Reinforcement Learning – Homework #4: Gym-Retro Experiment

Leonard Christopher Limanjaya – 20225087

1. Introduction

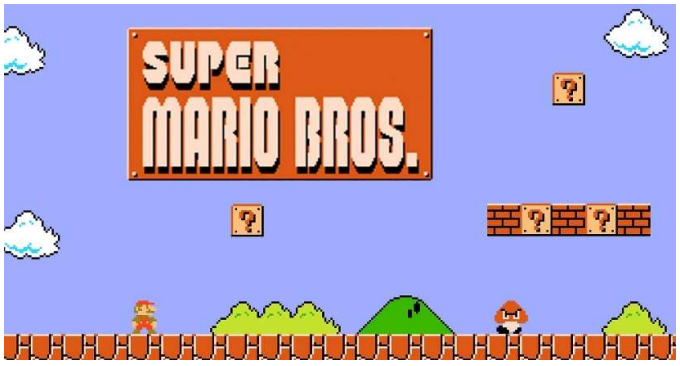
In this report, we will be exploring training a deep reinforcement learning agent through the lens of the gym-retro experiment. Specially, we will be focusing on one of the most iconic video games of all time: Super Mario Bros on NES. With the help of advanced machine learning algorithms, we will be analyzing different techniques and strategies for playing this classic game, and evaluating their effectiveness in improving gameplay and achieving higher scores or completing the game. The reinforcement learning algorithm used in this experiment is Deep Q-Networks (DQN) and Advantageous Actor-Critic (A2C) with Proximal Policy Optimization (PPO).

We will start by providing an overview of the Gym Retro Experiment and how it works, including the tools and resources used for this project. We will then delve into the specifics of Super Mario Bros, discussing its history and gameplay mechanics. Next, we will explore different strategies for playing these games using DQN and A2C algorithms, comparing the effectiveness of various approaches and evaluating their impact on gameplay. We will analyze the results and discuss the strengths and limitations of DQN and A2C algorithms in this context.

Finally, we will present our findings and conclusions, highlighting the potential of the Gym Retro Experiment, DQN algorithm, and A2C algorithm for future research and development in the field of retro gaming and machine learning. We hope that this report will inspire new ideas and approaches for further exploration of this exciting field and provide valuable insights.

2. Environment

a. Super Mario Bros



Super Mario Bros is the classic side platform video game developed and published by Nintendo for the Nintendo Entertainment System (NES). It was first published in Japan in 1985 and become one of the most popular video games of all tie. The mission of the game is to guide Mario through the Mushroom Kingdom to rescue Princess Toadstool from the evil Bowser and his minions. Mario must navigate through eight worlds, each with four levels, and defeat Bowser's minions along the way. The ultimate goal is to reach Bowser's castle and defeat him in the final boss battle, rescuing Princess Toadstool and restoring peace to the Mushroom Kingdom. Along the way, players can collect power-ups, coins, and extra lives to help them overcome the game's challenges and progress through the levels.

For this experiment, we will make the reinforcement learning agent for the very first chapter of the game. The first chapter of Super Mario Bros is World 1-1, and its mission is to introduce players to the basic mechanics and gameplay of the game. The levels are relatively simple, with few enemies and obstacles to overcome. The mission of World 1-1 is to guide Mario through the level, collect coins, and reach the end goal – a flagpole that marks the end of the level. Along the way, players can reveal hidden secrets and power-ups that will help them in later levels. Completing World 1-1 sets the state for the rest of the game and introduces players to the challenges and thrills that await them in the rest of the Mushroom Kingdom.

b. Felix The Cat

Felix the Cat is a 2D side-scrolling platformer game released for the Nintendo Entertainment System (NES) in 1992. The game is based on the classic cartoon character of the same name and was developed and published by Hudson Soft. In the game, players control Felix the Cat as he embarks on a mission to rescue his girlfriend, Kitty, who has been kidnapped by his arch-nemesis, The Professor. The game consists of six worlds, each with three levels, and a boss battle at the end of each world.

