Int. j. inf. tecnol.
https://doi.org/10.1007/s41870-022-01047-z

ORIGINAL ARTICLE

A deep reinforcement learning technique for bug detection in video games

Geeta Rani[1] · Upasana Pandey[2] · Aniket Anil Wagde[1] · Vijaypal Singh Dhaka[1]

Received: 8 April 2022 / Accepted: 22 July 2022
© The Author(s), under exclusive licence to Bharati Vidyapeeth's Institute of Computer Applications and Management 2022

Abstract The objective of this research is to design the deep reinforcement neural learning-based model that detects the bugs in a game environment. The model automates the bug detection and minimizes human intervention. It makes effective use of the Deep-Q-Network to design and develop the model 'RLBGameTester' for measuring the high dimensional sensory inputs. The model modifies the environment to intercept the game screen. It also adds faults to the game before submitting it to the Deep-Q-Network. It calculates the values of the loss function at different iterations. The differences in the values of the loss functions in a bug-free and the bug containing game environment point out the presence of a bug. It also locates the position where the bug appears. The proposed model is useful for multiple game environments with minimum customization. Its applicability for blurred as well as non-blurred inputs at different platforms proves its efficacy. Employing this model may prove a game changer in the game designing industry.

Keywords Deep-Q-Network · Reinforcement learning · Bug · Testing · Deep learning · Neural network

✉ Vijaypal Singh Dhaka
prof.dhaka@gmail.com

Geeta Rani
geetachhikara@gmail.com

Upasana Pandey
coe.upasana@gmail.com

Aniket Anil Wagde
aniket.wagde@gmail.com

[1] Department of Computer and Communication Engineering, Manipal University Jaipur, Rajasthan, India

[2] Department of Computer Science and Engineering (Artificial Intelligence), ABESIT College of Engineering, Ghaziabad, India

1 Introduction

Increasing the interest of young minds towards technology is a crucial reason for spending significant time on screen. It has attracted the researchers to focus on designing and developing techno-games such as Space Invaders, The Legend of Zelda, Mario and Pacman, etc. These games are a source of entertainment for people of all age groups. Moreover, Vlachopoulos, and Makri discussed that computer-based games are a good source of enhancing learning from kindergarten to higher education [1]. These games help to improve the use and to enhance the coordination into different sense organs.

We observe a consistent rejection of computer games due to programming errors. Delicate product bugs are disappointing for players, expensive for publishers, and destructive to their notoriety. For instance, a bug in the 'Nintendo' release of 'The Legend of Zelda: Breath of the Wild' in 2017 facilitates the players to progress playing but allows them to bypass the effects of temperature on the game. It leads to a significant degradation in the entertainment of playing the game [2]. Moreover, the 'NBA Elite 2011' got rejection due to issues in its programming quality [3].

There is a vital requirement to detect the programming bugs to minimize the chances of rejection of the developed games as early as possible. It gives time for bug removal from a game before its actual implementation. It is also helpful for frameworks where it is difficult to adjust a mistake once launched or delivered. Therefore, a need to introduce effective testing mechanisms arises. State-of-the-art reveals a significant contribution in bug detection. For example, the researchers Cruz, and Uresti give details of the impacts of Artificial Intelligence (AI) on game developers [4]. They claimed the use of AI and its tools in the game designing, developing, and testing.

Further developments in AI introduced the use of reinforcement learning in different domains. It is applied in various application areas such as performing troublesome aerobatic stunts with a helicopter [5], playing Backgammon robotics [6], playing Atari game [7], and video game testing [8]. Reinforcement learning is applied to develop automatic and artificial reasoning, enabling a machine to play aggressively against the global data of sports champions [9]. It creates advancements in virtual benchmarks set to test the reinforcement agents. Reinforcement learning is also a promising way to deal with numerous issues such as in-genuine assignments, checking generalizability of formulations in a controlled setting [9], simulating social intelligence to facilitate human-machine interaction, and creating coordination in robots to synchronize their movements [10].

It is clear from the above discussion that reinforcement learning is helpful in problem-solving and sequential decision making. Hence, it can fulfill the requirements of game designing, developing, and testing. Moreover, the existing models require human supervision for testing the performance of a player at each state. It motivated the authors to apply reinforcement learning for developing the tool for automated testing of a game environment without human supervision.

In this manuscript, the authors propose the model 'RLBGameTester' that makes effective use of reinforcement learning for designing and testing a game environment. This AI-based self-learning model identifies the discrepancies such as blank screens and incorrect refreshing of the game state under a specific set of conditions. The model aims at minimizing the requirement of manual testing and reducing the cost of testing a game. The major contributions of this research are as follows:

- Developing a reinforcement learning based tool for automatic bug detection in a video game.
- Minimize the human intervention in bug detection.
- Reduce the requirements for manual and periodic testing of a game.
- Minimize the cost of testing a video game.

The remaining part of the manuscript is organized as; Sect. 2 presents the related literature. Section 3 offers the background, and Sect. 4 demonstrates the materials and methods. Section 4 also covers the experimental results. Section 5 contains the discussion, and Sect. 6 presents the conclusions of the work.

2 Literature review

Artificial intelligence has proven its potential in contrast enhancement [11], object detection [12], and medical imaging [13]. Recent advancements in artificial intelligence increased the applications of reinforcement learning in different domains such as neuroscience, optimization control, business strategy planning, aircraft control, robot control, and game designing. Here, the authors focus on the research works available in-game designing and testing.

The authors in reference [2], presented the review of game designing models. They discussed that the ad-hoc methods of game designing that are driven by experience are the most opted strategies in the game industry. Next, the authors in [14] gave a systematic review of the software life cycle adapted for game development. They concluded that there is a lack of a standard software life cycle for developing the game. They also highlighted that most of the research works focus on the game development stage. Game testing is still a less exploited area of the game industry.

A majority of environments are designed to offer the rewards only at the final state (s) of a task. For example, in a game, the player is rewarded with points on completing a level of the game. On the other hand, the earned reward points are decreased as a punishment for losing a life in the game. There is no track of the rewards at intermediate states. Thus, it is difficult to track the performance of a player in medium conditions. To address this issue, the authors in [3] presented the techniques for improvement in game designing. For further improvements Giannakopoulos, and Cotronis successfully implemented the Q-learning network [15]. It is a model-free algorithm to train the agent for making the best suitable decision in a specific set of conditions. This technique of reinforcement learning simplifies the game environment. A simple game environment favors an agent to understand the far-reaching implications of acting [4].

