Deep Reinforcement Learning for Super Mario Bros: Deep Q-Networks vs. Policy Optimization

Jiwoo Jung
Kansas State University
1601 Roof Drive, Manhattan, KS
synoeca523@ksu.edu

Abstract

Training reinforcement learning (RL) models to play moderately complex games is a challenging problem, requiring the acquisition of visual features and optimal policy formulation. In this report, I tackle the intricate task of developing an algorithm to guide Super Mario through the initial game level. The primary focus is on the Markov Decision Process, Policy Gradient methods, and the exploration of the Double Deep Q-Network. The research systematically compares the efficacy of Proximal Policy Optimization with Generalized Advantage Estimation (PPO + GAE), Double Deep Q-Network (DDQN), and Policy Gradient. The results highlight PPO + GAE as the most effective approach. To further advance this project, attention is directed towards overcoming gradient challenges through the implementation of Batch Normalization and Gradient Clipping. Additionally, the exploration of techniques to identify and navigate "trap states," especially in scenarios with immotile agents, is proposed for future investigation.

1. Introduction

1.1. Problem Statement and Overview

Sequential decision problems can be encountered throughout our lives. Whether we're taking a shower, driving, or playing a game, all decisions are made sequentially over time. Reinforcement Learning (RL) is a methodology designed to solve these important problems, and the process of learning to correct behavior through trial and error to maximize cumulative reward in sequential decision problems is called reinforcement learning.

Training RL models to play moderately complex games is a challenging problem. While algorithms such as Deep Q-learning are starting to perform well on some games, applying reinforcement learning to games in general remains challenging. One reason for this is that RL algorithms start with no information about what behaviors are effective in a given game, and when using untrained convolutional networks as input for their agents, they do not have the same intuition about what visual cues are important for decision making as humans do.

In this project, I train a RL agent that plays Atari's Super Mario Bros. released in 1985 with three algorithms: Policy Gradient (PG), Double-Deep Q-Network (DDQN), and Proximal Policy Optimization (PPO) + Generalized Advantage Estimation (GAE) to compare mean reward and sampling efficiency to gain insight into the strategy of gameplaying RL agents.

2. Background and Related Work

2.1. Schematic sequential decision-making problem

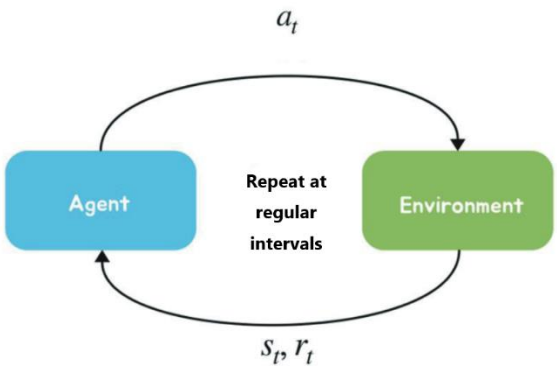

Figure 1. Agent and environment schematic.

An agent is the subject of reinforcement learning and is an entity that learns and interacts with the environment. The agent interacts with the environment by deciding which action a_t to take in the current situation s_t. The

environment changes state through the action a_t received from the agent, and as a result, the state changes from s_t to s_{t+1}. Additionally, the environment jointly calculates the reward r_{t+1} to give to the agent and send s_{t+1} and r_{t+1} to the agent.

In the real world, unlike the above figure, the flow of time may be continuous, but in sequential decision-making problems, the flow of time is considered discrete. The unit of time is called time step (t).

2.2. Markov Decision Process

$$p(s', r|s, a) = Pr\left[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right]$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

Figure 2. A Markov decision process (MDP) is a tuple (S,A,R,p,γ). From "Policy gradients in a Nutshell," by S. Kapoor, 2018, Medium.

As mentioned earlier, reinforcement learning is a methodology for solving sequential decision-making problems, but in fact, it is still a bit abstract. Sequential decision-making problems can be accurately expressed through the concept of Markov Decision Process. In a Markov decision process, an agent appears as a decision-making entity.

A set of states 'S' is a collection of all possible states. Action set 'A' is a collection of actions that an agent can take. The reward function' R' refers to the reward received for choosing action 'a' in state 's'.