Felix can jump and attack enemies with various weapons, including a magic bag that can transform into different items, such as a boxing glove, a car, or a hot air balloon. Felix can also collect power-ups throughout the game that give him extra lives, restore health, or grant invincibility. The game's levels are designed to be nonlinear, allowing players to explore and discover hidden areas and secrets. Each level is full of enemies and obstacles, including moving platforms, traps, and bottomless pits, that must be navigated to progress through the game.

3. Task Definition

The objective of the Mario Game is to guide Mario through a level and reach the end flagpole while avoiding enemies and obstacles. During training, the agent can receive observations of the game state base on the image. Based on these observations, the agent selects an action such as moving left, right, jumping, or attacking enemies. The agent receives a reward based on their performance, with a positive reward for completing some objectives and negative rewards given for not moving or losing a life. The agent will be trained using a DQN and DDQN. We will go through the algorithm in detail in section 4.

To take action, the agent can choose the action from the list of possible actions in each state. By default, the NES game in Open AI GYM Retro provides 9 possible actions to choose from, each representing the buttons on the NES controller, which are: B, A 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. These actions are:

- No actions
- Right
- Right + Jump
- Jump

When an agent interacts with the environment, it will get a reward depending on how the agent act. The reward can be depending on the x-axis of the agent. The more agent goes to the right, the higher the reward is. This reward function will motivate the agent to get the most right of the map, where the goal is.

The objective of the Felix The Cat game is similar to Super Mario Bros. The RL agent would be trained to guide Felix through a level while avoiding obstacles, defeating enemies, and collecting power-ups. The RL agent's objective is to learn a policy that maximizes the expected reward over time, similar to Super Mario Bros. This would involve learning to make decisions that lead to the highest possible reward in the long term.

To achieve the objective, the agent would receive observations of the game state from the image. Based on the observation the agent would select an action to take, such as jumping, attacking enemies, or using a special ability. To take action, the agent can choose the action from the list of possible actions in each state. By default, the NES game in Open AI GYM Retro provides 9 possible actions to choose from, each representing the buttons on the NES controller, which are: B, A 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. These actions are:

- No actions
- Left
- Right
- Punch
- Left + Punch
- Right + Punch
- Left + Jump
- Right + Jump
- Jump

When an agent interacts with the environment, it will get a reward depending on how the agent act. The reward depends on the score of the game. The agent can gain a score, by collecting points or killing the enemies.

4. Approach

In this section, we will see how the deep reinforcement learning used in this project works.

a. Q-Learning

Q0keabrubg is a value-based reinforcement learning algorithm used to find the optimal action-selection policy by using a Q-function. It is a value-based approach based on the Q-Table. The Q-table calculates the maximum expected future reward, for each action at each state which is called Q-values. With this information, we can then choose the action with the highest reward. To get the optimal Q-table first is to initialize the Q-table (all zeros), then choose an action using the epsilon-greedy exploration strategy, and after that update, the Q-table using the Bellman equation. The agent will keep acting and update the Q-table until the learning no longer improves. The Bellman equation to update the Q-table is as follows:

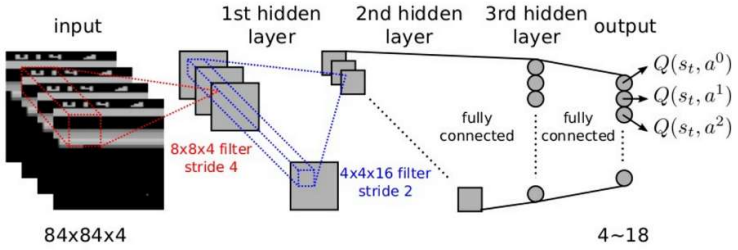
$$Q(S_t, A_t) = (1 - \alpha)\, Q(S_t, A_t) + \alpha * (R_t + \lambda * max_a\, Q(S_{t+1}, a))$$

The Bellman equation is used to update the Q-values for each state-action pair. The agent will update the current value with the estimated optimal future reward which assumes that the agent takes the best current known action. The agent will search through the actions for a particular state and choose the pair with the highest Q-value. To update the Q-value, it needs a learning rate (a value between 0 and 1) where the learning rate determines how fast will the agent adapt the newly learned Q-value. The higher the learning rate, the faster the agent adopts.

