Reinforcement Learning – Homework #2: Gym-Retro Experiment

Indra Imanuel / 20215136

1. Introduction

This report describes the training of a deep reinforcement learning agent to play a retro game. The retro game in particular is "Sonic the Hedgehog 2", released back in SEGA Genesis on 1992. Training and testing of the agent will be done on the first level of the game, "Emerald Hill Zone Act 1". The reinforcement learning algorithm used is Deep Q-Networks (DQN), which combines Q-learning and Neural Network, which will act as the function approximator. The game environment is made using Open AI Gym Retro. The content of this report can be summarized as follows:

- Explanation about Sonic the Hedgehog 2 video game.
- The environment description, along with some explanation about the agent's action and reward system in Sonic the Hedgehog 2.
- Explanation about the how the deep reinforcement learning algorithm, DQN, works and used to train the agent to play Sonic the Hedgehog 2.
- The details on how the experiments is run. This includes how the DQN is trained; the networks' input, output, architecture; hyperparameter details; and explanation on how the input is preprocessed.
- Discussion and analysis of the experiment results.
- Conclusion and takeaways from the experiment.

An accompanying 5-minute video is also provided along with this report to give a more visual explanation about the experiment. The video can be seen in this link: https://youtu.be/ZgS3YwhV_5w . The code and steps on how to run the code can be downloaded and seen in this GitHub repository: https://github.com/Indraa145/Sonic2_DQN

2. Sonic the Hedgehog 2 Video Game

Sonic the Hedgehog 2 was released for SEGA Genesis on 1992. The game features Sonic the Hedgehog, a blue hedgehog which the player controls; and his orange fox side-kick, Tails Miles Prower, which follows Sonic and controlled by a bot or a second player. The game is divided into levels, referred to as "Zone", where each Zone is divided into multiple stages referred to as "Act". The goal of the game is to get to the end of the stage, which is located on the right-most of the stage. The game is structured for the player to start on the left-most of the stage, and has to move to the right to reach the finish on the right-most of the stage. There are enemies and stage hazards between the start and finish of the stage.

Sonic the Hedgehog 2 is a fast, momentum-based platforming game, meaning Sonic's movement is fast and many of the level design tropes make use of Sonic's speed to build up momentum. For example, there are a lot of down and up slope in the level, in which when Sonic runs down the slope, he'll naturally gets faster; whereas to run up the slope, Sonic needs to build up speed first. With

enough speed while running, Sonic can turn into a ball, which will help him build speed faster when going down a decline or down slope, similar to a wheel getting faster when going downwards. Sonic can also do a move called “spin-dash”, where he would stop in place and charges his spin. When the charge button is released, Sonic will go into a direction he’s facing in high speed in his ball form. While in ball form, Sonic is also immune to the enemy and can destroy them. Sonic also jumps in ball form, meaning any enemy came in contact with him while jumping will get destroyed. For a more visual explanation, please refer to the accompanying video.

3. Task Definition

In reinforcement learning, the agent interacts with the environment by taking an action based on the current state the agent in, and then transition to a new state and getting a reward from the environment. The state in our case is screenshot-like or frame of the game in each timestep. Although, each state is made out of n stack of consecutive game’s frame, rather than just a single frame. This stack of frames will be the input of the DQN neural network. We’ll talk about the reason for this and explore it more in detail in section 4, where we’ll talk about the DQN neural network and the input data pre-processing.

Figure 1. An example of state, action sequence when playing Sonic the Hedgehog 2

As for the agent, it will choose an action from the list of possible action in each state the agent currently in (each timestep). By default, because Sonic the Hedgehog 2 is a SEGA Genesis game, Open AI Gym Retro provides 12 possible actions to choose from, each representing the buttons on SEGA Genesis controller, which are: B, A, C, Y, X, Z, MODE, START, UP, DOWN, LEFT, RIGHT. The number of possible actions is modified to fit our game better, because some buttons result in the same action for our game. The actions are redefined to 8 possible actions for our case, which are:

- **NO OPERATION,** Sonic stands still.
- **LEFT**, Sonic moves to the left.
- **RIGHT,** Sonic moves to the right.
- **LEFT + DOWN**, Sonic moves to the left and rolls into a ball.
- **RIGHT + DOWN**, Sonic moves to the right and rolls into a ball.
- **DOWN**, Sonic ducks or rolls into a ball.
- **DOWN + B**, Sonic charges his spin-dash, when these buttons are released, Sonic will launch to the direction he’s facing in ball form.
- **B**, Sonic jumps.

