to cheat detection within multiplayer games using machine learning. One of the papers studied only the logs of the game, which means it did not intrude the privacy of the player, but studies on files that are written when playing on the server. While reading logs, they must extract the most important properties on which their model learns to use supervised learning [3]. Within this paper, collecting logs from

the server are not needed. With reinforcement learning, the agent continuously studies all players and, with the help of certain rewards, figures out who is cheating.

The following paper demonstrated cheating detection by using game outcomes [4]. The authors proposed a general framework for various types of games, where one needs to know the current rank of the player. It is assumed that the cheating player has an artificially increased rank, and with such an assumption, the authors tried to detect the cheater by calculating probabilities. This is an interesting way of cheat detection, but it has some limitations, for example, some games do not have ranks. In this paper, the proposed solution is more generalized in a way that it can be used in any Unity game.

The paper [5] is about finding bots within the game Quake II by analyzing their trajectories. The work is based on the fact that the player almost never uses a similar or identical path, while the bot does. Using these settings, the authors tried to find the difference between a human player and a bot. By training the reinforcement learning agent, it is possible to detect a human player assisted by artificial intelligence. An agent can understand that a player has much faster reactions than those of other players, that a player has access to more data than other players have, and so on. It is necessary to properly set the problem and the environment in which the agent learns. Players using illicit aids are not detected within this work, but the agent can certainly be adjusted to find them.

Another cheat detection by using machine learning is performed in Unreal Tournament III [6]. The authors created a framework consisting of a server that collects data, a preprocessing backend, and the frontend that analyzes the data. Supervised learning was used with five different techniques: decision trees, Naive Bayes, random forest, neural networks, and support vector machines. Results showed that the framework could handle most of the test scenarios. There are some similarities with their paper and our research, but since there are many data that a computer needs to process, a reinforcement learning agent might prove better in some domains. The authors showed promising approaches for cheat detection and the number of various techniques used for learning.

Khalifa, in his paper, analyzed cheating inside FPS games [7]. The chosen game was Counter-Strike Global Offensive, where the main task is to find players that use aimbots. To achieve this, data from each player are analyzed, generating features and utilizing Markov models to see if the player is cheating. Since the main goal in this paper is to find the player using the aimbot, it studies certain things on the basis of the player's direction of viewing. This method is specific because it only explores the player's aim without any other forms of cheating. As already said, in this paper, the agent only distinguishes a bot from a human player, but can be adjusted to perform more complex cheat detection, such as detecting aimbots and forbidden scripts.

Staying on the game Counter-Strike Global Offensive, Willman was trying to identify aimbot players using neural networks [8]. Khalifa was analyzing logs from the game, but with a different approach. The aim of his paper was to create a machine-learning algorithm that would achieve cheat detection accuracy over 90%, but only 50% was achieved. Some problems are similar to this paper, as he stated that more training data were needed to achieve better results. When using reinforcement learning, the longer the training is, the better the result. However, an agent with reinforcement learning should also be able to detect more than just aimbots if that is intended.

Islam and others used another interesting way of finding cheats [9]. The goal was to train a machine-learning model observing network traffic. Their dataset was also from Counter-Strike, but 1.6 and not Global Offensive. In their experiments, they attempted to find out if a player is cheating by aimbot, speed hack, or wall hack. Their paper covers a slightly larger context than other papers do, considering they attempted to detect three different cheats, but it is also similar to other papers because they are trying to determine if a player is cheating using logs, while in this paper, a reward-driven agent is just dropped inside the environment in which it needs to learn what its goal is. So, in order to do that correctly, the length of training is important.

Platzer, in his paper, created a sequence-based solution to detect bots inside massive multiplayer online games (MMOG). The solution was implemented inside the game World of Warcraft. The approach for finding a bot is based on a combat sequence when it fights with an enemy. The bot probably uses repeated combat actions and the human player probably does not. If someone uses the bot as a tool to achieve an advantage, their account is not able to level up. Results showed that only 10 of 70 people leveled up in one month. None of the human players was detected as a bot, which is impressive [10]. Results show that a sequence-based solution can reliably differentiate the human player from the bot, and the importance of bot detection in some multiplayer games. This algorithm cannot be upgraded to detecting other cheats, but a similar solution can be used in other games in order to detect bots. The approach of this paper is that the agent needs to learn by itself what it needs to do and how. Because of that, if training is not long enough, the result could be a bit worse, but it is surely able to perform its task.