This algorithm can work well on not complex games with small state-action space. For a complex game with a large state action, this method might not be suitable. It can take a lot of time and computational cost. One of the reasons can be in conventional Q-learning, all of the Q-values for each state, and action pair have to be stored in Q-table. To overcome this problem there is an algorithm called Deep Q-Network (DQN).

b. Deep Q-Network (DQN)

Deep Q-Network uses the Q-Learning idea, and DQN uses a neural network that takes a state and approximates the Q-values for each action based on that state. They use neural networks because using a basic Q-table is not scalable, in a more complex game, Q-table will get too complex and cannot be solved efficiently. The learning process is still the same with the iterative update approach, but instead of updating the Q-table, we update the weights in a neural network.



Source: [Vanilla Deep Q Networks. Deep Q Learning Explained | by Chris Yoon | Towards Data Science]

The Super Mario Bros game and Felix The Cat game can be considered complex games because we give the input as images of the video game. It is almost impossible to use basic Q-Learning to solve this problem. In this deep Q-Network, we use CNN to extract the important features of the input. As the output will be Q-value for each possible action. The size of the input will be depending on the dimension of the image and the output will depend on the number of actions. As input, we will give the 4 frames of a gray image which each is scaled down to 84x84. We will feed them to the input of a convolutional neural network which outputs a vector containing the Q-value of each possible action.

In DQN we used experience replay where these techniques can stabilize the training of the agent by allowing the agent to revisit and learn from experience. To gather the experience data, at each time step the agent state, action, reward, and next state will be stored in replay memory. During the training, the agent will randomly sample a batch of experiences from

the replay memory to update the network. The experience will be stored in memory replay, where the memory has limited size and is used to store N size most recent experience.

$$L_i(\theta_i) = \underbrace{\mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}}_{\text{Samples from experience replay buffer}} \left[\left(\underbrace{r + \gamma \max_{a'} Q(s',a';\theta_i^-)}_{\text{Target network reward estimate at t+1}} - \underbrace{Q(s,a;\theta_i)}_{\text{Q-network reward estimate}} \right)^2 \right]$$

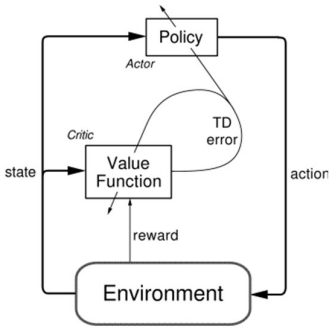
Source: [Deep Q-Networks Explained - LessWrong]

DQN uses two models of the neural network, policy network, and target network. Both will use the same architecture and have the same weight at the beginning. The target network will be used to calculate the optimal Q-value. To update the weight, we use the mean squared error between the expected Q and the target Q. After some iteration, we will soft update the weight of the target network based on the policy network like the equation below.

$$\theta^- = \theta \times \tau + \theta^- \times (1 - \tau)$$

Soft update means that we do not update the target network at once, but little by little. The target network will move slightly to the policy network value, since it will update very small, the update should be frequent so that the effect will be noticeable.

c. Advantageous Actor-Critic (A2C).



We can say that Actor-Critic is a Temporal Difference (TD) version of the Policy Gradient. It has two networks: Actor and Critic. The actor will decide which action should be taken and the critic inform the actor how good was the action and how it should adjust. The learning of the actor is based on the policy gradient approach. This algorithm will reduce the variance of Reinforce algorithm and can train the agent faster and better by using a combination of policy-based and value-based methods. The actor will be the policy function and the critic will be the value function.