Moreover, Q learning provides the false values of the reward at each state. This value is named the Q value of the state. The continuously assigned tips reveal information about the degree of progress achieved at each state. Thus, it resolves the problem of tracking the performance of a player in intermediate conditions.

DeepMind builds the Deep Q-Network (DQN) model that uses a deep learning model to predict the Q value of each state [7]. It uses multiple layers of the artificial perceptron to achieve higher levels of abstraction. These values are evaluated to get the best possible next step. It ensures the maximum potential Q value for the next state. For a wide variety of games designed for humans, the DQN model is efficient in playing at the human level or higher level(s) than human beings.

The study of the research in-game testing shows that the video game testing industry is entirely dependent on human testers. Very little innovation is observed in this field. The authors in [8] and [16] attempted to introduce automation for detecting logical bugs in a game. However, it requires an independent testing tool for each game. Moreover, it requires

detailed information about the internal state(s) of a game. Also, it is ineffective in identifying the visual bugs in a game. For example, it is unable to detect a bug if the change in value affects the speed of players without visualizing any effect on the screen. In such a scenario, DQN considers the entire screen and the scores as inputs. DQN uses the Convolutional Neural Networks (CNN) to handle the raw pixel inputs of the screen. It is completely autonomous and does not require a human player to play the game while searching the bugs. It is efficient in playing the game and searching for bugs simultaneously. Based on the advantages, the authors employed the DQN to develop the model 'RLBGameTester' proposed in this manuscript.

3 Background

In this section, the authors present the details about the structure of a video game, techniques used to develop and test a video game.

3.1 Structure of a video game

The literature study teaches that most techno games are designed using the concepts of object-oriented programming. In this approach, each element of a game is an object. The centralized piece of code is written for the game. This code is named the game loop. Each object present in this loop is monitored and recalculated. However, no change in the state is done without receiving a response from a user.

A mechanism is included in each game to ensure that it is played at a pre-set speed. The number of Frames Per Second (FPS) determines the rate of playing a game. FPS is the number of new frames displayed to the player in one second. The studies demonstrate that human beings can observe changes up to 500 FPS [17]. However, FPS is set to 60 for a pleasant viewing experience while playing. There are two methods to control the FPS: (i) fixed frame rate and (ii) variable frame rate. In a fixed frame rate, the computer runs the game loop faster than the required rate and then calls a sleep function. This function sets the refresh rate precisely equal to the value of the input parameter. If sleep function takes a longer time, then the game appears to lag.

On the contrary, in the variable frame rate, the computer shows a new frame at the end of each game loop. Here, the refresh rate can be higher than the set parameter. It may lead to inconsistency because it takes variable periods in different iterations to complete the game loop. The Deep-Q-Network (DQN) algorithm [18] is employed for game designing. The algorithm uses 'Q' value of each state of the game and learns the best action to perform at a given state. The deep neural network of DQN is useful for estimating 'Q' value function.

3.2 Techniques of finding bugs in video games

The attack of bugs in the running environment of a video game is a cause of unexpected interruptions in the game environment. To find the bugs, the game developers rely on the adoption of one of the following testing techniques in the life cycle of a game project.

3.2.1 Manual testing

In this strategy, the testing team needs to write rigorous test cases for testing the gameplay environment and the usability of the game. At an initial stage, an implicit testing process is carried out to perform a set of tests to remove the errors from the code. For further testing, an external and internal testing team(s) design rigorous test cases. Each game tester manually reports every bug identified with a detailed description. Now, the programmers address these bugs with utmost priority [19].

In case an updated version of the game is launched, it is sent to the bug testers. In such cases, the main objectives are to test the new features and functionalities added to the previous version of the game. Introducing new features in a game can potentially create bugs in earlier modules of a game. Thus, there is a vital requirement of a complete assessment of the code and to report the bugs in an updated version of the game.

This manual testing strategy requires a workforce. Designing test cases and identifying bugs manually is a time-consuming process. Hence, it is costly. Moreover, the number of test cases is limited. In this strategy, testers can only report bugs visible to them on the Graphical User Interface (GUI). Also, the detection of bugs is dependent on the knowledge of the testing team(s). There can be a possibility that one or more bugs remain unnoticed, or one or more essential features of the game can be flagged as a bug(s).

3.2.2 Runtime monitoring

In this strategy, the monitoring of the game is done at runtime for dynamically checking and enforcing the constraints. This is successful in monitoring the working of a game at the finer level [19].

In this strategy, the game is constantly evaluated based on the pre-set rules. For example, the jump time cannot exceed 3-s, and jump height is limited to 7-units in a particular game. If any of the pre-set rules are violated, the monitoring system will reevaluate values and attempt to bring consistency again.

The runtime monitoring requires a specific set of rules for each game. Also, the game tester needs to understand

the details of the code. The requirement to perform the game-specific code intrusion and understand each game's working mechanism is quite tricky. This created the demand for developing a generalized solution that requires minimal code intrusion.

The research community contributed to developing the game monitoring systems with minimal code intrusion. They developed monitoring systems such as JavaMOP [20] and LARVA [21]. But, all these systems focus on monitoring method calls in Java programs. Thus, these are game-specific as well as dependent on the programming language. Moreover, there is a negligible scope to employ these systems in other programming languages. Therefore, it becomes a point of concern for the game developers who develop their programming languages and Unreal Script [22] for developing the games. In such cases, the pre-developed monitoring systems fail to perform the runtime verification of games.

3.2.3 Reward hacking

In reinforcement learning, 'Reward Hacking' is when the model learns to take advantage of the unintended consequences of one or more actions in the simulated world. This happens due to the inability of human beings to specify the expectations that an AI-based system can fully understand. This inability is called the value-alignment problem. Furthermore, the failure to predict the exact action of an agent makes the method unreliable because it can cause a threat to human life in critical situations [23].

In the modern era, erroneous coding practices cause an exponential increase in in-game releases with bugs. The high complexity of games makes the debugging process extraordinarily lengthy and challenging. The reward hacking is beneficial to game developers and testers to uncover bugs from the complex games.

For example, in the game 'Coast Runner's boat racing', the agent learns how to maximize the game's score by spinning in circles and colliding with objects rather than racing. This indicates that the reward function is inappropriate for the racing game. In one more example, QBert Atari proposed the reinforcement learning algorithm and uncovered a bug in the game that remained unknown to humans [7]. These algorithms identify the inconsistencies in the game. Thus, these can learn ways to cheat in a game.