The reward, also referred to as "score" for the rest of this report, is the horizontal offset from the player's initial position. This translates to as the further the agent goes to the right, the higher the reward score is. This makes sense, considering the end of the stage is located on the right-most of the stage. Reaching the goal will give the agent rewards of 9000, and a completion bonus up to 1000 depending on how fast the agent finishes the stage. This means, when the reward score is more than or equal to 9000, the agent has successfully finished the stage.

4. Approach

In this section, we'll see how the deep reinforcement learning algorithm used in this project, DQN, works along with its properties.

4.1 Q-Learning

Q-Learning is the basis of DQN, as DQN is basically a Q-Learning that uses neural network as its function approximator. Q-Learning aims to find the optimal policy, π_*, by learning the optimal Q-values, q_*, for each state-action pair (s, a). The agent will choose the action with the highest Q-value for its current state. The Bellman optimality equation for q_* given state s and action a is defined in Equation 1, where R_{t+1} is reward in the next timestep, and γ is the discount factor.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] \tag{1}$$

The Bellman equation above is used to iteratively updates the Q-values for each state-action pair, until the Q-values converges to q_*. Q-values for each state and its corresponding possible action is stored in something called Q-table. Initially, all the Q-values inside this table is set to 0, but overtime it'll be updated as the agent takes action and moves in the state space over several episodes of the game. The update is done to minimize the loss/error, which is the difference between q_* and q given the same state-action pair, defined in Equation 2 and 3.

$$loss = q_*(s,a) - q(s,a) \tag{2}$$

$$loss = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{3}$$

To update the Q-value, a learning rate, α, is needed. α is a value between 0 and 1 and determines how fast the agent adopts the new learned Q-value and discard the old Q-value in the Q-table. The higher the learning rate, α, the faster the agent adopts the new learned Q-value. Formally, it can be said that the updated Q-value is a weighted sum of the old Q-value and the learned Q-value, and is defined in Equation 4.

$$q_{new}(s,a) = (1-\alpha) \cdot q_{old}(s,a) + \alpha \cdot \left(R_{t+1} + \gamma \max_{a'} q(s',a')\right) \tag{4}$$

As mentioned before, this update will be done iteratively until the Q-values converges to q_*. Q-learning works relatively well in the case with small state-action space. However, for real game application, where the state-action space is very large, this method is not suitable. This is because, in conventional Q-learning, all of the Q-values for each state, action pair have to be stored in the Q-table.

In the case of very large state-action space, using the Q-table is not preferable, as it will take so much memory. This is where DQN comes into play, which will be explained in the next section.

4.2 Deep Q-Networks (DQN)

As explained before, DQN can solve the problem of very large state-action space in regular Q-learning. Instead of taking the Q-value from Q-table, DQN calculates the Q-value using neural network, with the corresponding state as the input, as shown in Figure 2 The agent will then choose to take action with the highest Q-value.

