Playing Super Mario Bros and Doom with Reinforcement Learning

Tarık Bulut
Student of Firat University Faculty of Technology
Firat University Faculty of Technology
195541035@firat.edu.tr

Abstract

The purpose of this paper is to compare and evaluate the performance of models using PPO and DQN algorithms in reinforcement learning for games more complex than Pong, Space Invader, and Breakout, such as Super Mario Bros and Doom (1993).

In addition to utilizing Convolutional Neural Networks (CNNs), methods such as frame skipping, correlation, and reward shaping are also employed. The approach primarily involves processing raw images to generate outputs corresponding to button presses on a game controller. The results show progress up to the second level in Super Mario Bros and successful completion of corridor and center defense maps in Doom.

1 Introduction

Training high-quality data (audio or visual) with artificial intelligence has always been a challenging process. Particularly in games, enabling characters to play as if they were human requires a significant amount of high-quality data, often leading to suboptimal and inefficient results. However, with Reinforcement Learning, there is no need for prior data input. The model can independently make inferences, determine its own weights, and minimize latency during these processes. This has made Reinforcement Learning a prominent approach in games and new-generation AI algorithms like Chat-GPT [1], leading me to choose this model for my work.

Reinforcement Learning, in short, involves an artificial intelligence agent taking actions based on its environment and receiving rewards or penalties as a result of these actions [2]. The goal is to improve the agent's actions based on the rewards or penalties it receives, ultimately bringing it closer to the desired outcome. I chose Super Mario Bros and Doom as the focus for Reinforcement Learning because their goal-oriented gameplay allows for easier reward system adjustments.

The project is implemented using Python code, which includes libraries, algorithms, and tools, all based on C++.

For the environment, I utilized the gym [3] library, specifically gym-super-mario-bros [4] for Mario and Vizdoom [5] for Doom. These libraries enabled data collection from the games and facilitated input integration for the AI without the need for any external emulators.

2 Pre-processing and introduce Environments

2.1 Super Mario Bros

Super Mario Bros is a 2D, Nintendo-based game where players progress along the X-axis with the objective of reaching the flag at the end of the level without dying. To run the game in a Python environment, I made enhancements to the *gym-super-mario-bros* [4] library, which leverages the NES [6] library. For the environment setup, I used the standard version, "SuperMarioBros-v0."

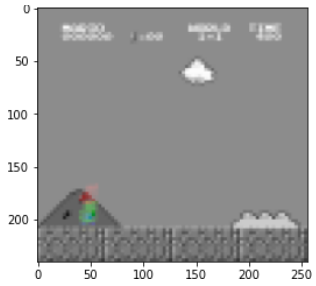


Figure 1.1: An image of the game and the environment used. Figure 1.2 : Image after preprocessing.

When no controller from the NES library was used, the algorithm's output consisted of 256 different options, which caused the AI to experience slowness due to the large number of choices during decision-making. By using the Joypad [4] library with SIMPLE_MOMENT, this number was reduced to 7 options.

The actions are as follows: NOOP, right, right-a, right-b, right-a-b, a, and left. To avoid incorrect learning [7] since the target is on the right side in the reward environment, the actions were restricted. During testing, when random actions were performed for 1000 frames, it was observed that the AI never moved to the left, ensuring that the output process was functioning correctly.

Next, for better optimization of the RL algorithm, additions were made to the environment provided by the library, and some features were removed. First, the Joypad feature was added, and then, with AtariWrapper [8], modifications were made to the image.

AtariWrapper first converted the screen to grayscale. The reason for this is that in current image formats, pixels are represented with matrices, while colors are indicated with three separate matrices for R, G, and B, meaning that Figure 1.1 consists of 2402563 = 184,000 pixels. When converted to grayscale, the number of pixels is reduced to around 61,000. This reduction in pixels increases the AI's FPS value, optimizing its performance and reducing the chances of encountering an OOM (Out of Memory) error. Additionally, by reducing pixel density, unnecessary details were removed.

AtariWrapper secondly performs a resize operation. In preprocessing steps, the library can result in different size values, so to prevent this, the game's window size was fixed to a static value.

Thirdly, AtariWrapper uses FrameSkip [9]. In short, FrameSkip avoids making a decision for every frame (a value of 4 was chosen for the project), providing significant optimization support. Since Super Mario Bros and Doom are not highly reflex-based games, and the FrameSkip value is not large, it did not cause any issues with the AI's gameplay.

With these three modifications, the final image of our environment is shown in Figure 1.2. The subsequent preprocessing steps are Reward Shaping [10] and VecFrameStack [11].

Reward Shaping can be briefly described as providing small intermediate goals to help the agent reach its main objective. In the Mario project, in addition to the reward increase for moving towards the right side, a reward system was added based on the score within the game. Additionally, when the flag at the end of each stage was reached, extra points were awarded; if not reached, negative points were given.