The above discussion shows that manual and run-time testing challenges can be resolved by applying the deep learning models. The effectiveness of these models in designing and testing the games gives the inspiration to develop a deep reinforcement learning model 'RLBGameTester' for game playing as well as monitoring the internal loss functions of the game to identify a bug.

4 Materials and methods

In this section, the authors will discuss the experimental setup and the dataset used to evaluate the model 'RLB-GameTester'. They will also discuss the experimental results obtained on performing the set of experiments.

4.1 Experiments

The authors used python 3.6 for developing the model. They used the 'Tensorflow' for implementing the Deep Q-Network (DQN). The authors performed the set of experiments using the 'paperspace' cloud platform. They executed their model on 'Ubuntu 14.04' Operating System (OS) and the graphic card 'Nvidia Quadro p4000' on GPU.

The DQN based model 'RLBGameTester' is designed by implementing the Q-learning algorithm. The authors applied the standard approach given at [24] to re-calculate the value of 'Q' for each possible action. This approach requires a forward pass for each step. Each pass linearly adds the computational cost of increasing the number of possible activities. The architecture of 'RLB-GameTester' is designed so that a single forward pass can calculate the values of 'Q' for all possible actions [3] resolves the problem of an increase in computational cost.

4.1.1 Hyper parameters of DQN

The proposed model 'RLB-GameTester' uses the following hyperparameters.

(i) Batch Size: Batch size represents the number of games running in parallel. In this manuscript, the authors chose the batch size of 32 instances.
(ii) Random start: It represents the maximum frame number where the game can start. In this manuscript, the random start is set at 30. This shows that the game can start randomly from any fame number from 0 to 29.
(iii) CNN_format of data layout: The 4-Dimensional (4-D) array that contains the information in the form of Number of Images (N), number of feature maps (C), image height (H), and the image width (W). The data layout' NCHW' is the default layout in tensor flow. Here, 'N' denotes the number of batches, 'C' is the number of channels, 'H' is the height, and 'W' is the width of the data. It is ideal for training with Nvidia GPUs available at [24]. Therefore, in this manuscript, the authors used the 'NCHW' data layout for experiments.
(iv) Discount: This is an arbitrary value multiplied by the target Q-value to avoid the problem of overfitting. For the experiments, the authors set the value of discount as 0.99 based on its impact on the performance of

the model. The increase in its value to 1 or above forces the model to assign a higher weightage to gains obtained from the forthcoming steps. This may lead to a hike in the reward of any step up to infinity. On the contrary, the decrease in its value forces the model to assign the low weightage to the forthcoming steps. Therefore, the model takes shortsighted steps to boost its rewards.

(v) Learning_rate: The learning rate of the model is a parameter that determines the change in the weights per epoch. It represents the initial learning rate of the model. The authors set the value of Learning_rate as 0.00025 based on the results obtained on performing a set of experiments. On increasing its value beyond the pre-set value, the model becomes ineffective to converge at the point of global minima. This happens due to its constant overestimation of the weight change because of a high learning rate. On the other hand, the decrease in its value takes a long time to converge to the global minima. This happens because the learning steps are too long.

(vi) Learning_rate_minimum: This is the minimum learning rate of the model. The authors used its default value, 0.00025.

(vii) Learning_rate_decay: This represents the rate of change in the learning rate. The authors set the value of Learning_rate_decay as 0.96 based on the set of experiments. Increasing or decreasing its value up to 4% did not degrade the performance of the model. This is due to the fact that; the current learning rate of the model is equal to the minimum learning rate that does not decrease further.

(viii) Learning_rate_decay_step: It represents the step number after which the learning rate of the model changes. The authors set the Learning_rate_decay_step as 500. It indicates that the learning rate of the model varies after every 500 steps. On performing the set of experiments, the authors observed that this value does not affect the model's performance. This is because the current learning rate is equal to the minimum learning rate, and it will not decrease further.

(ix) History_length: It is the number of frames stored in the history of the model. The model refers, its history length at every training state. In this manuscript, the authors set the value of History_length as 4. The value is selected based on the number of frames required to gather enough information needed for decision making. For example, in a turn-based game such as chess, a single frame is sufficient. On the contrary, the car racing game requires multiple frames to understand the position and speed of cars, direction of movement, and acceleration. On performing the set of experiments, the authors observed that there is no change in the performance of the model on increasing its value beyond four.

(x) Train_frequency: It is the number of frames encountered between two successive training steps. In this manuscript, the authors set the value of Train_Frequency as four based on the set of experiments. The training step is not required at each frame because tiny information gain occurs between two consecutive frames. Information increases significantly at every stage 4th frame.

(xi) Learn_start: It is the step number at which training of the model starts. The authors set the value of Learn_start as 500. It indicates that the training of the model begins at the 500th step. In earlier steps, the model records the significant information about the mechanism of the game.

(xii) Min_delta: It is the minimum value of reward fed to the model. The authors set their value as − 1. Therefore, the value of the negative reward function can reach up to − 1. It is useful in avoiding the problem of attaining the extreme values of the reward function. Thus, it resulted in efficient training of the model.

(xiii) Max_delta: It is the maximum value of reward fed to the model. The authors set its value as 1. Therefore, the maximum reward function can attain the highest value 1. It helps in avoiding the problem of extreme values of the reward function. Hence, the training of the model is effective.

(xiv) Screen_width and screen_height: These are the dimensions of the screen in terms of its width and height. The model received these as input. The authors set the value of dimensions 84×84 for the model 'RLBGameTester'.

4.1.2 *Test plan*

The authors used the above mentioned list of hyper parameters and implemented the test plan given in algorithm 1 for automatic bug detection in a game.

Algorithm 1: Test plan for the proposed 'RLBGameTester'

1. Constantly monitor the average value of loss function while executing the DQN.
2. Train the DQN in the environment without any bugs inserted explicitely.
3. Train the DQN repeatedly after inserting different bugs at various stages
    a. Compare the behavior of DQN when bugs are inserted early in training versus when they are inserted at later stage of training
    b. Compare the values of loss function reported without bug, or various bugs are encountered.
4. Compare the values of loss function at all stages of executing the DQN.

After following the test plan shown in algorithm 1, it was observed that that the values of loss function of the DQN rises when it encounters unfamiliar states such as bugs.

4.1.3 Training and testing the model

The authors selected the game 'Space Invaders' for training and testing the proposed model. The following two reasons favor the selection of 'Space Invaders'.