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

By using the value function as the baseline function, we subtract the Q-value term from the V-value. It means how much better it is to take a specific action compared to the average, general action at the given state. This is called the advantage value. Because there is a relationship between the Q and V of the Bellman equation, we can rewrite the advantage value with:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

With that, we can only need one neural network for the value function.

d. Proximal Policy Optimization (PPO)

PPO can improve the stability of the actor training by limiting the policy update at each timestep. To know the impact of some action update, we use the ratio between the probability of action happening under some current policy divided by that action happened under the previous policy. If the ratio is more than one, it means that the action is more probable in the current policy than in the old policy. But between zero and one, it means that the action is less probable for the current policy than the old one. From that ratio, the objective function will be like the image below.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_\theta^{old}(a_t|s_t)}$$

Source: PPO loss functions - TensorFlow Reinforcement Learning Quick Start Guide [Book] (oreilly.com)

But, without some constraint, if the action taken is much more probable in the current policy than in recent policy, it can make lead to a large policy gradient step. To avoid that we can use the constraint of the objective function by using the PPO clip probability ratio.

e. Exploration Strategy

In reinforcement learning, exploration strategy refers to the approach an agent takes to exploring its environment and learning the optimal policy. The exploration strategy is a critical component of the RL algorithm because it determines how the agent interacts with the environment and collects data to update its model. This algorithm is necessary for RL because the agent needs to try different actions and observe their consequences to learn which actions are most rewarding in a given state. Without exploration, the agent may converge on a suboptimal policy or fail to find the optimal policy.

The exploration strategy we used in this experiment is the epsilon greedy. The agent selects the action with the highest estimate value with probability 1- epsilon and selects a random action with probability epsilon. The value of epsilon determines the balance between exploration and exploitation. If epsilon is set to a high value, the agent is more likely to select random actions, and hence explore more. If epsilon is set to a low value, the agent is more likely to select actions with the highest estimated value, and hence exploit more. To balance it, the value of epsilon is set to decrease from 0.99 to 0.9 over time when training as shown in the equation below.

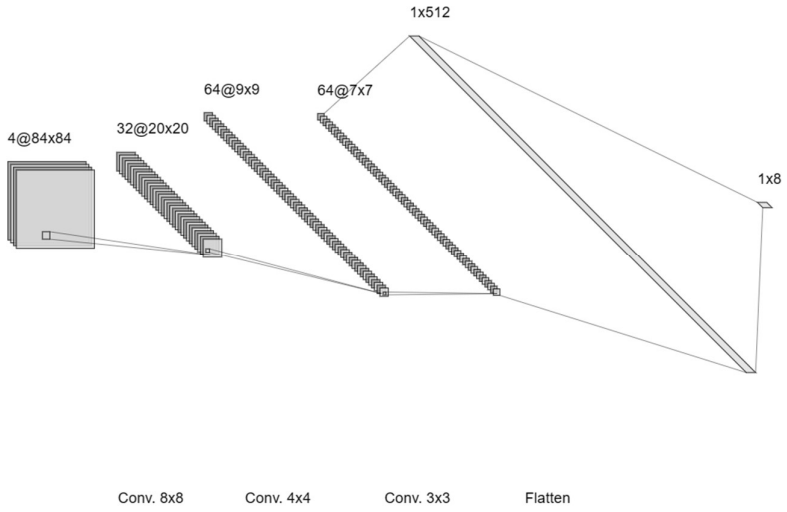
$$\epsilon = \epsilon_{start} + (\epsilon_{start} - \epsilon_{end}) \times \epsilon^{-(\frac{episode\ t^{th}}{decay\ rate})}$$

When n is the number of episodes and d is the rate of epsilon will decay.