The following paper did not address finding cheats, but focused on training the reinforcement learning agent to play the game Doom. To allow for an agent learn, the authors used positive rewards for kills and object pickup, and negative rewards for suicide, losing health, and losing ammo (shooting). After configuring the hyperparameters, the training started, and after training for 65 h, significant results were achieved [11]. Rewards need to be properly determined, so that the agent can learn more efficiently. In this paper, the agent uses simple rewards, where positive reward are given if the agent correctly distinguishes a bot from a human player or human player from a bot, and a negative reward if the agent does it wrong. Naturally, the longer the training is, the better the final results.

3. Methods and Implementation

While creating initial sketches of the application and contemplating about implementing reinforcement learning, we created a simple game in which reinforcement learning agents would be implemented. In short, there would be a security agent who could distinguish a bot from a human player. Implementation steps were the following:

- the development of a multiplayer game;
- the implementation of state machines (bots);
- the implementation of security agents using ML-Agents.

3.1. Game Implementation

The first steps in developing a simple game without complicated logic serve as a good template for an agent to learn. This concept of an agent can be implemented in any Unity game, but for each game, the agent needs to be trained. The game is called "Courier" and is a multiplayer game that consists of server and client sides tested only on Windows platforms and a local area network. The main goal of the game was to transfer five boxes to the opposite side faster than the rival team. One round lasts 185 s, and if neither of the teams transfers five boxes in that period, the winning team is the one that carried more boxes. If the score is even between teams, the result is a draw. The map is mirrored and contains a water and maze area. If a player falls into water, they respawn at the starting location. For additional complexity, artillery was added on each side that can hit players from both teams. If the player is hit by a missile, they are respawned. The maximum number of players is 10 (5 per each team), and the number of boxes is 12 per team. If the box falls into the water or out of the map, it respawns after two seconds. Players are randomly spawned at certain locations near their base.

The map was created using the libraries ProGrids [12] and ProBuilder [13] inside Unity. These libraries allow for the easy creation of maps and their mirroring. There are no materials on the map, but only on the required elements like player, boxes and similar. Controls are simple, using W, A, S and D for movement, SHIFT for faster running, and the left mouse click for picking up and dropping boxes. Players cannot collide with each other, but boxes can collide with other boxes and missiles. Figure 1 shows the game map and an in-game screenshot.

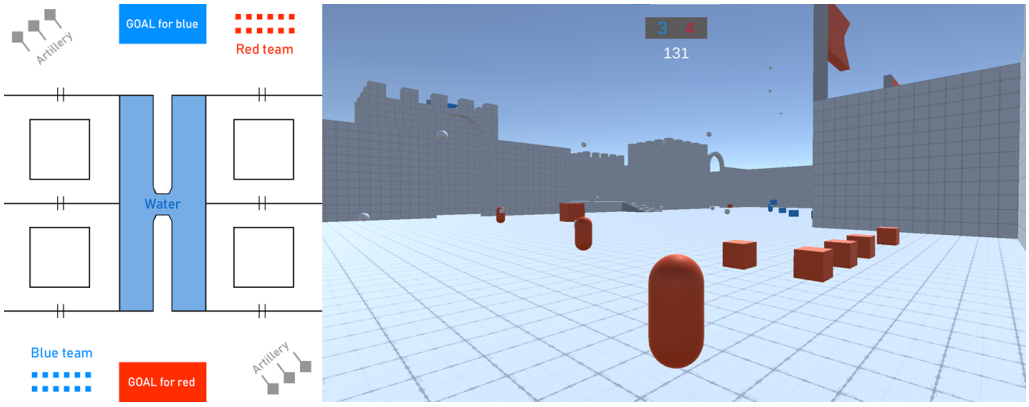

Figure 1. The game map (**left**) and an in-game screenshot (**right**).

3.2. Implementation of Bots

After the new game had been developed, bot implementation was required. For this purpose, state machines were used where the bot has a certain number of states, and upon completion of one state, it passes to another state [14]. All transitions between bot states are shown within Table 1.

Table 1. Transitions between bot states.