With this system, the agent is encouraged not only to move to the right but also to kill monsters while maintaining its momentum and collecting points. This resulted in the creation of four reward metrics:

With this system, the agent is encouraged not only to move to the right but also to kill monsters without losing momentum and collect points. As a result, four reward metrics were established:

1. Positive points for moving right on the map, negative points for moving left.
2. Negative points for staying in the same place based on game time and frame status (to prevent stagnation, negative points are given if the frame remains the same for too long).
3. Negative points for each death.
4. Positive points for collecting in-game scores (e.g., killing monsters or reaching the flag).

The final step in the preprocessing process, *Frame Stack*, allows the AI to make more accurate decisions by considering multiple independent environments. For example, while the agent typically takes one step per environment, this process enables it to take n steps (in this project, n = 4). This process, similar to the RGB method, creates additional matrices, which can negatively impact optimization but have a positive effect on training the algorithm.

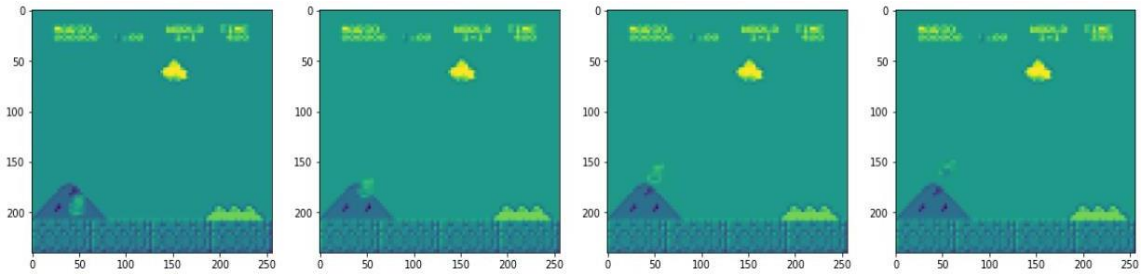


Figure 2: An example showing past frames stored using VecFrameStack.

2.2 DOOM 1993

DOOM 1993, being an open-source and popular game, is a 2D game widely used in various programming projects. After researching a project following Mario, I found that environment processes using VizDoom could be implemented just as easily as in the Mario project.

The VizDoom library allows playing Doom not directly, but through scenarios across different maps. The maps used in this project were Basic, Defend Center, and Deadly Corridor, and training was conducted on these three maps. Three different scenarios can be extracted from these maps. While there are differences in the preprocessing steps, the basic operations remain the same.

2.2.1 DOOM_BASIC & DOOM_DEFEND_CENTER

In *Doom Basic*, the agent is expected to shoot and kill randomly placed monsters using four possible actions: NOOP, left, right, and shoot. In *Doom Basic*, when a monster is between the agent and the wall, the agent should shoot and kill the monster. In contrast, in the *Defend Center* scenario, monsters surround the agent in a circular formation and move towards them.

The reason for processing the *Doom Basic* and *Defend Center* scenarios together is that, apart from the map differences, there are no significant differences in preprocessing steps.

The goal of these scenarios is to quickly detect the monster's location in Doom Basic and kill it while minimizing ammo usage.

In *Defend Center*, monsters move towards the agent and deal damage when they get close enough. The objective is to survive as long as possible with minimal ammo usage. The only way to avoid damage from the monsters is to kill them.

Figure 3.1 : Scene Basic

Figure 3.1 : Scene Defend Center

In the preprocessing process for the Doom project, unlike in Mario, where a ready-made gym environment was used and modified, a custom GymEnvironment[12] is created using the game files provided by VizDoom. In this custom environment, the following functions are written.

Init**:** This function initializes the *Doom* game, sets up the frame configurations, and includes the observation_space and action_space[13] modules. These spaces are customized to ensure that random actions and observations are obtained according to the game's format. The observation values range from 0 to 255, with frame dimensions of 100x160x1 and a noise effect, and 3 different action matrices.

Step: This function is called to execute actions. Similar to the Mario setup, the movement and reward structure is implemented here. The agent earns -1 point when a monster survives, -5 points when a bullet is fired, and 101 points when a monster is killed. Additionally, the frameSkip function is used in this step, with a value of 4. In Defend Center, a monster killed awards +1 point, and a monster's death results in -1 point.

Render**:** Since the *VizDoom* render function is used, this function is left empty.

Reset: This function restarts the game and provides a new stage.

Grayscale Similar to the Mario preprocessing process, this function converts the images to grayscale to reduce the matrix size and decrease pixel density using Inter_cubic interpolation.

Close: This function is responsible for closing the game.