5. Experiment Details

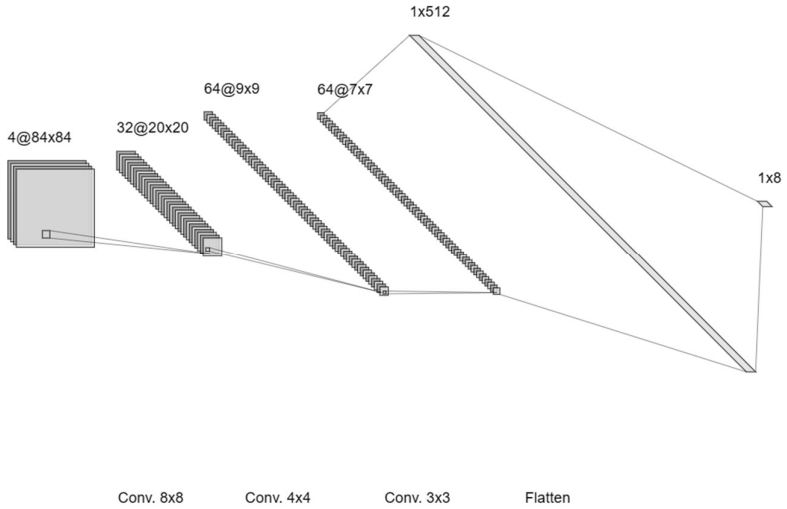
a. Experiment Settings & Network Architecture

This experiment is run on a computer with the following specification: Intel Core i7-9700K CPU, 16 GB RAM, and NVIDIA GeForce GTX 1080Ti GPU. As for the environment, we use a gym-retro environment. We implement the neural network using PyTorch and trained using Adam optimizer. As the architecture of the model can be seen in the figure below.

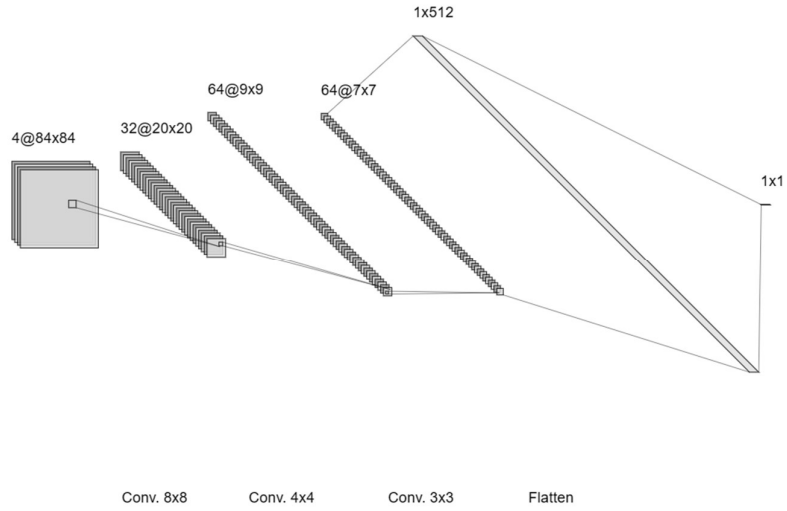


This experiment runs for 1000 episodes with a maximum of 10000 timesteps on each episode. The hyperparameters value for this experiment can be seen in the table below.

Hyperparameter	**Value**
Discount Factor	0.99
Replay Memory Size	100000
Batch size	32
Learning Rate	0.0001
Target network soft update	0.001
Timestep interval to update the neural network	50
Timestep when the network starts learning from the replay memory	2000
Epsilon start	0.99
Epsilon end	0.09
Exploration decay rate	150



The image below is for the critic network, for the output layer, we use softmax.



The hyperparameters value for the actor and critic experiment can be seen in the table below.