No.	Current State	Next State	Condition
1.	Search	Go toward box	If box found
2.	Go toward box	Search	If stuck for more than 0.5 s
3.	Go toward box	Pick up the box	If near box
4.	Pick up the box	Search	If box is not picked
5.	Pick up the box	Go to the destination	If box is picked
6.	Go to the destination	Drop the box	If arrived at the destination
7.	Drop the box	Search	If not carrying the box
8.	From any state	Dead	If health is zero or less than zero
9.	Dead	Search	If health is greater than zero

Each bot has six states. The first state when spawned on the map is 'Search'. In this state, the bot checks which box is the closest. When it finds the box, the condition 'If box found' is satisfied, and the bot goes in next state, which is called 'go toward box'. All bot movements are produced by a Navigation mesh agent (NavMeshAgent) that is attached to each bot, allowing for it to travel through a baked map [15]. If the bot is stuck or does nothing at the state 'go toward box' for more than half of a second, it starts searching for another box. If close enough to a box, it picks it up. Sometimes it happens that the bot just starts to pick up the box, but before someone else takes it, which results in blocking the bot. Therefore, transition number 4 is implemented, where if the bot failed to pick up the box, it starts to look for a new one. After the bot manages to pick up the box, it moves to condition 'go to the destination'. The destination is the area where the box needs to be placed in order for the bot to win one point for its team. When it reaches its destination, it enters the state 'drop the box' and wins a point for the team. When the bot drops the box, it goes to the 'search' state and everything repeats.

While in any state, a bot can die from missiles or by falling down into the water. If the bot has zero or less health, it goes to the 'dead' state. After respawning, it return to full health and transitions to the 'search' state.

3.3. Implementation of Security Agents (ML-Agents)

The implementation of security agents is achieved using ML-Agents, which is an open-source library for Unity based on Pytorch. According to the official documentation, there are three supported reinforcement learning algorithms: proximal policy optimization

(PPO), which is used in this paper, soft actor-critic (SAC), and multiagent posthumous credit assignment (MA-POCA) [1].

3.3.1. Proximal Policy Optimization

Proximal policy optimization is one of the most successful algorithms in reinforcement learning [16]. The algorithm uses a neural network to approximate an ideal function that maps the agent's observations to the best action that the agent can bring in a given state. The PPO algorithm is implemented within TensorFlow that communicates with Python processes [17]. More specifically, this algorithm serves to optimize step updates within a gradient descent, where it was proven to be a better solution than a standard gradient descent. It allows for more stable and faster training [18]. More about the configuration of PPO algorithm can be seen in Section 3.3.4.

3.3.2. Observations

Each agent is assigned to each player, but all agents share the same "brain", meaning that they are learning one from another. If one agent learns something on the certain player, the other agents use it [19].

In order to allow for a security agent learn, it needs to observe the environment and players. A reward-driven agent is able to reach decisions on these observations. On one player, the agent next observes:

- local position;
- local rotation on Y axis;
- velocity;
- is player carrying a box;
- box positions of your team and rotations;
- blue team score;
- red team score

Bots and players are implemented in such a way that they can be recognized within the server, but the agent only receives movement data from the player, regardless of whether he is a bot or not, and according to this information, the agent must learn the difference on its own.

3.3.3. Rewards

The agent uses discrete actions, meaning that only three possible decisions are available. Those decisions are in values of 0 (true), 1 (neutral), and 2 (false). If the agent marks the player with decision 0, then it concludes that the player is a bot. Decision 1 states that the player is neither a bot nor a real player, giving a neutral decision. Otherwise, if the agent reaches decision 2, then it concludes that the player is a human player.

In order to allow for the agent learn, it is necessary to provide rewards. An agent's main goal is to maximize the reward. If an agent concludes that the human player is indeed a human player, it receives a positive reward (+1), but if it concludes that the human player is a bot, then it receives a negative reward (-1). The same works for the bot: if the agent reaches a decision that the bot is a bot, it receives a positive reward (+1); if it concludes that the bot is a human player, it receives a negative reward (-1). If the agent chooses to remain neutral, it is not rewarded. The agent can wrongly identify a bot as a human player (false negative) or a human player as a bot (false positive). When the agent initiates one of these conditions, it receives a negative reward, steering it away from such assessments. If the agent correctly identifies a human player or a bot, it receives a positive reward, and there is increased tendency to repeat such actions.

