Training an AI-Powered Doomguy Leveraging Deep Reinforcement Learning

Boyi Xiao

College of Artificial Intelligence, Tianjin University of Science & Technology, Tianjin, 300000, China
21201108@mail.tust.edu.cn

Abstract. Reinforcement learning (RL) has recently gained significant attention due to the impressive successes of self-driving cars and human-like performance in games such as Go or StarCraft. However, approaching this subject can be intimidating. In this research, the author aims to explore how to train a RL agent to play various Doom scenarios. This will provide an opportunity to explore various aspects of RL, such as curriculum learning, reward shaping, and machine learning in general. The author will also address how to monitor the agent's progress during training and how to fix any issues that arise. Monitoring is crucial to ensure that the agent is learning effectively and behavior is appropriate. If any issues are detected, they can be fixed by adjusting the training process or the reward structure. The ultimate goal of this research is to train an RL agent to play deathmatch against real human players, with the author replacing humans with in-game bots. At the end of this article, you will see a human-like agent playing against bots like a real player. The author hopes to demonstrate the potential of RL in creating intelligent and autonomous agents that can compete against humans in complex environments.

Keywords: Reinforcement Learning, Doom Game, Deep Learning.

1 Introduction

Reinforcement learning (RL) is a type of machine learning that allows an agent to learn from its interactions with an environment. An RL agent repeatedly tries out new actions and receives rewards or penalties based on its behavior. Over time, the RL agent learns how to take actions that maximize its rewards [1].

Deep reinforcement learning (DRL) is a combination of RL and deep learning [2,3]. Deep learning is a type of machine learning that allows neural networks to learn complex relationships. DRL agents use deep learning to learn how to make the best decisions in complex environments. DRL has been successful in a variety of domains, including games, robotics, and finance. For example, AlphaGo is a Go program developed by Google AI that defeated world champion Lee Sedol in 2016. AlphaGo used DRL to learn how to play Go [4].

B. H. Ahmad (ed.), *Proceedings of the 2023 International Conference on Data Science, Advanced Algorithm and Intelligent Computing (DAI 2023)*, Advances in Intelligent Systems Research 180,
https://doi.org/10.2991/978-94-6463-370-2_26

In the above cases, the goals are all achieved through a lot of training, and video games are an indispensable part of training. The ideal environment for deep learning is often considered to be games due to several important characteristics and advantages they offer, making them an ideal setting for researching and training deep learning algorithms for the following reasons:

Firstly, games can generate a large volume of data, including images, videos, audio, etc., which can be used for training deep learning models. Additionally, games can provide accurate annotation information such as scores, win-loss outcomes, player actions, etc., aiding in supervised learning model training.

Secondly, games can provide a virtual simulation environment where various parameters and conditions can be controlled, providing a well-controlled testing scenario for deep learning algorithms. This is valuable for researching and developing new algorithms and techniques.

Thirdly, many games feature intricate plots, strategies, and difficulties, making problem-solving within games a challenging task. Deep learning algorithms can learn and optimize within these complex environments, enhancing their performance and generalization capabilities.

Fourthly, fames often require real-time decision-making and interaction, demanding that deep learning models make accurate predictions and decisions within short time frames. This helps drive the development of deep learning algorithms, enabling them to better handle real-time tasks.

Finally, in games, various innovative algorithms and techniques can be experimented with to explore their effectiveness in different scenarios. This contributes to advancing the field of deep learning, encouraging researchers to develop more powerful and intelligent models and methods.

In summary, games, as rich, challenging, and interactive simulation environments, provide valuable opportunities for researching and training deep learning algorithms. They help drive the continuous development and innovation of artificial intelligence technology [4].

The original Doom, published in 1993, is a first-person shooter game developed by id Software, considered one of the earliest popular 3D shooter games. The gameplay is straightforward, where players assume the role of a soldier who must battle demons and monsters within a research facility on Mars. The game gained renown for its fast-paced action, smooth controls, and innovative 3D rendering, making it a classic in the history of gaming.

In this report, the author will use DRL algorithms to train the agent, also the Doomguy. The agent will get screenshot as input, without knowing the internal state of the environment, which is an open-source project called Vizdoom, used to train agent to play Doom. This work will test some algorithm's performance in Doom and compare its difference performance between different parameters. Such as the frame it skipped.

The author believe that this paper will show how DRL can be used to train agents to learn how to perform tasks in complex environments. It can be a powerful machine learning technique that has a wide range of potential applications.

2 Method

2.1 The Game Environment

In 1996, id Software released the source code for Doom three years after the game's release, which was a very rare thing at the time. This allowed many developers to modify and improve the game and created many new game modes and maps. Doom's open source also promoted the development of open-source game engines and enabled the development of many other games [5].