The transition probability matrix 'p' defines the probability that the next state will be 's'' when the agent chooses action 'a' in the current state 's'. The important thing to note here is that the state reached when action 'a' is selected from state 's' is not deterministic. Since there is a probability distribution over 's'' and it is part of the environment, p defines the process dynamics.

The discount factor 'γ' is a number between 0 and 1 and is a parameter that indicates how much more importance is given to the reward obtained now compared to the reward obtained in the future in RL.

G is the return and refers to the sum of attenuated rewards received in the future from time t. As the figure shows, returns are defined through future rewards without considering past rewards.

2.3. Policy Function

$$\pi(A_t = a|S_t = s)$$

$$\forall A_t \in \mathcal{A}(s), S_t \in \mathcal{S}$$

Figure 3. A policy (π) is defined as the probability distribution of actions given a state. From "Policy gradients in a Nutshell," by S. Kapoor, 2018, Medium.

In real-world MDPs, the number of states can be in the tens of billions, and the number of actions is even larger. In these complex MDPs, what we eventually want to find is which action 'a' should be chosen for each state 's' to maximize the sum of rewards. We call this a policy, a function that determines which action to choose in each state.

2.3.1. State Value Function

$$V(s) = \mathbb{E}_{\pi_\theta}[G_t|S_t = s]$$

Figure 4. State Value is defined as the expected returns given a state following the policy π_θ. From "Policy gradients in a Nutshell," by S. Kapoor, 2018, Medium.

The return of the state value function depends on the agent's policy function (πθ). Therefore, the value function is dependent on the policy function.

2.3.2. State-Action Value Function

$$Q^\pi(s, a) = E_\pi\{G_t|s_t = s, a_t = a\}$$

Figure 5. State-Action Value Function focusing on the particular action at the particular state. From "What is the Q function and what is the V function in reinforcement learning?", by S. Shafiq, 2016, Data Science Stack Exchange.

A state value function is a function that evaluates a state given a state, so why not also evaluate an action in each state? If we could evaluate the actions, we could evaluate all the actions we could choose in each state, and then choose the most valuable one.

A state-action value function q(s, a) takes a pair of states and actions as arguments and evaluates them simultaneously. Like the state value function v(s), it measures value through the return G_t that we will get in the future. And because the future state depends on the

policy π, we evaluate it with π fixed: it's the expected value of the return get from choosing 'a' from 's' and moving along π thereafter.

2.4. Bellman Optimality Equation

$$Q^*(s,a) = E_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

Figure 6. The Bellman equation (the next state s' is sampled from the environment's transition rules P). From "A simple guide to reinforcement learning with the super mario bros.", by N. Schneider, 2023, Medium.

When an MDP is given, the value calculated by selecting the best π (i.e., with the highest value of v_π(s)) among all π existing in the MDP is called the optimal value.

And if you follow each policy π*, which has the optimal value in all states, you will get a higher value than any other policy in all states. We call this policy π* the optimal policy.

The optimal state-action value function q*(s, a) is a function that returns the state-action value when π* is followed by agent.

3. Methodology

3.1. Reward Function

In the context of the gym-super-mario-bros environment, the reward function is designed to align with the game's objective of advancing as far right as possible, maximizing the agent's x-axis value promptly while avoiding death. The reward computation involves three distinct components:

Instantaneous Velocity (v) represents the difference in the agent's x-axis position between states and serves as a measure of the agent's instantaneous velocity for the given step. The formula is defined as v = x1 - x0, where x0 is the x position before the step, x1 is the x position after the step (Kauten, 2022).

Game Clock (c) is the game clock component that addresses the difference in the game clock between frames, discouraging the agent from remaining stationary. The formula is given by c = c0 - c1, where c0 is the clock reading before the step, c1 is the clock reading after the step (Kauten, 2022).

Death Penalty (d) encourages the agent to avoid death, penalizing the agent when in a state of demise (Kauten, 2022).

Finally, the overall reward (r) is calculated as the sum of these components: r = v + c + d. The reward is then clipped to fall within the range (-15, 15). The accompanying info dictionary returned by the step method contains valuable information about the game state, including the number of collected coins, flag reach status, remaining lives, cumulative score, current stage, Mario's status, time left on the clock, world, and Mario's position in the stage (Kauten, 2022).