3.3.4. Training Configuration

A basic configuration is needed before proceeding to training. This is achieved using a YAML configuration file that contains information about the used reinforcement learning algorithm and its parameters, which are analyzed in the following sections. It was

important to correctly adjust the parameters, because training might otherwise not have given results consistent with our goals. While testing the training, several configurations were conducted. The main goal was to straighten the line of cumulative reward as much as possible, so that there were no excessive deviations from the curve. The main reason for this is more stable training updates, which allow for better training efficiency. After a few versions of configurations, the following was created and used in training.

```
behaviours:
  ACSystemDiscrete:
    trainer_type:  ppo
    hyperparameters:
      batch_size:  32
      buffer_size:  10240
      learning_rate:  0.0003
      beta:  0.005
      epsilon:  0.2
      lambd:  0.95
      num_epoch:  3
      learning_rate_schedule:  linear
    network_settings:
      normalize:  false
      hidden_units:  128
      num_layers:  2
      vis_encode_type:  simple
    reward_signals:
      extrinsic:
        gamma:  0.99
        strength:  1.0
    max_steps:  500000
    time_horizon:  64
    summary_freq:  10000
```

The mostly adjusted parameters were batch size, buffer size, learning rate, and beta. ML-Agents have typical range values for each parameter, so some of them were left as default. ACSystemDiscrete is the name of the YAML file configuration that needs to be equal to 1 inside Unity. The next parameter is trainer type, which shows the used algorithm, in this case proximal policy optimization (PPO). Parameter batch size is the number of experiences in each iteration of the gradient descent. Several changes were conducted, but leaving it at the default value proved to work best. Since buffer size needs to be multiple times greater than batch size, after some testing, it was set to 10,240 and proved to be the most stable during training. Buffer size is the number of experiences that must be collected before updating the policy model [20]. While testing, entropy was monitored all the time, which shows the randomness of the agent actions [21]. If entropy is high, the agent is conducting random actions in order to conclude and learn something, but if entropy stays high all the time, the agent is not able to learn anything from it. Because of that, parameter beta needs to be properly set since it shows the strength of the entropy regularisation. The last one edited was the learning rate, showing the strength of update for each gradient descent step [20]. After testing out the parameters, the training of agent was able to start.

3.3.5. Description of Agent Decisions

Above every player within the game world, there is a description of agent decisions. In the first row, the latest agent decision is written (true, neutral or false) and in the second row is a number which represents the percentage of player cheating. The last 3000 decisions from the agent are taken, and only the number of positive decisions is

extracted. For example, if an agent has marked a player with a positive decision 2700 out of 3000, the following calculation gives the percentage of cheating on that player.

$$\frac{2700}{3000} * 100 = 90\% \quad (1)$$

Therefore, the probability of cheating is 90%, and it can almost certainly be concluded that the player is a bot. Since the percentage varies all the time, each trained model has a representative result that is taken as the mean value of ten rounds. A percentage is taken each second, and the mean value is calculated at the end. This description is only shown at the server side of the game and is not visible to the players.

3.3.6. Testing

Since the training is a time-consuming process, it is important to ensure that the game functions properly, and that the agent can perform its activities. ML-Agents could implement a heuristic where the user controls the agent by testing its functionalities. Keyboard controls were assigned to the agent and after testing out the decision-making process, and ensuring that everything was working properly, the experiments were initiated.

4. Experiments

All experiments were monitored by Tensorboard, which enables the continuous observation of statistics of the model and was mostly used during configuration of the YAML file in order to get better results. Each training was analyzed in the next sections using Tensorboard [22].

Training was performed in two parts, 1v1 and 2v2. For training initialization, a command line program called Anaconda was used, allowing for simple package management [23]. In order to start training in ML-Agents, one command was used inside Anaconda:

```
mlagents-learn config/ppo/ACSystemDiscrete.yaml -run-id=ACS_RLU_v0.72
                                -time-scale=1
```

The first parameter is the path to the YAML file and the second one is the ID of the run. Each run has a unique ID that is shown inside the Tensorboard. Moreover, there are certain parameters that can be added, for instance, the time scale that allows for training at normal speed. By default training, time scaling is set at value 20 and accelerates the process of training [24]. In this case, this is not desirabl, because human players are participating in training.

After the end of each training, the model is generated with file extension ONNX. This file can be inserted in Unity, allowing for an agent to work in production. Generated models are used later for result evaluation.