Doom's open source also promoted the development of the open-source movement. Doom was a very popular game, and its open source made many people meet open-source software for the first time. This made open-source software more popular and promoted the development of the open-source movement.

Vizdoom is an open-source, Doom-based RL environment. It is designed for research and development of RL algorithms. Vizdoom uses the same engine as Doom, so it can use all of Doom's features, including graphics, sound, and physics simulation. Vizdoom also provides an easy-to-use API that allows users to easily create and customize tasks [6].

Vizdoom works with scenarios. Each scenario describes a different situation with its own rules and rewards. A scenario is usually composed of two files. A ".cfg" file which parametrize the learning environment by defining for example which buttons are available to the agent or what is the screen input format. A ".wad" file which is a binary file that contains the maps and other resources specific to Doom.

Vizdoom has been used to research and develop a variety of RL algorithms, including Q-learning, Double-Q learning (DQL), and Asynchronous advantage actor critic (A3C). These algorithms have been shown to be able to learn complex tasks in Vizdoom, such as playing Doom and navigating a maze [6].

2.2 The Model

This paper will be using Proximal Policy Optimization (PPO) with an Advantage Actor Critic (A2C) policy. Thanks to stable baselines, training a PPO agent will be much easy than author thought [7,8].

The only caveat is that stable baselines expect a gym environment. A gym is a very simple interface to standardize the interactions between RL agents and various environments [9]. Therefore, this work will need to write an adapter to make it work with Vizdoom [6].

The architecture of the model in this work is demonstrated in Fig 1. The input to this model, or observation space, will always be one (or more) color images of shape (height, width, channels) whose pixel values are integers in the range 0 to 255. When the program gets the frame, it will blur the frame with OpenCV, like Fig 2 [10]. The output of the model will be an integer representing the index of the ideal action. The final model is based on a custom convolutional neural network from the previous article with a couple of "size" changes. 512 neurons for the first flat fully connected layer. 256 neurons in a fully connected layer for the value net. 256 neurons in a fully connected layer for the action net.

In the repository you might see different dimensions for the input layer. This is due to the usage of frame stacking. However, the author has noticed no difference in performance with or without frame stacking. The illustration below shows the model architecture without the layer norms.

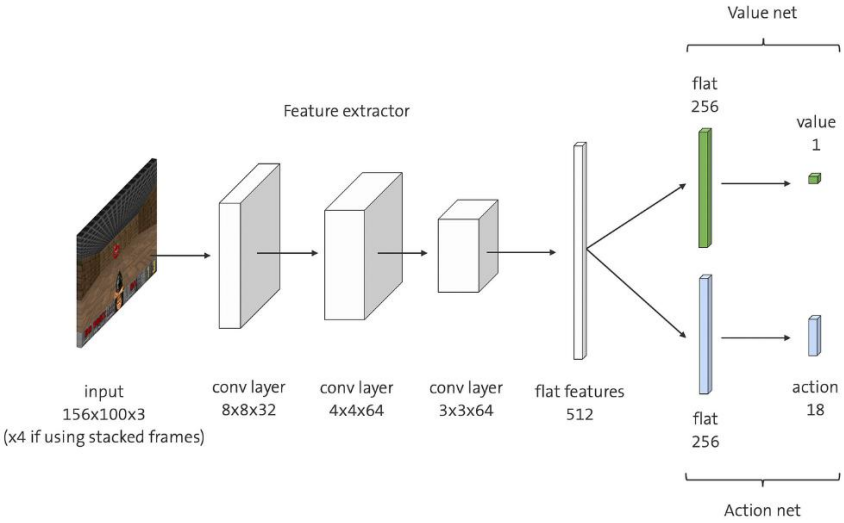

Fig. 1. Architecture of the model with layer norms omitted (Figure credit: Original).

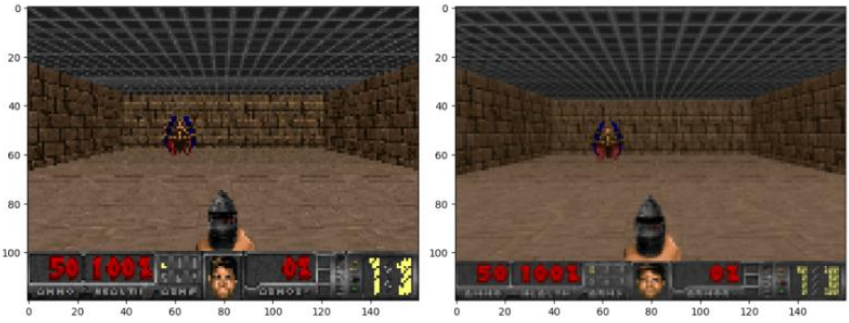

Fig. 2. The original and blurred frame (Figure credit: Original).

