Hindawi
Complexity
Volume 2021, Article ID 9963018, 14 pages
https://doi.org/10.1155/2021/9963018

Research Article

A Novel Behavioral Strategy for RoboCode Platform Based on Deep Q-Learning

Hakan Kayakoku,[1] **Mehmet Serdar Guzel**,[2] **Erkan Bostanci**,[3] **Ihsan Tolga Medeni**,[4] **and Deepti Mishra**[5]

[1]*Aselsan Company, Ankara, Turkey*
[2]*Robotics Laboratory, Computer Engineering Department, Ankara University, Ankara, Turkey*
[3]*SAAT Laboratory, Computer Engineering Department, Ankara University, Ankara, Turkey*
[4]*Ankara Yildirim Beyazit University (AYBU), Ankara, Turkey*
[5]*Department of Computer Science (IDI), NTNU-Norwegian University of Science and Technology, Gjøvik, Norway*

Correspondence should be addressed to Deepti Mishra; deepti.mishra@ntnu.no

Received 11 March 2021; Accepted 8 July 2021; Published 16 July 2021

Academic Editor: Kalyana C. Veluvolu

This paper addresses a new machine learning-based behavioral strategy using the deep Q-learning algorithm for the RoboCode simulation platform. According to this strategy, a new model is proposed for the RoboCode platform, providing an environment for simulated robots that can be programmed to battle against other robots. Compared to Atari Games, RoboCode has a fairly wide set of actions and situations. Due to the challenges of training a CNN model for such a continuous action space problem, the inputs obtained from the simulation environment were generated dynamically, and the proposed model was trained by using these inputs. The trained model battled against the predefined rival robots of the environment (standard robots) by cumulatively benefiting from the experience of these robots. The comparison between the proposed model and standard robots of RoboCode Platform was statistically verified. Finally, the performance of the proposed model was compared with machine learning based-customized robots (community robots). Experimental results reveal that the proposed model is mostly superior to community robots. Therefore, the deep Q-learning-based model has proven to be successful in such a complex simulation environment. It should also be noted that this new model facilitates simulation performance in adaptive and partially cluttered environments.

1. Introduction

In the last decade, studies on machine learning and robotics have shown notable improvements compared to previous years with the evolution of technology and the demands of the sector. Research studies have shown that one of the most substantial application areas of machine learning applications is adaptive control. Adaptive control systems are able to deal with uncertainties without demanding any external intervention. Reinforcement learning (RL) is considered as a solution to this concept, which is a leading machine learning technique and inspired by behavioral science. RL, in essence, provides an agent to learn in a defined environment by trial and error using feedback from its own actions and experiences [1, 2]. RL is a learning discipline that allows an agent to gain self-thinking ability based on a rewarding and punishing mechanism. Aforementioned computer science literature was adapted from the science of psychology [3]. According to Skinner, a famous psychologist and behaviorist, behaviors are affected by the results [4]. In this context, reinforced learning is the process of shaping the behavior by controlling the results of the behavior. Skinner basically suggests changing behavior by punishment and reward where rewards are used to strengthen the desired behavior, and penalties are used to prevent unwanted behavior. This theory was used in accordance with the principles mentioned in computer science. It was

inspired by the idea that an agent could be conditioned to a particular movement or series of actions by giving a penalty or reward at the end of each move.

Q-learning, one of the most popular RL techniques, is a successful model and has a solid foundation in the Markov theory of decision processes [5]. This learning technique, in essence, utilizes the formal framework of Markov decision processes to describe the interaction between the agent and the environment in terms of situations, actions, and rewards. This framework is a simple way to represent the key features of most artificial intelligence- (AI-) based problems. Due to its simplicity, it has been applied to different fields, from robotic control to computer vision. Recently, it is also adapted in video games [6, 7]. The main purpose of this algorithm is to examine the actions that an agent will perform in the environment in which it is defined, to see the reward that the agent will win according to these movements and to act in accordance with the maximum reward. The prizes to be won in the environment in which the agent is defined and the places where the agent is expected or not are determined by the user beforehand, and these values are written in a reward table. The forward-looking experiences and moves of the agent are determined according to this table. The agent uses the experiences gained for each move applied to the award determined by the user to select the moves to be applied. It keeps these experiences in a table called "Q-table." This table is initially assigned to zero since the agent has no experience with the environment. This allows the agent to move randomly until finding a reward in the environment. Along with the first award estimated, the agent starts inserting the "q value" regarding the relation between the state and action into the Q-table. Subsequently, the agent starts to predict and reach the prize by maximizing the values of forward-looking moves at each iteration [8].

Despite having a strong and stable learning model, this will cause extreme growth of the Q-table to be created for situations and actions when the number of situations and actions increases excessively, thus causing problems in accessing and updating the table. For instance, for an environment using 10,000 states and 1000 actions, the size of the Q-table that the agent will access for each action to be performed must consist of 10 million cells, which causes problems in terms of both memory space and processing time. For the solution of this problem, it is envisaged to use an artificial neural network that can approximately determine the values obtained with the Q-table. Thus, by combining deep learning and Q-learning algorithms, the concept of deep Q-learning has emerged [9–11]. Primarily, deep Q-Learning is a Q-learning algorithm that employs deep neural networks to approximate the Q value essential for the agent to predict the move to be applied in the specified environment. Overall, a supervised learning approach is adapted to estimate Q values, which allows Q-learning algorithm to be efficient for not only discrete but also continuous space problems.

In recent years, it has been observed that the deep Q-learning concept has been used especially in video games and successful results have been achieved. Especially the article written by Mnih et al. [10] is a pioneer in this subject. Within the scope of the mentioned article, seven popular Atari 2600 games were trained with images obtained from the arcade games through the convolutional neural network using the deep reinforcement learning model, and agents that can play 7 games with very high scores were obtained [10]. This study has been revolutionary in terms of satisfactory results and the use of reinforced learning and convolutional neural networks. The concept mentioned in the study was later used in various fields such as image processing, computer vision, robotics. The aforementioned study motivates authors to adapt the deep Q-learning algorithm in a more complex simulation environment, involving multiagent systems and battling strategies. Accordingly, in this paper, a new model is designed and also a data acquisition strategy is developed for the RoboCode simulation environment using a deep Q-learning algorithm. RoboCode is an open-source war simulator program developed by IBM using Java programming language in 2001 [12, 13]. The purpose of this simulator is to program a war robot using the classes offered by the platform to the users in a two-dimensional environment and to measure the performance of the programmed agent by fighting these robots in the environment provided by this simulator which is called the arena. Each robot body consists of parts named as barrel and radar, and in simulation, users are provided to control these parts with certain interfaces. Considering the anatomy and mobility of the robot in question, the situation and action number of the simulation is more complex than "Atari 2600" games. In this respect, rules have been determined to optimize the number of situations and actions throughout the study. Evolutionary neural networks were created by applying reward and penalty formulas along with the determined situations and actions, and the network was trained using the screenshots of the simulation. The aim was to get the highest score from the simulation by applying the next movement of the robots with the q values formed with the trained network. Trained robots fought in the arena with the predefined robots in the RoboCode application which helped to test the proposed model by making score measurements.

The rest of the paper is structured as follows. Section 2 introduces the RoboCode platform and the existing literature, while Section 3 presents the proposed adaptive learning model and system for the RoboCode platform. Section 4 presents the results of the proposed model based on different scenarios and compares the results with the state-of-the-art robots. Section 5 summarizes the results of this study.

2. RoboCode and Background

RoboCode is a combat simulator developed on the basis of the Java programming language, the first version released by "IBM AlphaWorks" in 2001, where it is necessary to code a robot war tank in order to fight against the enemy robots in a determined war arena [14, 15]. In this simulator, the player is the programmer of the robot and, as a rule, does not have a direct effect on the robot in the simulator. Instead, the players run the customized model in defined environments called the arena by coding the robot's artificial intelligence

capabilities, in other words, how the robot on the battlefield behaves and reacts to events in an adaptive manner. Battles fought in the arena are instantly broadcasted on the screen. Figure 1 illustrates an example from the arena.

