University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Benchmarking reinforcement learning algorithms in NES games

Erin-Louise Connolly

April 19, 2021

Abstract

Many different reinforcement learning algorithms exist, with some performing better than others. This project set out to benchmark two state-of-the art reinforcement learning algorithms, Asynchronous Advantage Actor Critic (A3C) and Proximal Policy Optimization (PPO) , in a set of Nintendo Entertainment System games. The games are of different genres to vary the types of tasks the agents must learn to deal with. We looked at each algorithm's learning and reward graphs, time taken to train and final scores achieved. A3C was faster and had less score variation between runs, however PPO managed to get higher scores in 4 out of 5 games it was tested in. Thus, we recommend PPO for cases where performance is the most important metric, and A3C for speed and stability.

Acknowledgements

I would like to thank my supervisor, Dr. Gerardo Aragon-Camarasa for his support and guidance throughout this project, and my friends, family, and cats for helping me get through university.

Contents

1 Introduction

This chapter will introduce the Benchmarking Reinforcement Learning Algorithms in NES Games project, briefly explaining what reinforcement learning is, why we want to benchmark it, and the aims of the project. The paper will then go on to detail specific areas of the project, showing the background, design, implementation and results. We will then evaluate the results and offer conclusions about the algorithms we are benchmarking, detailing some ideas for future work and reflections on the project as a whole.

1.1 Overview

Reinforcement Learning is an area of machine learning which sees agents interacting with an environment based on the consequences of its actions, with the aim of maximising a numerical reward over time (Sutton and Barto 2015). It can produce powerful results in video games, enabling agents to learn and complete a video game without any input or supervision (Shao et al. 2019). There are many different reinforcement learning algorithms, and this paper looks at a comparison between two popular algorithms in a set of Nintendo Entertainment System (NES) games to see their advantages and disadvantages.

1.2 Motivation

Reinforcement learning, and machine learning in general, can have many applications in video games. Bergdahl et al. (2020) found that use of deep reinforcement learning in testing helped increase test coverage and find unintended game mechanics and exploits. If an AI agent can find bugs that are passed over in normal testing environments, this could lead to a more enjoyable experience for players. More widespread use of reinforcement learning could make game development as a whole more efficient and more profitable.

Machine learning can also be used to help develop tool assisted speed runs of games. These are speed runs of games done within an emulator to get the "perfect" playthrough, usually meaning the fastest run possible. Currently this is done by using frame-by-frame runs, save states and macros, however reinforcement learning could aid the discovery of new tactics. Chrabaszcz et al. (2018) found that their agent had discovered a bug in the Atari game QBert, enabling it to quickly gain a very high score. If similar bugs can be found in other games, it may lead to new strategies for speed runs.

Reinforcement learning is not restricted to use in video games: it has been successfully used in healthcare to aid diagnosis and suggest treatments (Yu et al. 2020). This technology could lead to greater healthcare outcomes, for example higher life expectancies. Different algorithms would again be suited to different tasks, and lessons learnt in-game performance could be applied to areas such as healthcare.

With these use cases in mind, it is clear that comparisons between reinforcement learning algorithms are needed. We want to find a benchmark that tests different algorithms in a variety of tasks and finds their advantages and disadvantages. This benchmark could then be applied to multiple algorithms beyond the ones presented in this paper.

1.3 Aims

This objective of this project is to benchmark two popular reinforcement learning algorithms using Nintendo Entertainment System games. The games are of different genres, to represent a variety of tasks an algorithm may have to undertake. The benchmark will examine high scores produced by the agent, as well as its learning curves, to get a look at how well it performs but also how quickly it can learn. We will then perform an overall comparison of how the two algorithms fare in the games.

1.4 Summary

This chapter introduced the overview of the project and the motivation and aims behind it. The remainder of the dissertation is structured like so:

- **Chapter 2** explains what reinforcement learning is and the details of A3C and PPO, the two algorithms we will be benchmarking, and goes over some related work.
- **Chapter 3** introduces the Nintendo Entertainment System, the games we are benchmarking, the platform used to implement them, and what we are measuring for each algorithm.
- **Chapter 4** goes into more detail about the implementation of the benchmarking software, from the Gym features we are using, to an overview of the code, then finally the experiment setup.
- **Chapter 5** shows the results of the benchmarks, from high scores to learning and reward curves, comparing them with each other and a random agent, before evaluating the results, discussing how each algorithm performed and its advantages and disadvantages.
- **Chapter 6** presents a summary of the dissertation, ideas for future work and a reflection on the project as a whole.

2 Background

This chapter will examine the background of this paper, explaining the fundamental concepts of reinforcement learning. The main references for these concepts is Sutton and Barto (2015), *Reinforcement Learning: An Introduction*. We will then discuss the algorithms we have chosen to benchmark and the concepts behind them, finishing with a discussion of some related work in benchmarking reinforcement learning algorithms and what we can learn from them.

2.1 Reinforcement Learning

Reinforcement learning is a machine learning paradigm consisting of an agent making actions in an environment based on maximising a numerical award. The agent must explore the environment and learn from its actions, discovering which actions lead to the highest reward. The two most defining features of reinforcement learning are that the agent learns through trial and error and the reward it received can be delayed, where actions may affect all subsequent rewards and not just the immediate reward.

2.2 Components of a Reinforcement Learning System

Most reinforcement learning systems have are comprised of these four components: a policy, a reward signal, a value function and a model.

A policy (π) is how agents decide to behave, taking in states (s) and producing actions (a). Policies can be either deterministic, where they map states directly to actions ($\pi(s) = a$) or stochastic, where each time step (t) the policy outputs a probability distribution over actions ($\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$).

A reward signal shows the goal of the reinforcement learning problem. This could be moving right, earning points or finishing a level. Each time step, the agent receives the reward sent by the environment. The agent's objective is to maximise how much reward it earns over time. The reward signal shows what actions are good or bad: if an action taken by a policy gives low reward, the policy may be changed to take another action.

A value function specifies which actions are good in the long term, as opposed to the short term reward given by the reward function. This can also be used to select actions to take given a certain state. While a state may have low reward, it can have a high value. The value function is computed by $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = S)$ where G_t is the total reward, also known as return. Return is defined as $\sum_{i=t+1}^{\infty} \gamma^{i-t-1} R_i$ for some $\gamma \in [0, 1]$. γ is the discount factor (Quaedvlieg 2020). This is used as the representation of the environment may not be perfect, so it discounts the impact the future has on the return.

A model predicts what the environment might do next. Given a state and action, the model can predict what the next state and reward may be. They work on a transition model, where the transition probability to next state is computed given an action, and the probability of a reward given a certain action in a given state. This project focuses on *model-free* algorithms, which rely directly on the environment and trial-and-error, not predicting what will happen next.

2.3 Markov Decision Processes

In order to fully define reinforcement learning, we look to Markov decision processes. A Markov decision process is a framework which models decision making where there is a delayed reward, with actions influencing immediate and delayed rewards. In a Markov decision process we decide the value $q_*(s, a)$ of a potential action a in state s, or with optimal action selections the value $v_*(s)$ of a state. A state is said to be Markov if and only if it has the Markov Property, where $\mathbb{P}(S_t + 1|S_t) = \mathbb{P}(S_t + 1|S_1, ..., S_t)$, i.e. the probability of future states depends only on the current state and not previous states. The state must contain enough information to make decisions on without needing any other states.

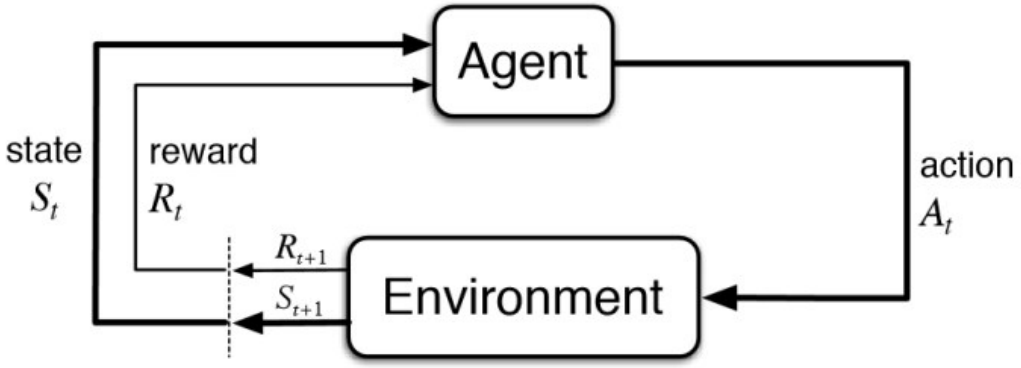

Figure 2.1: *The agent-environment interaction in a Markov Decision Process (Sutton and Barto 2015).*

Markov decision processes consist of the elements (S, A, P, R, γ). These can be seen in Figure 2.1. Each timestep t, the agent receives a representation of the environment's state s. The agent then selects an action a, and one timestep later receives a reward r. Almost all reinforcement learning problems can be expressed as Markov decision processes.

