Optimizing Learning Rates in Proximal Policy Optimization: An Exploration Within Super Mario Bros

Moe Khalil
Stanford University
Stanford, CA
mfkhalil@stanford.edu

Thomas Adamo
Stanford University
Stanford, CA
tadamo@stanford.edu

Walker Stewart
Stanford University
Stanford, CA
walker23@stanford.edu

1 Abstract

The utilization of reinforcement learning algorithms, specifically Proximal Policy Optimization (PPO), in complex environments such as video games has become a common practice in artificial intelligence studies. This paper presents an exploration of the impact of varying learning rates on the PPO algorithm's performance within the well-known video game, Super Mario Bros. In this context, a convolutional neural network (CNN) serves as the policy model for interpreting the game's visual inputs. Through a series of tests, it was found that a learning rate of 3e-4 yielded the most consistent performance for our AI agent in navigating the first level of the game. This research contributes to the ongoing dialogue about optimizing learning parameters in reinforcement learning applications, particularly with respect to PPO in gaming environments. However, while these findings provide valuable insight, it should be noted that they represent a specific use case and further investigations are warranted to generalize these results.

2 Introduction

Reinforcement learning is a fast-growing subset of machine learning with numerous applications in various sectors. One of the most captivating challenges in this domain lies in the arena of game playing, specifically the complex, dynamic environments found in games like Super Mario Bros. By developing algorithms that can learn to play and succeed in such games, we not only make strides in artificial intelligence but also provide valuable insights for complex real-world problems such as autonomous vehicle navigation and financial trading strategies.

In this study, we sought to refine the application of a popular reinforcement learning algorithm, Proximal Policy Optimization (PPO). While PPO has gained recognition for its balance of computational efficiency, ease of implementation, and sample complexity, it often requires meticulous tuning of hyperparameters for different problem domains. Among these, the learning rate plays a crucial role in algorithm performance, influencing the speed and stability of learning.

Our study revolves around the use of the PPO algorithm in the Super Mario Bros game environment. The input to our algorithm is a multidimensional array representing the game screen, including characters' positions, enemies, and various obstacles. Using the PPO algorithm, we output decisions pertaining to Mario's actions to navigate the game environment and maximize the game score. This research paper is particularly focused on investigating how different learning rates can impact PPO's performance in this context, ultimately discovering a learning rate of 3e-4 to be optimal.

3 Related Work

Reinforcement learning has been applied in numerous game environments, with Super Mario Bros being a popular choice. Such environments offer unique challenges for reinforcement learning algorithms. Kauten's adaptation of the Super Mario Bros game for the OpenAI Gym environment in his gym-super-mario-bros GitHub Repo provides a platform for this experimentation. Moreover, several class projects have taken on the task of training an agent to play Super Mario Bros, each with a distinct approach.

For example, some class projects have used traditional Q-Learning approaches, such as Liao, Yi, and Yang's project from 2012 and Klein's project from 2016. In contrast, our project leans on Proximal Policy Optimization (PPO), an algorithm from a paper published in 2017 with documented efficiency and superior performance in reinforcement learning tasks. Notably, PPO forms part of the Stable Baselines3 library, which is a collection of high-quality implementations of reinforcement learning algorithms.

Our project distinguishes itself by investigating the impact of varying learning rates on the PPO algorithm within the Super Mario Bros environment. This exploration aims to optimize the effectiveness of training reinforcement learning agents in this environment and offers an interesting contrast to previous projects. By examining different learning rates, we hope to glean insights that can improve the efficiency of reinforcement learning training in similar complex environments.

4 Method

4.1 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a type of policy optimization method that falls under the umbrella of policy gradient techniques in reinforcement learning. Its goal is to learn a policy that maximizes the expected cumulative reward. The central innovation of PPO over previous policy optimization methods, like vanilla policy gradient or actor-critic methods, is its objective function, which adds a penalty term to discourage excessive deviation from the previous policy during updates.

In traditional policy gradient methods, each update can significantly change the policy, which can lead to instability and harm the agent's performance. PPO addresses this issue by ensuring the new policy doesn't stray too far from the previous policy, effectively limiting the policy update at each step. This stability allows the algorithm to take multiple epochs of mini-batch updates on the same set of data, improving sample efficiency.