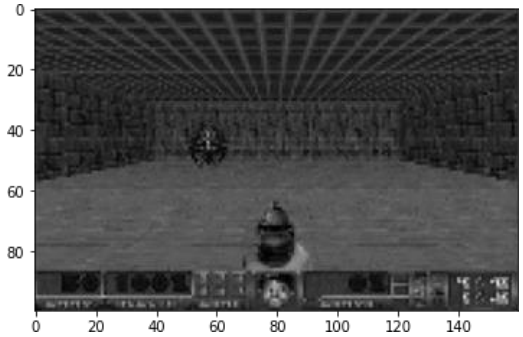


Figure 4.1 : Basic after Preprocessing

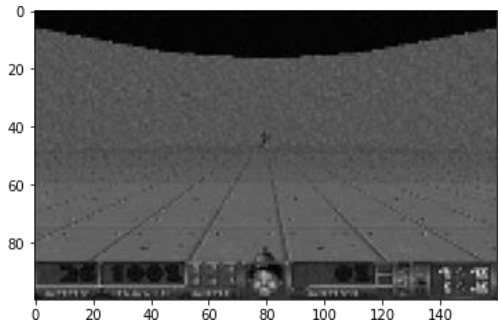
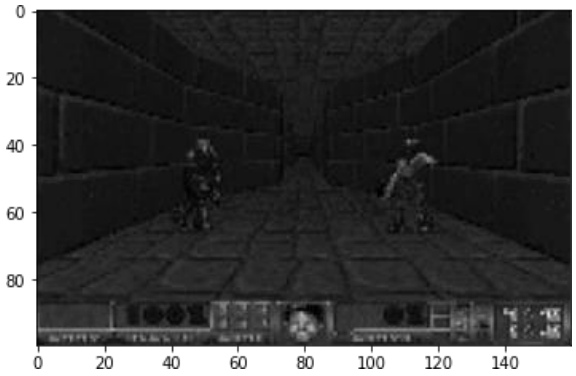


Figure 4.2 : Defend Center after Preprocessing

2.2.2 DOOM_DEADLY_CORRIDOR

The Deadly Corridor scenario is essentially about reaching the target ahead, but there are a total of 6 enemies, 3 on the left and 3 on the right, who shoot at the agent. There are 6 possible actions in total: NOOP, turn-left, turn-right, move-left, move-right, and shoot.

Figure 5.1 : Scene Deadly_Corridor



Figue 5.2 : Deadly_Corridor after Preprocessing

The reason I separate the Deadly_Corridor section is because two different methods are used: Reward Shaping[10] and Curriculum Learning[14].

As explained in Mario, Reward Shaping aims to prevent the agent from learning incorrectly by providing intermediate goals to help it reach the main goal more consistently. In this scenario, the values of damage_taken, hitcount, and ammo are used.

The basic principle is to increase the rewards as the agent moves forward, and also to calculate the delta of the acquired values (the difference between the current and the previous frame) and prioritize these values.

For example, the value hitcount_delta*200 encourages the agent to hit more enemies. If the value was 0 in the previous frame and 1 in the current frame, the delta will be 1, and this will affect the reward as 200 points rather than just 1. Conversely, if the value decreases, a penalty of -200 points is applied. The main goal here is to prevent the agent from completing the level without hitting any enemies. This will be explained in more detail in the Curriculum Learning[14] section.

The values used in the project are:
"reward = movement_reward + damage_taken_delta10 + hitcount_delta200 + ammo_delta*5".

Before proceeding to Curriculum Learning, it's important to understand the enemy levels. In Vizdoom, the enemy levels increase from 1 to 5, with level 5 being quite difficult. Since the map uses level 5 enemies, to help the AI learn better, the difficulty should increase gradually.

Curriculum Learning is about starting with the enemies at the lowest level so the agent can adapt more easily, and gradually increasing the difficulty until it reaches the main difficulty level. As mentioned in Reward Shaping, our goal is for the agent to hit enemies. If the agent finds a way to pass the first level without hitting enemies, it won't be able to successfully complete the later stages, so it must learn to hit enemies.

With this, we have completed the preprocessing steps for both Mario and Doom. From this point onward, the processes will be the same as in the Deadly_Corridor scenario, except for Curriculum Learning, which will be explained separately.

3 Configure Callback and Reinforcement Learning

Under this heading, we will briefly discuss the Callback process, models used in Reinforcement Learning, results, and comparisons.

The Callback process[15] is a function that helps us save the trained data, collect log records, and is commonly found in the same format on the internet. Since it works integrated with stable_baseline3[16], it can function properly in any stable_baseline3 training model.

4. Setup Callback

In [18]:
```
# Import os for file nav
import os
# Import callback class from sb3
from stable_baselines3.common.callbacks import BaseCallback
```

In [19]:
```
class TrainAndLoggingCallback(BaseCallback):

    def __init__(self, check_freq, save_path, verbose=1):
        super(TrainAndLoggingCallback, self).__init__(verbose)
        self.check_freq = check_freq
        self.save_path = save_path

    def _init_callback(self):
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self):
        if self.n_calls % self.check_freq == 0:
            model_path = os.path.join(self.save_path, 'best_model_{}'.format(self.n_calls))
            self.model.save(model_path)

        return True
```

In [20]:
```
CHECKPOINT_DIR = "./train/train_deadly_corridor"
LOG_DIR = "./logs/log_deadly_corridor"
```

In [21]:
```
callback = TrainAndLoggingCallback(check_freq=50000, save_path=CHECKPOINT_DIR)
```

Figure 6 : Callback code block

In RL processes, the artificial intelligence algorithms we will use are PPO and DQN models, which have shown the best performance in games. However, before introducing the models, let's briefly talk about Stable-baseline3, where these models are located.