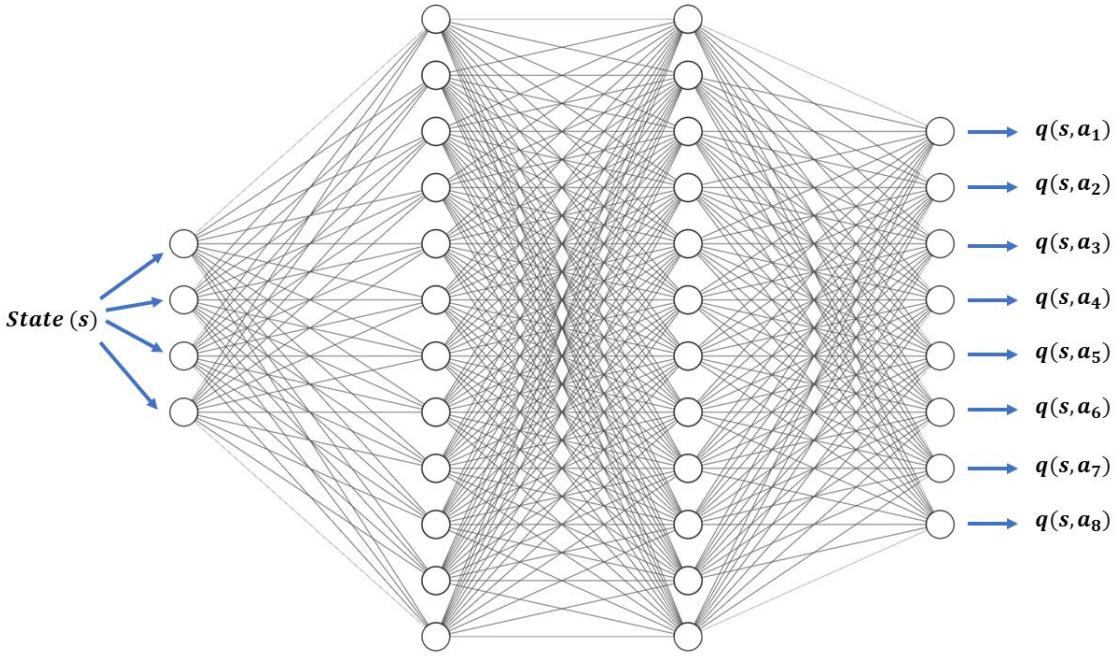

Figure 2. DQN takes state s as input, extracts the features and outputs a list of possible Q-value for that state.

To train a neural network, a loss function and dataset is needed. As explained in Equation 2, the loss is difference between the optimal Q-value, q_*, and estimated Q-value, q. In our case, we use mean-square error (MSE) between them to calculate the loss. We can get the estimated Q-value by inputting the current state to the neural network, and as shown in Figure 3, the neural network will output all possible Q-value in accordance with number of possible actions. This neural network is referred to as the policy network.

Every time the agent takes action and steps into a new state, the agent will save the "experience" in a Replay Memory. The replay memory will have limited size and used to store N size most recent experience. The policy network will sample batch of experience from this replay memory every N timestep and be trained on it, basically making these experiences as the dataset. We'll see more about Replay memory in Section 4.3.

Other than the policy network, in DQN there's also something called the target network. This target network will help to calculate the optimal Q-value, q_*. Recall the q_* formula in Equation 1, we can see that it needs to calculate the maximum optimal, or in this case estimated optimal, Q-value given the

next state and action, $\max_{a'} q_*(s', a')$. In the case of regular Q-Learning, this can simply be done by taking a look in the Q-table. Because we don't have a Q-table in DQN, we need another neural network, the target network, to calculate this. The target network will take the next state, s', as the input and outputs all the possible $q(s', a')$ in accordance with number of possible actions in state s'.

The target network has the same architecture as the policy network, and both are initialized with the same weights. The target network weight is updated from the policy network weight, using a soft update method. This update formula is defined in Equation 5, where τ is a hyperparameter, θ' is the target network's weight, and θ is the policy network's weight.

$$\theta' = \theta \cdot \tau + \theta' \cdot (1 - \tau) \tag{5}$$

4.3 Replay Memory

Replay memory, as mentioned in Section 4.2, is what used to store experiences. Experience is a tuple made out of current state, action, reward at the next timestep, and state at the next timestep: $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Replay memory has buffer size, which defines the limit of how many experiences can be stored. Storing all experience is not preferable because it'll quickly fill up the memory. In our case, the buffer size is set to 100,000, so only 100,000 most recent experiences are stored.

The policy network will sample a random batch of experiences from the replay memory every N timestep and use it as dataset to train itself. The purpose of replay memory is to let the policy network learn from experiences that are not just consecutive to the agent's current state. Without replay memory, the policy network would only learn from states that are next to each other of the current state, and will learn it in sequence. Learning only from consecutive states can be bad, as those states can be highly correlated and would lead to inefficient training. That's why replay memory and random sampling from it is used, by doing so it would break the aforementioned correlation problem.