(i) In this game, the model achieved a better performance than human beings. It indicates that the model successfully learned the entire mechanism of the game.
(ii) The game 'Space Invaders' includes a large number of moving objects. Thus, the model becomes resistant to visual bugs. The insertion of a bug degrades the performance of the model. In the worst case, the model fails to detect bugs due to the high complexity of the game.

To evaluate the performance of the model 'RLBGameTester,' the authors created three different bug propagations and testing environments using DQN. They trained the model for all the three bug propagations and testing environments on a single game running platform. In the training session, the bugs are propagated manually. The values of loss function are calculated before and after the bug propagation. A sudden spike in the loss function and decrement in the reward function shows that there is a bug in the running game environment. Thus, the system fails to play as efficiently as it could play in a bug-free environment. The authors explain the building, preparation of the environment, and training of the model in the following two phases.

Phase 1 This phase involves the building of the model and preparation of the environment.

The authors reduced the size of the input screen to 84×84. Also, they reduced the number of channels to one. Thus, the model receives a single 84×84 grayscale display as input. At the initial step, a point is identified, where the game screen is fed as an input to the model. At the next step, a point is identified where the changes in the input are made before it is fed to the model. Identifying these points is essential for defining the number of iterations required to insert a bug into the game and developing the following three types of bugs.

(i) White dots: This bug inserts a random number of pixels into the screen before its image is given as input to the model. The random number is pre-set in the range of 10–20 white dots of dimension 4×4 pixels. The model randomizes the positions of these pixels in each frame. This bug leads to the appearance of 10–20, flickering and random white dots on the screen to a human player.
(ii) Black Screen: Here, the value of the current reward is given as an input to the model. But, no visual information is fed to the model. The model is still in the control of sprite. Therefore, it can control the rewards up to a pre-set extent. This bug makes the appearance of the entire screen black. It seems that the game is completely crashed.
(iii) Sprite Disappearing: The bottom 20% of the screen disappears when this bug is encountered in the game. It implies that the player's sprite disappears, and the spaceship is not visible. It leads to the loss of enormous information. But, it does not have a significant impact on the actual screen fed to the model.

After developing the bugs, the authors created different functions that received 84×84 display as input and warped

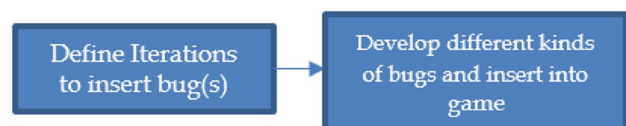


Fig. 1 Process of Phase 1

the images to mimic the appearance of bugs. Now, the model is trained and tested at different stages. On performing a set of experiments, the authors claim that 420,000 iterations are sufficient for training the model.

In Phase 1, the model defines the iteration where the bug is to be inserted as an input to the game. The identified iterations and different types of bugs developed are useful for understanding the capabilities of the proposed model in the detection of bugs into a game. Figure 1 shows the process involved in Phase 1 of the experiment.

Phase 2 This phase is designed to monitor the changes in the value of loss function on executing the model at the preset parameters. It completes the rigorous training and testing of the model in three steps.

In the first step, the model is trained without any bug. The training is performed for 420,000 iterations. It familiarizes the model with the details of the code and the working mechanism of the game. The loss function values during this training step are recorded in a graph, as shown in Fig. 3. These values are used as a reference point to compare the model's performance after inserting the bug. At the next step, one or more bug(s) are introduced in the game. Now, the model is trained for the next 30,000 iterations. The values of change in the loss function are recorded in a graph. The model is again executed in a bug-free environment at the last step, for the iteration range from 450,000 to 500,000. It is crucial to test the robustness of the model when it is reverted to the original gameplay. This test is required because, in most of the games, the bugs do not impact the complete life cycle of the game. These bugs appear and disappear sporadically. Figure 2 shows the set of activities performed in the phase-2 of the experiment.

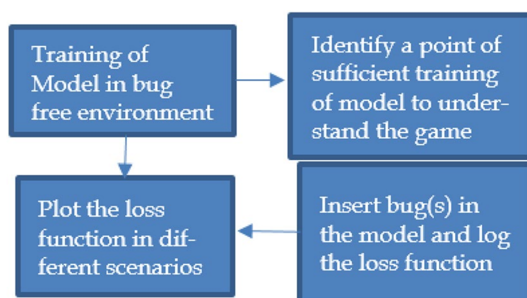


Fig. 2 Process of Phase 2

4.1.4 *Mechanism of bug detection*

A game encounters numerous types of bugs. These bugs can be simple such as the crashing of the game, which leads to the appearance of a black screen while playing the game. On the other hand, the bugs can be involved, such as inappropriate dialogue of a character and repeating of the word(s) by a character.

One model is not sufficient to deal with all kinds of bugs. Therefore, different models are designed to provide the best possible solution for a particular type of bug(s).

The proposed model understands the game as a collection of interconnected states. It does not consider any temporal relation in these states. This model considers each frame as an independent state and identifies a state that has never been encountered. The authors in reference [8] claimed that the existing model does not consider the outputs obtained in previous steps while making the decision. For example, the game shows unusual behavior for a particular input. This behavior is not considered in decision-making when the same input is given to the model at subsequent steps. However, 'RLBGameTester' is the best suitable model for monitoring and flagging instances of visual bugs. The authors developed a function to intercept the screen before it is given as an input to the model, test the efficacy of this model in monitoring, and flag the instances of visual bugs. This function also inserts visual bugs in the model and monitors the average value of the loss function.

4.2 Results

For rigorous testing of the model 'RLBGameTester,' all the test cycles are executed for 490,000 iterations. The data logs created on executing the test cycles are collected, analyzed, and displayed in graphical form using the tensor board. The x-axis of each graph represents the iteration number, and the y-axis represents the value of the loss function. A smoothening function is applied to make the graph easier to visualize and improve its readability.

The authors evaluated the performance of the model in three different types of testing environments. They intentionally inserted the three types of bugs, namely White Dots, Black Screen, and Sprite Disappearing, to test the model's bug detection ability and robustness.