2.4 Policy Gradient Algorithms

For this project we will be focusing on Policy Gradient Algorithms, which target modelling and optimising the policy directly (Weng 2018). Policy gradient algorithms represent a policy with a parametric probability distribution $\pi_\theta(a|s) = \mathbb{P}[a|s; \theta]$, stochastically selecting action a in state s according to parameter vector θ (Silver et al. 2014).

Many reinforcement learning algorithms have no guarantee of convergence and some may even diverge (Peters 2010). Policy gradient algorithms do not suffer from this problem and are guaranteed to converge, improving performance. They are also much better at dealing with continuous states and actions whereas many reinforcement learning algorithms cannot deal with them, working only with discrete states and actions.

2.5 Asynchronous Advantage Actor Critic

Asynchronous Advantage Actor Critic (A3C) is an Actor Critic method. Actor Critic methods consist of two models, a critic, which updates the value function's parameters and an actor, which updates the policy parameters in accordance with the critic's suggestions. In A3C, the critic learns the value function whereas multiple actors are trained in parallel to estimate the policy, occasionally syncing with global parameters (Weng 2018). Each actor gains its own environment,

allowing exploration and aiding learning, contributing to the global network with diversified training data. The pseudo code for each actor thread can be seen in figure 2.2.

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\,(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 2.2: *Pseudo code for Anonymous Actor Critic, taken from Mnih et al. (2016).*

A3C keeps a value function $V(s_t;\theta_v)$ and policy $\pi(a_t|s_t;\theta$, updating them after every t_max actions or when a terminal state is reached. This update can be seen as $\Delta_{\theta'}\log\pi(a_t|s_t;\theta)A(s_t,a_t;\theta,\theta_v)$, where $A(s_t,a_t;\theta,\theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1}\gamma^i r_{t+1} + \gamma^k(Vs_{t+k};\theta_v) - V(s_t;\theta_v)$, where k can vary from state to state and is upper-bounded by t_{max} (Silver et al. 2014). The gradients in A3C are accumulated in training to improve stability. A3C is known for being fast to train, with Silver et al. (2014) showing it beating DQN, SARSA and Q-learning in every task.

2.6 Proximal Policy Optimization

Proximal Policy Optimization (PPO) tries to avoid large changes in policy, which could lower performance and stability. This is done by using a ratio to calculate the difference between the old and new policy, and clipping it to ensure it is not too large. If it is too large, this can lower performance and stability. This probability ratio is defined as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, where $r(\theta_{old}) = 1$. PPO uses the novel objective function seen in Figure 2.3, where θ is the policy parameter, $\hat{E}_t$ denotes the empirical expectation over timesteps, r_t is the ratio of probability under the new and old policies, $\hat{A}_t$ is the estimated advantage at time t and ϵ is a hyper-parameter, usually 0.1 or 0.2 (OpenAI 2017).

The pseudo code for PPO is shown in Figure 2.4.

Each iteration, N parallel actors collect T timesteps of data, and then construct the surrogate loss on the NT timesteps of data, optimising it for K epochs (Schulman et al. 2017). PPO is known for its performance, being the first algorithm to beat human players in the eSports game Dota 2 (OpenAI et al. 2019).

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\,]$$

Figure 2.3: *Objective function for Proximal Policy Optimization, taken from OpenAI (2017)*

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 2.4: *Pseudo code for Proximal Policy Optimization, taken from Schulman et al. (2017)*

2.7 Related Work

Much work has been done to compare existing reinforcement learning algorithms, using different methods and measures. In this section we will examine some of these papers and discuss their findings.

Barati and Chen (2019) proposes a deep reinforcement learning algorithm and an attention mechanism in a multi-view environment. A3C and PPO are used as a baseline in this paper along with some other single-view algorithms to compare with their novel algorithm. Agents were trained in multiple tasks, such as a self-driving car and physics simulators. PPO performed well, managing to beat the novel algorithm agents in some tasks. A3C did not perform as well, even coming last in most tasks. While this paper was not focused on A3C and PPO, it gives an indication of their performance, with some of the physics tasks being similar to video games.

Stooke and Abbeel (2019) looks at how to optimise deep reinforcement learning algorithms, paralellising them to speed up learning time without impacting performance. It takes the two main groups of reinforcement learning algorithms, policy gradient methods and q-value learning methods, and demonstrates multi-GPU versions of them. The algorithms are tested using Atari 2600 games via the Arcade Learning Environment. It found that scaling the algorithms successfully increased performance. As with the previous paper, PPO outperformed A3C, however it was noted that A3C takes less time to complete 50 million steps. The benchmarking performed in this paper does not use parallelised versions of the algorithms, however we get a sense for their performance from the versions with fewer GPUs.

Bhonker et al. (2017) compares a set of Deep Q-Learning Network algorithms in Super Nintendo Entertainment System (SNES) games, using the Retro Learning Environment. They trained each agent until it converged or 100 epochs had passed. The results were normalised by subtracting a random agent's score and dividing by a human player score. Using F-Zero and Super Mario Bros, it showed how the reward used could have a large impact on how fast an agent can learn, with some games experiencing delayed rewards struggling to learn quickly. It also showed that most algorithms struggle against humans, with only Dueling Double Deep Q-Learning beating a human in one game, Mortal Kombat. It concluded that the Retro Learning Environment provided some great challenges for reinforcement learning algorithms that could be studied further, such as delayed rewards and noisy background. The Retro Learning Environment was

succeeded by Gym Retro, the environment used for benchmarking in this paper.

Mnih et al. (2013) presented Deep Q-learning, showing its performance in a suite of Atari 2600 games. It studied the learning graphs and also the average total reward for each game. It also compared the agents against human players and a random agent. Their results were even able to beat the human players in some games.

2.8 Chapter Summary

This chapter introduced the concept of reinforcement learning, discussing the theory behind it, the category of policy gradient methods and how Asynchronous Advantage Actor Critic and Proximal Policy Optimization work. We then discussed some related work and what we learned from these papers, and how they relate to the work we present in this paper.

3 Design

This chapter will show design choices made for our benchmark and why these were suitable choices for this work. We discuss the choice of the Nintendo Entertainment System games console, the specific games we used, the platform we used to implement them, and how what measures we used to benchmark the algorithms' performance.

3.1 Nintendo Entertainment System

The Nintendo Entertainment System is a video game console released in 1983 by Nintendo. Its release transformed the gaming landscape, particularly in the West, where the industry was in a large recession caused by market saturation and poor game quality (Vega 2019). Game franchises started on the NES are still popular today, from Super Mario Bros to The Legend of Zelda. The console has an 8-bit architecture, allowing for complex games in comparison to the competition at the time, for example the Atari 2600. The controller has two action buttons, A and B, a 4-directional pad and Start and Select buttons.

Much research has already been done on reinforcement learning in Atari games, with Mnih et al. (2013) finding success with a Deep Q-Networks agent. Liang et al. (2016) also showed that a Blob-PROST agent could successfully learn to play Atari games. Atari games are much simpler than NES games, as can be seen in figures 3.1 and 3.2. We wanted to show reinforcement learning can achieve results in more complex NES games. Similarly, games for the Super Nintendo Entertainment System, the NES' successor, have also been tested, however Bhonker et al. (2017) noted that because of SNES games' impressive graphics, there is often meaningless background noise that offers no information about a game's state, confusing the agent. NES games offer a sweet spot between the simplicity of Atari games, and the complexity of SNES games, so we chose these for our benchmark.