Stable-baseline3 is a library that provides ready-to-use functions for many tasks in Reinforcement Learning. In the project, it is a comprehensive library that can be used in almost every area, such as model usage, preprocessing tasks, and logging processes.

The first model used is the PPO[17] (Proximal Policy Optimization) model. Developed by OpenAI, PPO is a policy-based algorithm that does not use a replay buffer, meaning it learns from the current state rather than past experiences. The model attempts to learn immediately, distributing its learning equally rather than focusing on the later stages of training.

One limitation of the PPO model is its tendency to rely on memorization. For example, in the Doom_basic scenario, since enemies can appear either on the left or the right, the model might guess that enemies are always on the left with a 50% probability and constantly move left. To address this, entropy_regularization can be used. However, this model was not implemented in this case, as it may not always choose one action consistently. Additionally, reward shaping partially mitigates this issue. In the Deadly_corridor scenario, when reward shaping does not work, a small value like 0.01 can be assigned instead of 0.

In summary, without going too deep into the mathematical model, we can summarize the core principle of the PPO model as:

E(log(input) * optimize weight).

However, instead of focusing on the mathematical model, we should focus more on the inputs from the Stable_baseline3 library. This library provides the necessary parameters and building blocks for training and

optimizing the PPO model. These inputs make up the most important components regarding how the model operates and is trained.

model = PPO('CnnPolicy', env, tensorboard_log=LOG_DIR, verbose=1,
learning_rate=0.00001, n_steps=8192, clip_range=.1, gamma=.95, gae_lambda=.9)

The model uses CnnPolicy, which is based on Convolutional Neural Networks (CNN). In summary, the input data, which typically consists of image matrices, is simplified through convolution and pooling processes. In the final step, the data is transformed into one-dimensional input arrays using an activation function. Although there are other policies like MLP (Multilayer Perceptron), I decided to use CNN in both PPO and DQN model

Env is the Environment we created during the preprocessing process.

Tensorboard_log is a directory where we store the log records and graphs generated by TensorBoard. In the code, this directory is assigned to a variable called LOG_DIR.

Verbose, when set to 1, prints the numerical values of the log records shown for each epoch.

Learning rate, also known as the small learning value, helps reduce noise and ensures slower but more reliable learning. It plays a role in reaching the global optimum rather than getting stuck at the local optimum.

N_steps determines how many times the neural network loop will run during an epoch. When the learning rate is small, it is recommended to increase n_steps in multiples of 64.