2.3 Scenarios

Basic. This is a simple scenario which have a box-shaped room and a randomly spawned enemy. The agent needs to shoot it as fast as possible while remaining accurate. In Fig 3 a) the triangle means player, and white square is a randomly spawned enemy, there will be only 1 enemy spawn each time.

There are rewards which used: +101 for killing an enemy. -1 per time step. -5 for missed shots. The episode ends after 300-time steps. And the agent has 3 action to do: move left or right and shot.

Defend the Center. This scenario include a round shape room and tons of randomly spawned enemies. In this scenario, the agent will stand at the middle of the map, which cannot move. The hostile appears around him randomly. The Doomguy only has a pistol which can one-shot any enemies but only have 26 bullets. In Fig 3 b), the triangle means player, and white square is a randomly spawned enemy, there will be multiple enemies spawn each time. And the Enemy will attack player.

There are rewards which used: +1 per kill. -1 per death. The 3 actions agent can do: turn left or right and shot.

In real deathmatch situation, the player needs to move around to find enemy to kill, so after the basic training, the author add new actions into the config file. The action after edit includes turn left or right, move left or right, go forward, and shot.

Deathmatch. Deathmatch is a 2:30-minute free-for-all in which players compete to get the most kills. There are no teams. A kill is defined as killing an enemy. Each player starts with a pistol and a brief period of invincibility. After dying, a player respawns at a random spawn point on the map. The scenario is demonstrated in Fig 3 c).

The agent can perform the same actions in both the easy and difficult variants of "defend the center". These actions are moving, turning, and shooting, which are all defined in advance.

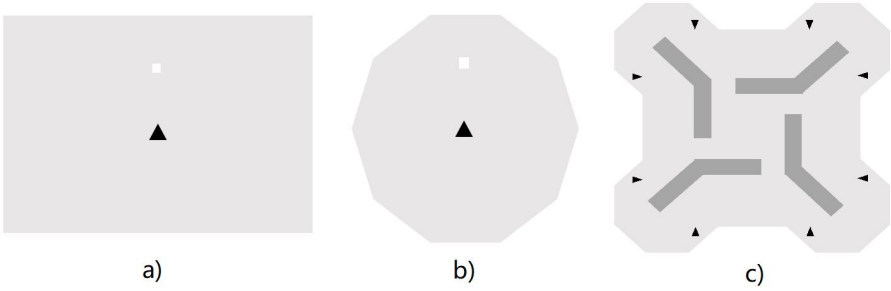

Fig. 3. The shape of a) Basic, b) Defend the Center, and c) Deathmatch (Figure credit: Original).

3 Results

Training this agent to the final goal requires a step-by-step training, from simple to difficult. Although some steps are not necessary, the author still followed them to make it easier to understand and compare.

3.1 The Result of Basic

The training process is very simple, as paper mentioned before, with the help of stable-baseline3, creating an agent became very straight forward. By calling the PPO constructor, an agent with an A2C policy will be created. The training process is doing by stable-baselines, which results are demonstrated in Fig 4.

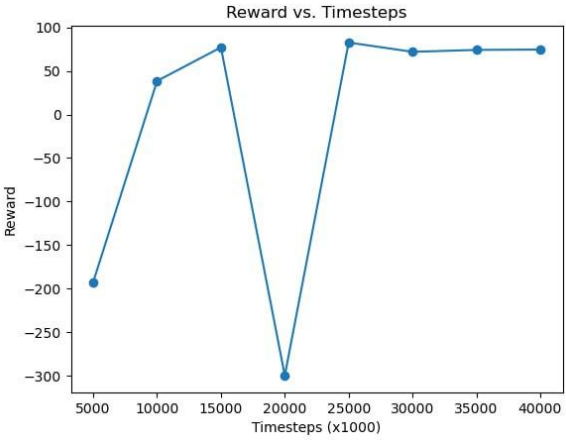

Fig. 4. The result of Basic (Figure credit: Original).