4.4 Exploration Strategy

As with other Reinforcement Learning Algorithms, an exploration strategy is also used in this project to balance out the exploitation and exploration. The strategy that is used in this project is the epsilon greedy strategy. In epsilon greedy strategy, we have exploration rate, ϵ, which will determine whether the agent do exploration or exploitation. Every time the agent wants to take an action, it'll generate a random number between 0 and 1, and compare it with the ϵ. If the random number is greater than the ϵ, the agent will do exploitation for its action (getting Q-value from the policy network). Otherwise, if the random number is smaller than ϵ, the agent will do exploration for its action, which means the agent will do an action randomly from the list of possible actions. This strategy is called the Epsilon-Greedy Strategy.

The value of ϵ starts at 0.99 and will slowly decay for each episode to 0.01 at the lowest. The ϵ decays following formula defined in Equation 6.

$$\epsilon = \epsilon_{start} + (\epsilon_{start} - \epsilon_{end}) \times e^{-\left(\frac{episode\ i^{th}}{decay\ rate}\right)} \tag{6}$$

The parameters $\epsilon_{start}, \epsilon_{end}, decay\ rate$ are hyperparameters in which the value can be modified. We'll see the effect of changing these hyperparameters to different value in Section 6.2.

5. Experiment Details

In this section, explanation about the experiment settings & network architecture, and input data preprocessing will be provided.

5.1 Experiment Settings & Network Architecture

The experiment is run on a computer with the following specs: Intel Core i7-9700K CPU, 16 GB RAM, and NVIDIA GeForce RTX 2080 Ti GPU. As mentioned in the beginning, OpenAI Gym-Retro is used as the environment manager. The neural network is implemented using PyTorch and optimized using Adam optimizer, where the architecture detail can be seen in Figure 3.

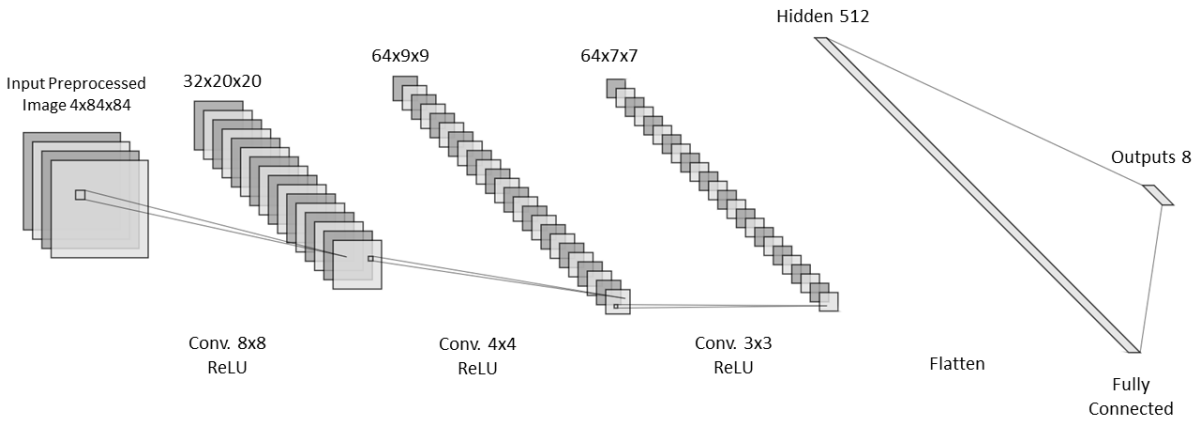

Figure 3. Network architecture of the Policy and Target network. They're made out of 3 convolutional layer, each followed by ReLU activation function. After the convolution, the features are flattened and inputted to fully connected layers. The number of output is equal to the number of possible actions of the agent

The experiment is run for 1000 episodes, with each episode has a maximum of 10,000 timesteps (roughly around 2 minutes 47 seconds in gametime). If the timesteps goes over 10,000, the episode will end automatically. This is done to prevent the agent going in circle without making any progress for too long. The episode will also end when Sonic dies. The hyperparameters value for the experiment can be seen in Table 1. Results will be discussed in further details in Section 6.1.

Table 1. Hyperparameters of the experiment.