The operation of PPO can be described in a sequence of steps. Starting with interacting with the environment to collect training data, the agent uses its current policy to select actions and receives rewards. This interaction produces a trajectory of states, actions, and rewards. From this trajectory, PPO calculates the advantages and returns, which serve as an estimate of how good the actions were relative to the average. The policy is then updated by optimizing a surrogate objective function, which incorporates these advantages and the penalty for large policy changes.

PPO is designed to maintain a balance between exploration (trying out new, potentially better policies) and exploitation (sticking with the known, effective policy). By controlling the step size of policy updates with the penalty term, PPO can effectively explore the policy space without causing harmful instabilities.

In this study, we test the PPO algorithm's sensitivity to different learning rates. The learning rate is a crucial hyperparameter that determines the step size of updates during the optimization process. An optimal learning rate allows the algorithm to learn efficiently from the reward signals, improving performance and stability. By studying the impact of varying learning rates, we aim to provide insights on the effective tuning of PPO for tasks like Super Mario Bros, contributing valuable knowledge to the field of reinforcement learning.

4.2 Value Function

The value function for our agent is defined by the objective of the game, which is to move as far to the right as possible, as quickly as possible, without dying. We model this through a reward function that is comprised of three variables:

Velocity (v): This measures the difference in the agent's x values between states, effectively serving as the agent's instantaneous velocity for a given step. Mathematically, this is expressed as $v = x_1 - x_0$ where x_0 represents the x position before the step and x_1 represents the x position after the step. A positive v value signifies movement to the right, and a negative v value signifies movement to the left.

Clock Difference (c): This represents the change in the game clock between frames. This variable is critical as it penalizes the agent for standing still, encouraging constant movement.

Death Penalty (d): This variable penalizes the agent for dying in a state, therefore incentivizing the agent to avoid death. If the agent is alive, d equals zero, but if the agent dies, d equals -15.

In summation, the total value function is given by

$$r = v + c + d \tag{1}$$

4.3 Preprocessing

Preprocessing is crucial for running our RL algorithm, as it allows us to manipulate the input data such that the program can "understand" it and interact with it efficiently.Specifically, our three main methods of preprocessing for this project are:

1. `GrayScaleObservation:` `GrayScaleObservation` transforms each input screenshot from RGB color to gray-scale, which in turn decreases the complexity of the input data. The model itself does not need a color input to learn successfully - it doesn't matter if the obstacle is red or blue, it needs to be avoided no matter what.

2. `DummyVecEnv:` As the model can only operate on numerical inputs, the input data must be transformed from visual and actionable interfaces to vector representations, which is done by `DummyVecEnv`.

3. `VecFrameStack:` To increase the model's efficiency in learning which actions result in different outcomes, we used `VecFrameStack` to stack multiple 'screenshots' of the game in real-time. By stacking this data, the model is better equipped to observe the transitions that occur more efficiently.

4.4 Creating the Environment

Our training environment, SMB Level 1-1, is created using the `gym_super_mario_bros` library. This library provides a gym-like interface that allows developers and researchers to easily incorporate SMB as an environment for training and evaluating RL agents, which is perfect for our project. We wrap this environment with `JoypadSpace` to simplify the action space to a set of simple movements, then transform it to gray-scale, vectorize it, and frame-stack it using the preprocessing methods outlined above.

We also define a callback function, `CallbackFunction`, that saves the model periodically during training. This callback function is passed to the training method of the RL model and is called at each step of the training. The frequency of saving is defined by `save_frequency`.

To monitor our agent's performance throughout the training process, we made use of the `Monitor` class from the Stable Baselines 3 library. This tool records the agent's reward signals at each time step, which serves as an important metric of the agent's learning progress and effectiveness. This, alongside the callback function allows us to evaluate the performance of our agent at different stages of training and identify the best performing model.

5 Experiments

5.1 Metrics

To systematically evaluate the effectiveness of our model, we utilized two primary metrics: the reward function over time and the average episode length.

The reward function over time is an essential measure as it reflects the agent's learning progress. By tracking the rewards received by the agent at each step of the training process, we gain insight into how effectively the agent is learning to navigate the environment. A rising reward function over time generally indicates the agent's improved ability to make beneficial decisions, signaling a successful learning process.