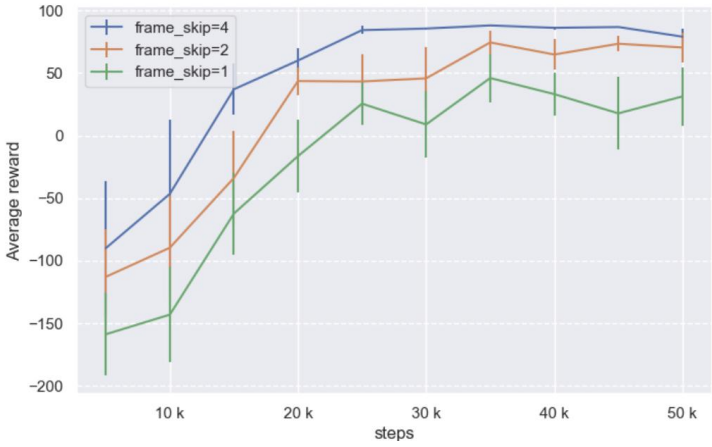

Fig. 5. The result with different frame skips (Figure credit: Original).

When set the frame skip parameter to 4, the agent will shoot to early, so author did a series of experiments with varying frame skip sizes, as shown in Fig 5.

Using 160 x 120 pixels image to input is a bit unnecessary. Cutting the meaningless parts means that this model will need significantly less trainable parameters. If modify the environment and use instead a cropped version of the input frame, it can reduce the number of parameters by 2.6M.

In addition, the PPO class from stable-baseline3 allows us to directly modify the architecture of the model using keyword arguments. In particular, the default number of features produced at the end of the convolutional layers is 512. This is also unnecessarily large. Decreasing the size of trainable parameters by a factor 7. The training process are much faster than before.

3.2 The Result of Defend the Center

When came to Defend the Center, the training method is same as before. And the author defined a little helper function that will streamline the training and evaluation process as needed to repeat it several time. The function simply creates the environments, instantiate an agent based on the PPO algorithm and start training and optionally evaluating the agent for a given number of steps. As demonstrated in Fig 6, When author training about 100k steps, this agent already gets some pretty good result, so it's enough for training.

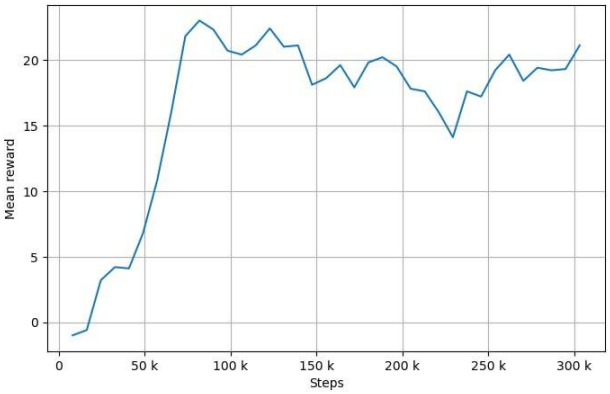

Fig. 6. The result of Defend the Center (Figure credit: Original).

As paper mentioned before, A real Deathmatch situation is very complex than stand still and shot. So, the author added some new actions for the agent, making it can move around.

Vizdoom provides a list of button states, but model uses a discrete action space. A simple solution is to create a discrete action space by generating all possible combinations of buttons and using each combination as a single command. More like a real human player will do. Although, the number of all the possible combination of output is 2^n where n is the amounts of possible commands. And in this situation, the agent has 64 available move set to choose.

The agent needs to disallow specific combinations in order to reduce the combinatorial explosion of the action space. Allowing, for instance, both LOOK_RIGHT and LOOK_LEFT to be outputted at the same moment does not make much sense. As a result, the author eliminated the combinations LOOK_RIGHT and LOOK_LEFT as well as GO_RIGHT and GO_LEFT.

Additionally, the author forbids using anything other than the shoot button. Because in original Doom game, most weapons have a cooldown between two attacks anyhow, so the agent performance won't be affected by. In the end, the total number of different possible actions is given by the cartesian product of {ATTACK} + {TURN_RIGHT, LOOK_LEFT, NOTHING} x {GO_RIGHT, GO_LEFT, NOTHING} x {MOVE_FORWARD, NOTHING}. Since the author don't want the all-zero state (no button pressed) the agent will offer us a final total of 18, which is considerably better compared to the 64. As shown in Fig 7, More training time will be needed to train the model in this more complex scenario.

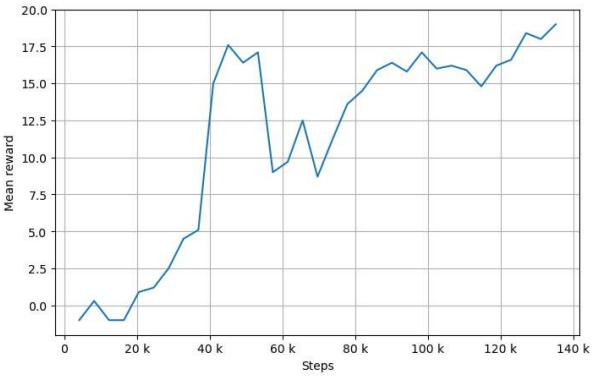

Fig. 7. The result of edited version of Defend the Center (Figure credit: Original).