Hyperparameter	**Value**
Discount Factor (γ)	0.99
Replay Memory Size	100,000
Batch size for experience sampling	32
Learning Rate (α)	0.0001
Target network soft update parameters (τ)	0.001
Timestep interval to update the neural network	100

Timestep when the network starts learning from replay memory	10,000
Exploration rate start (ϵ_{start})	1
Exploration rate end/minimum (ϵ_{end})	0.01
Exploration rate decay rate	$\frac{frame\ index}{100}$

5.2 Input Preprocessing

The input data, state, which are represented by stack of screenshot-like frames are preprocessed first before getting feed into the network. Preprocessing is done to reduce complexity of data, as some information in the data can be removed and simplify to lessen the computation burden and fasten the training process. The preprocessing includes:

- Converting the frames to grayscale, as the color channels doesn't really matter for our case.
- Resizing the frames to 84x84.
- Normalizing the pixel values to a range of 0 to 1.
- Stacking 4 consecutive frames together, to give more sense and context to the network as explained in Section 4.2.

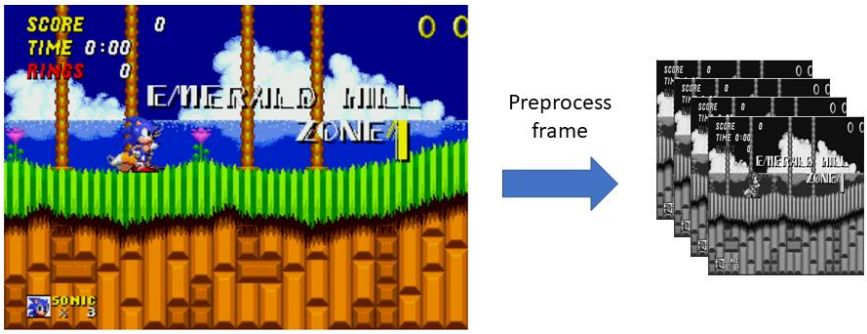

Figure 4. The frame is preprocessed before being fed into the network.

6. Results and Analysis

In this section, experiment results and analysis of it will be provided. We'll also see how different reward function during training affects the agent's performance. As for testing, the agent will be tested on the level 100 times, and calculated how many times does the agent succeed in clearing the level (in percentage). The same ϵ will be used for all of the experiment testing, which is set to 0.01. The agent training process and result can be seen in the accompanying video of this report: https://youtu.be/ZgS3YwhV_5w

6.1 Base Results

The base experiment uses the hyperparameter said in Table 1, and a reward function that was explained in Section 2. The base reward function is based on Sonic's horizontal position. The further to the right Sonic is (closer to the goal), the higher the reward is. The agent will get a total of 9000 reward for reaching the goal, with a speed bonus up to 1000 for reaching the goal faster. The agent

will be punished if its only standing in one place without moving for a long period of time. The reward graph over 1000 episodes of training can be seen in Figure 5.