2.1. The Rules of the RoboCode Game. Figure 1 illustrates the RoboCode battle environment where robots fight with each other until the last robot remains in the arena. Robots can fight individually or join a robot group to fight other groups. A battle consists of several rounds and the winner of the game is the robot or the group having the highest score at the end of the war. Robots in the arena initially have "100" energy levels and are destroyed once the energy level drops below zero. Robots fight with each other using projectile shots. Each robot is equipped with a radar (local sensor) that can scan other robots up to "1200 pixels" away, which helps to get information about distance, route, speed, name, and energy about another robot entering its scanning area.

Radar, gun turret, and robot body can rotate 360 degrees independently. The firepower of the turret is the key variable. In this manner, the turret can damage target robots of different values depending on the firepower set by the player. The heating feature of the turret was also added to the simulation as a disadvantage of increasing firepower. Bullets with high firepower cause the robot's turret to heat up more, which means that the robot cannot fire for a while. In addition, low-power shells move faster, while high-power shells move slower. All battles take place in a rectangular arena of variable sizes surrounded by walls (see Figure 1). On the other hand, RoboCode has a limited physical structure, restricting some actions. These drawbacks, in essence, are deliberately added to the platform so as to make the game more realistic and complex [16]. For instance, the robot cannot move at full speed while rotating. Firing the weapon causes a robot to lose the same amount of energy as its firing power. If the bullet hits a target, the lost energy is recovered by three times the bullet power. Since it runs on a single thread, the operations of the robots defined in RoboCode are sequential. Consequently, a limited time is allowed for a robot to make its moves. When all robots have completed their sequence, the actions are repeated. This means that there must be a compromise between the complexity of the actions a robot can perform and the time elapsed. For this reason, the robot may lose in the arena if aiming to perform a complex action that requires extremely complex calculations and spends the time allocated to it to perform an action. RoboCode relies on a scoring system to determine the winner of a match. In order for a robot to be a winner, it is not sufficient only to survive on all rounds because robots can earn more points than the winner due to some offensive actions. A list of the rules for the RoboCode platform can be seen in [8, 9]. The flow diagram of the process cycle used by the RoboCode engine can also be seen in Figure 2.

2.2. Battling Strategy and the Literature. Strategies are the most critical guidelines for how to act in a particular situation. Effective strategies to be implemented in the arena are of great importance for international tournaments that are organized annually for the RoboCode war simulator [13]. It should be noted that, while a robot can perform well in a one-on-one battle, other adaptive strategies may be required for close combat. It is a general belief that a good strategy in melee combat in the RoboCode battle arena is to stay away from the battlefield and not draw attention at all [17]. With this method, other robots are expected to kill each other. Afterward, it is easier to kill the remaining robots due to possible damage. However, when the opponent has full energy, this strategy does not work equally well in one-on-one battles. Another argument against this strategy is the scoring system used in RoboCode. Due to the principles of the RoboCode simulator, the war is not won only on the basis of being the last agent standing. In addition to the simulator, the damage done and the number of robots skilled by the robot also has a huge impact.

Nowadays, RoboCode war simulation is accepted as a member of the serious game category [18, 19] and is employed as an artificial intelligence training tool in many parts of the world. These robots, which are developed by researchers, are published on the platform called "Robot-Rumble," where they score by fighting in arenas and are ordered accordingly. It is noted that, in these tournaments, robots having the highest scores are implemented based on AI algorithms. Apart from a few of the robots that belong to the platform called RobotRumble, the vast majority are applications that are not academically documented. The first academic papers in this area are based on the RoboCode simulator several years after the release date. First, Eisenstein worked on a robot based on genetic algorithms (GA) and explored how to best implement a combat robot based on assumption and behavior-oriented languages such as REX [20]. In the following years, it is seen that most of the AI robots developed for the RoboCode war simulators are designed based on genetic algorithms [18, 21]. Shichel et al. made the first attempt to introduce evolutionarily designed robots in international RoboCode competition [21]. In another study, a Bayesian network is defined to choose between an aggressive or defensive strategy [22]. On the other hand, particle swarm optimization (PSO), an efficient metaheuristic strategy, is also adapted for RoboCode agents [23]. In a similar study, a neural network model is trained by an adaptive PSO algorithm by considering different parameters so as to approximate optimum turning angle for a RoboCode agent [24].

Reinforcement learning (RL) is also applied for the tank battling scenario of the RoboCode platform. In a recent study, the performance comparison between the GA and RL algorithm is done using different scenarios defined for the RoboCode environment [25]. An improved Q-learning technique in Semi-Markov decision processes is validated by using the RoboCode environment in [26]. Q-learning is one of the leading off-policy RL algorithms, preferred in another recent study due to its efficiency and popularity. However, in this study, an artificial neural network is designed to approximate Q values instead of trying to keep them in a "Q-table," which is essentially not possible for such a continuous space problem [8]. The neural network has a very modest structure, involving only two layers. The first layer denotes

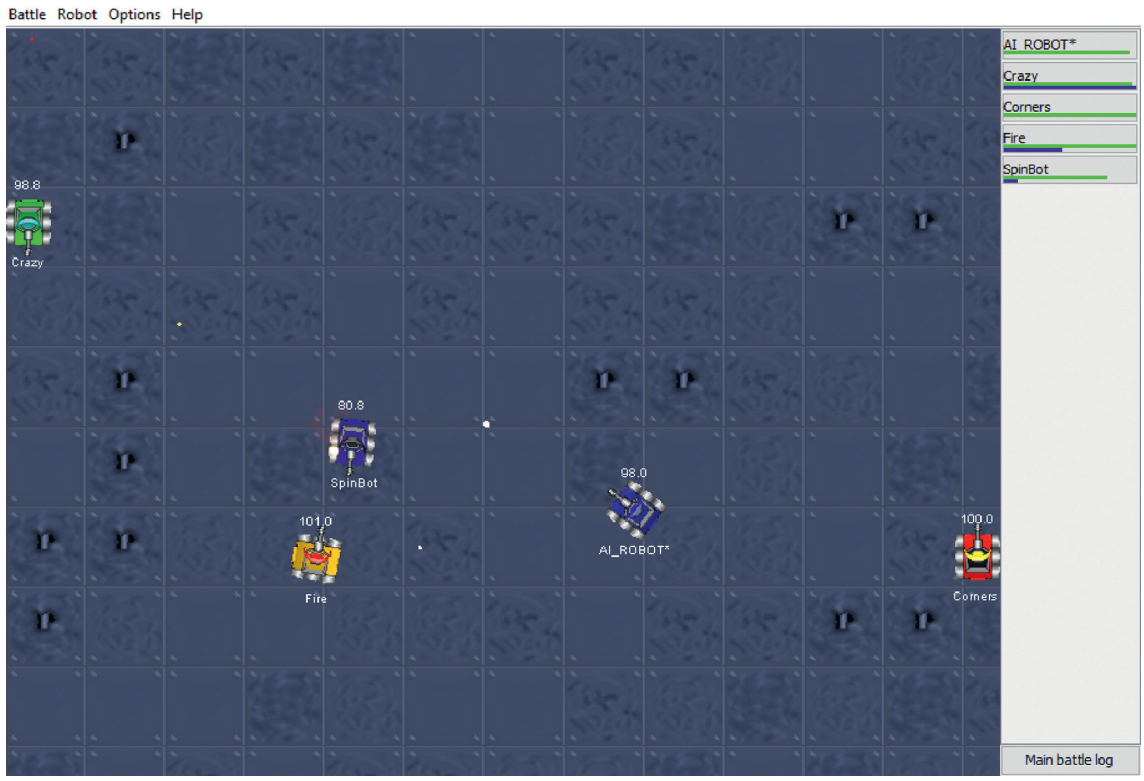


Figure 1: An example screenshot obtained from the RoboCode battle environment.