Hyperparameter	**Value**
Discount Factor	0.99
Actor Learning Rate	0.0001
Critic Learning Rate	0.0001
Soft Update	0.95
Batch Size	32
PPO Epoch	5
Clip Parameter	0.2
Timestep interval to update the network	1000

For the Mario game this is the reward function that we use:

```
if score > score_max:
    reward += 1000
    score_max = score

if xscrollLo > xscrollLo_prev:
    reward += 50
    xscrollLo_prev = xscrollLo
    counter = 0
else:
    counter += 1
    reward -= 0.1 * (counter//10)

if xscrollHi > xscrollHi_prev:
    reward += 2000
    xscrollHi_prev = xscrollHi
    xscrollLo_prev = 0
```

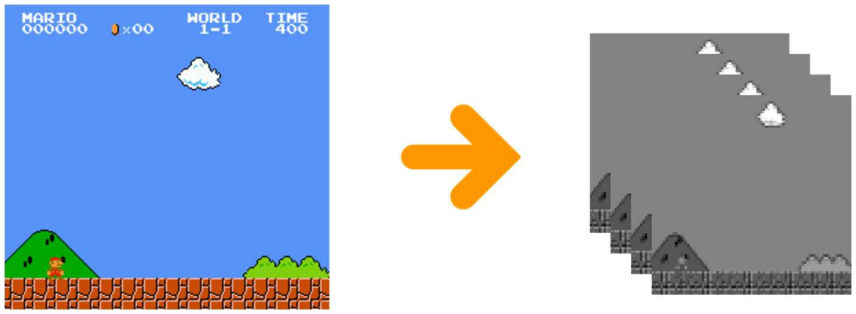
Mario will get the reward when going right based on xscrolHi and xscrolLo. When the agent gets a higher score than the maximum score, then the agent will get an additional reward. As the architecture of the actor can be seen in the figure below. The reward function for Felix The Cat will be in the figure below:

```
if (prev_state['score'] == info['score']):
    steps_stuck += 1
else:
    steps_stuck = 0

if prev_state['lives'] > info['lives']:
    reward -= 1000
prev_state = info

if score > high_score:
    reward += 500
    high_score = score
else:
    reward -=2

if (steps_stuck > 20):
    reward -= 1

reward_e += reward
```

b. Input Pre-processing

As the input data, it will be represented by stacked images from the video game. The image will be preprocessed first before feeding them into the neural network. This is done to make the neural network easier to understand. The preprocessing is:

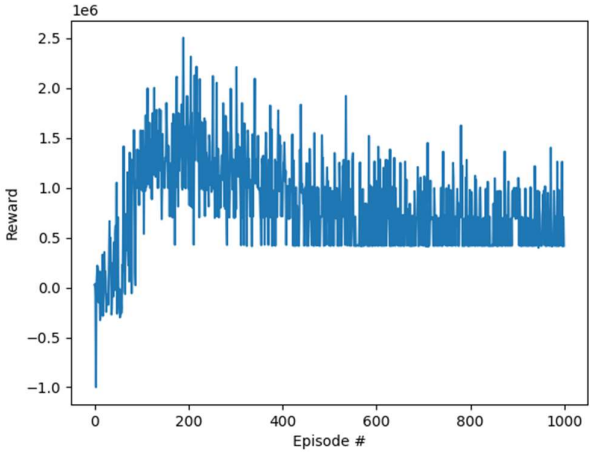
- Convert the RGB image into grayscale
- Resizing the image to 84x84
- Crop the image to remove the unnecessary part
- Normalizing the pixel value from 0-255 to 0-1
- Stack 4 sequence frames together