C3 fx 2317.73965908924

	A	B	C	D	E	F	G	H	I	J	K	L
1	Trial	Score	N_steps	Gamma	Learning	Clip Range	GAE			Ran for 100k steps		
2	13	1983	7338.129721	0.9609188446	1.74E-05	0.1519145195	0.9014927448					
3	7	1715.992815	2317.739659	0.9559806628	2.38E-05	0.3938390658	0.9324839315					
4	22	1588	2077.923454	0.8567310643	1.27E-05	0.112031946	0.8054916811	Low n_steps,				
5	15	1484	2419.817619	0.8869301265	1.18E-05	0.1146372208	0.9305203738					
6	0	1300.034421	6080.901046	0.8985678895	2.95E-05	0.2153423364	0.9802054034					
7	21	1144	2109.766457	0.8656152936	1.26E-05	0.1050032449	0.8102508326					
8	26	1090.499976	4251.656062	0.8596195568	1.06E-05	0.1060692409	0.9887147433					
9	17	871	3105.547249	0.9367927056	1.97E-05	0.1508463399	0.9329797035		Notes			
10	10	476.4931641	3556.540545	0.9196811704	2.48E-05	0.3788951351	0.8832078638		High n_steps / low clip range			
11	20	476	4354.685383	0.9858412842	1.09E-05	0.3962483857	0.8765196884		Low n_steps / high clip range			
12	28	445.9870178	4135.970845	0.9981059765	9.03E-05	0.220636109	0.9850057584		Increasing gae_lambda reduces impact of this and smoothes			
13	11	445.9763489	4408.775531	0.931931913	4.62E-05	0.2787497169	0.8444721077					
14	8	435.9992294	5227.688267	0.9974732723	2.52E-05	0.3065264184	0.942830563					
15	18	428	3255.989451	0.9527220478	9.70E-05	0.3637975235	0.876730357					
16	29	384.4927063	4254.721967	0.8671992867	6.58E-05	0.2287512228	0.9873331316					
17	9	367.9967072	5707.562909	0.8983397317	1.59E-05	0.2643964763	0.8196138977					
18	35	355.9763489	3778.697975	0.8880937226	7.10E-05	0.2324109788	0.9839430632					
19	24	345.9989319	4034.663093	0.8545507377	8.97E-05	0.1007764099	0.8044994392					
20	36	305.9922424	3691.966036	0.9749863144	6.46E-05	0.2293185058	0.9623028925					
21	40	305.9920898	3651.105562	0.9080842663	6.43E-05	0.1864068885	0.9610397858	lr too high and high n_steps, high clip				
22	30	305.9914993	4135.314883	0.9938744676	6.28E-05	0.224175639	0.9856033658					
23	2	295.9915085	6090.065849	0.8200338633	4.56E-05	0.3223791141	0.8811567848	learning rate too high?				
24	6	247.4871765	3231.309387	0.8342323688	4.66E-05	0.3078926746	0.907669503	learning rate too high?				
25	32	225.9930222	3848.485012	0.854731949	6.49E-05	0.2258173753	0.9851209166	learning rate too high?				
26	34	212.4896408	3757.382335	0.8782143551	6.44E-05	0.2147468969	0.9827969177					
27	25	186.992662	4259.001854	0.8662720211	9.98E-05	0.1081790685	0.9849126941					
28	37	185.9756287	3572.89101	0.909821753	6.62E-05	0.2276774644	0.961778689					
29	4	165.9888367	7723.658995	0.8685778342	7.75E-05	0.2519001453	0.8532210334					
30	16	131	5004.632596	0.8037376124	1.08E-05	0.188493166	0.8253314752					
31	23	122	4249.288743	0.8521314294	8.91E-05	0.1380914967	0.8027320959					
32	27	95.99982605	4125.592909	0.8723476005	9.43E-05	0.1097433619	0.9820072663					
33	33	95.99403381	3676.770305	0.8664460522	6.85E-05	0.2208680791	0.9647696914					
34	1	75.97639465	2791.835911	0.8116466393	3.85E-05	0.1978220561	0.853059856					
35	19	65	7386.066016	0.9542026638	4.73E-05	0.292066171	0.8439208746					
36	14	-700	6205.861809	0.8343085622	1.54E-05	0.3953572734	0.9522813826					
37	5	-1019.224593	3102.66606	0.8125930738	6.68E-05	0.1783128436	0.813471332					
38	31	-1048.122929	4104.82548	0.8535002663	6.85E-05	0.2330965447	0.9667910654					
39	3	-1245.397221	4116.907281	0.8829709566	4.05E-05	0.1613871313	0.9698500734					
40	12	-1461	4534.335697	0.9566269153	8.59E-05	0.280853711	0.9381173352					
41												

Figure 7 : compare to paramaters

The reason why StackFrame is used in Mario but not in Doom is because Doom uses DummyVecEnv and VecTransposeImage. When StackFrame is used, these are not used because puppet vectors cannot be used in multivectors.

The studies conducted with PPO will be presented along with DQN in Chapter 4.

The second model used is the DQN [18] model. DQN, which stands for Deep Q Network, utilizes the Future Reward technique. Unlike PPO, which learns based on the present moment, DQN moves according to the best reward it receives based on past rewards. In other words, it trains itself using previous data and then moves using these learned experiences.

DQN operates on a similar principle to PPO but uses an Iterative Learning algorithm. Initially, it moves randomly to explore the environment, and once it has learned enough about the environment, it takes actions that aim to maximize the reward.

DQN Simple Formula = CurrentQ + learning_rate * (reward * discount_rate * maxQ value – currentQ) The discount rate is a number between 0 and 1, representing the trade-off between immediate rewards and future rewards. It determines how much future rewards are valued compared to immediate ones.

model = DQN('CnnPolicy', env, tensorboard_log=LOG_DIR, verbose=1, learning_rate=0.00001, batch_size=64, buffer_size=10000, learning_starts=5000, gamma=.95)

The variables that differ are buffer_size and learning_starts. Buffer_size represents the portion of memory allocated for action values, while learning_starts is the number of steps given to the model to explore the environment before starting the learning process. The Q-value can be considered as a controller in this context, and it can also be indicated as the output value. When using the DQN model, the StackFrame method used in Mario is not applied because DQN does not support multi-progress.

In the fourth section, we will compare the results and training graphs, starting with Mario followed by Doom.

4 Training and Compare to RL Algorithms

4.1 Super Mario Bros

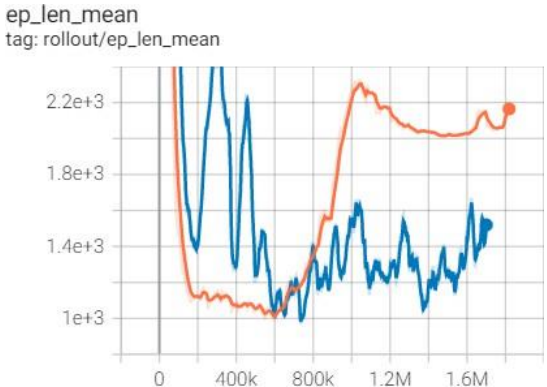
Firstly, in the Super Mario Bros game, approximately 2 million iterations were carried out for both the PPO and DQN models. As a result of PPO training, the model successfully completed the first level in the fastest possible time while also focusing on accumulating in-game points, but it was not able to pass the second level. The size of the trained data is approximately 280MB, and since we used a callback to save every 200,000 steps, the total generated data is around 3.5GB.