4.2.1 *Testing environments*

Initially, the game is seeded with three types of bugs: (1) White Dots, (2) Black Screen, (3) Sprite Disappearing. These bugs are seeded before applying the pre-processing techniques to detect the bug that may appear in the game before pre-processing the screen and the actual starting of

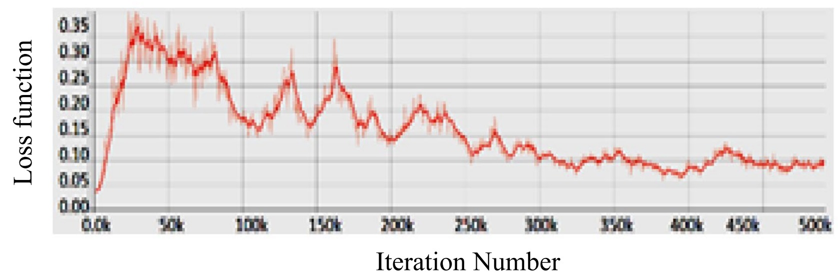


Fig. 3 Loss function without bug propagation

the game. Inserting these bugs is essential to visualize the impact(s) of each type of bug on the model's performance. Now, the pre-processing is applied on the game screen to reduce its dimensions. The screen size is reduced from 210×160 to 84×84 frames. The number of input channels is reduced from three (RGB) channels to a single grayscale channel. Thus, the computation time and the cost of the 'RLBGameTester' model decrease.

At the next step, the game screen is given as an input to the model. The insertion functions are named as (1) insert white dots, (2) insert black screen, (3) hides the players' spaceship. These functions enable embedding a specific type of bug after a pre-set number of iterations and at a pre-set location of code.

At the last step, the 'RLBGameTester' is executed to evaluate its performance in the presence of one or more bugs in the code. The change in the loss function indicates the presence of the bug in the code. Figure 3 shows the decline of a loss function in a game with a bug-free environment. The graph peaks show the growth of bugs. The value of the loss function is high at the peaks. It indicates that the 'RLBGameTester' model is shifting its weights drastically and learning new information. The maximum new information is given at these peaks through the screen to improve the model's adaptability. The new information can be an entirely new screen that never appeared before or maybe a new bug. It is clear from the graph shown in Figure 3 that the model achieves the maximum value of loss function 0.33 at the iteration number 20,000. The trends showing the change in values of the loss function indicate the updating weights of the model. The higher the value of the loss function, the greater will be the changes in weights. A higher value of the loss function also shows that the new information is given to the model. If an abrupt and sharp increase in the loss value is reported, then it is a clear indication that either a bug appeared in the game or a huge amount of new information is given as an input to the model. The testing team does not add any new screen to the game used for testing. Therefore, the peaks in the graph showing the abrupt rise in the values of loss function report the abnormal behavior of the game caused by one or more bugs present or inserted into the game. The subsequent sub-sections present the propagation of different types of bugs and their impact on the value of loss function.

4.2.2 White dots propagation

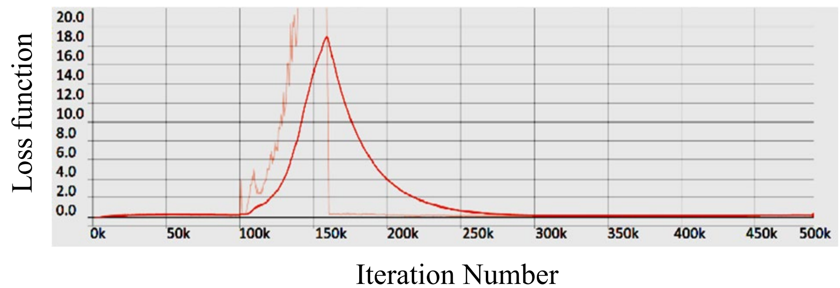
The authors designed the function 'white dots propagation' to insert a random number of square-shaped dots in a preset range into the screen. The function receives the size of dots and the number of dots appearing on the screen as the input parameters.

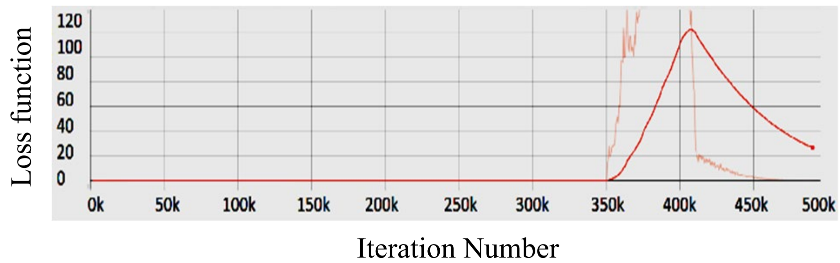
Implementation: For performing the experiments, the authors set the dot size of 2×2 and the number of dots in the range of 5–20 per screen. The value is one is set for the pixels showing the white dots. Whereas the float values in the range of 0–1 are set for other grayscale pixels. The model uses the 'randint' function of NumPy to choose the number of dots appearing on the screen. The function 'white dots propagation' randomizes the positions of dots for each frame. It makes the screen non-static and disturbs the player. However, in some cases, the model regains its normal behavior even after inserting the bug into it. It ignores the white dots and does not cause distractions for the player. It happens when the model is rigorously trained until it learns that the white dots are irrelevant information.

Expected Behavior: The bug 'white dots' is highly visible. It is very distracting to a human player. It also causes an enormous change to the input model. Thus, it is easy to spot this bug.

Actual Behavior: The model shows its expected behavior before the bug is introduced. Its loss function remains in the range of 0–2 in a bug-free environment. After introducing the bug, the model shows a significant spike in the loss function. The value of the loss function rises beyond 100. The abrupt and high change in the value of the loss function makes this bug easily identifiable.

To instantly identify the bug, the threshold value of the loss function is set 3. The value of the threshold, being higher than the default value of the loss function, ensures that the model will not mislead if there is no bug in the game. Figure 4 shows the change in values of the loss function in a game environment with 'white dots' bug inserted into the game at different iterations.





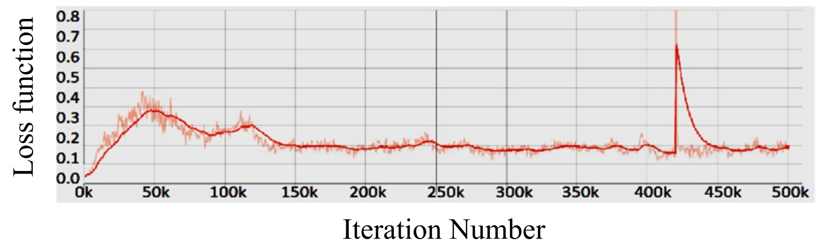


Fig. 4 **a** Loss function with 'White dot bug' inserted at the iteration range from 100,000 to 150,000. **b** Loss function with 'White dot bug' inserted at the iteration range from 350,000 to 420,000. **c** Loss function with 'White dot bug' inserted at the iteration range from 420,000 to 450,000

On inserting the bug at the range of iterations from 100,000 to 150,000, the model reports the highest value of the loss function 16, as shown in Fig. 4a. It is clear from the graph that the model achieves this highest value and shows the peak at the iteration number 150,000. On the contrary, the highest value of the loss function beacomes 110, when the bug is inserted at the range of iterations from 350,000 to 420,000. It is shown in Fig. 4b that the model achieves this highest value and shows the peak at iteration number 410,000.