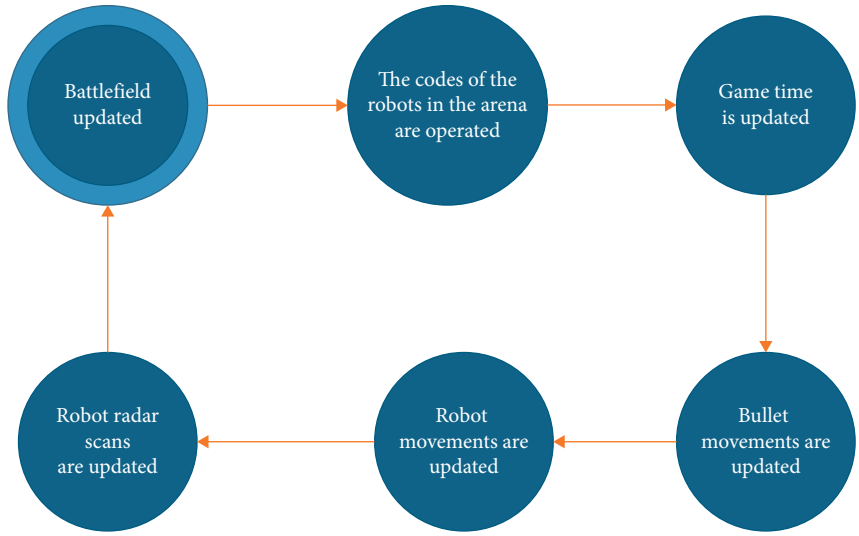


Figure 2: The flow diagram of the process cycle used by the RoboCode engine.

inputs, namely, "X position," "Y Position," "the distance of the robot to its opponent," "the bearing angle," "action value," and the "bias value," whereas the hidden layer is a fully connected layer and the output aims to approximate the "Q values." Consequently, despite some metaheuristic and machine learning based research conducted for RoboCode, Deep Q-learning based model has not been designed and applied to this problem. The model essentially gathers and processes images taken from the RoboCode simulator and then reinforced them using the deep convolutional neural networks without requiring any feature extraction process. According to the best of our knowledge and experience, there is no study and a robot model developed in this direction, which is one of the main contributions of this study to the field. It should be noted that RoboCode involves a larger and more complex environment than the environment of Atari Games, the efficiency of the model on those games has been previously proved by [10].

3. Methodology

With the latest developments in computer vision and speech recognition, it has been observed that deep neural networks are effectively trained in a wide range of training sets. The most successful approaches are trained directly from raw inputs using mild updates based on stochastic back gradient processing. It is often possible to learn better presentations than handmade features by feeding enough data to deep neural networks [27]. In addition to all of these, the studies that efficiently process educational data by connecting the reinforcement learning algorithm directly to the deep neural network working on RGB images and using stochastic backward gradient updates and thus developing agents that can play video games with superhuman performance have also been very successful [10, 11].

3.1. Q-Learning. Q-learning is a model-free and values-based reinforcing learning algorithm. The purpose of this learning model is to learn a policy that trains an agent on what action to take under what conditions. It does not require a model of the environment and can solve problems with stochastic transitions and rewards without requiring adaptation. For any finite Markov decision process, Q-learning finds an optimal policy, starting from the current situation, in terms of maximizing the expected value of the total reward of all consecutive steps. Given the unlimited discovery time and a partially random policy, Q-learning can determine the most appropriate action selection policy for any Markov decision process. The q-function used in the Q-learning algorithm uses the Bellman equation and takes two inputs: state (s) and action (a). The Q-function is represented by equation (1). The equation expressed in the formula refers to the expected total reward for a situation and action at the moment t, which is applied.

$$Q^{\pi}(s_t, a_t) = E\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} +, \ldots, |S_t, a_t\right]. \quad (1)$$

The Q values are kept in a state-action-size table called the Q-table, and the relevant table values are updated according to the reward value that the agent receives for each move it applies:

$$Q_n(s, a) = Q(s, a) + \sigma\left[R(s, a) + \gamma \max Q'(s', a') - Q(s, a)\right]. \quad (2)$$

Here, the value indicated by "σ" is the learning factor and it is a hyperparameter that determines how old the newly acquired information overrides. The value of Q (s, a) represents the current q value, whereas "γ" is the discount factor. The expression R (s, a) refers to the reward for performing this action in the specified state, whereas expression $\max Q'(s', a')$ defines the highest reward value for each action in the set of all possible actions for the current situation.

3.2. Deep Q-Learning. The agent can find the action in which it can get the greatest reward for the situations defined in the environment with the Q-learning algorithm. Although this is a simple but very powerful algorithm, creating a (*status x action*) size Q-table requires huge memory space and processing time, which is an expected case, especially for continuous space problems. This problem has been solved with the idea of obtaining q values using machine learning models such as deep neural networks instead of calculating q values and storing them in a q-table. In deep Q-learning, the deep neural network model is mainly used to approximate the q value function. This network is given as an input and the Q value of all possible transactions is output. By choosing the highest data among the obtained Q values, the agent learns. The comparison between Q-learning and Deep Q-Learning (DQL) algorithms is shown in Figure 3. DQL is revealed as one of the most successful learning models by compensating for the drawbacks of the conventional Q-learning algorithm.

3.3. Proposed Architecture. Tesauro's TD-Gammon [28] and Mnih's et al. [10] architectures are considered as a starting point for the proposed architecture. The aim of these architectures is to modify the parameters of a network that estimates the value function using the algorithm's interactions with the environment. Unlike TD-Gammon, in Atari architecture, the experience repetition concept, in which the network is trained by using the values of the past experiences of the agent, is used [29]. During the internal cycle of the algorithm with repetition of experience, the Q-learning updates or minibatch training updates are applied to the randomly drawn experience samples in this pool of repetition of stored experience and the network is trained. After the training, the agent selects an action in accordance with the epsilon greedy policy and carries out this action. The experience repetition method has many advantages [30]. First, in contrast to the experience repetition method, diversity is insufficient due to the strong correlations between the inputs in the direct learning model using sequential inputs. Randomizing the inputs prevents these correlations and therefore reduces the variance of network updates. In this way, repetition of experience averages many of the behavioral distributions of previous situations and prevents oscillations or deviations in parameters by softening learning. It has been seen that this concept gives successful results in Atari architecture. For this reason, repetition of experience was used in the learning of the deep neural network structure within the scope of this study. The algorithm used in this context stores only the last n series of experience in memory and randomly takes samples from this series while performing the updates. This approach is problematic in some respects because the memory buffer cannot distinguish important transitions and is always overwritten because of the finite memory length (n) [10]. Within the scope of the study, firstly, screenshots of the arena were recorded periodically for the input of convolutional neural networks, and it was decided to take action based on these images. Since RoboCode does not have any periodic screen capability feature, first, a program that can take screenshots periodically was implemented by running the RoboCode program. However, initial results prove that this process has several disadvantages. Due to the difference in frame rate (fps) between the RoboCode

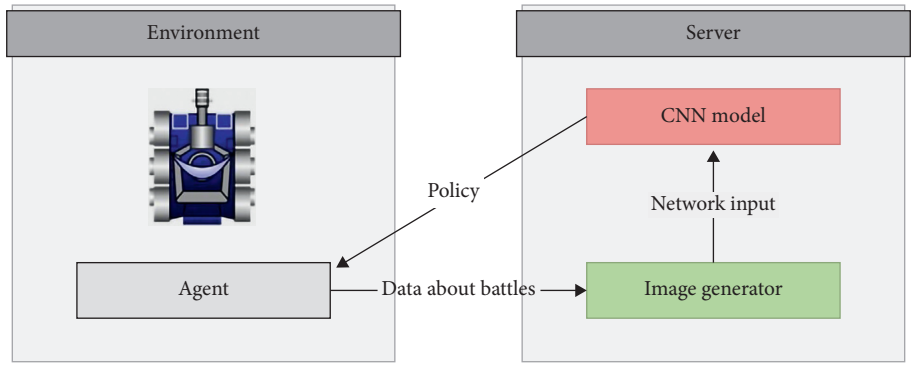


Figure 3: The flowchart diagram illustrating the relationship between agent and server.

arena and the developed program, it is not possible to store all frames of the arena. It should also be noted that training the model by taking a screenshot using the application's visual interface significantly increases the learning time of the model. Consequently, a program has been written that enables the image to be given as an input to the network using the position and direction information of the robots in the RoboCode arena and the position information of the bullets shot in the arena. With this program, images similar to the display of the arena were obtained, and the disadvantaged screenshots were eliminated. The screenshots that will be given as input to the evolutionary neural network architecture represent the current state of the RoboCode arena and have limited knowledge for training the network. In other words, it gives the positions where robots and bullets can be detected for screen input at time "t," but there is also a need for knowledge of the direction of movement of robots and bullets in order to escape bullets and carry out effective attacks. In order to obtain this information, the screenshot at "$t+1$" can be considered. Essentially, giving input to the convolutional neural network that will be defined by processing more than one frame at the same time will also protect the flow of information between these frames and make the network more powerful. For this reason, four consecutive frames (obtained from t to $t+3$ time instants) are employed in the model, designed, and given to the network as input.