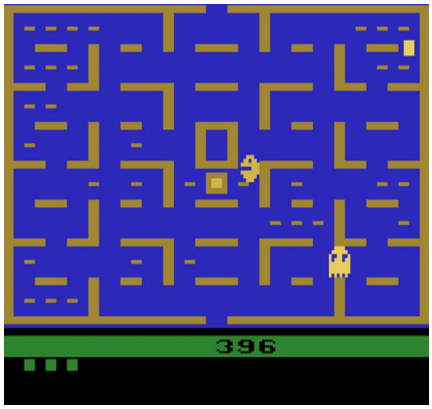

Figure 3.1: *Pac-Man on the Atari 2600. Adapted from DeMarco (2017).*

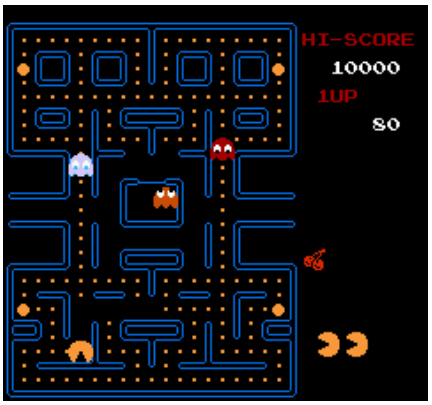

Figure 3.2: *Pac-Man on the Nintendo Entertainment System.*

3.2 Games

Five games were chosen to benchmark, with a wide range of genres to ensure different aspects of each algorithm's performances could be compared. All games chosen can be played with 1 or 2 players, but were operated in 1 player mode. This section shows the games used in the benchmarks and describes their gameplay, reward, done condition (when an episode ends) and any additional details of note.

3.2.1 Arkanoid

Arkanoid is a block breaker game where the main objective is to clear the screen of blocks by bouncing a ball towards it, not letting the ball fall to the bottom of the screen (Taito 1986). The player controls a paddle called a 'Vaus', which is used to bounce the ball up the screen to break the blocks, which are described as 'space walls'. The Vaus is moved left to right with the D-Pad, and the A button is used to fire the ball or use power-ups. Different coloured bricks give different scores, and some contain power capsules which give the user extra points and powers, for example slowing the movement of the ball, splitting the ball into three or advancing to the next level. Different bricks also have varying amounts of hits required to be destroyed, with this increasing the higher the round. Obstacles also appear, which can change the direction and angle of shots, but give extra points if they are destroyed. The reward used is 1 reward point for 1 in-game score point, and the done condition is running out of lives.

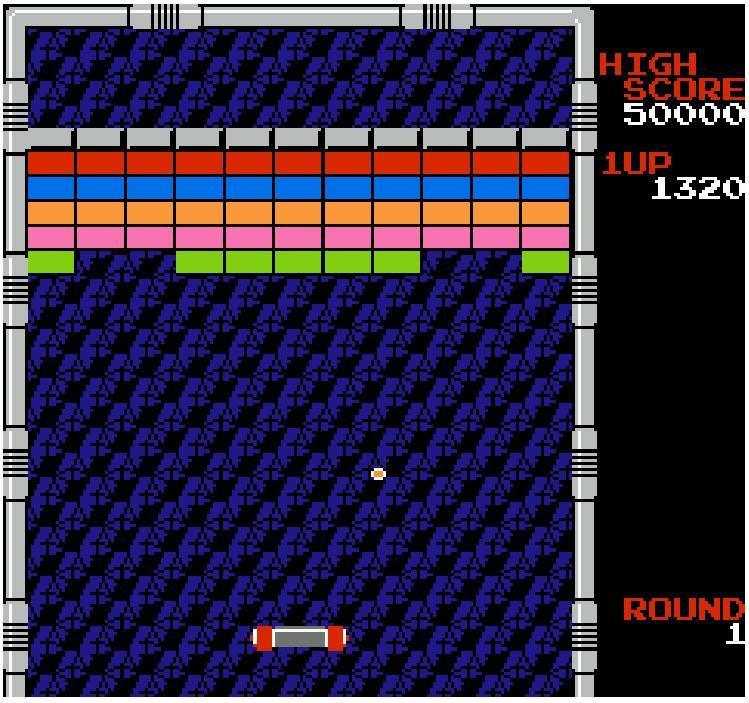

Figure 3.3: *Screenshot of Arkanoid, with the paddle at the bottom, ball in the middle and bricks at the top.*

3.2.2 Gradius

Gradius is a side scrolling shoot 'em up game (Konami 1986). The main objective is to survive through several waves of enemies while progressing through the game map. The player controls a spaceship using the D-pad, using A to shoot and B to use a power boost. The game has a power-up system where the player can collect power capsules that fill up a bar at the bottom of

the screen with a selection of rewards, like increased speed, double missiles or invulnerability. The game environment also poses hazards, such as volcanoes that can kill the player. The reward used in this game is 10 reward points for 1 point of in-game score, and the done condition is running out of lives. OpenAI (2018) states that Gradius is a good fit for reinforcement learning as it has frequent and incremental rewards for each enemy killed, so agents can learn quickly since most of the difficulty comes from fast reaction times, which is easy since the agents play the game one frame at a time.

Figure 3.4: *Screenshot of Gradius, with the player spaceship in the centre left, shooting at an enemy to its right.*

3.2.3 Pac-Man

Pac-Man is a maze chase game where the main objective is to collect all the dots on the screen while avoiding ghosts (Namco 1983). The player controls Pac-Man with the D-pad. The maze is filled with Pellets, which give Pac-Man points, and Power Pellets, which give extra points and also allow Pac-Man to eat ghosts for a short duration of time. Eating multiple ghosts in a single Power Pellet's duration awards more points. Twice in each maze, fruits appear which give a lot of bonus points, increasing as the game progresses. Once Pac-Man has eaten all the pellets in the maze, he is brought to a new maze. The layouts of each maze are identical, however with each new maze the faster the game plays, and the power pellets last for less time. The reward used for this game is 10 reward points for 1 in-game score point, and the done condition is running out of lives.

3.2.4 Space Invaders

Space Invaders is a fixed shooter game (Taito 1985). The player controls a cannon, where they can move it horizontally across the bottom of the screen and use A or B to shoot. Rows of enemies steadily move down the screen, attacking the player. The player can hide behind a row of shelters, which get destroyed by the enemies or player shooting them. Once all enemies are

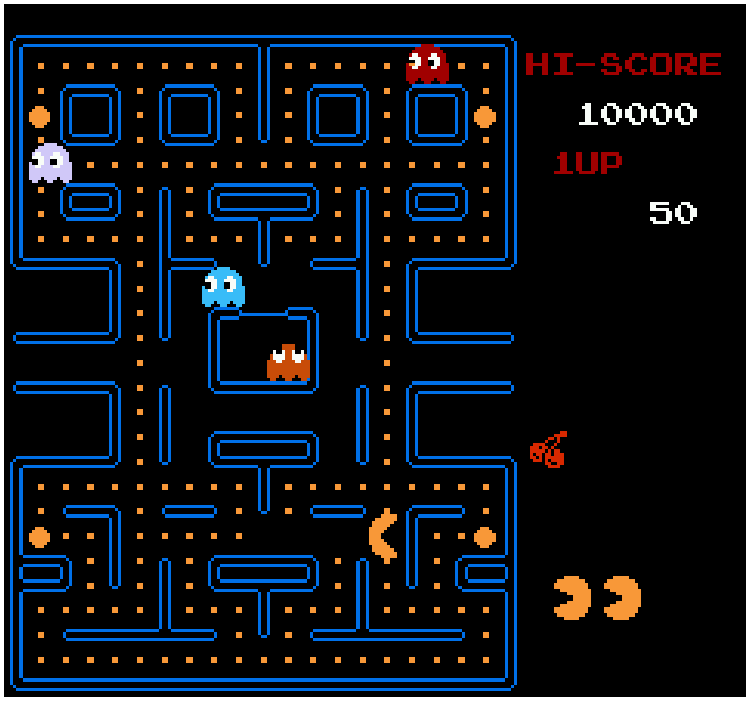

Figure 3.5: *Screenshot of Pac-Man, with Pac-Man at the bottom right, ghosts at the top and the maze filled with pellets.*

defeated, a new and faster wave spawns, repeating until the player runs out of lives. Different enemies give different amounts of points when killed. Occasionally a special enemy will appear and the player can kill it for bonus points. The reward used is 10 points of reward for 1 in-game score point, and the done condition is running out of lives.