In parallel, we employed the average episode length as a supplementary metric. This measure provides an average of the lengths of episodes experienced by the agent during training, with an episode defined as the sequence of states and actions from the start of a game to its termination. A longer episode typically suggests that the agent has learned to survive for an extended period in the environment, often indicating a more skilled agent. Thus, by observing both the reward function over time and the average episode length, we could form a comprehensive evaluation of the model's performance and the effectiveness of the varying learning rates in our PPO algorithm.

5.2 Experimental Setup

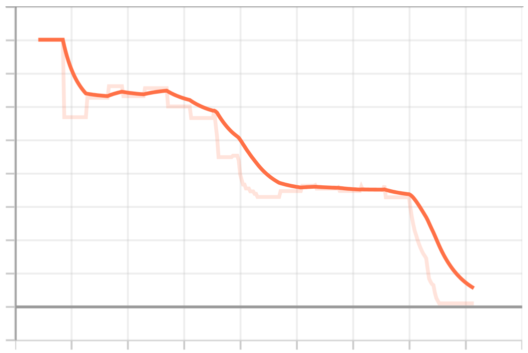
In order to investigate the effect of varying learning rates on the performance of the Proximal Policy Optimization algorithm, we devised an experimental setup that encompassed four different learning rates. These values were specifically chosen to represent a broad range of possible rates, with the intention of studying their impacts on the model's learning dynamics. The learning rates selected for this study were 10^{-5}, 10^{-4}, $3 * 10^{-4}$, and 10^{-3}.

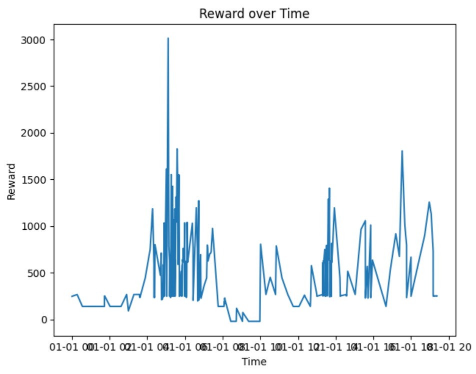
6 Results

The results of our experimentation showed a wide range of different outcomes depending on the different learning rates, all of which make sense in the context of the experiment. In our training, we set the number of steps to 2.5 million. The results were as follows:

6.1 Learning Rate: 10^{-3}

When using a learning rate of 10^{-3}, the output of the value function oscillated, sometimes reaching a high output but never maintaining it. This indicates that the agent is struggling to consistently improve its policy; it learns something valuable in one step, but then it unlearns it in the next. Simultaneously, the average episode length decreased fairly consistently throughout the experiment. This indicates that the agent is never truly able to maintain its performance over time, and although it decreases the number of steps it takes per trial, the decrease is not associated with a higher reward function output. This means that the learning rate was likely too large to converge on a policy, and is causing the algorithm to take leaps that are too large.


Average Episode Length over Time (LR=10^{-3})

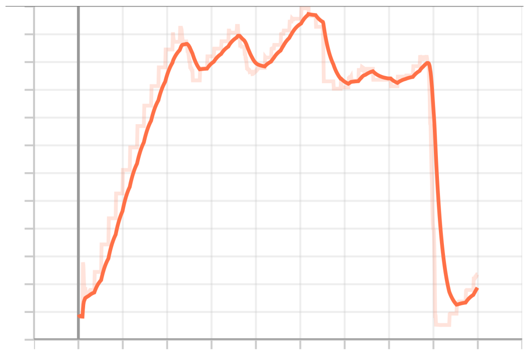


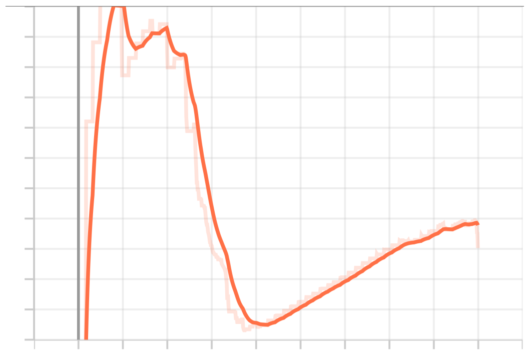
Reward Function over Time (LR=10^{-3})

6.2 Learning Rate: 10^{-4} and $3 * 10^{-4}$

When using a learning rate of 10^{-4}, the output of the value function also oscillated slightly but had an overall increasing trend, unlike with a learning rate of 10^{-3} where it did not trend in any direction. Interestingly, however, unlike the larger learning rate of 10^{-3}, the average episode length over time increased for a while during the first $100 - 200k$ steps before stabilizing at a relatively high value, then eventually had a sudden decrease followed by a slight increase towards the end. This is likely because the agent started out optimizing for a high reward function by moving right, then when that had been fully optimized, it began to further increase the reward function by taking fewer steps to get as far right as possible.