Using Evaluate Policy[19], we can find the average reward across 10 games. In the PPO model, this value was found to be 5542.2, while in the DQN model, the value was 2430.8.

The DQN model is slower compared to PPO. While the PPO model was trained in 50,000 seconds, the DQN model took around 60,000 seconds. The DQN model failed against the PPO model as it couldn't even reach the flag in the first level, and its mean_reward value was lower.

The size of the trained data is approximately 375MB. The total data is close to 4GB

The graph comparing the PPO and DQN models is shown below. This graph was obtained using TensorBoard.



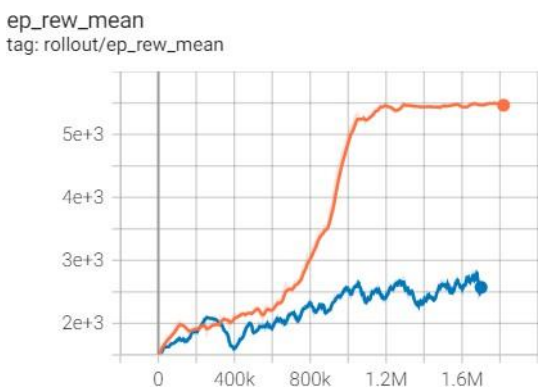


Figure 8 : Compare to DQN and PPO models

The visual on the left compares the episode lengths. The orange line represents PPO, while the blue line represents DQN. The PPO model shows more stable episode lengths, while DQN has fluctuating episode lengths due to its need for continuous feedback-based learning. In terms of rewards, PPO has a significant lead, accumulating more points overall.

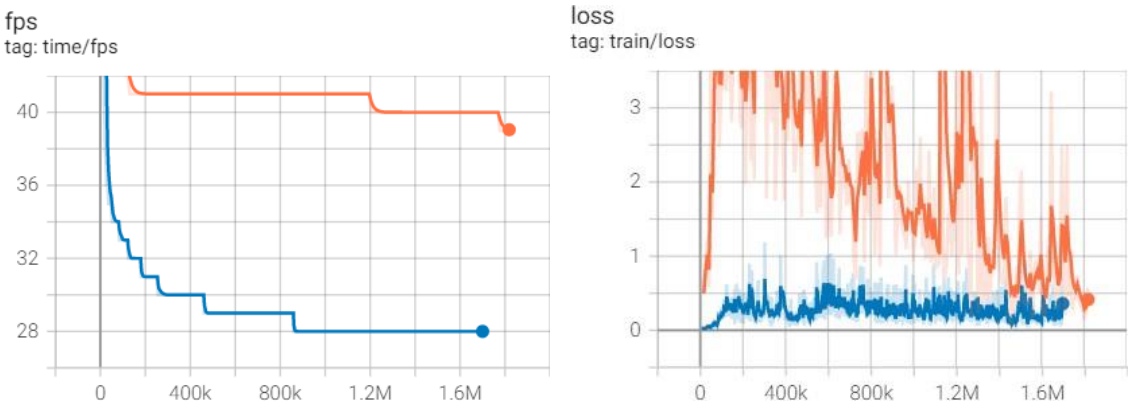


Figure 9 : FPS and Loss values on graphs

In terms of FPS, DQN shows a lower FPS, indicating it is behind in optimization compared to PPO. However, in the Loss value, DQN performs better, maintaining a more stable and lower level than PPO, making it more successful in this regard.

Since the remaining graphs are not available for the DQN model, they are not shown. Based on these results, the PPO model has demonstrated a better performance compared to the DQN model in the Super Mario Bros game.

4.2 DOOM 1993

In the Doom game, both the DQN and PPO models were compared. Since the Doom game consists of three different scenarios, as mentioned in the Preprocessing section, we will process the Basic and Defend Center scenarios together, and examine the Deadly Corridor scenario separately under a different heading.

4.2.1 DOOM_BASIC & DOOM_DEFEND_CENTER

For Doom Basic, both the PPO and DQN models underwent 70,000 iterations. The PPO model successfully learned the task, achieving an average reward score of 89.4 using the evaluate_policy function [19]. On the other hand, the DQN model performed less successfully, with an average reward score of -27.2. The reason for the low score is that the PPO model used one bullet and reached the target in the shortest time, while the DQN model used more bullets than necessary and did not reach the target as quickly.