One of the difficulties in raising a reinforced learning agent in the RoboCode application is that the agent works in a heavy sandbox, which is not guaranteed to be shared between games. Robots have limited access to resources on the host computer for security reasons and also to prevent robots from cheating in the game. Moreover, no external Java library is available. For this reason, it is not possible for robots to report the necessary information for the DQL model, such as instant locations, health conditions, points, and rewards, to any location on the host computer. For the solution to this problem, it was concluded that the training of the deep neural network model and the registration of the trained model were carried out outside the robot development process.

For the solution of the problem mentioned in the study, an out-of-process Python code has been implemented that periodically saves information from the network to a file using the "http" data protocol, by listening to the experience samples obtained from the agent during the war, storing them in memory, updating the learning model according to the inputs from the robot, using the status information sent by the robot to the robot, giving feedback to the robot by deciding what the optimal action is. With this program, the necessary communication with the robot takes place via the http server to train the network and implement the most appropriate action. After starting the game, the agent opens a UDP connection with the server and sends the inputs that will feed the model to the server. The agent also applies the response by questioning the action decision made by the model from the server for each situation in the war started in the arena. The study on how to overcome the limitations of RoboCode simulation is shown in Figure 3. In the scenario expressed in the figure, the agent sends the environmental data instantly to the server. Using this data sent by the agent, the server produces and displays the image needed by the convolutional neural network model. As a result of the given input, the most optimal action accessible by the model is sent to the agent via "UDP" connection through the server. Data about the war sent by the agent to the server for training are described as follows.

The corresponding parameters between the agent and the server are defined as follows:

(i) *Coordinates*. The agent sends the x and y coordinates to the server, involving the agent coordinates and the opponent robot location acquired by the radar scan.

(ii) *Direction*. Directions of the agent and the opponent robot, found as a result of radar scanning, are sent to the server.

(iii) *Bullet*. Agent sends the locations of enemy bullets to the server. The coordinates of the bullets launched by the agent are not included in the information sent to the server.

(iv) *Energy*. The agent sends the energies that he and the rival robot found as a result of radar scanning to the server. This information is used for reward signals, not image production.

The information sent by the robot to the model server is converted into images suitable for the convolutional neural network model with the image generator utility. The positions of the robots and the bullets that will make the agent

prevail in the arena and the direction of the robots should be included in the produced image. Unlike RoboCode, it was decided to produce robot objects in a triangle formation. The image presented by the RoboCode arena and its rendered state to be used as the convolutional neural network input is shown in Figure 4.

A basic preprocessing step is implemented to reduce input sizing to adapt the input to the convolutional neural network model. Raw images were first converted from RGB color range to gray scale and then downsampled to a lower resolution, 80×60. Since the ease of matrix operations required for the image inputs, suitable for the convolutional neural networks model, facilitates network design, it is more beneficial to work with pictures having a square frame [27, 31]. Since robots can access every point in the arena, cropping of the picture does not cause loss of information. Subsequently, the resolution is converted to 80×80 pixels. The action sets to be determined for an agent in the RoboCode arena can be very diverse. Within the scope of this study, the following action sets are determined.

(i) Turn 10 degrees to the right
(ii) Turn 10 degrees to the left
(iii) Move forward (5 pixels)
(iv) Move backward (5 pixels)
(v) 1-shoot bullet power

As mentioned before, the RoboCode environment allows different values for right-to-left turns, back-and-forth movements, and the power of the bullet. These values are fixed to simplify the problem. Although the RoboCode environment gives a score based on the actions taken at the end of the war, it is not possible to get this score directly and transmit it to the network. Since RoboCode cannot provide a direct reward signal, it is necessary to artificially create a reward signal. The most standard approach imaginable in this regard is to give a "+1" or "−1" reward only in terminal situations, depending on whether the agent has won the war or not. The preliminary work was to train the network using the first mentioned approach, but with this method, it was observed that the reward signal had a very rare frequency, and the agent could not receive enough signals to learn a reasonable policy. Once the general winning strategies applied in order to prevail in the RoboCode environment are examined, it is the general belief that the main way to be superior for robots in an arena is to have more energy than the rival robot. Therefore, it was thought that the main focus for the reward signal should be a function dependent on the energies of the robots. Considering the energies possessed, the differences between the energies of the robots at "t" moment give information about which robot is ahead on the way to win, but the energies of the robots at "t−1" are also considered to be included in the reward function. This means that the reward function also includes bullet drop information. The reward function containing the energies of the robots at the moments "t" and "t−1" is determined as follows:

$$R_t = A_t - A_{t-1} + R_{t-1} - R_t. \tag{3}$$

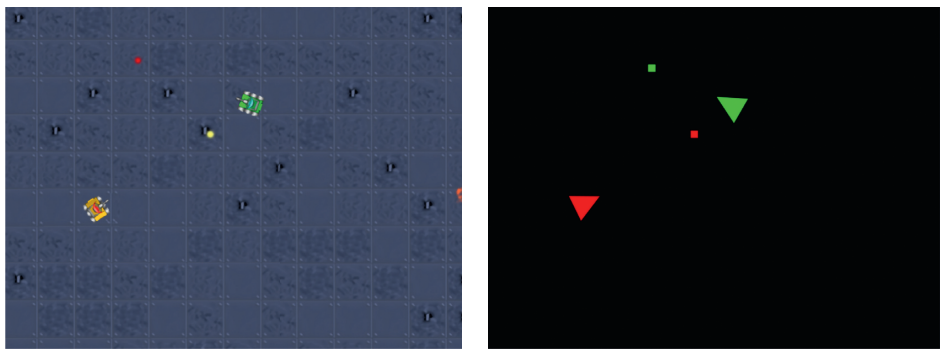
Here, A_t refers to the agent and R_t refers to the rival robot. This formula allows the agent to receive a consistent positive reward as it progresses over time. The discount factor (γ) value, illustrated in equation (2), is defined as "0.99." By considering this value, a higher priority is given to the prizes. The epsilon greedy policy was applied in the training of the agent within the scope of this study. It should be noted that the discount factor is a critical hyperparameter, regulating how much the RL agents care about rewards in the distant future corresponding to those in the immediate future. The value of the discount factor parameter is mainly adapted from [32, 33]. Experimental results validate the parameter value adapted for this study. In this context, the process of selecting the action of the maximum value of "q," which the neural network model applies when choosing an action, is limited by the possibility of "1-epsilon" selection. In this way, the model can choose random action as much as the possibility of epsilon. In this way, during the learning of the agent, it is prevented from converging to a point with randomness according to the value of epsilon.

The proposed network structure, illustrated in Figure 5, is created with "$80 \times 80 \times 4$" input referring 4 consecutive frames (images) having "80×80" resolutions. The first hidden layer of the convolutional neural network is composed of 16 "8×8" filters (4 strides) with an input image, each applying a nonlinear rectifier [34]. The second hidden layer involves 32 "4×4" filters with 2 steps and then applies the nonlinear rectifier once again. The last hidden layer of the network consists of a neural network model consisting of 256 rectifier units that are completely connected to each other. The output layer is a fully connected linear layer with a single output for each valid process. Networked outputs consist of 5 previously specified actions, namely, "turn 10 degrees to the right," "turn 10 degrees to the left," "go 5 pixels forward," "go back 5 pixels," and "shoot with 1-shot power." The structure of the specified model is illustrated in Figure 5.

4. Experimental Section

This section introduces the experimental work and also details the results. In order to train the models and perform the experiments, a computer of medium configuration is utilized. The computer has an Intel Core i7, 3.40 GHz CPU, and 8 GB RAM. It should be noted that the RoboCode simulation has some limitations on access to resources related to security reasons. In order to avoid these restrictions, a server that can communicate with the robot and transmit the future actions of the robot is established, and the robot is enabled to talk to this server.

The experimental section is divided into four sections, namely, experimental section with primitive robots, experimental section with advanced robots, and experimental section with community robots. The first two sections mainly involve experiments by using standard robots of the RoboCode platform. The third section presents the experiments against community robots. The final section presents the discussion and statistical analysis of experiments based on standard robots.



(a) (b)

Figure 4: A screenshot of the RoboCode arena (a). Input to the deep learning model (b).