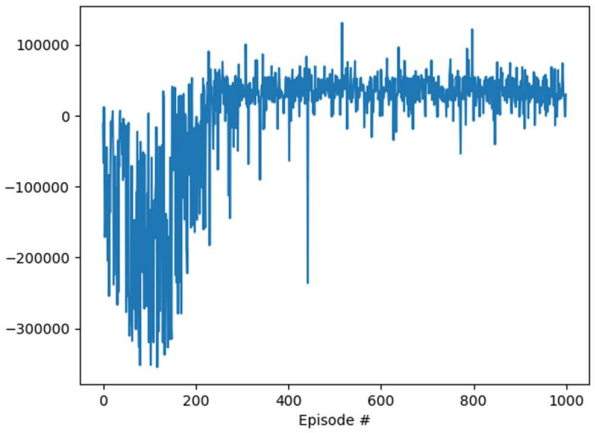
6. Result and Analysis

a. Super Mario Bros – DQN

At first, we try to train the agent without adding a new reward function, only by using their base reward function. We can see the result of the reward on 1000 episodes in the image below. To train this agent it takes about 4 hours.



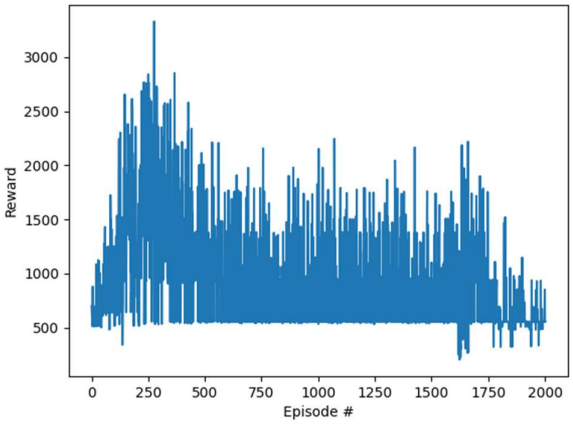
Then we try to train the agent with an additional reward function, where we will give the agent some reward when they keep going to the right, and give punishment when the agent gets stuck or not going to the right. We want to make the agent more motivated to go to the goal. Because from the previous experiment, the agent doesn't motivate enough to keep going to the right, instead just keeps dying. We will run this on 1000 episodes, the result can be seen on the image below.



From the observation, we can know that the reward of the agent can, get lower or stuck after some episodes. This can be happened because of the overfitting problem. As the agent learns more about the environment, it may start memorizing the optimal actions for specific states rather than learning a general policy. This memorization may lead to overfitting, where the agent becomes too specialized in some certain place. The overfitting of the Q-function can make the agent select the wrong action because the Q-value function is overfitted. If the agent keeps doing the same mistake, it can make the agent worse. In this scenario, the agent keeps falling into the same pit over and over, without trying to increase the reward. To make the agent not die, we can try to make the agent only have one life and punish them when the agent dies, so the agent can avoid some enemy or pit.

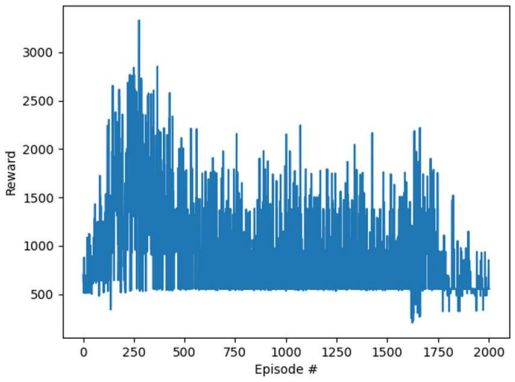
To overcome the problem, we can try to adjust the hyperparameters again to improve the performance, such as the learning rate, discount factor, and the target network update frequency. Also, we can try to use the Dueling DQN which separates the estimation of the state-value function and the action advantage function. This separation can lead to better estimates of the Q-values and improved performance. We can update the agent with the most informative replay that can lead to faster learning and improved performance y using prioritized experience replay. Prioritized experience replay will help the agent to prioritize the replay buffer based on the TD error.