The DQN model was insufficient with 70,000 iterations for this environment. Given that DQN relies on past data, it is estimated that with an increased number of iterations, such as 200,000, it could achieve similar or better performance compared to the PPO model.

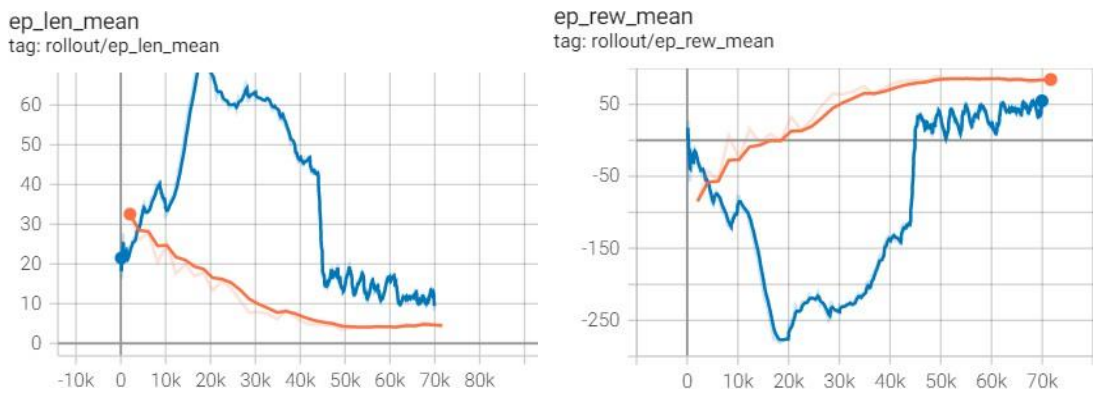


Figure 10 : Compore to DQN and PPO models

In the table shown in Figure 10, the orange color represents the PPO model, while the blue color represents the DQN model. Looking at the episode lengths, we can see that PPO has consistently shortened the episode length, while the DQN model has shortened it in a more inconsistent manner. The shortening of episode length over time indicates that the model is becoming more successful.

The reason for the DQN model's more erratic behavior is due to the learning_starts parameter and its use of past data for learning. As a result, the episodes are longer in the beginning, but towards the end, a sudden decrease in episode length is observed.

When looking at the reward table, we see that the PPO model has steadily increased its rewards, as shown in the first graph. On the other hand, DQN has also demonstrated inconsistency. The inverse relationship between the graphs is due to the nature of the environment. In other words, the shorter the episode, the more successful the model is.

For the Defend Center scenario, the PPO model underwent 100,000 iterations, with a log being recorded every 10,000 iterations. This resulted in a total of 560MB of data, with each log file being 56MB. Similarly, the DQN model also underwent 100,000 iterations, generating a total of 750MB of data from an individual log file of 75MB each.

Using the Evaluate Policy function, we can find the average reward across 10 games. In the PPO model, the average reward was 17.1, while in the DQN model, it was 10.3.

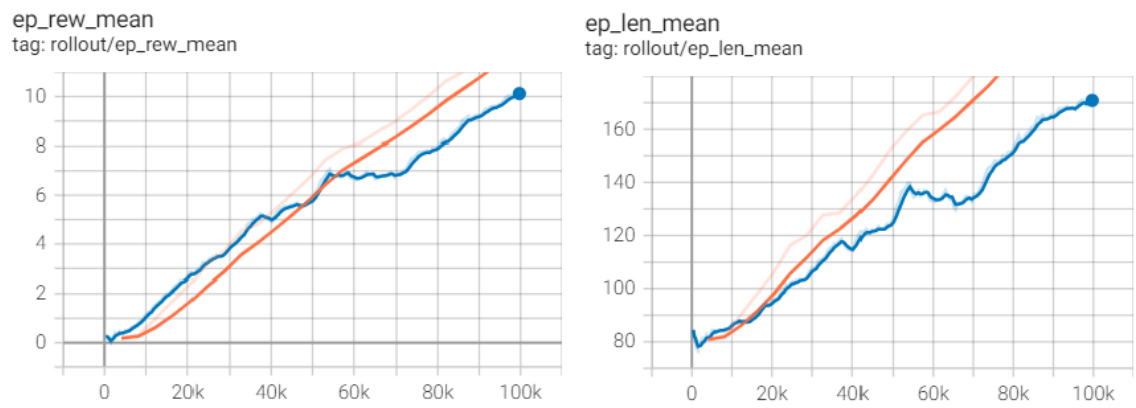


Figure 11 : Compore to DQN and PPO models

As shown in Figure 11, the PPO model (orange) not only survived longer but also dealt more damage to enemies, as observed in the reward graph.

The reason for the PPO model surviving longer is its efficient use of ammunition, similar to the Basic level. In contrast, the DQN model wastes more ammunition unnecessarily.

The results from the Basic and Defend Center levels show that the PPO model performs better. We observed that the PPO model handles the target-reward relationship more effectively than the DQN model.