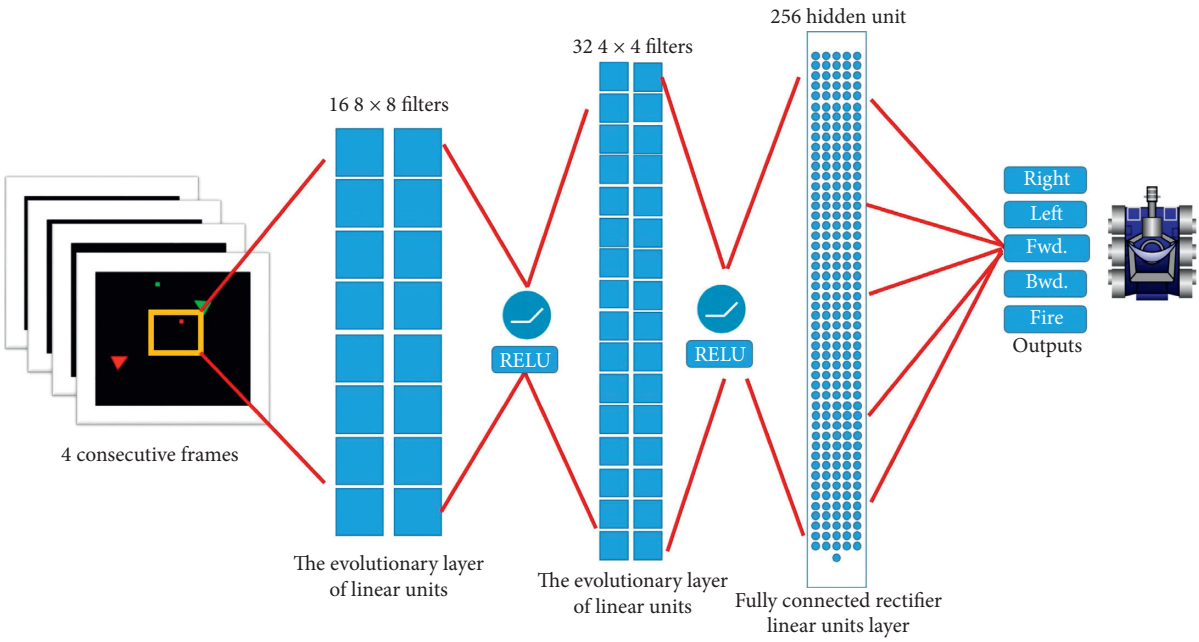


Figure 5: The proposed evolutionary deep Q network model for the RoboCode simulation environment.

4.1. Experiments by Using Primitive Robots of RoboCode Platform. Within the scope of the training process, it is aimed to train the convolutional neural network by choosing a predefined robot, "SittingDuck" that does not move and does not perform any projectile firing action. The weight values of deep neural network cells are recorded at the end of each round with input from the agents. These recorded values are reloaded with the start of the tour and robots are trained with previously trained models. In this way, it is aimed to increase the cumulative experiences of the trained model and achieve better scores in each battle played by the agent in the arena. Essentially, in this way, it is assumed that the robot, which fights with the rival robot and started to prevail, will tend to learn the correct bullet shooting action at the end of the training. For this reason, the SittingDuck robot was fought in the arena until %100 success was achieved by the agent robot, and the agent finally learned the targeted policy after over 4000 battles. The greedy strategy was followed during the training phase and the "ϵ" parameter was kept at "0.1." Another noteworthy point throughout the learning is that the robot intensifies the projectile firing action over time. In order to report this situation, all actions performed by the agent are stored and then visualized. The amount of actions that the agent has applied throughout the education is given in Figure 6, and the distribution of these actions in the iterations range is given in Figure 7.

The proposed model trained with the "SittingDuck" carried out approximately 4000 battles with the Corners robot in the same environment of RoboCode. However, as it is expected, it was observed that the model failed with the existing approaches against aggressive opponents like Corners robot. Essentially, the robot failed against the Corners robot due to not being educated against rival robots, which can fire bullets. Accordingly, several optimization techniques have been applied to train the network, detailed below. One of the critical learning challenges of a RoboCode agent is that the robot rarely reaches the reward even in the

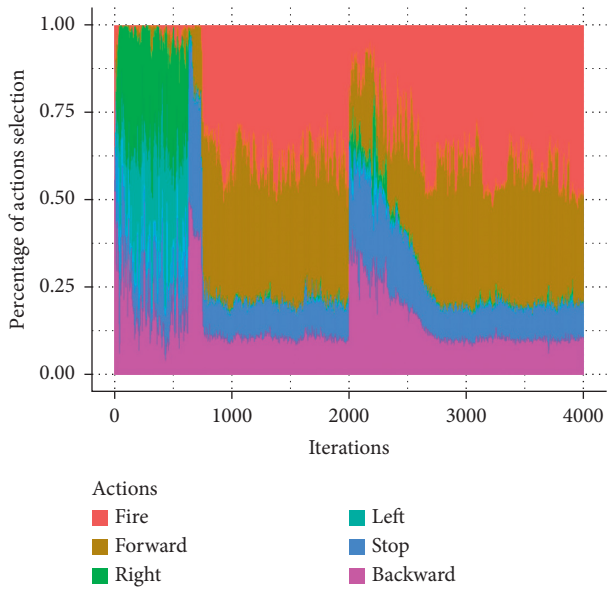


Figure 6: Percentage of actions selected during model training in combat with the Robot "SittingDuck."

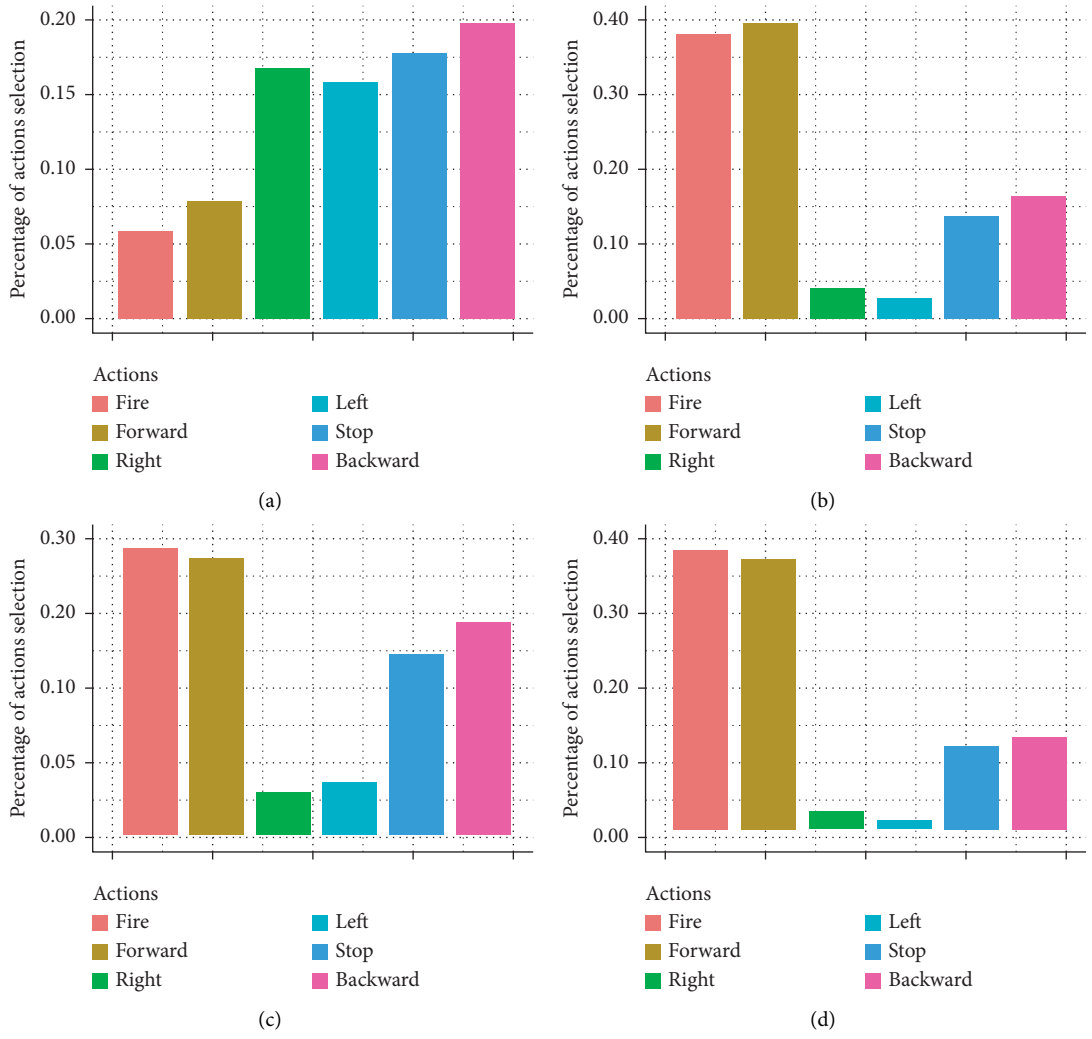


Figure 7: Percentage of actions selected in the 1st, 2nd, 3rd, and 4th quarters of the training phase against "SittingDuck."