3.2.5 Super Mario Bros.

Super Mario Bros. is a 2D side-scrolling platforming game where the main objective in each level is to reach the flagpole at the end while avoiding enemies and obstacles (Nintendo 1985). The player controls Mario with the D-pad, and can run and jump with A and B. The game is divided up into 8 worlds, each with 4 stages, with the 4th containing a boss fight. Super Mario Bros. contains an in-game score, with the player earning points by collecting coins, defeating enemies, breaking bricks and reaching the end of the level, with more points being given for a fast completion. ? blocks at certain points of the levels contain power ups, either increasing Mario's size, giving him a fire power, or granting invincibility for a short duration.

A custom reward is defined, with the main reward received by Mario's scroll position, and extra awarded for gaining score points, and more for reaching the flag pole. The custom reward was defined as rewarding only the in-game score does not incentivise the agent to properly complete the level, instead only defeating enemies and collecting coins. In training, the agent loops through Level 1-1, repeating it if it reaches the flagpole. The done condition is running out of lives.

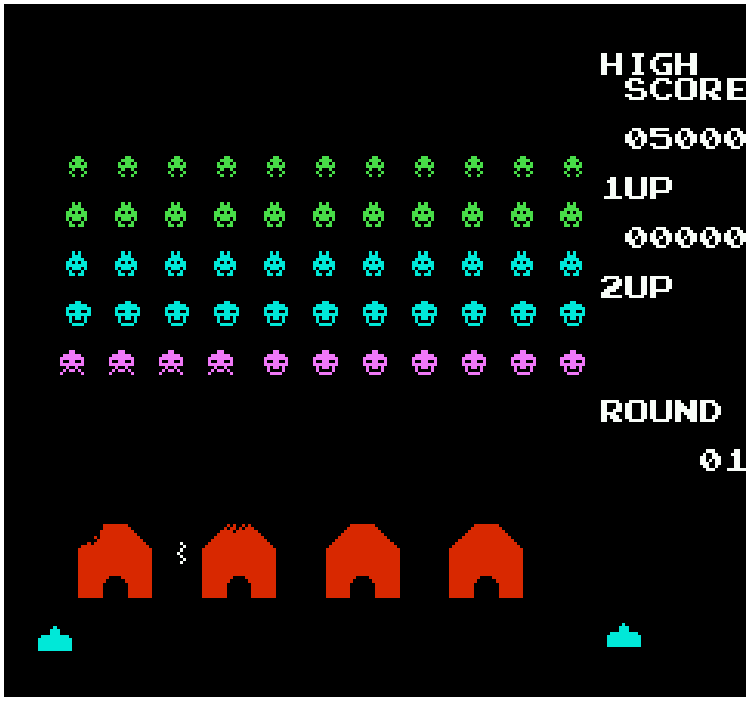

Figure 3.6: *Screenshot of Space Invaders, with the player spaceship at the bottom right and enemies at the top, shooting at the player.*

3.3 Gym Retro

We decided to use Gym Retro, a Python platform for reinforcement learning research in games. OpenAI (2018) released this tool to allow for easy study of how reinforcement learning algorithms generalise in-games. While we will not be looking into generalisation, their platform includes the data needed for thousands of games to be played by a reinforcement learning agent, such as reward and game variable information. Games from multiple platforms including Sega Genesis and Super Nintendo Entertainment system are available, but we focused on Nintendo Entertainment System games. Gym Retro offers a tool to integrate games yourself (OpenAI 2019*a*), allowing you to create save states and look at memory locations to assign game variables. However, it was easier to use the games that Gym Retro already provided with data and rewards given.

We looked through the Gym Retro dataset to find a wide variety of games to test the reinforcement learning agents on, performing some initial training runs to see how agents did in the games. A few games achieved little or no learning and so weren't suitable for this benchmark: agents managed to get through Kirby's Adventure levels with some success but could not find the door to exit the level, and failed to make any significant progress in Donkey Kong. After trying a few games, we narrowed the selection down to the final five.

3.4 Measures

There are many measures of a reinforcement learning algorithm's performance. Machado et al. (2017) provides an overview of common performance measures used in the Arcade Learning Environment, the Atari 2600 platform for reinforcement learning. We will discuss some of the common statistics used and what we will be using for this benchmark.

Evaluation after training is where agents are trained for a period, then their score evaluated in a

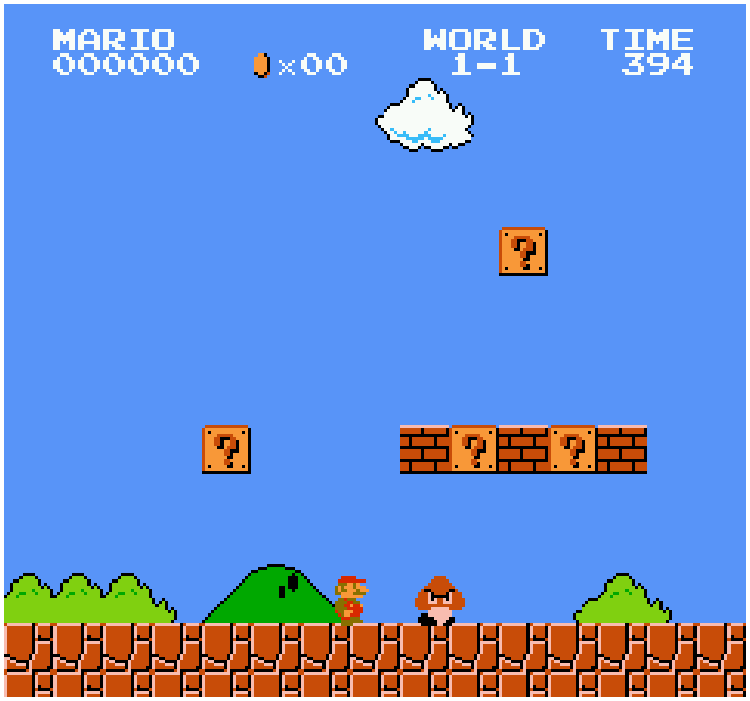

Figure 3.7: *Screenshot of Super Mario Bros. showing Mario in the centre, an enemy to his right, with bricks and ? blocks above them.*

number of episodes without learning. Bellemare et al. (2013) uses this approach, using five games for a training set and choosing randomly from 50 games for the test set. This approach has some disadvantages, such as agents behaving differently during the training and evaluation periods leading to misleading performance. There is also a greater implementation cost, with twice the number of games needing to be implemented.

Evaluation of the best policy, like evaluation after training, trains agents for a fixed training period, but then evaluates the best policy in a number of evaluation episodes without learning. Wang et al. (2016) also uses the Arcade Learning Environment, training on a set of 57 Atari 2600 games, finding the best policy and then evaluating this in a number of episodes with no training. This method has the same downsides as evaluation after training in that performance could be misleading, and it also does not show the stability of the agent's progress, for example if it learns quickly but then has a large drop in performance.

Area under the learning curve was first proposed in Stadie et al. (2015), where the agents were trained on 14 games from the Arcade Learning Environment, with the area under their learning curves then computed using the trapezoid rule. The area is then normalised by 100 times the maximum game score achieved. This metric was suggested as it gives a holistic view of the agent's learning rate and does not require running the games for 1000 epochs, however the metric, like evaluation of best policy, does not give an idea of the stability of the agent.

Many of these measures are aimed at looking at an agent's performance during training but don't give an accurate idea of its final performance, and also don't indicate an algorithm's stability. We will present the learning curves for each algorithm per game, the time taken to train two million timesteps and the final scores achieved by a trained agent.We will also show the reward over time and how it differs in each algorithm. These metrics allow us to analyse how stable each algorithm is, how quickly it learns, and how well it can perform in a game after training. With our focus on video games, we believe final scores are the most important metric, with the

goal of agents in video games usually being getting a higher score or successfully completing a task. Stability and time to train are important too: if an algorithm can learn well but experiences random dips in performance or takes a very long time to learn, it may be of less practical use than stabler, quicker algorithms that perform worse.

3.5 Chapter Summary

This chapter has detailed some of the design decisions behind our benchmark. We discussed our reasons for choosing the Nintendo Entertainment System games console, noting that its games are complex, yet not too complicated for an agent to understand, and our use of Gym Retro to emulate the NES. We then showcased the games we will be using for this benchmark, showing screenshots, describing gameplay and defining rewards. Finally, we gave an overview of different metrics used for benchmarking reinforcement learning algorithms and what will be used in this paper.