On inserting the bug at the range of iterations from 420,000 to 450,000, the highest value of the loss function decreases sharply, and it becomes 0.6. It is clear from the graph shown in Fig. 4c that the model achieves this highest value and shows the peak at iteration number 420,000.

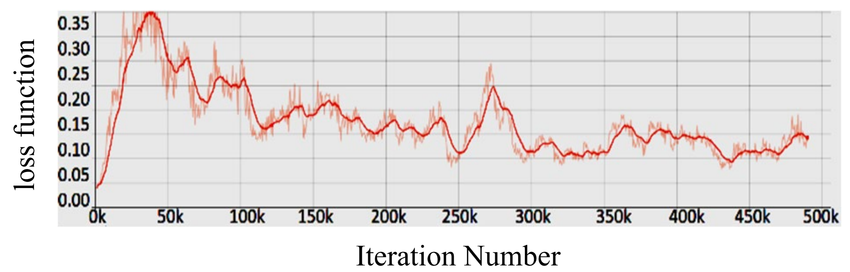
It is observed from the graphs shown in Fig. 4a–c, that the model reports the variations in the highest values of loss function when the 'White Dots' bug is inserted at a different range of iterations. This is because the model continuously learns the mechanism of the game and updates its weight with a change in the values of the loss function.

4.2.3 *Black screen propagation*

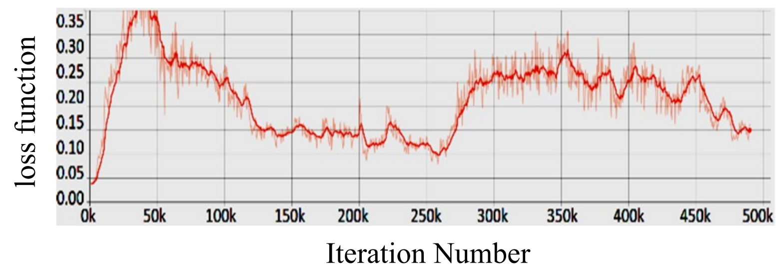
The function 'Black Screen' is designed to simulate the crashing of the complete game. The game with the bug 'Black Screen' displays only a black screen. The DQN is still fully capable of controlling the game and feeding the inputs.

Implementation: To implement the bug 'Black Screen', the authors multiplied the input matrices of the screen with zero. This fills the complete matrices of the input screen of 84×84 size with zero values. All entries with zero values represent that the game has been crashed and it will never recover. The function needs to restart for recovering the game and continue playing. The effect of this bug is permanent. Thus, the authors executed the model for more iterations. This is useful for the successful training of the model. For example, the other bugs are removed after the 30,000 to 50,000 iterations. However, this bug is allowed to run as a part of the game from 50,000 to 200,000 iterations.

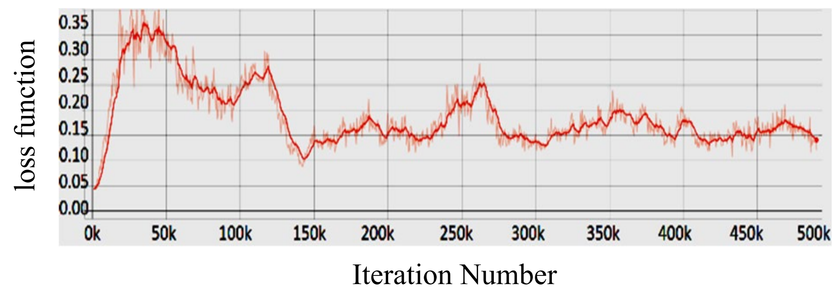
Expected Behavior: The loss function is a combination of the actual value of screen inputs and the uncertainty of the model. Here, uncertainty the degree of variation in the



(a)



(b)



(c)

Fig. 5 **a** Loss function with 'Black Screen' bug Inserted at the iteration range from 100,000 to 150,000. **b** Loss function with 'Black bug inserted at the iteration range from 250,000 to 450,000. **c** Loss function with 'Black Screen bug' inserted at the iteration range from 420,000 to 470,000

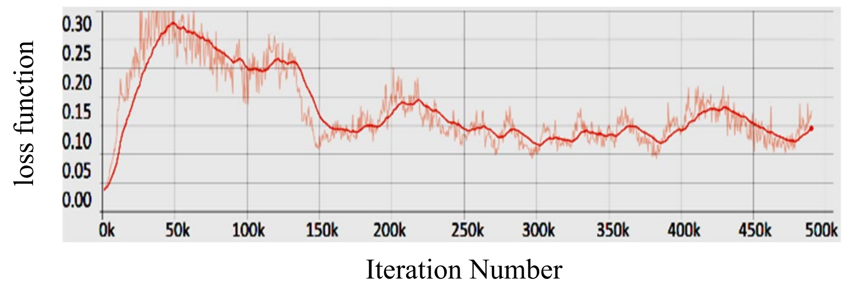
model's behavior to adapt to the changes fed into it. The encounter of the bug 'Black Screen' results in the abrupt rise in the loss function. The bug leads to an abrupt decrease in the value of loss if there is a lack of input given to the model.

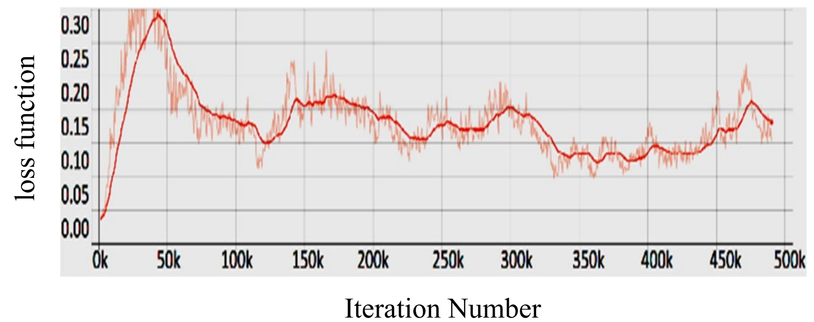
Actual Behavior: At the initial stage, before introducing the bug in the game environment, no deviation from the expected behavior of the model is observed. As the time for executing the game progresses, the loss function of the model fluctuates slightly. However, it seems that the fluctuation in the loss function is constant. This shows that the model is not learning anything new, and the environment is bug-free. The deviation in the loss function makes the bug easily identifiable. Figure 5 demonstrates the variations in the values of the loss function when the bug 'Black Screen' is inserted into the game.