presence of a predetermined reward signal. The reason for this is that considering the determined reward function, the robot must shoot the opponent robot in random directions in order to see any positive reward. Considering the similarly awarded function, every projectile that does not hit the target will cause the agent to receive a negative reward since the projectile shot by the robot will decrease its energy. As a result, the agent can only see a few positive prize passes thanks to randomness, while the epsilon value is only "1" in the early period. The agent has a positive effect on achieving the reward after increasing the randomness that it will be influenced at the beginning. For this reason, it is foreseen that the epsilon value starts with a value of 1 at the beginning of the training and then decreases linearly to 0.2 per 100 iterations. In addition to all these, the firing and going actions learned by the "SittingDuck" robot were decided to train the network from the beginning with the said optimizations so that the agent could produce more balanced actions compared to the previous model during the learning process. The network was trained by repeating 4000 battles previously carried out against the Corners robot. The score values and action distributions of this network trained are shown in Figure 8.

4.2. Experiments by Using Advanced Robots of RoboCode Platform. In the light of the graph presented, it was interpreted that the balanced lands given by the educated agent with the Corners robot were reflected on the RoboCode score and the model obtained was more successful. The target training model obtained was recorded and continued to be used to fight with predefined robots in the arena in the RoboCode simulator. In order to train the agent well, 100 iterations of war are executed for each of the predefined robots so that the model could be trained cumulatively by benefiting from the experience of all robots. This model has been applied for each of the 100 iterations and the robot has gained cumulative learning ability. Each of the predefined robots of the artificial intelligence agent RoboCode simulator participated in 2800 different battles with robots, each with a different combat strategy, and RoboCode scores for these battles are shown in Figure 9. This figure, in essence, contains the game score between the proposed robot (AI robot) and each advanced robot, defined in the RoboCode platform. Additionally, the average training performance is added as a separate graph.

4.3. Experiments by Using Community Robots. After the success of the proposed AI robot against predefined robot is validated (see Figure 9), it is aimed to fight with the robots in the platform named "RobotRumble," where users can load the robots they have developed. Therefore, leading robots of this platform, namely, "Bl4ck," "QBot," "Net," have been compared with the developed robot. It should be noted that these leading robots were also trained by machine learning algorithms.

Figure 10 presents the results of the first 400 matches of these encounters. In addition, it was also planned to combat these machine learning-based robots of "RobotRumble" platform with the predefined robots of RoboCode platform, namely, "Walls," "Corners," "Tracker", "SpinBot," "RamFire." In order to compare these with the proposed model, machine learning based robots and the proposed robot fought in the arena for 100 iterations against the same predefined robots. The graph including the scores obtained is shown in Figure 11.

4.4. Discussion and Statistical Analysis. Figure 9 illustrates that once the number of battles increases, the superiority of the proposed model over enemy robots defined in the RoboCode environment is validated. Besides, it has been proved that the proposed model is mostly superior to other RoboCode community robots against predefined robots defined in the RoboCode simulation environment, as shown in Figure 11. The only exception is the "Net" robot which produces a higher score against the "Corners" robot. In addition to the aforementioned experiments, a statistical test, T-test, was applied in order to provide a benchmark comparison between the proposed AI Robot and standard robots of RoboCode platform. It measures whether the difference between two sets of data is random or statistically significant [35]. It should be noted that that RobotRumble platform involves nonstandard robots that may be removed, modified, or restricted by developers. Hence, they are not used in statistical analysis. Consequently, in order to provide test data, "Corners," "Walls," "Tracker," and "SpinBot" robots defined on RoboCode and previously defeated by AI ROBOT (proposed robot) were selected. These robots were subjected to 10, 100, 500, 1000, 5000 arena battles with themselves, respectively. Win percentages were formed by proportioning the number of victories achieved by the robots. In the second phase, the default robots were subjected to the same number of battles with AI robot, respectively. Victory (Winning) percentages were formed by proportioning the number of victories achieved by AI robot after the war. AI ROBOT fought with each robot by resetting the learning model before the war. In this way, the learning tendency of the model was also being observed. Overall, T-test (two-sample assuming unequal variances) techniques were applied to the proportioned victory percentages table. Results, presented in Tables 1 and 2, verify the efficiency of the proposed model.

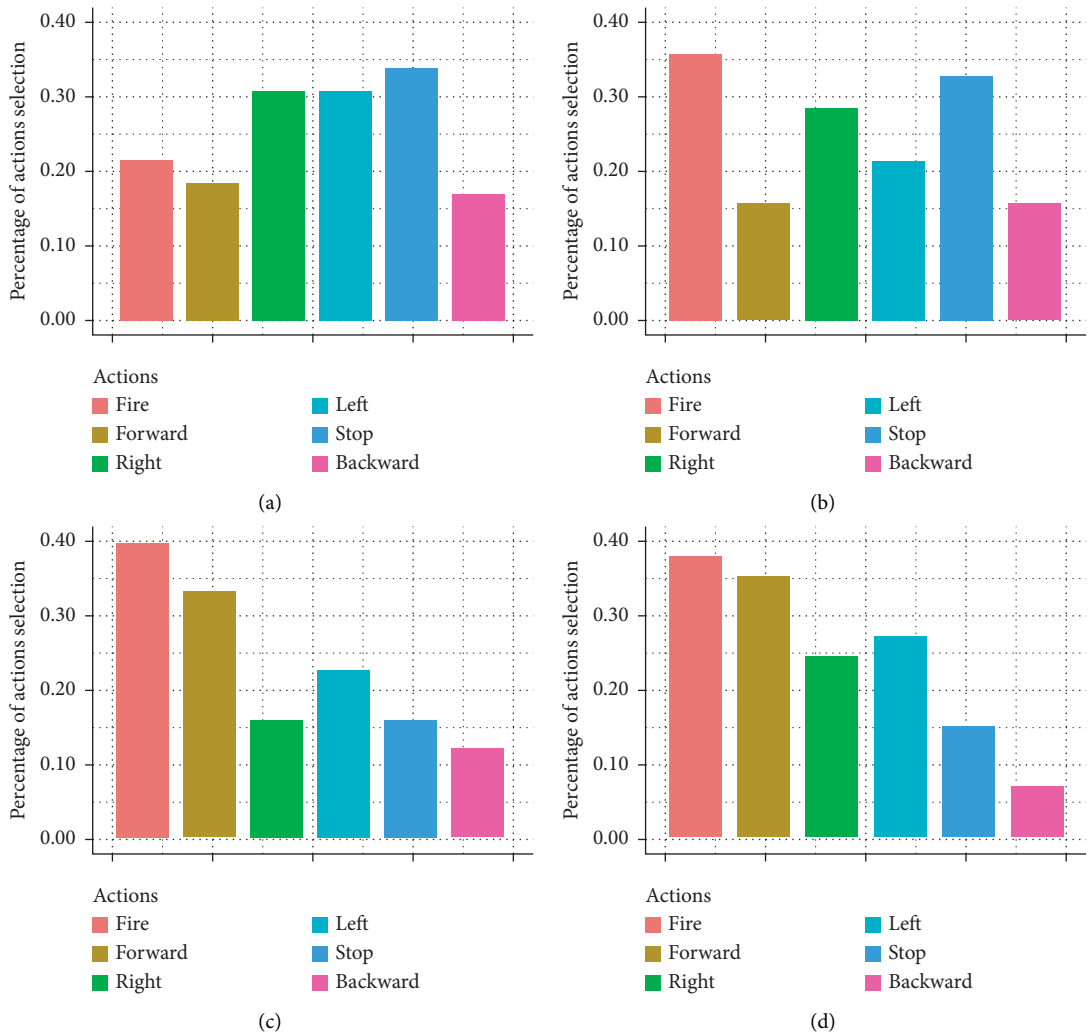


Figure 8: Percentage of actions selected in the 1st, 2nd, 3rd, and 4th quarters of the training phase against "Corners robot."