4 Implementation

This chapter will discuss the implementation part of the project, showing how our benchmarking code worked, from the Gym features used, to an overview of the code itself. We will then show the setup used for all training and testing of the reinforcement learning algorithms.

4.1 Gym

As mentioned in Section 3.3, we are using Gym Retro to create the environments for our agents to play in. This section will go over some of the Gym features we used to implement our benchmarks.

4.1.1 Overview

Gym Retro creates Gym environments, which are used for comparing reinforcement learning algorithms (OpenAI 2016). The agent's main way of interacting with the environment is with the step() function, which returns four values:

- `Observation`, an object representing an observation of the environment. In Gym Retro, this can be either an RGB image, or RAM values. RAM values are smaller than RGB images and allow for more direct observation. These allow for faster computation, however RGB images are more detailed and are the default option. All our game training and testing is done with an RGB image observation.
- `Reward`, a float showing the amount of reward the previous action gave. The agent's goal is always to increase this reward.
- `Done`, a Boolean that decides if the environment should be reset again. In most games, this is met by dying, completing a level, or running out of time. If done is True, this means the environment is reset.
- `Info`, a dictionary showing game info. This is dependent on the game data, for example it may contain the current lives and score, or be much more detailed, showing the current level, co-ordinates, types of enemies on screen, and time left.

4.1.2 Retro Integration

The `Retro Integration` folder contains all the data needed for each game to run. This is different for each game, but generally contains:

- `Data` contains all the game info. This can be any game variable from lives to score to the player's position on screen.
- `State` defines where the game starts and with what settings. This allows the agent to skip the start-up screen or start at a later level.
- `Metadata` is data from the Gym Retro implementation. This may be default states, recommended hyper-parameters or tags indicating how complete the implementation is.
- `ROM` files are not included in Gym Retro, but are essential for running the game and can be found by their SHA which is included.

- `Scenario` defines the game reward and done condition. This could be simple as 1 reward for 1 in-game score point and done when lives are zero, or a completely custom reward that penalises the agent given certain conditions.

4.1.3 Actions

Every time the agent produces an action, this is encoded as binary vector representing the button combinations on an NES controller, where 0 means the button is not pressed and 1 means it is. The Nintendo Entertainment System controller has 8 buttons: Up, Down, Left, Right, Start, Select, A and B. It is common for games to use Start and Select only for the title screen and pausing, so most gameplay only uses the directional buttons and A and B. In our code, the action space can be customised, with three main choices ('noop' represents no buttons pressed):

- RIGHT_ONLY : {['noop'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B']}
- SIMPLE_MOVEMENT:{ ['noop'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B'], ['A'], ['left']}
- COMPLEX_MOVEMENT:{['noop'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B'], ['A'], ['left'], ['left', 'A'], ['left', 'B'], ['left', 'A', 'B'], ['down'], ['up']}

Some games are playable with right only and simple movement, like Super Mario Bros., however some require complex movement, like Pac-Man. For our training and testing, all games were played with the complex movement action space.

4.1.4 Custom Reward

A custom reward is also implemented. This takes in the reward set in the scenario file and transforms it slightly, adding the current score and dividing by 40. For Super Mario Bros., the reward is also incremented by 50 and divided by 10 if the flagpole was reached that episode. This prevents the reward from being too high, which can make the agent choose a less than optimal policy.

4.1.5 Custom Skip Frame

As is customary in reinforcement learning solutions (Mnih et al. 2015), our implementation contains a custom frame skip. This allows the agent to only see and selects actions on every 4th frame, its last action repeated on the skipped frames. This allows the agent to play 4 times more games without significantly increasing the run time, since this requires less computation. This is why we refer to our unit of time in the benchmark as timestep: it represents around $\frac{1}{15}th$ of a second, instead of the $\frac{1}{60}th$ it would be without.

4.2 Code Overview

For our implementation of the A3C and PPO agents, we used existing codebases written by Nguyen (2019-2020). These implementations use Python and PyTorch to train and test the models. They use the Python package gym-super-mario-bros, which provides an OpenAI Gym environment for the games Super Mario Bros. and Super Mario Bros. 2 (The Lost Levels). Since we needed to implement more than one game, we converted the code to use Gym Retro instead, which supports all the games we used. This conversion to Gym Retro was based on a PPO conversion already written by Aragon-Camarasa (2020).

The code layout is similar for both algorithms. The purpose of each file will be explained in Sections 4.3 and 4.4. The structure of the file directory is like so:

```
Arkanoid
├── A3C
└── PPO
    ├── src
    │   ├── env.py
    │   ├── helpers.py
    │   ├── model.py
    │   ├── process.py
    │   └── retrowrapper.py
    ├── trained_models
    ├── test.py
    └── train.py
├── Gradius
├── Retro-Integration
└── (...)
```

The benchmarking set up is split into two main sections: training and testing. Training was performed first, then testing.

4.3 Training

The training process involves these files: `env.py`, `helpers.py`, `model.py`, `optimizer.py`, `process.py` and `retrowrapper.py`. In this section we will explain each file's use:

- `env.py` sets up the Retro environment for the game. This is where the actions, custom reward and custom frame skip are defined, as wrappers around the environment.
- `helpers.py` contains some helper functions for the the Retro environments. It defines the controller space, mapping binary values to discrete buttons. For Super Mario Bros., it contains functions that detect if Mario has reached the flagpole and the level is over, which indicates that the environment should be reset.
- `model.py` contains the neural network implemented to control learning. Its dimensions depend on the number of inputs allowed and the action space. It contains layers for actors and critics.
- `optimizer.py` is a part of A3C only, where it defines a custom version of the PyTorch Adam optimizer (Kingma and Ba 2017).
- `process.py`'s role differs slightly in A3C and PPO. In A3C, it contains the code for both training and test processes. The test processes render the environment, showing the training process in game. In PPO, it contains only the test process as the training is done in `train.py`.
- `retrowrapper.py` is a wrapper used in PPO, to spawn several retro environments at once.
- `train.py` differs in A3C and PPO. For both A3C and PPO, it takes in hyper-parameters and options such as save paths. It also sets up CUDA acceleration if it is available. For A3C, it then creates the multiprocessing processes, one test and a variable amount of training depending on how many the user wants, each with their own environment. For PPO, the separate test process is created and launched in train.py and then the training also happens in train.py.

4.4 Testing

The training process is simpler than training, being started from `test.py`. It looks for a trained model in the game's `trained_models` directory, then sets up a game environment and plays

it, until the done condition is met. An example of the done condition is running out of lives. It then outputs the maximum score achieved during that run and exits.

To test if our A3C and PPO agents were actually learning, we also implemented a random agent for comparison. Adapted from `random_agent.py` in the Gym Retro repository (OpenAI 2019*b*), the agent chooses a random action from the available action space every step. The environment and states used are the same as the ones used when training A3C and PPO agents. The random agent runs until the done condition is met, which differs in each game but is usually met by dying, running out of time, or finishing a level. Once the agent is finished, it outputs its highest score achieved in the run, which we took for comparison with the trained agents. If a trained agent can beat a random agent, we take this to mean that the agent has successfully learned.

We also implemented a process to automate the processing of data, taking the CSVs that were output during training and testing, turning them into Pandas dataframes and then producing graphs from them, and calculating the time to train and final scores.

4.5 Experiment Setup

The software used provided a large variety of hyper-parameters that could be adjusted for each algorithm. Modifying these for each game could lead to greater results for this comparison each game used the same hyper-parameters per algorithm to keep the tests fair. All benchmarking was performed on a PC running Ubuntu 20.10 with a Ryzen 5 5600X processor, 32GB of 3600 MHz DDR4 RAM and a RTX 3070 graphics card.