The graph is shown in Fig. 5a demonstrates the deviation in the loss function in the presence of the bug 'Black Screen' in the game environment.

It shows the spike of a loss function in the game environment when the bug is inserted at the iterations from 100,000 to 150,000. The graph peak is obtained at the value of 0.35 for the loss function. On inserting this bug for the iterations from 250,000 to 450,000, a slight change in the highest value of the loss function is observed. The highest value of the loss function becomes 0.36, as shown in the peak plotted in Fig. 5b. The bug 'Black screen' causes a slight decrease in the value of loss function when it is inserted in the iteration range from 420,000 to 470,000. As shown in Fig. 5c, the graph peak is obtained at the 0.33 value of the loss function.

It is observed from the graphs shown in Fig. 5a–c), that the model reports the slight variations in the highest values of loss function when the 'Black Screen' bug is inserted at a different range of iterations. This is because the model gradually learns the mechanism of the game and updates its weights with a change in the values of the loss function when the bug 'Black Screen' is inserted into the game.





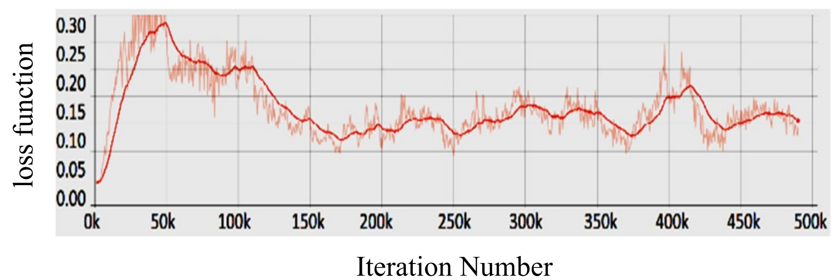


Fig. 6 **a** Loss function with 'Sprite Disappearing' bug inserted at the iteration range from 100,000 to 150,000. **b** Loss function with 'Sprite Disappearing' bug inserted at the iteration range from 350,000 to 400,000. **c** Loss Function with 'Sprite Disappearing' bug inserted at the iteration range from 420,000 to 450,000

4.2.4 'Sprite disappearing' propagation

Protecting the spaceship is the main objective of the game. Thus, making it invisible enormously affect the gameplay. The authors performed a set of experiments to observe the impact of 'player's spaceship sprite' on the game. They removed the player's spaceship sprite and observed the impact on the gameplay.

Implementation: The bug 'Sprite Disappearing' makes the bottom 20% of the screen black. This bug has an insignificant impact on the screen because it blackens only the bottom 20% pixels of the screen. The objective of this bug is only to remove the players' sprite, which appears only at the bottom of the screen.

Expected Behavior: The bug 'Sprite Disappearing' is difficult to detect because it causes a negligible variation in loss function value.

Actual Behavior: The behavior of the model is indistinguishable in the presence and absence of the bug 'Sprite Disappearing' because it does not significantly change the value of the loss function.

Figure 6 demonstrates the deviation in the value of the loss function on inserting the bug 'Sprite Disappearing' in the game at different iterations.

It is clear from the graphs shown in Fig. 6a, b that inserting the bug 'Sprite Disappearing' at the iteration range from 100,000 to 150,000 and 420,000 to 450,000, the model shows the exact value of the loss function as 0.28. Whereas, inserting the bug from 350,000 to 400,000 iterations, a small increase of 0.1 is observed in the value of the loss function. In this case, the value of the loss function becomes 0.29, as shown in Fig. 6c.

It is observed from the graphs shown in Fig. 6a–c) that the model reports negligible variations in the highest values of loss function when the 'Sprite Disappearing' bug is inserted at a different range of iterations. The model gradually learns the mechanism of the game and updates its weights with a change in the values of the loss function.

5 Discussion

In this manuscript, we achieved the objective of designing an automatic deep reinforcement learning based model 'RLBGameTester' for automating bug detection in a video game. The proposed approach is game independent, and need minimum human intervention even to find unknown bugs. Also, the proposed model is independent of the game environment. It ensures the robustness of the model. The 'RLBGameTester' is efficient in detecting logical mistakes as well as visual bugs present in the game. Moreover, this model marks the bug or abnormal behavior encountered in the game. We explored the possibility of using deep learning techniques to detect bugs that cannot be detected by pattern matching. Whereas, the game testing technique proposed in reference [8] is game specific. It needs to be customized for each game. This computer vision based strategy automatically detects the anomalies in the video outputs of the game. But, it looks only for the visual bugs and ignores the logical mistakes in the game. The authors in reference [16] discussed that the techniques available in the literature require human supervision and coordination. It requires manual monitoring only to view and edit notable instances. Therefore, it reduces the need for human supervision. The minimum human supervision, high robustness, automatic testing of the game, and platform independence may prove helpful to apply in the game industry. It is recommended to apply on a segment of the game rather than a complete game because this model requires high computational power.

Next, the bug finder proposed in [25] is based on recent developments in applying reinforcement learning to aid video game developers. However, it is a tool designed that can be used by video game testers rather than automatic bug detection. Thus, it incurs extra human efforts and cost to bug detection.

Similarly, the authors in [26] proposed a very through testing engine for bug detection. But, it requires "test oracles" or tools to determine whether the current stage in the game is valid, or need to be flagged as an error. On the other hand, our approach uses the reinforcement learning algorithm to understand anomalous behavior, and enabling end-to-end autonomous testing. It requires very little human intervention only to confirm that a particular behavior is a bug or an intended feature.

6 Conclusion

In this manuscript, the authors applied the DQN algorithm to develop the model 'RLBGameTester' to understand the mechanism of a game thoroughly. The model is trained for 490,000 iterations. It is trained and tested in a bug-free game environment as well as by inserting three different types of bugs 'white dots' or 'black screen' or 'sprite disappearing'. Figures 4, 5 and 6 showcase the variations in the values of the loss function for observation of the impacts of each type of bug on the game environment.

The proposed model relies solely on an input screen and the prebuilt reward system. It can be employed in a video game of a high level of complexity and learns to play the game without any human supervision. Moreover, the model is efficient in playing multiple games which are entirely different in the playing styles and game mechanics. It implies that the model will prove an effective testing agent for versatile games and in different environments.