The learning rate of $3 * 10^{-4}$ had very similar behavior the rate of 10^{-4}. However, the main difference is that while the learning rate of 10^{-4} maintained a high average episode length for a while, the learning rate of $3 * 10^{-4}$ maintained it for a way shorter period before dropping and re-optimizing with a smaller number of steps. This likely means that it was sensitive enough to adjust quickly while not being way too oversensitive and not actually every converging on a good policy.


Average Episode Length over Time (LR=10^{-4})


Average Episode Length over Time (LR=$3*10^{-4}$)

6.3 Learning Rate: 10^{-5}

With a learning rate of 10^{-5}, we surprisingly get results similar to the learning rate of 10^{-3}. Once again, the reward function oscillates, however this time it never reaches values as high as those reached with a learning rate of 10^{-3}. Additionally, the average episode length over time also decreases consistently, which shows that the agent optimizes the reward function by taking fewer steps rather than actually making any progress. So, we can consider the learning rate of 10^{-5} as having failed.

7 Conclusion

In this study, we examined the impact of different learning rates on the Proximal Policy Optimization (PPO) algorithm within the Super Mario Bros game environment. The findings illustrate that the learning rate significantly influences the performance of the PPO algorithm, with an optimal learning rate of 3e-4 identified for consistent performance in the given gaming environment.

Higher learning rates (10^-3) led to high variability and inconsistency in performance. Lower learning rates (10^-5) also yielded suboptimal performance, although this manifested in low policy learning effectiveness rather than instability. Moderately set learning rates (10^-4 and $3 * 10^-4$) demonstrated more stable and effective learning over time, with the latter offering the most consistent performance overall.

To build upon this work, future studies could explore different reinforcement learning algorithms and their sensitivity to learning rate variations. This would offer a broader perspective on the impact of the learning rate, extending beyond the boundaries of the PPO algorithm. It would also be valuable to investigate other hyperparameters that can affect the performance of reinforcement learning algorithms in complex environments, providing a more comprehensive understanding of effective hyperparameter tuning strategies in reinforcement learning.

8 Contributions

Our team has been able to develop an RL agent for SMB with contributions from each member:

8.1 Walker

Walker researched the suitability of RL for this project, evaluating several RL algorithms. He recommended PPO as the main algorithm to use for the project, and explored past projects that had used PPO.

8.2 Thomas

Thomas investigated preprocessing techniques to expedite training. He recommended the use of grayscaling and frame stacking to streamline the training process. He also played a large part in writing up the report and creating the poster.

8.3 Moe

Building on the groundwork laid by Thomas and Walker, Moe developed the training code and ran it on an Amazon EC2 instance. He then used TensorBoard to present the results and the trained model, allowing for easy visualization.

References

[1] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). *Stable-Baselines3: Reliable Reinforcement Learning Implementations.* Journal of Machine Learning Research, 22(268), 1-8. Retrieved from http://jmlr.org/papers/v22/20-1364.html

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms.* arXiv preprint arXiv:1707.06347.

[3] Kauten, C. (2018). *Super Mario Bros for OpenAI Gym.* Retrieved from https://github.com/Kautenja/gym-super-mario-bros

[4] Anonymous. (n.d.). *Learning an Agent to Play Super Mario Bros [Notebook].* Retrieved from https://colab.research.google.com/drive/1Dj1dm-UiBAXzEGpGqFfzkAaIqJmNOrGi?usp=sharing

[5] Klein, S. (2016). Deep Q-Learning to Play Mario. CS229 Final Report, Department of Computer Science, Stanford University. Retrieved from https://cs229.stanford.edu/proj2016/report/klein-autonomousmariowithdeepreinforcementlearning-report.pdf