4.1. Training 1v1

In the first training, there was a bot in the blue team, and a human player in the red team. No swapping between teams was made during the training. It lasted for 1 h 51 min and 17 s, and the cumulative reward is shown in Figure 2.

With almost every step, the agent increased the reward, meaning that it acquired some level of knowledge at the end to distinguish the bot from the real player. The agent started with the negative reward, because it was randomly trying to take actions. After the initial tryouts, the reward increased steadily every 10,000 steps.

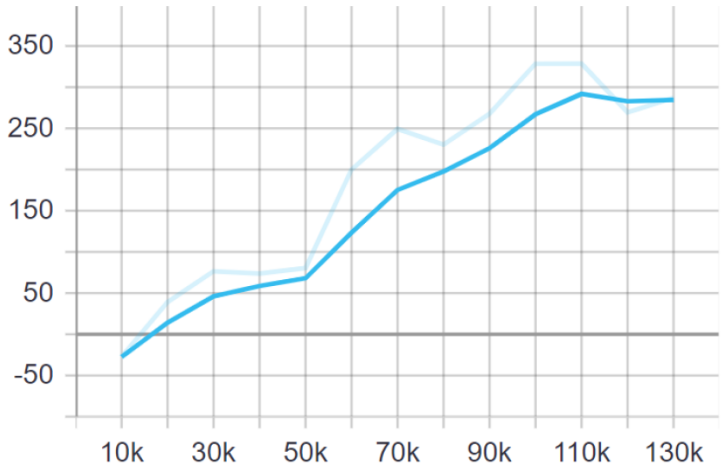

Figure 2. Cumulative reward of 1v1 training (Tensorboard).

4.2. Result Evaluation for 1v1

In order to obtain representative results from the model, five normal games were played; afterwards, the players swapped the teams and played five more games. While training, a human player was inside the red team and a bot in the blue team, and in that order, the first test was carried out. After that, player and bot changed teams, and the second test was performed. Both tests are shown in Table 2.

Table 2. Result of 1v1 training.

	Blue Team		**Red Team**	
	Player	**Cheating Percentage**	**Player**	**Cheating Percentage**
Test 1	Bot	84.15%	Person	12.18%
Test 2	Person	71.92%	Bot	6.34%

For Test 1, the cheating percentage for a bot was 84.15%, while for a real player it was 12.18%. Since the training was short, the obtained results were promising, as the agent could distinguish a bot from a human player. However, within Test 2, the cheating percentage of the human player in the blue team was 71.92%, while that of the bot in the red team was 6.34%. This indicated that the agent was not working properly. The reason for this was because training occurred only when the player was in the red team and the bot in the blue team. Therefore, the agent concluded that the real player would most certainly play in the red team, and the bot in the blue team. The bot in the training was also only using one side of the map, which was the closest path to score for its team, while the real player was using multiple paths. Because of that, if the real player went to the side on which the bot mainly was, the agent would mostly mark him as the bot. For these reasons, it was necessary to change the way of training.

4.3. Training 2v2

Within this training, there was one human player with one bot in the blue team, and one human player with one bot in the red team. The same training model was used, and the training continued. So, the cumulative reward of both training is shown in Figure 3, where the first training is marked with blue and the second training with red.

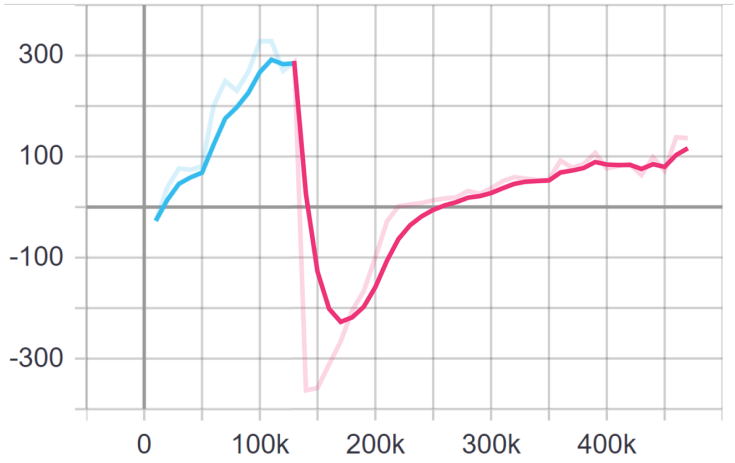

Figure 3. Cumulative reward of 2v2 training (Tensorboard).