4.5.1 A3C Hyper-parameters

We use the following hyper-parameters for all A3C training runs:

Hyper-parameter	**Value**
Action Type	Complex
Learning Rate	1e-4
Reward Discount Factor	0.9
Generalised Advantage Estimation	1.0
Entropy Coefficient	0.01
Number of Local Steps	50
Number of Global Steps	2e6
Number of Processes	4
Max Actions	200

Table 4.1: *Hyper-parameters used when training A3C models.*

Action type is the action space used, according to those defined in Section 4.1.3. Learning rate controls how much the weights of the neural network are adjusted with respect to the loss gradient (Zulkifli 2018). A low learning rate means training is very slow, whereas a high learning rate means the agent may converge at a sub-optimal solution. Reward discount factor corresponds to γ in our definition of a value function, where it discounts the impact the future has on decision making. Generalised advantage estimation (Schulman et al. 2018) is used in policy gradient algorithms to estimate the value functions. Entropy coefficient is used to define the amount of entropy in calculations, which encourages exploration and avoids getting stuck in local optima (Argerich 2020). Number of local steps is how many steps can be used in one episode, and number of global steps is how many are used in total. Number of processes is how many training processes are used. Max actions is how many repetition steps are used in the test

phase.

4.5.2 PPO Hyper-parameters

We use the following hyper-parameters for all PPO training runs:

Hyper-parameter	Value
Action Type	Complex
Learning Rate	1e-4
Reward Discount Factor	0.9
Generalised Advantage Estimation	1.0
Entropy Coefficient	0.01
Clipped Surrogate Objective	0.2
Batch Size	16
Epochs	10
Local Steps	512
Global Steps	2e6
Processes	4
Max Actions	200

Table 4.2: *Hyper-parameters used when training PPO models.*

Many of the hyper-parameters used in PPO are the same as A3C. Clipped surrogate objective is used in PPO to minimise large gradient updates. The batch size defines how large a stochastic gradient descent is used in PPO. The epochs specifies how many epochs this stochastic gradient descent is used on.

4.6 Chapter Summary

This chapter has detailed the choices made in the implementation of our benchmarking. Starting with an overview of Gym, we showed some features we were using, including actions and observations. We then discussed why we were using custom frame skips and custom rewards. We then gave an overview of our code, explaining how it is split into training and testing sections, and the use of each file involved. Finally, we showed our experiment setup, from hardware to hyper-parameters.

5 Results

In this chapter we perform the benchmark, showing for each algorithm the learning curve, time taken to train, cumulative reward over time and the high scores attained by a trained model. We then analyse the results and offer some conclusions on the overall performance.

5.1 Training

Three models were trained for each algorithm in each game. Each model was trained for 2 million timesteps, which took around 2 and a half hours. Every model was trained using the same seed. The same hyper-parameters were used for each run to ensure a fair comparison.

5.1.1 Arkanoid

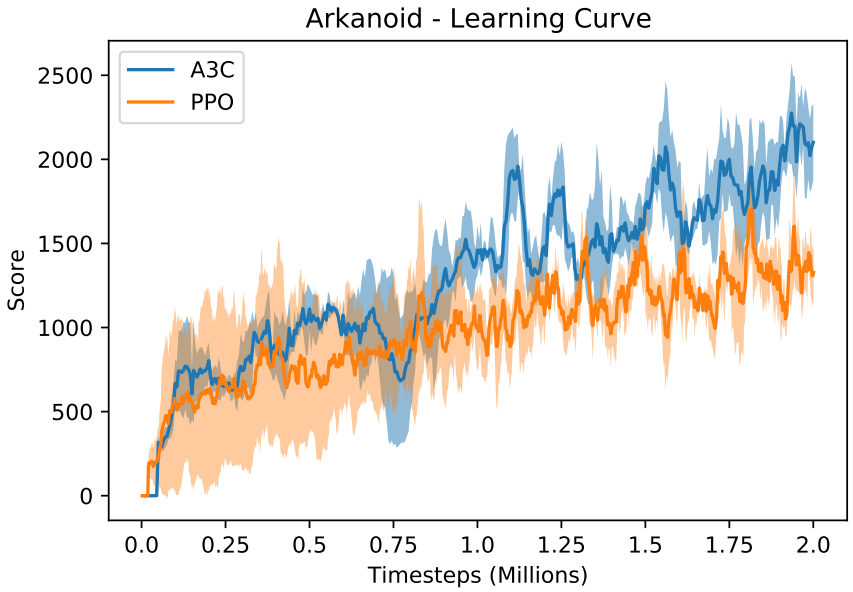

Figure 5.1: *Learning curve for Arkanoid. The score represents an average of the last 10 episodes. Averaged over three runs, the shaded area represents standard deviation.*

For Arkanoid, as can be seen in Figure 5.1, the A3C agent managed to reach much higher scores during training than the PPO agent did. At the start, the PPO scores have a very high variation in score, however this narrows as training goes on.

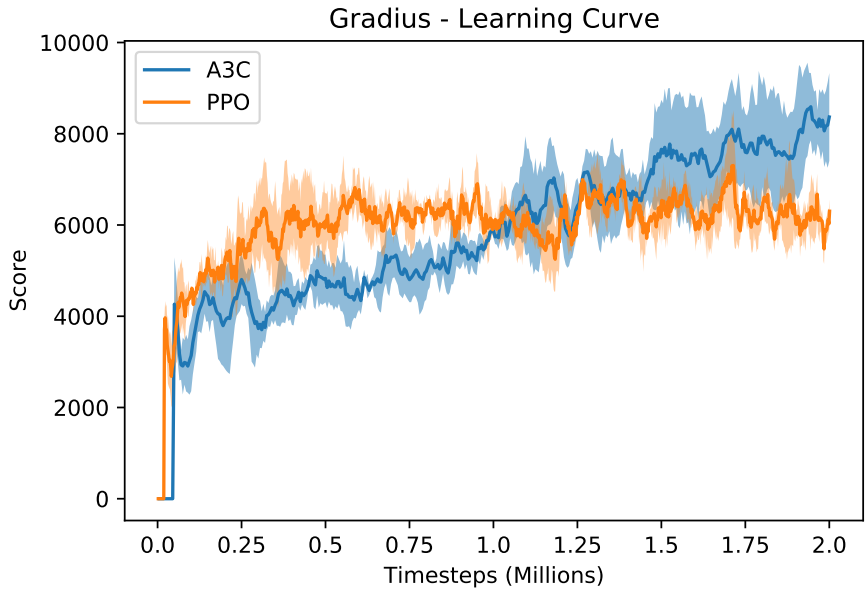

Figure 5.2: *Learning curve for Gradius. The score represents an average of the last 10 episodes. Averaged over three runs, the shaded area represents standard deviation.*

5.1.2 Gradius

As with Arkanoid, for Gradius the A3C agent reached higher scores than the PPO agent in training, seen in Figure 5.2, however their scores are quite close. The PPO agent reaches a higher score quickly initially, however levels out from there and doesn't make much more progress, whereas the A3C agent steadily reaches a higher score as training continues.

5.1.3 Pac-Man

PPO reaches the higher score during training for Pac-Man, as shown in Figure 5.3. Interestingly, the PPO agent seems to reach its peak score at around 1.5 million timesteps, and falls slightly from then on, however it is almost always above the A3C agent's score.

5.1.4 Space Invaders

The learning curve for Space Invaders, Figure 5.4 shows that the PPO agent consistently reaches a higher score than the A3C agent, however there is a high variation at all times, signifying that each run gave drastically different scores during training. The A3C agent seems to be catching up at the end, and perhaps with more time given would eventually beat the PPO agent.

5.1.5 Super Mario Bros.

The Super Mario Bros. learning curve is an interesting case, as can be seen in Figure 5.5. Both agents' scores are quite low until 0.5 million timesteps, when A3C shoots up. This is because the A3C agent has managed to reach the flagpole at the end of the level, granting lots of in-game points. The PPO agent doesn't manage to reach this, so its scores are very low in comparison. It should be noted that because of the custom reward defined for Super Mario Bros., the agent is rewarded more for moving right than gaining score directly like by killing enemies or breaking