3.2. Data Preprocessing

In the pursuit of training an effective reinforcement learning model for playing Super Mario Bros., careful data preprocessing plays a pivotal role in shaping the quality and efficiency of the learning process. The preprocessing steps aim to transform raw pixel observations from the game environment into a format suitable for consumption by a convolutional neural network (CNN). The following key preprocessing steps have been applied:

Frame skipping (SkipFrame Wrapper) expedite training without sacrificing critical information, a frame skipping mechanism is employed. This process involves skipping a fixed number of frames between each action, effectively reducing the temporal resolution of the input sequence (Schneider, 2023).

Grayscale conversion (GrayScaleObservation Wrapper) convert the RGB images captured from the game environment to grayscale. This not only reduces the dimensionality of the input data but also simplifies the learning task for the neural network by removing color information (Schneider, 2023).

Image resizing (ResizeObservation Wrapper) is crucial to standardize the input dimensions for the neural network. The images are resized to a specified shape, facilitating consistent input sizes across all training samples. The resizing process includes interpolation to maintain essential spatial information (Schneider, 2023).

These preprocessing steps collectively contribute to shaping the observations into a format that the neural network can effectively learn from. While these transformations significantly reduce the computational burden, they also introduce a degree of information loss, requiring a delicate balance to ensure the model receives

sufficient cues for effective decision-making during gameplay.

The processed images, now grayscale and resized, serve as the input to the convolutional layers of the neural network, enabling the model to learn spatial hierarchies and relevant features for making informed decisions in the Super Mario Bros. environment. The impact of these preprocessing choices is assessed in conjunction with the overall model performance during training and evaluation.

3.3. Double Deep Q-Network (DDQN)

Q-learning involves learning the best jack state-action value, Q*(s, a), using the Bellman Optimization Equation. Deep Q-learning extends the methodology of Q-learning to neural networks. In deep Q-learning, updates are made to reduce the difference between the correct answer, r + γ max Qθ(s', a'), and the current estimate, Q(s, a).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r'_{t+1} + \gamma \max_{a \in A} Q'(s'_{t+1}, a) - Q(s_t, a_t)$$

Figure 7. Update the Q-values according to the formula above. Here we estimate Q' out of the best Q values for the next state, but which action a' leads to this maximal Q is not quite important.). From "A simple guide to reinforcement learning with the super mario bros.", by N. Schneider, 2023, Medium.

As we continue to update the Q-value using the above expression, Q(s,a) will increasingly approach the optimal state-action value function Q*(s,a) (Schneider, 2023).

Double Deep Q-Network introduces two special methodologies to stabilize learning and improve performance while reinforcing the Q-function represented by the aforementioned neural network. These are Experience Replay and Target Network.

Reinforcement learning is essentially the process of reinforcing an agent with the experiences it has had. Experience replay literally starts with the idea of "wouldn't it be better if we reused the experience the agent had?". An experience is made up of multiple episodes, and an episode is made up of multiple state transitions. A replay buffer stores the n most recent episodes, and when you train, you randomly draw data from this buffer. Because of this randomization, each piece of data can be reused multiple times. Not only does this increase data efficiency, but by training with such a wide variety of data, we can learn more efficiently because there is less correlation between each data point than if we were to use consecutive data points from the same game (Mnih et al., 2015).

The second idea that enhanced DQN is the methodology of having a separate target network. The intuitive meaning of the loss function J(θ) is the difference between the correct answer and the estimate, and it is updated with θ to reduce this difference. However, in Q-learning, r + γ max Qθ(s', a') is used as the correct answer, so the correct answer is dependent on θ. So, every time θ is updated, the value corresponding to the correct answer keeps changing, which is detrimental to stable learning.

$$J(\theta) = E_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma Q' \left(s', \operatorname{argmax}_{a'} Q(s', a', \theta), \theta^- \right) - Q(s, a, \theta) \right)^2 \right]$$

where θ^- − parameters of the target network
θ − parameters of the online network

Figure 8. Double Deep Q-Network loss function. From "A simple guide to reinforcement learning with the super mario bros.", by N. Schneider, 2023, Medium.