The start of the 2v2 training was at 130,000 steps. Shortly after, the cumulative reward fell to −227.5, meaning that the agent found itself in a new environment where more players existed. Quickly after that, the reward started to climb continuously, and after 250,000 steps, it was positive. This training lasted for 3 h 26 min and 56 s with 340,000 steps and a cumulative reward above 100.

The cumulative reward in 1v1 grew faster because the agent had a clear situation, where one bot was in the blue team and one person in the red team. In 2v2 training, it was a bit more complicated to distinguish the person from the bot because the walking path of the person interfered with the bot's path.

Figure 4 shows parameters of beta and learning rate. They both continuously decreased, meaning that the agent was continuously learning in both training without major exceptions.

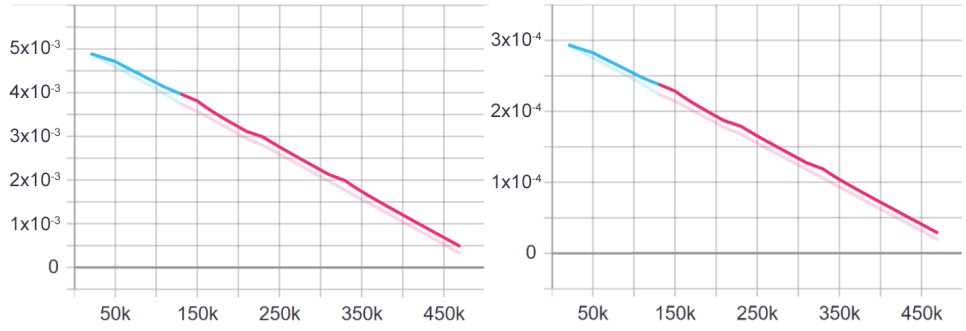

Figure 4. Beta (**left**) and learning rate (**right**) (Tensorboard).

The random actions of the agent are shown in Figure 5.

After 1v1 training or 130,000 steps, entropy was at the lowest point, meaning that agent decisions were not random. When 2v2 training started, the agent's rewards dropped really quickly, so it needed to change its behavior. Because of that, more random actions were taken in order to find a pattern in the behaviour of players. After 250,000 steps, entropy began to fall, meaning that the agent gained certain knowledge to distinguish bots from real players.

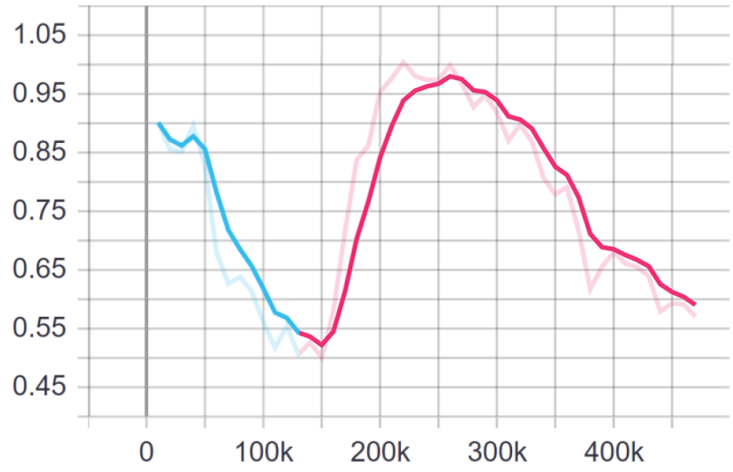

Figure 5. Entropy (Tensorboard).

4.4. Result Evaluation for 2v2

Same as for 1v1 result evaluation, ten games were being played. In the first test, five games were played and then, after the players had swapped teams, another test with five games was carried out. Tests are shown in Table 3.

Table 3. Result of 2v2 training.

	Blue Team		Red Team	
	Player	**Cheating Percentage**	**Player**	**Cheating Percentage**
Test 1	Person 1	28.98%	Person 2	14.28%
	Bot 1	38.43%	Bot 2	67.78%
Test 2	Person 2	17.90%	Person 1	31.72%
	Bot 2	40.54%	Bot 1	62.61%

From the tests, it can be concluded that the agent had a certain ability to distinguish the bot from the human player. The main problem was within the blue team. Since 2v2 training continued from 1v1, and the bot played on the blue team during 1v1 training, the performance of the blue team was less effective than that of the red team. If the training had lasted for longer or if had not continued on 1v1, these results would have certainly been better. However, inside the red team, the percentage of cheating for bots was more than 60%, while for human players, it was below 32%.