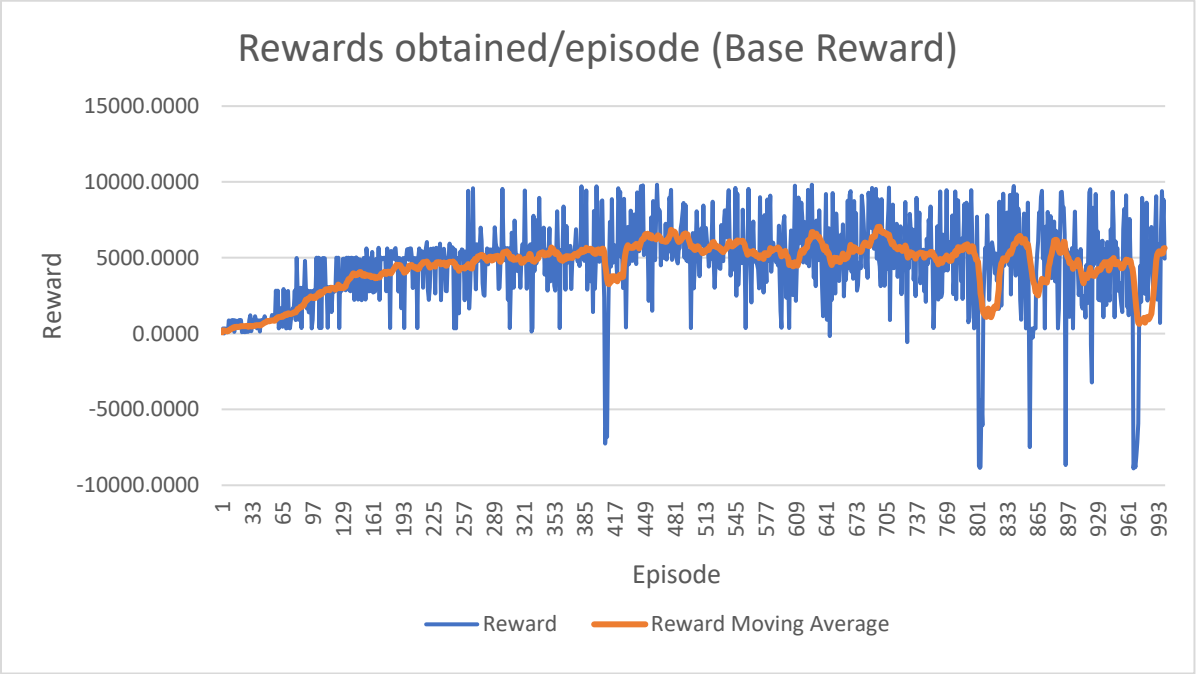

Figure 5. Rewards obtained by the agent during training in the span of 1000 episodes. The moving average is calculated in the range of 20 episodes. The agent uses the base reward function definition here.

We can see from Figure 5, the reward obtained by the agent slowly increases as the episode goes on. This indicates that the agent is learning, by improving its ability in choosing the actions that can lead to high reward. As explained before, the maximum reward the agent could get is 10,000; with over 9000 reward indicating the agent has successfully reached the goal. From Figure 4, it's shown that the agent successfully reach the goal for the first time in around the 250th episode, and was able to continue doing so in the following episodes. Albeit, the reward is going up and down, so the agent doesn't always succeed in reaching the goal. But still the agent is progressively getting more consistent in reaching the goal, as can be seen with the increase in the reward moving average.

The big decrease in the reward graph indicates that the agent is getting punished a lot. As explained before, the punishment happens when the agent stays still for a prolonged period of time. This means that during some of those episodes, the agent was stuck and couldn't progress.

It's also found from the experiment that training for 1000 episode is somehow too much for the agent. The agent becomes confused and "forgets" some of the important action it's taken in the previous episodes. This is also evident from Figure 5, where prolonged training makes the graph more erratic in going up and down. In my opinion, training for around 500 episodes is enough, as can be seen in the graph that not much improvement is also made after 500 episodes. The test result of this agent can be seen in Table 2. For a more visual explanation, please refer to the accompanying video.

6.2 Results with Different Reward Function

One problem that the agent continuously face is getting stuck in the incline road and loop de loop part of the game. The agent will continuously try to move forward by holding the right button, but because the road is incline, the agent lacks the momentum to go past it. The correct way to go through past it is either going to the left first and then go right to build up speed, or use the spin dash move. Sometimes there's also a spring on the left that can launch the agent to the right with enough momentum.

Figure 6. The agent often getting stuck in the loop de loop section of the level, as it keeps trying to move to the right, despite lacking the momentum to get through it.

Because of that, the base reward function is modified, by adding punishment when the agent is stuck. This works by decreasing the reward every time the agent stays in a certain range of x position for a prolonged period of time. This is different with just staying stills, as if the agent moves, but if that movement didn't bring the agent forward or backward in the x position, the agent will get punished. This is done by storing a moving average of the agent's x position in every timestep and comparing the previous moving average timestep with the current moving average timestep. If the value between them don't differ much, then it means the agent's x position doesn't change much. If that happens for more or equal than 100 timesteps, then the agent gets punished. The result of this experiment can be seen in Figure 7.