In 2016, Hado van Hasselt introduced a new way to estimate the value of Q called double Q-learning. You prepare two pairs of networks: a target network, which is the network you use to compute the correct answer, and an online network that is being trained. Then you estimate the value of Q while keeping the parameters of the target network frozen for a while, and the estimate has a stable distribution (Van Hasselt, 2016).

For this project, I implement DDQN by using experience replay and an online-target network to stabilize learning and improve performance,

3.4. Policy Gradient (PG)

$$J(\theta) = \mathbb{E}_\pi \left[r(\tau) \right]$$

Figure 9. Maximize the "expected" reward following a parametrized policy. From "Policy gradients in a Nutshell," by S. Kapoor, 2018, Medium.

In contrast to the Deterministic Deep Q-Networks (DDQN), the Policy Gradient (PG) approach operates as a probabilistic policy-based agent, distinctively focusing on the continuous refinement of a policy function, denoted as π(s, a), facilitated through neural networks. While DDQN relies on value functions for action selection, PG introduces a stochastic element, emphasizing the

enhancement of the policy to adapt to varying environmental states.

The policy function, π, encapsulates the probability distribution of selecting actions given a particular state. This probabilistic nature aligns with the goal of maximizing the expected value (𝔼) of a random variable, considering the inherent uncertainty in the policy's outcomes. In the reinforcement learning context, where rewards (r) play a pivotal role, PG seeks to optimize the policy to achieve the highest cumulative rewards over time (Kapoor, 2018).

A trajectory (τ) in PG refers to the sequential states and actions undertaken by the agent during its interaction with the environment. The parameters (θ) encompass the weights and biases of the neural network governing the policy function, representing the learnable components that adapt during training (Kapoor, 2018).

At the core of PG is the pursuit of maximizing the policy performance (J), a metric indicating the effectiveness of the policy in generating desirable actions. The primary objective is to find the optimal set of parameters (θ⋆) that lead to the highest policy performance. The policy gradient, representing the gradient of J with respect to the parameters θ, guides the iterative adjustment of θ, driving the policy towards configurations associated with increased expected rewards (Kapoor, 2018).

3.4.1. Policy Gradient Theorem

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\sim\pi_\theta}[R(\tau)] = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$$

Figure 10. The derivation of policy gradient takes the form. From “Policy gradients in a Nutshell,” by S. Kapoor, 2018, Medium.

The Policy Gradient Theorem is that the derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ (Kapoor, 2018).

In the context of this project, the chosen PG algorithm is "REINFORCE." This algorithm serves as a practical instantiation of the Policy Gradient Theorem, guiding the agent to adapt its policy iteratively based on the observed rewards and trajectories. The REINFORCE algorithm, known for its simplicity and effectiveness, involves adjusting the policy parameters in the direction that increases the likelihood of actions leading to higher rewards. This adaptive process, rooted in probabilistic decision-making, aligns seamlessly with the probabilistic nature of PG, making it a suitable choice for training agents in environments where uncertainty and adaptability are paramount considerations (Kapoor, 2018).

3.5. Proximal Policy Optimization (PPO) + Generalized Advantage Estimation (GAE)

3.5.1. Actor-Critic Method

The DDQN agent learned through the value network, and the PG agent learned through the policy network. So, couldn’t we train the two networks together?

$$\nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T}(R_{t+1} + \gamma V^\omega(S_{t+1}) - V^\omega(S_t))\nabla \log \pi_\theta(a_t|s_t)\right)\right]$$

Figure 11. θ : Actor’s parameter. ω: Critic’s parameter. From “Policy gradients in a Nutshell,” by S. Kapoor, 2018, Medium.

The policy network πθ parameterized by θ and the value network Vω parameterized by ω can be learned together. πθ plays the role of an actor, selecting the action 'a' to be executed, and Vω plays the role of a critic, evaluating the value of the selected action 'a'. This method of learning both policy π and value V during the agent's learning process is called actor-criticism. The critique is updated to learn the value of the current policy function π, and the actor learns by evaluating V by strengthening if the result was good and weakening if the result was bad. Actor-critic methods help reduce variance and increase the stability of the learning process.