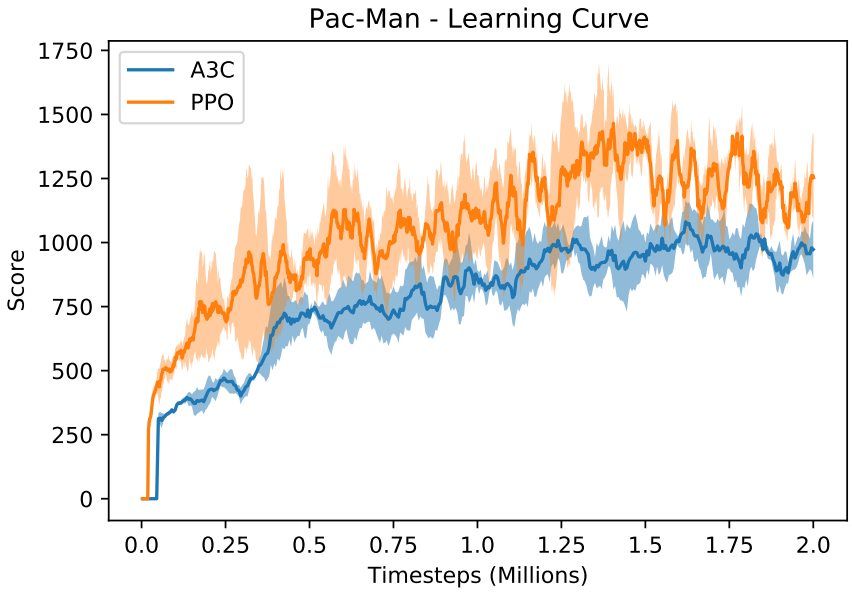

Figure 5.3: *Learning curve for Pac-Man. The score represents an average of the last 10 episodes. Averaged over three runs, the shaded area represents standard deviation.*

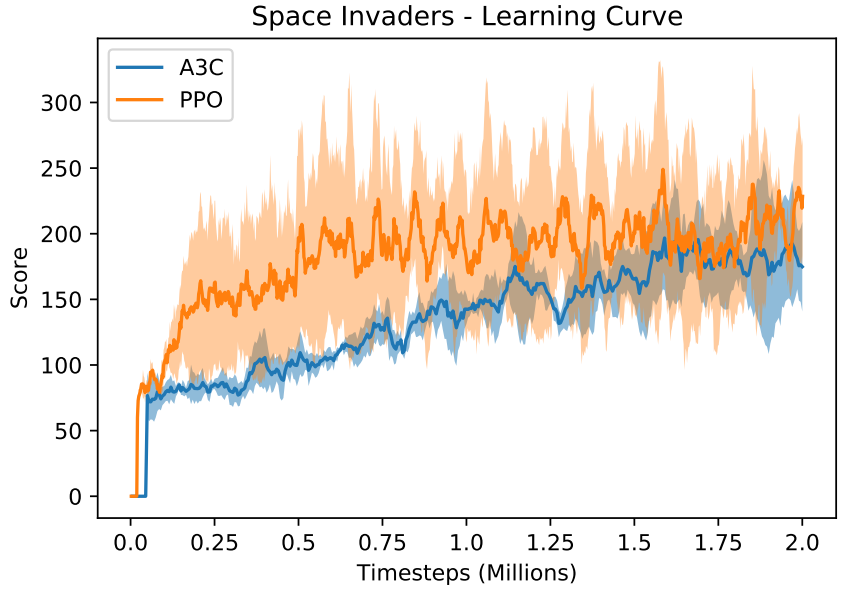

Figure 5.4: *Learning curve for Space Invaders. The score represents an average of the last 10 episodes. Averaged over three runs, the shaded area represents standard deviation.*

blocks. This is meant to encourage the agent to focus on finishing the level and reaching the flagpole sooner, however in PPO's case it does not seem to have worked.

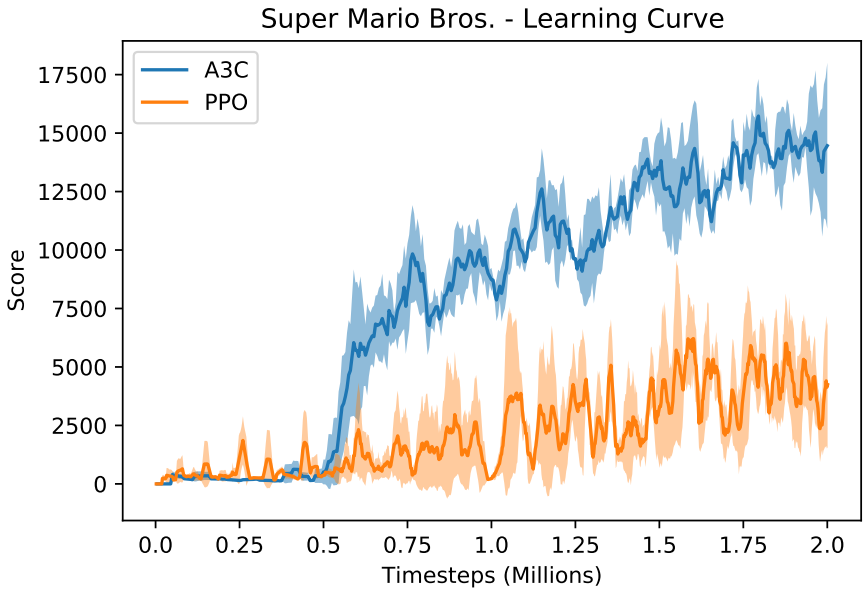

Figure 5.5: *Learning curve for Super Mario Bros. The score represents an average of the last 10 episodes. Averaged over three runs, the shaded area represents standard deviation.*

5.1.6 Training Speed

We now look at the training speed of each algorithm. Each algorithm was run for 2 million timesteps, but this took a different amount of time each run, depending on the game and algorithm.

	A3C	**PPO**
Arkanoid	145±1.6	167±3.6
Gradius	151±1.4	166±2.7
Pac-Man	150±1.0	161±2.8
Space Invaders	148±1.5	164±1.6
Super Mario Bros.	146±1.5	167±3.3
Average	148±2.9	165±2.9

Table 5.1: *Minutes to complete 2 million timesteps in training, averaged over three runs.*

We show in 5.1 the average time taken to reach 2 million timesteps in each training run. Per algorithm, the games take about the same time to finish. This was a somewhat surprising result as some games could take longer per episode than others, for example it takes longer to run out of time in a level of Super Mario Bros than it does to drop the ball in Arkanoid. Across all games, A3C was consistently faster than PPO, finishing almost 20 minutes faster than PPO on average. This is due to the asynchronous aspect of A3C.

5.1.7 Rewards

We will now show the cumulative reward over time for A3C and PPO for some games. The cumulative reward graphs for the games not shown here can be found in the Appendices.

We can see that for all games, the PPO reward increases in much smoother increments due to

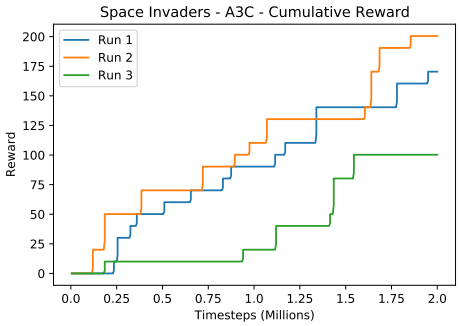

Figure 5.6: A3C Cumulative Reward for Space Invaders.

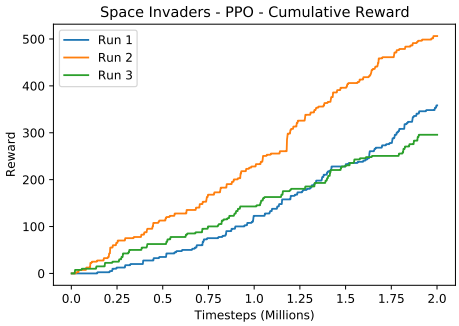

Figure 5.7: PPO Cumulative Reward for Space Invaders.

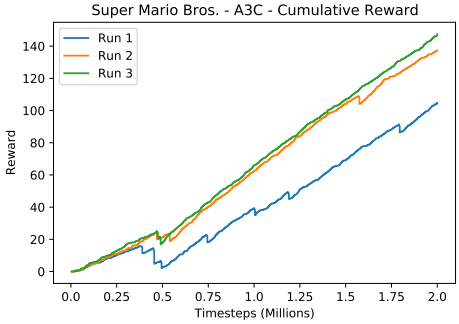

Figure 5.8: A3C Cumulative Reward for Super Mario Bros.

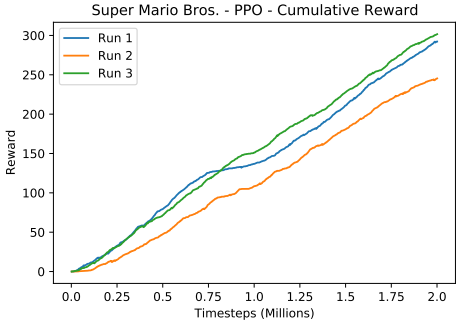

Figure 5.9: PPO Cumulative Reward for Super Mario Bros.

the policy clipping. In Super Mario Bros., we can see the cumulative reward even fall at times in A3C, showing periods of negative reward. With PPO's rewards being smoother, we can see that it is less likely to fall in performance and stability.