3.3 The Result of Deathmatch

Training an agent to play deathmatch is significantly harder than the previous scenarios the author have tried so far, also need to be patient and train for a long time. But the training process and the environment are basically same as Defend the Center, so the author just show the training result in Fig 8.

When reach this step, the goal is already achieved. The agent is already able to fight against the bot, and of course, it can play with human player too. Just by replace the bot with human.

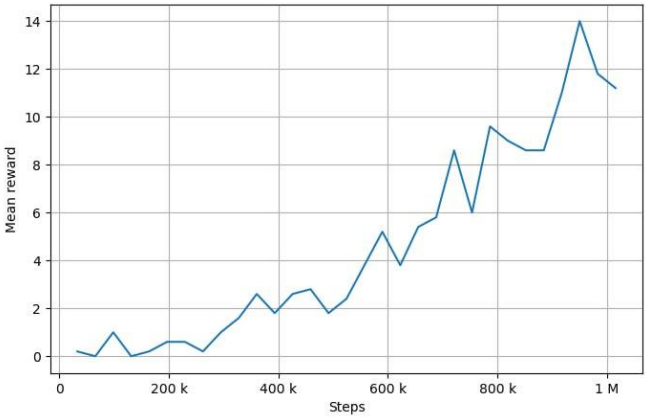

Fig. 8. The result of Deathmatch (Figure credit: Original).

4 Discussion

The in-game's bot is not real AI, their actions are determined from a set of predefined rules. After performing a few million training steps, the result started becoming some interesting behaviors coming from the agent. However, one substantial limitation of the setup was the scarcity of the reward signal. Indeed, in a naive approach where each enemy killed grants one reward point, obtaining such reward requires a lot of consecutive steps done just right. The agent must chase and aim at moving enemies while repeatedly shooting them in order to possibly obtain a reward. Also, it needs to do it while avoiding obstacles like walls. Since RL tries to increase the likelihood of behaviors that result in positive feedback, the fact that rewards could be observed from time to time means that the learning process will be very slow. So, to edit the reward to make the agent learning faster and more efficient is a real deal for that.

Also, the author can make a scenario which allow 2 agents against with each other. It will be a very interesting situation. And the author can spawn 2 instances of Doom, one for human player and the other for the trained agent.

You might have noticed that the agent has no concept of memory. This means that enemies that are not visible on the screen are immediately forgotten by the agent. Also, the AI does not keep track of places that it has already visited. This is not a big issue when playing against 8 programmed bots as there is always an enemy close by. However, when playing against a single opponent, the agent will revisit several times the same location or simply ignore some area of the map it should have explored.

A potential improvement here would be to use a model that has a concept of memory like a Long short-term memory (LSTM) neural network. The following paper shows that such a model could be used to play Doom effectively.

5 Conclusion

In this paper, DRL models are leveraged to establish a Doomguy. The performances of the model are validated in three different scenarios, including the basic, defend the center, and the deathmatch. The results show that this method can successfully train an agent to play the Doom game. The agent can learn how to avoid enemies, collect power-ups, and kill enemies. This method can also be applied to other games, such as Pac-Man and Space Invaders. The author believe that this work is an important step towards training agents to play games. It is expected that this work will inspire other researchers to work in this area, and eventually develop powerful agents that can play a variety of games.

References

1. Li, Y. Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274 (2017).
2. Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6), 26-38 (2017).
3. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. An introduction to deep reinforcement learning. Foundations and Trends in Machine Learning, 11(3-4), 219-354 (2018).
4. Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. A survey of deep reinforcement learning in video games. arXiv preprint arXiv:1912.10944 (2019).
5. Lewis, M., & Jacobson, J. Game engines. Communications of the ACM, 45(1), 27 (2002).
6. Kempka, M., Wydmuch, M., Runc, G., Toczek, J., & Jaśkowski, W. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In 2016 IEEE conference on computational intelligence and games, 1-8 (2016).
7. Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. Implementation matters in deep policy gradients: A case study on ppo and trpo. arXiv preprint arXiv:2005.12729 (2020).
8. Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. Stable baselines3, 1-4 (2019).
9. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. Openai gym. arXiv preprint arXiv:1606.01540 (2016).
10. Bradski, G. The openCV library. Dr. Dobb's Journal: Software Tools for the Professional Programmer, 25(11), 120-123 (2000).