One of the reasons why the percentage of human players differed is because the person in the red team (Person 2) was more familiar with the game, allowing him to play much faster than the person from the blue team (Person 1). Since Person 1 had more similar movements as bots in the beginning, the agent increased the cheating percentage.

5. Discussion

According to the results of this paper, the main goal was achieved: to develop an agent that is able to identify a human player from a bot. It is possible to configure this kind of agent in order to find other ways of cheating, such as finding the players using illicit aids (scripts). The agent could also have more than one goal, such as distinguishing bot from the human player and detecting the aimbot at the same time. in order to obtain better results from the training, the agent can be improved by finetuning the hyperparameters of the YAML file. The agent configuration that was used within this paper could be used in other Unity games, but at this stage, the agent would need to be retrained for each game instance.

One of the goals of related future research would involve a more general solution that could be transparently applied in more than one game. At this stage of research, however, the proposed solution was implemented as a proof of concept tied to a single domain, but with a tangible potential to continue and build more scalable implementations from this point. Moreover, it would be interesting to explore the application of the proposed solution to other, seemingly disjointed areas of research, where there is a need to identify some sort of anomalous behaviours or patterns, or differentiate between two types of certain dynamics.

6. Conclusions

With the continuous growth of the gaming industry, security demands are also on the rise. Gaining an advantage over other players is always the goal of a multiplayer game, but when proven difficult, some players turn to illicit ways. One of the possible scenarios is to create an artificial-intelligence entity that can play the game instead of a human player. The computer can play day and night without the need to rest. For such scenarios, we implemented an anticheat system based on a reinforcement learning agent, with the main goal of detecting bots among human players. The system was tested in the Unity environment; the game Courier was developed solely for this purpose.

In a relatively short time, the agent was able to distinguish the bot from the human player: it took around five hours of training to achieve these results. To boost the performance of the agent, more time would be needed for the training itself. With the currently trained model, it can be concluded that a player is a bot if the percentage of cheating is over 38%; otherwise, if it is lower than 32%, then the player is not a bot. These two values are close to each other, but by only looking at the red team, a player is a bot if it has more than 60% cheating percentage. The training would have had significantly better results if it had not continued from the initial training, which biased the final results.

One of the disadvantages of the proposed approach is that it takes time for an agent to be able to determine a bot from a human player with a high probability. This is probably the major restriction, because a reinforcement learning agent at the beginning "does not know what to do". The agent is reward-driven, meaning that it acts to maximize reward, but in order to obtain better results, much trial and error is needed. Library ML-Agents only exists within Unity and cannot be used inside any other game engine.

The next step for improving the proposed approach would include the training of the agent for a longer time and with larger number of players. In this way, the agent would have more data from the players, allowing for it to distinguish real players from bots with higher probability. In further research, the agent domain should be expanded to detect players that are using cheats such as scripting, aimbots, and other exploits. Since cheating is increasingly advanced, the approach including a self-learning agent might prove efficient for finding cheaters.

Author Contributions: Conceptualization, M.L. and I.T.; methodology, M.L. and I.T.; software, M.L.; validation, I.T. and A.B.; formal analysis, M.L.; investigation, M.L.; resources, M.L., I.T. and A.B.; data curation, M.L., I.T. and A.B.; writing—original draft preparation, M.L.; writing—review and editing, M.L. and I.T.; visualization, M.L.; supervision, I.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All data from this research is available via request: igor.tomicic@foi.unizg.hr.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. ML-Agents. ML-Agents Toolkit Overview. 2021. Available online: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md (accessed on 10 February 2022).
2. Mwiti, D. 10 Real-Life Applications of Reinforcement Learning. 2021. Available online: https://neptune.ai/blog/reinforcement-learning-applications (accessed on 25 January 2022).