To solve the overfitting of the network, we tried to use the pre-trained ResNet. The result can be seen in the image below. This agent will be trained for 2000 episodes, but as seen in the image below, the result is not good. This can be caused by the complexity of the network and ResNet is not trained for this image, because ResNet is trained for RGM images on different datasets.



b. Super Mario Bros – A2C & PPO

To solve the DQN problem, we try to use A2C as the RL algorithm because A2C can update the policy and value function in a single forward and backward pass. The algorithm combines the advantages of both policy-based and value-based methods, allowing for better sample efficiency and more stable learning. With the help of PPO, A2C directly optimizes the policy and can help the agent learn a more robust and stable policy. The image below will show the result of the A2C agent reward on 1000 episodes.

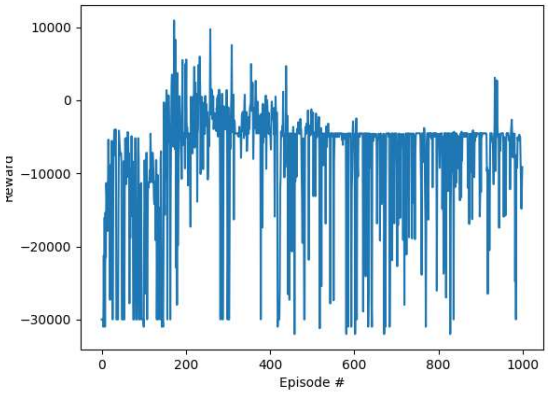


As seen in the image above, the reward is not stable and tends to get a lower reward than the DQN. This can be caused by the hyperparameter tuning and A2C requires the agent to estimate both the state-value function and the policy, which can be more computationally expensive than estimating only the state-action value function in DQN. The environment can be not suitable for the A2C algorithm, which can lead to training instability, especially if the gradients are too noisy. To overcome this problem, we can try to use different hyperparameters, more powerful neural network architecture, or ensure that the agent has a better exploration strategy.

c. Felix The Cat – DQN

In this experiment, the agent can get the reward when they get some score by getting some points or killing an enemy. When the agent did not get any score for a certain time, then the agent will get some punishment. When we run the experiment, the agent gets stuck at the same place, where the agent needs to go up instead of going left as seen in the image below.

The reward graph for over 1000 episodes of training can be seen in the figure below.



We can see from the figure below, the reward obtained by the agent increases as the episode goes on, but after certain episodes, the reward goes down again. This can be caused by overfitting the Q-function when selecting actions. This can be caused by training the policy for too long on the same data distribution. When the agent selects the wrong action, because the Q-value function is overfitted, the agent can not generalize well on how to recover from that mistake. In this scenario, the agent keeps choosing the wrong action by coming to the enemies or dropping into the hole.

To overcome the agent being stuck at the same place, we can increase the exploration decay rate, so the agent can explore more even in the mid to end of episodes. Therefore, the agent can explore more when the agent gets stuck at a certain place. The reward function can affect how the agent will act. For this game, there is not much information that can support us to make the reward function, if we can get some information about the position of the agent, it can help to make the reward function better and navigate the agent towards the goal. This game only provides information about lives and scores. That information cannot provide a good reward function to help the agent.

7. Conclusion

From the experiment above, we can conclude that the agent could make some progress on the game. The agent can learn how to play the game, but the performance is not good. This can be because of the overfitting of the network or the hyperparameter tuning. In the future, we can try to use another algorithm that is more suitable for the game, or add a new method to make the agent learn how to play the game well. The game state or information is very important to help the agent learn, for example, the Felix The Cat game is hard to learn for the agent because it's hard to make the reward function that can help the agent go towards the goal. But In Mario's game, it is easier because we can know where the agent's position is and make rewards and punishments according to the agent's position.

The experiment result video was uploaded on YouTube:

- Super Mario Bros - DQN: https://youtu.be/0xGgGnw-ZsU
- Super Mario Bros - A2C: https://youtu.be/nQbJq5mosaQ
- Felix The Cat - DQN: https://youtu.be/0Xd5TEFk4GU