3.5.2. Generalized Advantage Estimation (GAE)

Suppose the agent is lucky enough to reach a state ‘s’’ where the value is very high. At this time, two actions, ‘a0’ and ‘a1’, can be selected, and when ‘a1’ is selected, a better return is obtained than when ‘a0’ is selected. However, because ‘s’’ is in such a good state in the first place, the return agent get afterward is high no matter what action you choose on ‘s’’. In summary, ‘a1’ is clearly a better action than ‘a0’ in ‘s’’, and both are strengthened. If this feels inefficient, couldn't we look into the future and decide which actions to strengthen?

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{k} (\gamma \cdot \lambda)^l \delta_{t+l}^V$$

Figure 12. We take an exponentially weighted average of the advantages to balance bias and variance. This is called Generalized Advantage Estimation. From "A simple guide to reinforcement learning with the super mario bros.", by N. Schneider, 2023, Medium.

So, we can think of an alternative like the above formula. The expected value of δ, the difference between the future rewards and the predicted state value, is the advantage A(s, a). In other words, δ is an unbiased estimator of A(s, a). Even if you select the same action 'a' in the same state 's', the δ value will get a different value each time depending on how the state transition occurs. This means that if you collect several of these values and average them, the value converges to A(s, a). This convergence property is instrumental in obtaining a more accurate and stable estimation of the advantage, contributing to the effectiveness of GAE in reinforcement learning scenarios (Schneider, 2023).

By introducing GAE into the framework, we enhance our ability to estimate advantages in a manner that adapts to the dynamic nature of state transitions. This adaptability is particularly valuable in scenarios where the environment exhibits variability, and traditional methods may struggle to provide reliable advantage estimates (Schneider, 2023).

4. Result and Evaluation

4.1. Policy Gradient Performance

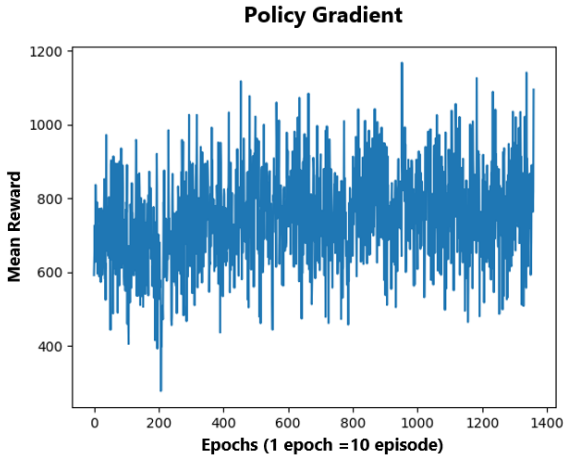

Figure 13. Mean Reward vs. Epochs graph of the Policy Gradient agent. The epoch on the x-axis consists of 10 episodes. The y-axis represents the average mean reward earned by the agent over 10 episodes. The computations were performed using CUDA on an RTX 3070 Laptop GPU.

Figure 13 plots the Mean Rewards earned by PG agents over approximately 13,000 episodes. The sample efficiency of PG agents is relatively low, which contributes to the high variance of the gradient estimates.

However, the PG algorithm is relatively the simplest to implement and, despite its slow convergence rate, shows a relatively steady learning gradient.

4.2. Double Deep Q-Network Performance

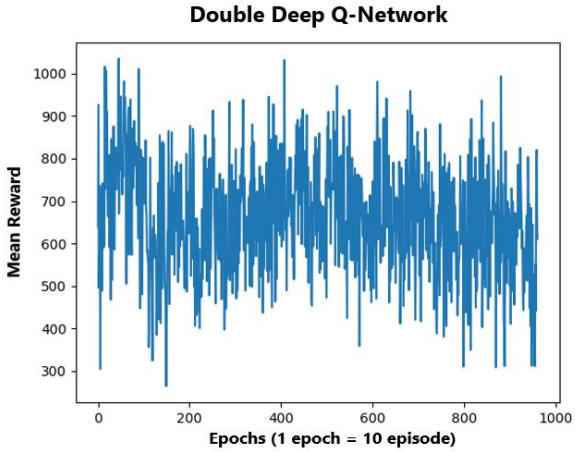

Figure 14. Mean Reward vs. Epochs graph of the Double Deep Q-Network agent. The epoch on the x-axis consists of 10 episodes. The y-axis represents the average mean reward earned by the agent over 10 episodes. The computations were performed using CUDA on an RTX 3070 Laptop GPU.