3. Alayed, H.; Frangoudes, F.; Neuman, C. Behavioral-Based Cheating Detection in Online First Person Shooters Using Machine Learning Techniques. 2013. Available online: https://ieeexplore.ieee.org/abstract/document/6633617 (accessed on 5 February 2022).
4. Chapel, L.; Botvich, D.; Malone, D. Probabilistic Approaches to Cheating Detection in Online Games. 2010. Available online: https://www.researchgate.net/publication/221157498_Probabilistic_Approaches_to_Cheating_Detection_in_Online_Games (accessed on 5 February 2022).
5. Pao, H.K.; Chen, K.T.; Chang, H.C. Game Bot Detection via Avatar Trajectory Analysis. 2010. Available online: https://ieeexplore.ieee.org/document/5560779 (accessed on 5 February 2022).
6. Galli, L.; Loiacono, D.; Cardamone, L.; Lanzi, P. A Cheating Detection Framework for Unreal Tournament III: A Machine Learning Approach. 2011. Available online: https://ieeexplore.ieee.org/abstract/document/6032016 (accessed on 14 February 2022).
7. Khalifa, S. Machine Learning and Anti-Cheating in FPS Games. 2016. Available online: https://www.researchgate.net/publication/308785899_Machine_Learning_and_Anti-Cheating_in_FPS_Games (accessed on 15 February 2022).
8. Willman, M. Machine Learning to Identify Cheaters in Online Games. 2020. Available online: https://www.diva-portal.org/smash/get/diva2:1431282/FULLTEXT01.pdf (accessed on 16 February 2022).
9. Islam, M.; Dong, B.; Chandra, S.; Khan, L. GCI: A GPU Based Transfer Learning Approach for Detecting Cheats of Computer Game. 2020. Available online: https://ieeexplore.ieee.org/abstract/document/9154512 (accessed on 16 February 2022).
10. Platzer, C. Sequence-Based Bot Detection in Massive Multiplayer Online Games. 2011. Available online: https://ieeexplore.ieee.org/abstract/document/6174239 (accessed on 23 March 2022).
11. Lample, G.; Chaplot, D. Playing FPS Games with Deep Reinforcement Learning. 2017. Available online: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456/14385 (accessed on 21 March 2022).
12. Unity. About ProGrids. 2020. Available online: https://docs.unity3d.com/Packages/com.unity.progrids@3.0/manual/index.html (accessed on 6 February 2022).
13. Unity. ProBuilder. Available online: https://unity.com/features/probuilder (accessed on 6 February 2022).
14. Tadevosyan, G. Unity AI Development: A Finite-State Machine Tutorial. Available online: https://www.toptal.com/unity-unity3d/unity-ai-development-finite-state-machine-tutorial (accessed on 21 March 2022).
15. Unity. NavMesh Agent. 2020. Available online: https://docs.unity3d.com/Manual/class-NavMeshAgent.html (accessed on 6 February 2022).
16. Wang, Y.; He, H.; Wen, C.; Tan, X. Truly Proximal Policy Optimization. 2019. Available online: https://arxiv.org/abs/1903.07940 (accessed on 10 February 2022).
17. ML-Agents. Training with Proximal Policy Optimization. 2018. Available online: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-PPO.md (accessed on 10 February 2022).
18. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal Policy Optimization Algorithms. 2017. Available online: https://arxiv.org/abs/1707.06347 (accessed on 10 February 2022).
19. Mattar, M.; Berges, V.P.; Cohen, A.; Teng, E.; Elion, C. ML-Agents v2.0 Release: Now Supports Training Complex Cooperative Behaviors. 2021. Available online: https://blog.unity.com/technology/ml-agents-v20-release-now-supports-training-complex-cooperative-behaviors (accessed on 21 March 2022).
20. ML-Agents. Training Configuration File. 2021. Available online: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md#ppo-specific-configurations (accessed on 11 February 2022).
21. Lisowski, E. What is Entropy in Machine Learning? 2021. Available online: https://addepto.com/what-is-entropy-in-machine-learning/?utm_source=rss&utm_medium=rss&utm_campaign=what-is-entropy-in-machine-learning (accessed on 11 February 2022).
22. ML-Agents. Using TensorBoard to Observe Training. 2021. Available online: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md (accessed on 21 March 2022).
23. Anaconda. Data Science Technology for Human Sensemaking. Available online: https://www.anaconda.com/ (accessed on 11 February 2022).
24. Zhang, J. Ultimate Volleyball: A Multi-Agent Reinforcement Learning Environment Built Using Unity ML-Agents. 2021. Available online: https://towardsdatascience.com/ultimate-volleyball-a-3d-volleyball-environment-built-using-unity-ml-agents-c9d3213f3064 (accessed on 21 March 2022).