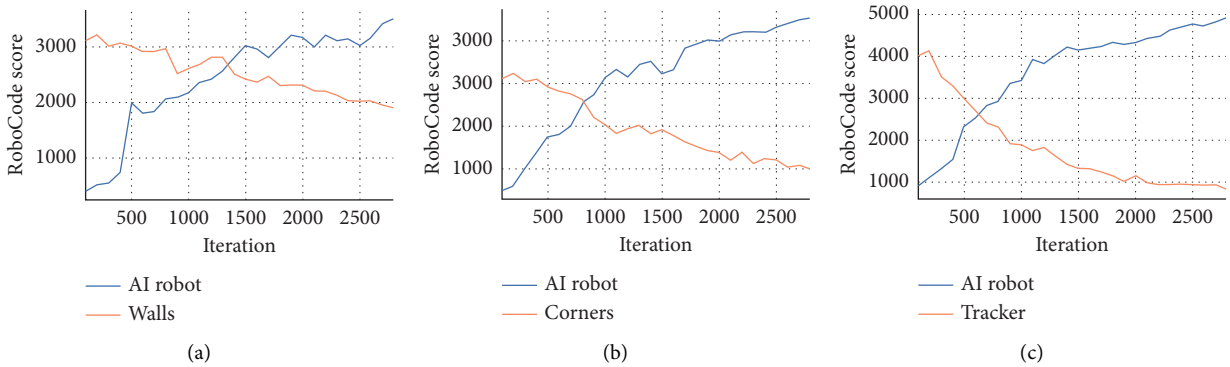


Figure 9: Continued.

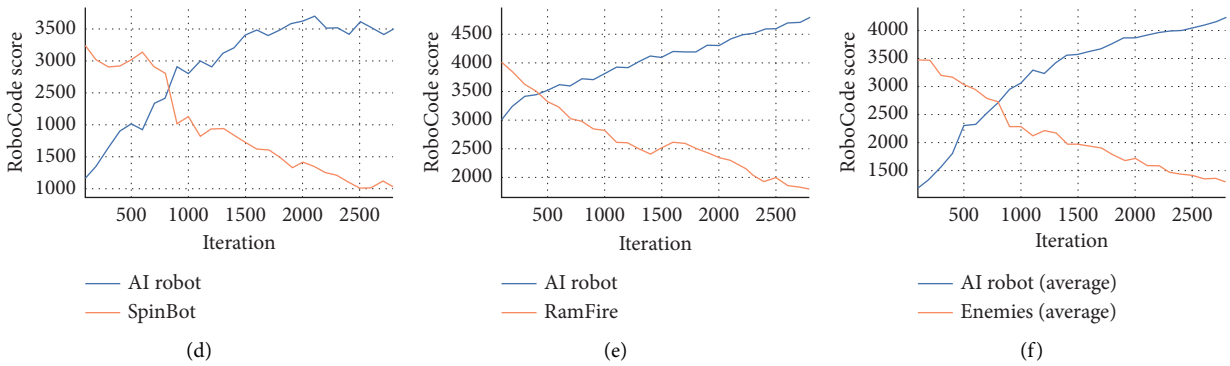


Figure 9: Training phase (against advanced predefined RoboCode robots).

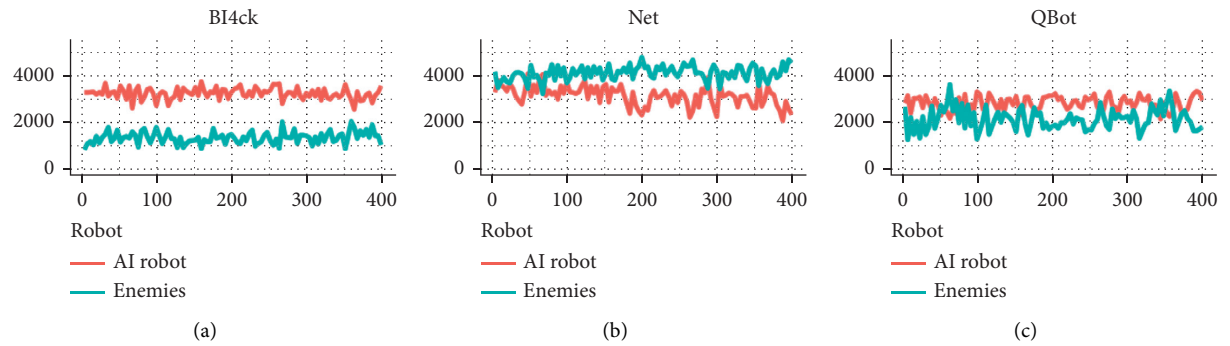


Figure 10: Percentage of actions selected in the 1st, 2nd, 3rd, and 4th quarters of the training phase (against community robots).

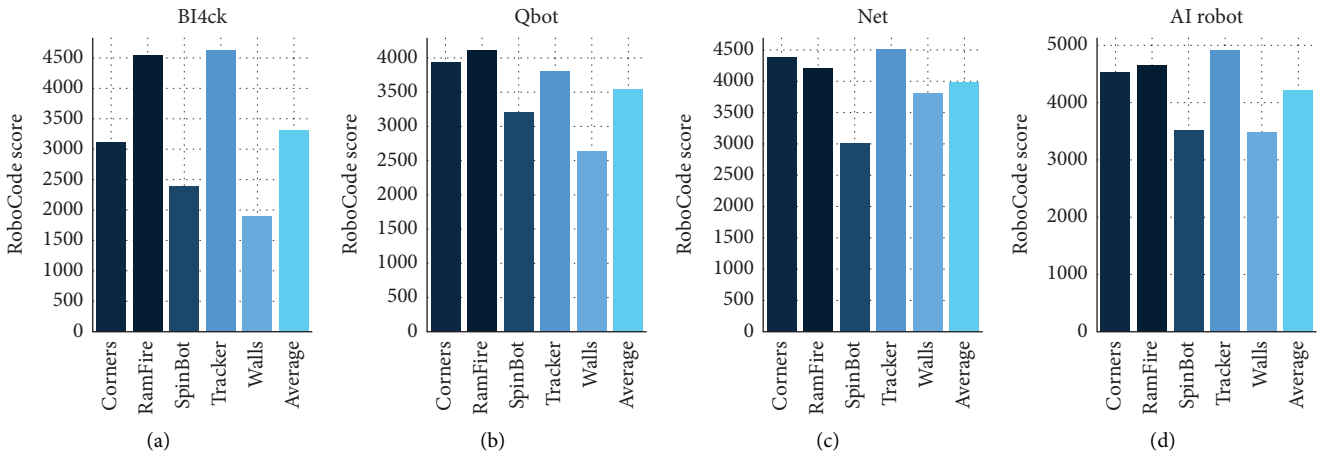


Figure 11: Scores are represented between AI robot and community robots against standard robots.

Table 1: The T-test results between the AI robot and Corners and Wall robots.

Corners vs. AI robot		Walls vs. AI robot	
t Stat	−10.8663	t Stat	−7.4899
P ($T \leq t$) one tail	5.73E−05	P ($T \leq t$) one tail	6.92E−05
t Critical one-tail	2.015048	t Critical one-tail	1.894579
P ($T \leq t$) two tail	0.000115	P ($T \leq t$) two tail	0.000138
t Critical one-tail	2.570582	t Critical one-tail	2.364624

Table 2: T-test results between the AI robot and Tracker and SpinBot robots.