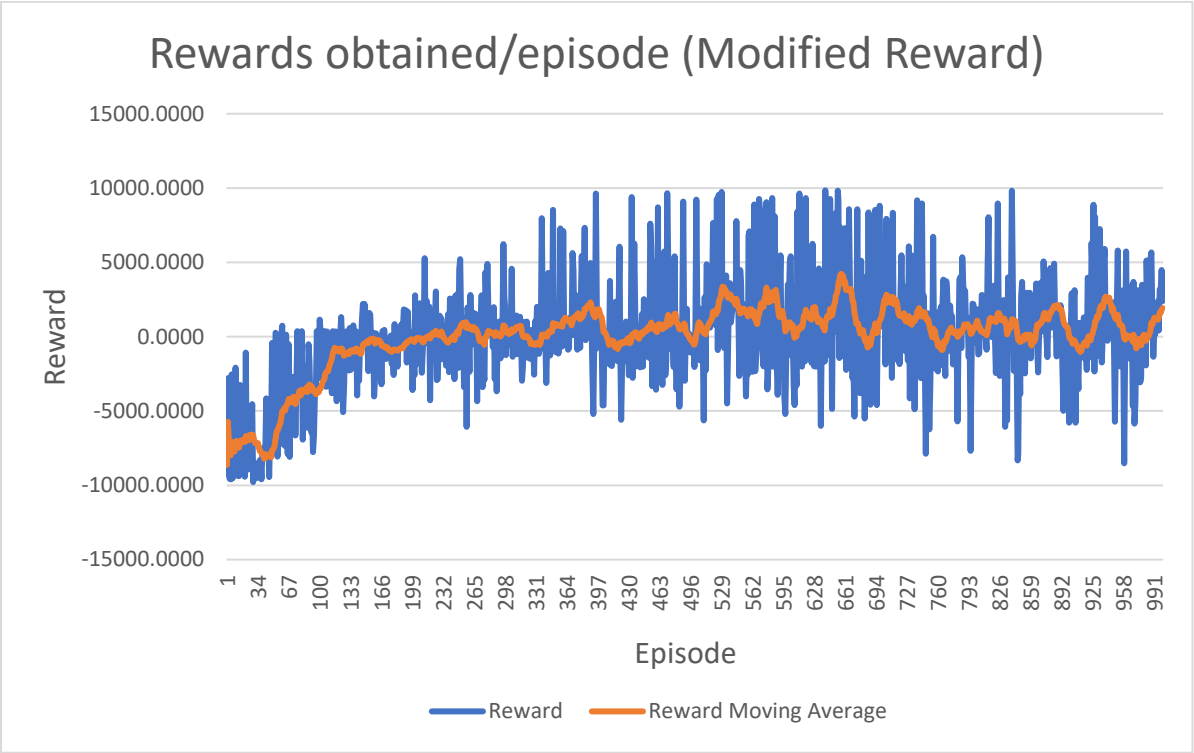

Figure 7. Rewards obtained by the agent during training in the span of 1000 episodes. The moving average is calculated in the range of 20 episodes. The agent uses the modified reward function definition here.

Compared to Figure 5, we can see that Figure 7 has more erratic change of reward in each episode. This is caused by the aforementioned punishment system, which indicates that the agent often gets stuck. We can see that the episode starts with a negative reward value, meaning in the beginning the agent wasn't really moving much in the x axis, which leads to big punishment value. But as the episode goes on, we can see that the agent gets better and get less stuck, although it still gets stuck time to time. But according to test result shown in Table 2, this agent was able to complete the level more times than the other agent, meaning this agent is able to get out of the stuck more often. For a more visual explanation, please refer to the accompanying video. It's also found from the testing that prolonged training is not good for this agent as well, around 500 episodes are enough for the training.

Table 2. Success Rate comparison of the two methods. Success Rate indicates how many times does the agent succeed in finishing the level out of 100 times playing the level.

Method	**Success Rate**
Using the base reward function	24%
Using the modified reward function	31%

7. Conclusion

Based on the experiments above, we can see that the deep reinforcement learning algorithm, DQN, can be used to learn and play complex high-dimensional-state video game such as Sonic the Hedgehog 2, to some extent. The DQN agent can learn to finish the first level of Sonic the Hedgehog 2 by learning the features of the input state, a stack of screenshot-like frame, using the policy neural network, which outputs Q-value of all possible actions in that state. The maximum Q-value will be chosen and translates to the action Sonic will take in that state.

Experiment results shows that the agent was able to finish the level 31% of the time. This success rate could perhaps be improved by using a more complex network architecture and reward function definition. Future work will try to work on this and improve the success rate.

Other interesting thing to consider for future reference is trying to generalize the agent to the different levels of the game. A little test was done to the current agent and it was able to finish the stage “Emerald Hill Zone Act 2”, even though it’s not trained on it, albeit with a low success rate of 12%, This might due to both levels having similar level design and environment, because the agent couldn’t really finish the other levels, as those levels have more complex platforming elements and enemies/obstacles.