DDQN agents show a faster initial learning speed compared to PG agents. However, despite implementing online-target network configuration and experience replay, it shows low training stability and potentially vanishing gradients. Fine tuning of hyperparameters may be required.

4.3. Proximal Policy Optimization + Generalized Advantage Estimation Performance

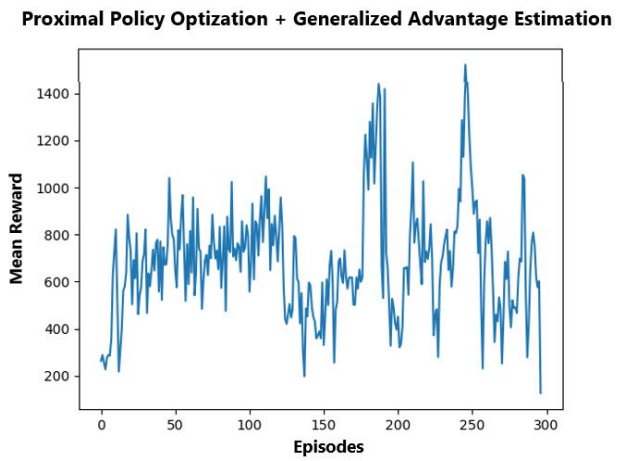

Figure 15. Mean Reward vs. Episodes graph of the PPO + GAE agent. The x-axis consists of a episode. The y-axis represents the average mean reward earned by the agent over 10 episodes. The computations were performed using CUDA on an RTX 3070 Laptop GPU.

The PPO + GAE agent shows an extremely fast initial learning rate compared to the other two agents and has the best overall sample efficiency. However, because the code implementation is very complex, two networks, actor and critic, are trained simultaneously, and an advantage calculation process is also added, the average time to play one episode was the longest.

4.4. Model Evaluation

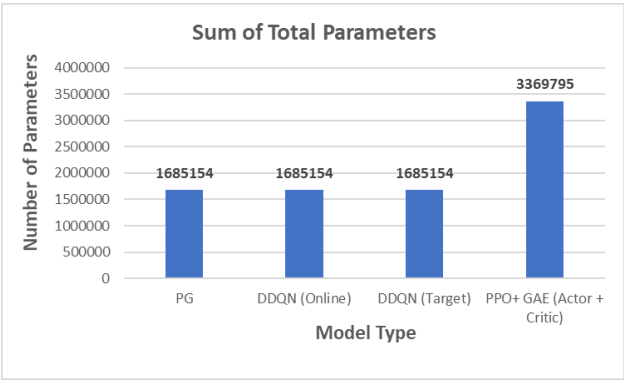

Figure 16. Sum of Total Parameter of PG, DDQN (Online), DDQN (Target), and PPO + GAE (Actor + Critic) models.

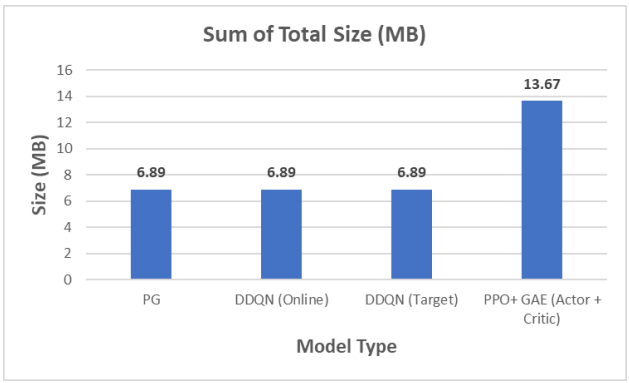

Figure 17. Sum of Total Size (MB) of PG, DDQN (Online), DDQN (Target), and PPO + GAE (Actor + Critic) models.

Since the DDQN agent is calculated by adding the Online network and the Target network, it has the same number of parameters and total size (MB) as the actor-critic network of the PPO + GAE agent. The PG agent had half the number of parameters and size of the DDQN agent and the PPO + GAE agent.