Tracker vs. AI robot		SpinBot vs. AI robot	
t Stat	−3.91654	t Stat	−7.43674
P ($T \leq t$) one tail	0.002887	P($T \leq t$) one tail	0.000346
t Critical one-tail	1.894579	t Critical one-tail	2.015048
P ($T \leq t$) two tail	0.005775	P($T \leq t$) two tail	0.000693
t Critical one-tail	2.364624	t Critical one-tail	2.570582

5. Conclusions

This paper introduces a new framework using a deep Q-learning algorithm for a complex simulation platform, namely, RoboCode. This platform offers predefined robots and allows AI-based customized robots developed by the community. Compared to Atari Games, RoboCode has a fairly wide set of actions and situations. In addition, the simulation has various restrictions on access to computer resources for security reasons. In order to avoid these limitations, a server model that can communicate with the robot and transmit future actions of the robot has been established and the robot is enabled to talk to this server continuously. In the light of the information obtained from the robot, the proposed convolutional neural network model has continuously experienced the arena and directed the agent. During the model training, it was determined that the in-game images given as input to the neural network were quite noisy and the agent failed. Hence, instead of using in-game images, images were produced by using the information obtained by the agent during the war in the arena. The proposed model was first tested on a simple predefined robot, "SittingDuck," where it is much easier to dominate the arena than other robots, and the model achieved %100 success, as it was expected. Essentially, the effect of the epsilon greedy algorithm on the success of the agent has been observed, which allows the agent to make random selections without being dependent on Q values. The agent's use of its combat experience against more aggressive robots led to unsuccessful results. Consequently, the model is trained with more complex robots, namely, "Corners," "RamFire," "Walls," and "SpinBot," offering different battling strategies. Furthermore, some optimization approaches have been applied to increase the overall performance of the model. With the cumulative learning method applied during the training of robots, it was observed that the training model achieved significant success against the aforementioned predefined robots and the results are validated statistically. The proposed model was also compared with machine learning based model defined in the literature. Results also reveal that the proposed model is mostly superior to models defined in RoboCode literature and community. In the light of these results, the deep Q-learning algorithm has proven to be successful in the RoboCode simulation environment as in Atari 2600 games.

Data Availability

The data belong to the authors and can be made available by Dr. Mehmet Serdar GÜZEL, Computer Engineering Department of Ankara University, Turkey (email: mguzel@ankara.edu.tr), upon request.

Disclosure

Part of this study was published in the MSc. thesis of the first author.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] H. Li, Q. Zhang, and D. Zhao, "Deep reinforcement learning-based automatic exploration for navigation in unknown environment," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 6, pp. 2064–2076, 2020.

[2] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, "Deep reinforcement learning for multiagent systems: a review of challenges, solutions, and applications," *IEEE Transactions on Cybernetics*, vol. 50, no. 9, pp. 3826–3839, 2020.

[3] R. J. Dolan and P. Dayan, "Goals and habits in the brain," *Neuron*, vol. 80, no. 2, pp. 312–325, 2013.

[4] B. F. Skinner, *Contingencies of Reinforcement: A Theoretical Analysis*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1969.

[5] D. Pandey and P. Pandey, "Approximate Q-learning: an introduction," in *Proceedings of the 2010 Second International Conference on Machine Learning and Computing*, Bangalore, India, February 2010.

[6] W.-S. Jung, J. Yim, and Y.-B. Ko, "QGeo: Q-learning-based geographic ad hoc routing protocol for unmanned robotic networks," *IEEE Communications Letters*, vol. 21, no. 10, pp. 2258–2261, 2017.

[7] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep reinforcement learning for general video game AI," in *Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, Maastricht, Netherlands, August 2018.

[8] Ü. Ulusoy, M. S. Güzel, and E. Bostancı, "A Q-learning based approach for simple and multiagent systems, multi agent system," Intech, London, UK, 2020.

[9] H. Kayakökü, "A new battle strategy for robocode based on reinforcement learning," MSc Thesis, Ankara University, Ankara, Turkey, 2019.

[10] V. Mnih, K. Kavukcuoglu, D. Silver et al., "Playing atari with deep reinforcement learning," in *Proceedings of the NIPS Workshop on Deep Learning*, Lake Tahoe, USA, December 2013.

[11] V. Mnih, K. Kavukcuoglu, D. Silver et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[12] G. Morten, K. Michael, R. A. Kjær et al., "Applying machine learning to RoboCode," Technical Report, Aalborg University, Aalborg, Denmark, 2003.

[13] U. Ulusoy, "A Q-learning based approach for simple and multiagent systems," MSc Thesis, Ankara University, Ankara, Turkey, 2019.

[14] J.-H. Hong and S.-B. Cho, "Evolution of emergent behaviors for shooting game characters in RoboCode," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, pp. 634–638, Portland, OR, USA, June 2004.

[15] K. Kobayashi, Y. Uchida, and K. Watanabe, "A study of battle strategy for the RoboCode," in *Proceedings of the SICE 2003 Annual Conference (IEEE Cat. No.03TH8734)*, pp. 3373–3376, Fukui, Japan, August 2003.

[16] D. G. Nidorf, L. Barone, and T. French, "A comparative study of NEAT and XCS in RoboCode," in *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1–8, Barcelona, Spain, July 2010.

[17] B. G. Woolley and G. L. Peterson, "Unified behavior framework for reactive robot control," *Journal of Intelligent and Robotic Systems*, vol. 55, no. 2-3, pp. 155–176, 2009.

[18] R. Harper, "Evolving RoboCode tanks for evo RoboCode," *Genetic Programming and Evolvable Machines*, vol. 15, no. 4, pp. 403–431, 2014.

[19] A. J. Abdellatif, B. McCollum, and P. McMullan, "Serious games quality characteristics evaluation: the case study of optimizing RoboCode," in *Proceedings of the 2018 International Symposium on Computers in Education (SIIE)*, pp. 1–4, Jerez, Spain, September 2018.

[20] J. Eisenstein, "Evolving RoboCode tank fighters," CSAIL Technical Reports, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, MA, USA, 2003.

[21] Y. Shichel, E. Ziserman, and M. Sipper, *Gp-robocode: Using Genetic Programming to Evolve RoboCode Players*, Department of Computer Science, Ben Gurion University, Beersheba, Israel, 2005.

[22] J. Frøjhær, M. L. Kristiansen, P. B. Hansen, I. V. S. Larsen, D. Malthesen, and R. Suurland, "RoboCode, development of a RoboCode team," Technical Report, Department of Computer Science, Aalborg University, Aalborg, Denmark, 2004.

[23] P. Solomon, P. Thulasiraman, and R. Thulasiram, "Collaborative multi-swarm PSO for task matching using graphics processing units," in *Proceeding of the 13th Annual Genetic and Evolutionary Computation Conference*, Dublin, Ireland, July 2011.

[24] V. Alexiev, "Machine learning through evolution: training algorithms through competition," CS Thesis, Trinity University, San Antonio, TX, USA, 2013.

[25] K. Kuchar, E. Holasova, L. Hrboticky, M. Rajnoha, and R. Burget, "Supervised learning in multiagent environments using inverse point of view," in *Proceedings of the 2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, pp. 625–628, Budapest, Hungary, July 2019.

[26] C. Hu and M. Xu, "Fuzzy reinforcement learning and curriculum transfer learning for micromanagement in multi-robot confrontation," *Information*, vol. 10, no. 11, p. 341, 2019.

[27] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1106–1114.

[28] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[29] L. Lin, "Reinforcement learning for robots using neural networks," Technical Report, Carnegie Mellon University, Schenley Park, Pittsburgh, PA, USA, 1993.

[30] R. S Sutton and A. G. Barto, *Reinforcement Learning. An Introduction*, MIT Press, Cambridge, MA, USA, 2017.

[31] M. S. Güzel and R. Bicker, "A behaviour-based architecture for mapless navigation using vision," *International Journal of Advanced Robotic Systems*, vol. 9, pp. 1–13, 2012.

[32] B. W. Know and P. Stone, "Framing reinforcement learning from human reward: reward positivity, temporal discounting, episodicity, and performance," *Artificial Intelligence*, vol. 225, pp. 24–50, 2015.

[33] F. S. Perotto and L. Vercouter, "Tuning the discount factor in order to reach average optimality on deterministic MDPs," in *Proceedings of the International Conference on Innovative Techniques and Applications of Artificial Intelligence (SGAI)*, Cambridge, UK, December 2018.

[34] V. Nair and G. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814, Haifa, Israel, June 2010.

[35] A. A. Yilmaz, M. S. Guzel, E. Bostanci, and I. Askerzade, "A novel action recognition framework based on deep-learning and genetic algorithms," *IEEE Access*, vol. 8, no. 1, pp. 100631–100644, 2020.