We observed that the DQN model requires more iterations and is more wasteful in terms of resource consumption by the agent.

4.2.2 DOOM_DEADLY_CORRIDOR

The Doom Corridor section is more complex compared to other Doom scenarios, so I decided it should be analyzed under a separate heading.

In the Corridor section, both PPO and DQN models underwent 600,000 iterations, with 400,000 + 50,000 * 4 repetitions. The reason for this approach is the use of the Curriculum Learning technique[14]. In brief, this technique involves completing 400,000 iterations at the first difficulty level, and then gradually increasing the difficulty level every 50,000 iterations, enabling the agent to fight against more challenging enemies.

In the Corridor section, the PPO model generated 56MB of data per iteration, totaling 672MB of data, while the DQN model generated 72MB of data per iteration, resulting in a total of 900MB of data.

In terms of mean_reward scores, the PPO model collected an average of 934.27 points, while the DQN model scored 770.20 points, falling behind the PPO model.

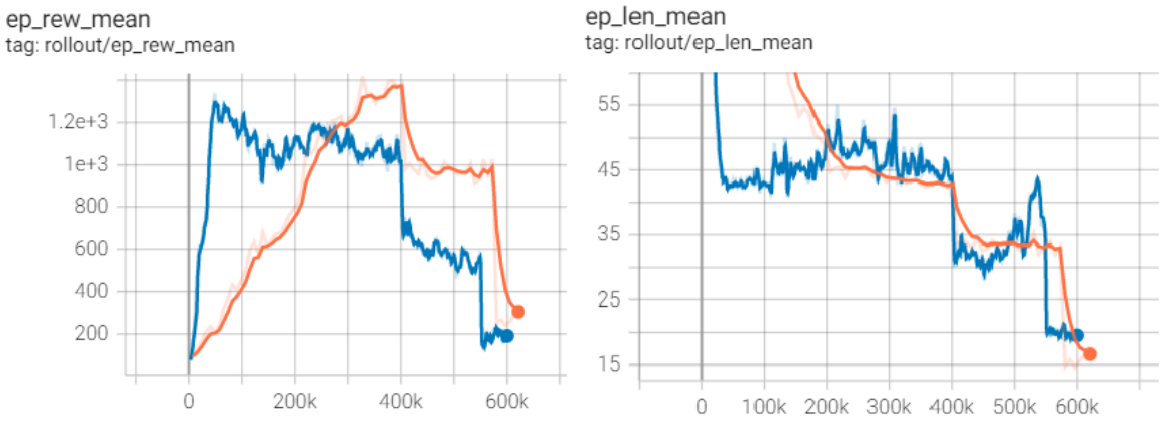


Figure 11 : Compore to DQN and PPO models

Looking at the graphs, the PPO model, represented in orange, progressed more steadily compared to the DQN model, and with a slight margin, it performed better than the DQN model..

In the tests, the PPO model, unlike the DQN model, preferred to use the weapons of the enemies it killed. This action indirectly increased the number of bullets, which positively impacted the reward. This shows that the PPO model applied reward shaping more effectively and made better decisions compared to the DQN model.

As a result, in the Doom section, the PPO model outperformed the DQN model.

5 Conclusion

With this paper, the performances of the PPO and DQN models used in Reinforcement Learning have been compared, with performance measured across a total of 2 games and 4 different maps. As a result, we compared the PPO and DQN models using CNN policy in both Super Mario Bros and Doom games. In this comparison, the PPO model outperformed the DQN model in terms of the number of iterations, performance, average reward score, and ability to reach the target.

6 References

1. https://towardsdatascience.com/self-improving-chatbots-based-on-reinforcement-learning-75cca62debce
2. https://www.researchgate.net/publication/323178749_A_Concise_Introduction_to_Reinforcement_Learning
3. https://www.gymlibrary.dev
4. https://github.com/Kautenja/gym-super-mario-bros
5. https://github.com/Farama-Foundation/ViZDoom
6. https://github.com/Kautenja/nes-py
7. https://www.cs.toronto.edu/~toryn/docs/AlizadehAlamdari2022considerate.pdf
8. https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html
9. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d893434d78a81f7c7a7c9aee44894b53c7b1359d
10. https://gibberblot.github.io/rl-notes/single-agent/reward-shaping.html
11. https://stable-baselines.readthedocs.io/en/master/guide/vec_envs.html
12. https://www.gymlibrary.dev/content/environment_creation/
13. https://medium.com/swlh/states-observation-and-action-spaces-in-reinforcement-learning-569a30a8d2a1
14. https://ronan.collobert.com/pub/2009_curriculum_icml.pdf
15. https://stable-baselines.readthedocs.io/en/master/guide/callbacks.html
16. https://stable-baselines.readthedocs.io/en/master/index.html
17. https://arxiv.org/abs/1707.06347
18. https://arxiv.org/abs/1312.5602
19. https://stable-baselines3.readthedocs.io/en/master/common/evaluation.html