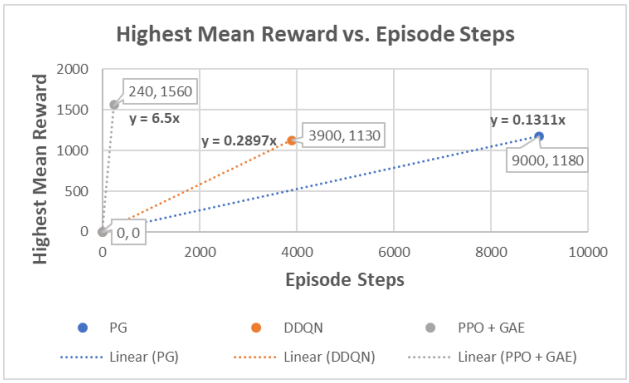

Figure 18. Highest Mean Reward vs. Episode Steps graph of PG, DDQN (Online), DDQN (Target), and PPO + GAE (Actor + Critic) models. PG agents had an average gradient of 0.1311, DDQN agents had an average gradient of 0.2897, and PPO+GAE agents had an average gradient of 6.5.

The highest Mean Reward earned by a PG agent was 1180, which took a total of 9000 episodes. Therefore, the average gradient for PG agents is 0.1311. The highest Mean Reward earned by a DDQN agent was 1130, which took a total of 3900 episodes. Therefore, the average gradient of the DDQN agent is 0.2897. The highest Mean Reward earned by a PPO+GAE agent was 1560, which took a total of 240 episodes. Therefore, the average gradient for PPO + GAE agents is 6.5.

4.5. Conclusion

Despite its fast initial learning rate, DDQN agents show an extremely unstable learning curve.

The PG agent has the slowest sample efficiency, but shows a more stable and less volatile learning process than the DDQN agent.

The PPO + GAE agent exhibits an extremely fast initial learning rate and the best overall sampling efficiency despite its slow per-episode learning rate, complex code implementation, and large parameter count and capacity. Therefore, I confirm that the PPO + GAE agent has the highest performance playing Super Mario Bros.

5. Troubleshooting and Future Work

5.1. Parallelism Issues on Windows 10

During the development and implementation of the reinforcement learning agent, a notable challenge arose concerning parallelism issues specifically on Windows 10. This issue was encountered in the context of utilizing multiple environments for training, where the attempt to parallelize the processing encountered unexpected limitations.

One significant aspect of the challenge was observed in the handling of communication between the main process and subprocesses. Specifically, the utilization of pipes for interprocess communication was met with limitations on Windows. The line of code `self.agent_conns[index].close()` was identified as a potential source of the issue, emphasizing the complexities associated with pipe usage on Windows systems (Mumasr, 2020).

The challenge was further exacerbated by the approach taken in the `MultipleEnvironments` class, where all environments were instantiated in the main process, and subsequent attempts were made to access each individual environment in a subprocess. This design choice, while effective in many environments, encountered specific issues related to Windows-specific sharing memory policies associated with the `_LIB` module in the `nes_py` library (Mumasr, 2020).

5.2. Vanishing and Exploding Gradient

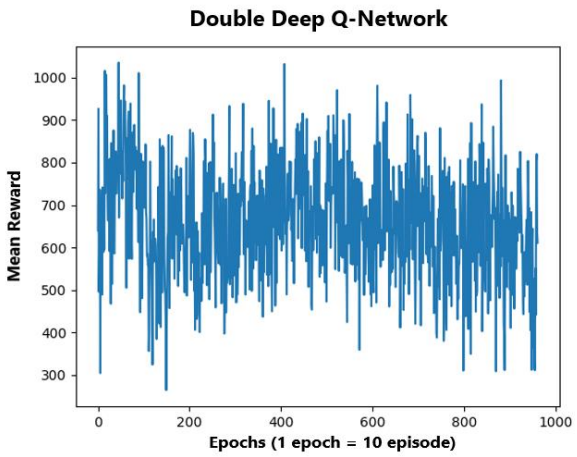

Figure 19. Mean Reward vs. Epochs graph of the Double Deep Q-Network agent. Showing vanishing and exploding gradient problem.