The experimental results prove that the model shows a sharp increase or decrease in the loss value when the bug is encountered in the game. The deviations in the values of loss function at different iterations are recorded in the graphs. The peaks in the graphs, reports the iteration number where the bug appears in the game. Thus, this model is useful in identifying the type(s) of bug(s) present in a bug and also the point where the bugs were encountered. The model will prove useful in the game industry for automatic testing the video games to detect the presence of the bugs 'white dots', 'black screen', and 'sprite disappearing'. However, it needs to be trained in an entirely bug-free environment also. It is feasible only in a closed system or a bug-free sample of the game.

However, our approach has potential of autonomous testing it is limited to Atari games. It can be scaled up for other games to add value to the video game industry. Next, the approach used in [26] is incredibly helpful in improving exploration and testing harder to reach stages of the game. Such approaches could be integrated to create a more generalizable tool for video game testing.

Funding The work presented in this manuscript is not funded by any private or Government funding agency.

Data availability NA.

Code availability The code is available with authors. The code will be submitted after receiving the decision for the submitted manuscript.

Declarations

Conflict of interest The authors declare that, there is no conflict of interest associated with this research.

References

1. Vlachopoulos D, Makri A (2017) The effect of games and simulations on higher education: a systematic literature review. Int

J Educ Technol High Educ 14(1):22. https://doi.org/10.1186/s41239-017-0062-1
2. Berg Marklund B, Engström H, Hellkvist M, Backlund P (2019) What empirically based research tells us about game development. Comput Games J 8(3–4):179–198. https://doi.org/10.1007/s40869-019-00085-1
3. Aleem S, Capretz LF, Ahmed F (2016) Critical success factors to improve the game development process from a developer's perspective. J Comput Sci Technol 31(5):925–950. https://doi.org/10.1007/s11390-016-1673-z
4. Arzate Cruz C, Ramirez Uresti JA (2017) Player-centered game AI from a flow perspective: towards a better understanding of past trends and future directions. Entertain Comput 20:11–24. https://doi.org/10.1016/j.entcom.2017.02.003
5. Coates A, Abbeel P, Ng AY (2017) Autonomous helicopter flight using reinforcement learning. Encyclopedia of machine learning and data mining. Springer US, Boston, pp 75–85. https://doi.org/10.1007/978-1-4899-7687-1_16
6. Tesauro G (2002) Programming backgammon using self-teaching neural nets. Artif Intell 134:1–2. https://doi.org/10.1016/S0004-3702(01)00110-2
7. Mnih V et al (2013) Playing atari with deep reinforcement learning
8. Ariyurek S, Betin-Can A, Surer E (2021) Automated video game testing using synthetic and humanlike agents. IEEE Trans Games 13(1):50–67. https://doi.org/10.1109/TG.2019.2947597
9. IBF de MSABJRCS JHJKJ, P. R. P. J. P. H. C. M. S. L. L. S. N. N. A. K. Z. Yunqi Zhao (2020) Winning isn't everything: enhancing game development with intelligent agents. Arxiv-Comput Sci-Artif Intell 1–14
10. Qureshi AH, Nakamura Y, Yoshikawa Y, Ishiguro H (2017) Robot gains social intelligence through multimodal deep reinforcement learning
11. Agarwal M, Rani G, Dhaka VS (2020) Optimized contrast enhancement for tumor detection. Int J Imaging Syst Technol 30(3):687–703. https://doi.org/10.1002/ima.22408
12. Rani G, Jindal A (2020) Real-time object detection and tracking using velocity control. 767–778. https://doi.org/10.1007/978-981-13-8406-6_72
13. Nitesh Pradhan HC, Dhaka VS, Rani G (2022) Machine learning model for multi-view visualization of medical images. Comput J 65(4):805–817
14. Aleem S, Capretz LF, Ahmed F (2016) Game development software engineering process life cycle: a systematic review. J Softw Eng Res Dev. https://doi.org/10.1186/s40411-016-0032-7
15. Giannakopoulos P, Cotronis Y (2018) A deep Q-learning agent for the L-game with variable batch training
16. Hernández Bécares J, Valero CL, Gómez Martín PP (2017) An approach to automated videogame beta testing. Entertain Comput 18:79–92. https://doi.org/10.1016/j.entcom.2016.08.002
17. Davis J, Hsieh Y-H, Lee H-C (2015) Humans perceive flicker artifacts at 500 Hz. Sci Rep 5(1):7861. https://doi.org/10.1038/srep07861
18. Deep Q-Network, PyTorch with (2022) https://towardsdatascience.com/deep-q-network-with-pytorch-146bfa939dfe. Accessed 02 Jul 2022
19. Varvaressos S, Lavoie K, Gaboury S, Hallé S (2017) Automated bug finding in video games. Comput Entertain 15(1):1–28. https://doi.org/10.1145/2700529
20. Meredith PO, Jin D, Griffith D, Chen F, Roşu G (2012) An overview of the MOP runtime verification framework. Int J Softw Tools Technol Transf 14(3):249–289. https://doi.org/10.1007/s10009-011-0198-6
21. Colombo C, Pace GJ, Schneider G (2009) LARVA---safer monitoring of real-time java programs (Tool Paper). In: Seventh IEEE international conference on software engineering and formal methods, 2009, pp 33–37. https://doi.org/10.1109/SEFM.2009.13
22. Barringer H, Havelund K (2011) TraceContract: a scala DSL for trace analysis. pp 57–72. https://doi.org/10.1007/978-3-642-21437-0_7
23. Cunneen M, Mullins M, Murphy F, Shannon D, Furxhi I, Ryan C (2020) Autonomous vehicles and avoiding the trolley (Dilemma): vehicle perception, classification, and the challenges of framing decision ethics. Cybern Syst 51(1):59–80. https://doi.org/10.1080/01969722.2019.1660541
24. 18. NVIDIA Isaac: Virtual Simulator for Robots: 2018.”
25. Joakim LG, Bergdahl C, Gordillo K, Tollmar (2021) Augmenting automated game testing with deep reinforcement learning. Arxiv Comput Sci Learn. https://doi.org/10.48550/arXiv.2103.15819
26. Zheng Y, Xie X, Su T, Ma L, Hao J, Meng Z, Liu Y, Shen R, Chen Y, Fan C Wuji: automatic online combat game testing using evolutionary deep reinforcement learning. s://crazynote.v.netease.com/2021/1011/e38c5f3d60a830785f1bdd8b69563c45.pdf

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.