During the learning process, the agent's learning curve was unstable, showing no consistent learning progress in most cases. As can be seen in Figure 19, the DDQN agent showed a tendency to gradually vanish even after learning about 9000 episodes. In the case of the PG algorithm, in extreme cases, the agent refused to move and degenerated into a Mean Reward of 0. The PPO + GAE agent shows extreme vanishing and explosion gradients despite high sampling efficiency.

5.3. Future Work

5.3.1. Parallelism issues troubleshooting

To address the parallelism problems of the reinforcement learning process on Windows 10, potential future work includes investigating alternative interprocess communication mechanisms that are better suited for Windows environments.

The parallelism issue particularly related to the usage of pipes and subprocesses on Windows, is more likely to be associated with the operating system's handling of interprocess communication (IPC) rather than being explicitly CPU- or GPU-related.

Parallelism issues, in this context, often stem from the intricacies of how different operating systems manage concurrent execution and communication between processes. Windows, in particular, has certain limitations and considerations when it comes to IPC mechanisms, and this can affect the implementation of parallel processing in Python scripts.

Exploring modifications or workarounds in the instantiation and access of environments in subprocesses

may offer insights into overcoming the observed limitations.

5.3.2. Vanishing and Exploding Gradient Troubleshooting

To address the challenges associated with Vanishing and Exploding Gradients, various techniques have been proposed. One prominent method is Batch Normalization, involving normalizing linear outputs by subtracting the batch mean and dividing by the batch standard deviation. This normalization helps prevent training from getting stuck in saturated regimes of nonlinearities (Jonathan, 2021).

Another effective approach is Gradient Clipping, aiming to avoid vanishing and exploding gradients during neural network training by constraining the magnitude of gradients (Li, 2023) (Slater, 2019).

In future work, implementing Batch Normalization and Gradient Clipping will be explored as strategies to mitigate the issues related to Vanishing and Exploding Gradients. These techniques are expected to enhance the stability and effectiveness of the training process.

Furthermore, Trap State Mitigation techniques will be investigated. This involves exploring methods to detect and address "trap states," with a particular focus on scenarios involving immotile agents. Identifying and navigating these trap states are crucial aspects of reinforcement learning, contributing to ongoing research in RL area.

References

Schneider, N. (2023, October 8). A simple guide to reinforcement learning with the super mario bros.. environment. Medium. https://medium.com/geekculture/a-simple-guide-to-reinforcement-learning-with-the-super-mario-bros-environment-495a13974a54%20

Kapoor, S. (2018, June 2). Policy gradients in a Nutshell. Medium. https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d

Shafiq, S., & aerin. (2016, February). What is the Q function and what is the V function in reinforcement learning?. Data Science Stack Exchange. https://datascience.stackexchange.com/questions/9832/what-is-the-q-function-and-what-is-the-v-function-in-reinforcement-learning

Kauten, C. (2022, June 20). Gym-super-mario-bros. PyPI. https://pypi.org/project/gym-super-mario-bros/

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. https://doi.org/10.1038/nature14236

Mumasr. (2020, December 21). OSERROR and eoferror2 · issue #14 · uvipen/super-mario-bros-PPO-pytorch. GitHub. https://github.com/uvipen/Super-mario-bros-PPO-pytorch/issues/14

Jonathan. (2021, June). How batch normalization layer resolve the vanishing gradient problem?. Data Science Stack Exchange. https://datascience.stackexchange.com/questions/95160/how-batch-normalization-layer-resolve-the-vanishing-gradient-problem

Li, K. (Yi). (2023, August 7). Vanishing and exploding gradients in neural network models: Debugging, monitoring, and fixing. neptune.ai. https://neptune.ai/blog/vanishing-and-exploding-gradients-debugging-monitoring-fixing

Slater, N. (2019, October). Relu for combating the problem of vanishing gradient in RNN?. Data Science Stack Exchange. https://datascience.stackexchange.com/questions/61358/relu-for-combating-the-problem-of-vanishing-gradient-in-rnn

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double Q-learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1). https://doi.org/10.1609/aaai.v30i1.10295

Jung, J. (2023, December 11). RLAgent. GitHub. https://github.com/Synoeca/RLAgent