5.1.8 Conclusion

During training, A3C reached higher scores than PPO in 3 out of 5 games, with Super Mario Bros being the most significant difference. PPO also showed the highest variance overall, with its standard deviation often high. PPO usually learns quickly at the start, but then levels off, with its average score increasing slowly. A3C tends to start off more slowly but steadily increases in score over time. A notable exception is Super Mario Bros. where the A3C score rapidly increases when the agent starts to reach the flagpole at the end of the level, which gives much more score than other ways of increasing score like killing enemies. When it comes to time taken to train, A3C beats PPO again, being on average 17 minutes quicker. A3C took less time to reach 2 million timesteps in every game. We also showed the reward graphs, with PPO's cumulative reward over time progressing much more smoothly than A3C's, due to its policy clipping preventing large changes in policy.

Overall, during training, A3C looks to be the superior algorithm, especially in terms of speed. It also has better stability, with its scores varying less during training, though its reward does drop at times and is unstable compared to PPO.

5.2 Testing

We present the trained models' high scores in each game, along with a random agent for comparison, to see if any learning did occur. The results shown in the following bar charts are the average of the highest score each model achieved during testing, with error bars representing the standard deviation. A full table of scores can be found in Table 1.

5.2.1 Arkanoid

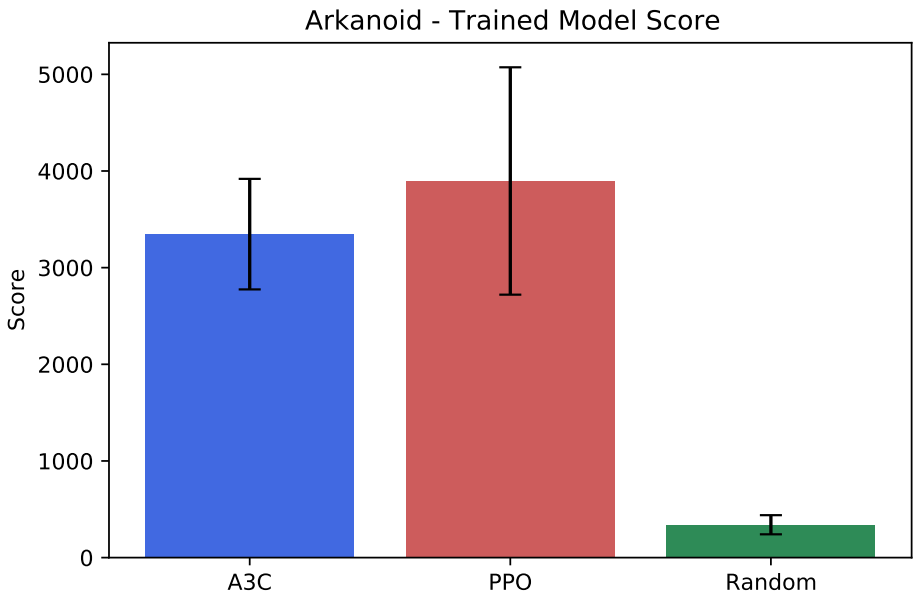

Figure 5.10: *Final test scores for Arkanoid. Averaged over three runs, error bars represent standard deviation.*

Figure 5.10 shows the high scores for Arkanoid. Despite it getting lower scores in training, PPO managed a higher score than A3C, however it should be noted that neither got very far in each level, only managing to clear the first row of bricks. PPO has a higher range of scores, with some scoring lower than the A3C average.

5.2.2 Gradius

For Gradius, shown in 5.10, both PPO and A3C failed to beat the random agent, signalling that not much meaningful learning happened. This was surprising as OpenAI (2018) mentioned this as a game that trains well due to its dense rewards. A key feature of Gradius is its 'Power Up bar', which offers accumulating power ups which can be of great benefit. The agents did not ever use these power ups, which could have massively improved their score. With more time to train, perhaps the A3C and PPO agents could eventually learn and perform better than the random agent.

5.2.3 Pac-Man

The Pac-Man test scores are seen in 5.12. PPO does better than A3C here, even though the A3C agent was the only one to use Power Pellets to defeat the ghosts in any run. In training, the PPO

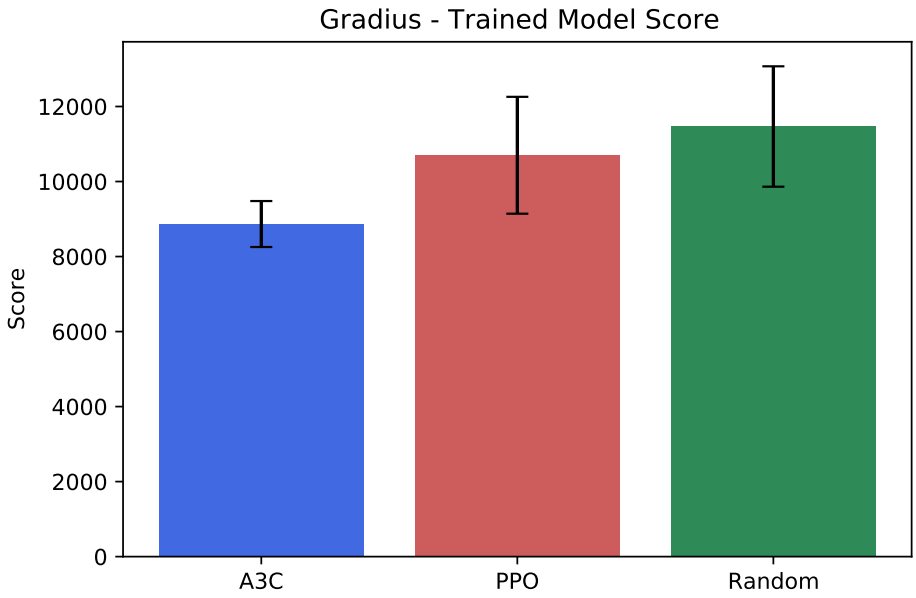

Figure 5.11: *Final test scores for Gradius. Averaged over three runs, error bars represent standard deviation.*

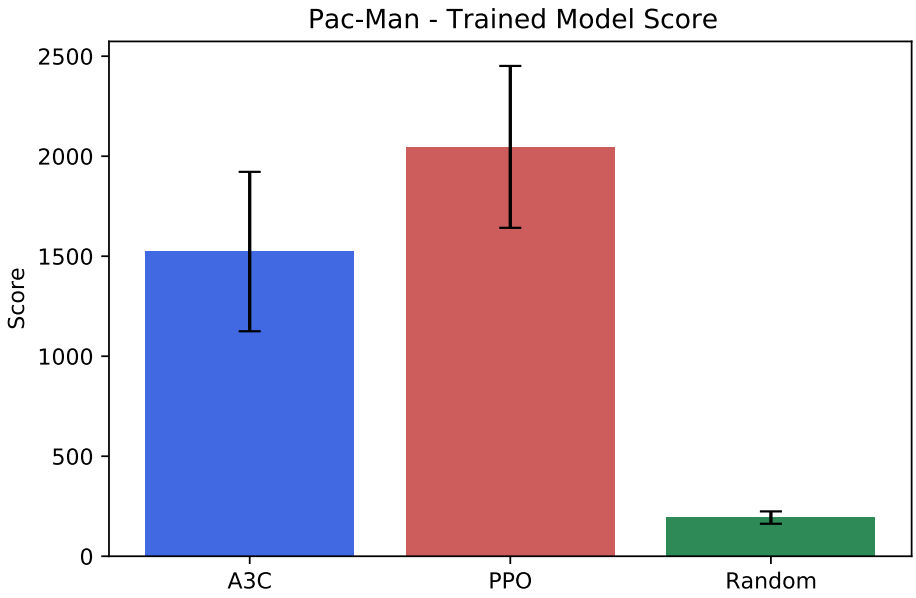

Figure 5.12: *Final test scores for Pac-Man. Averaged over three runs, error bars represent standard deviation.*

agent would often get stuck running against the maze walls, so it was expected to perform badly, however it still managed to out-perform A3C.

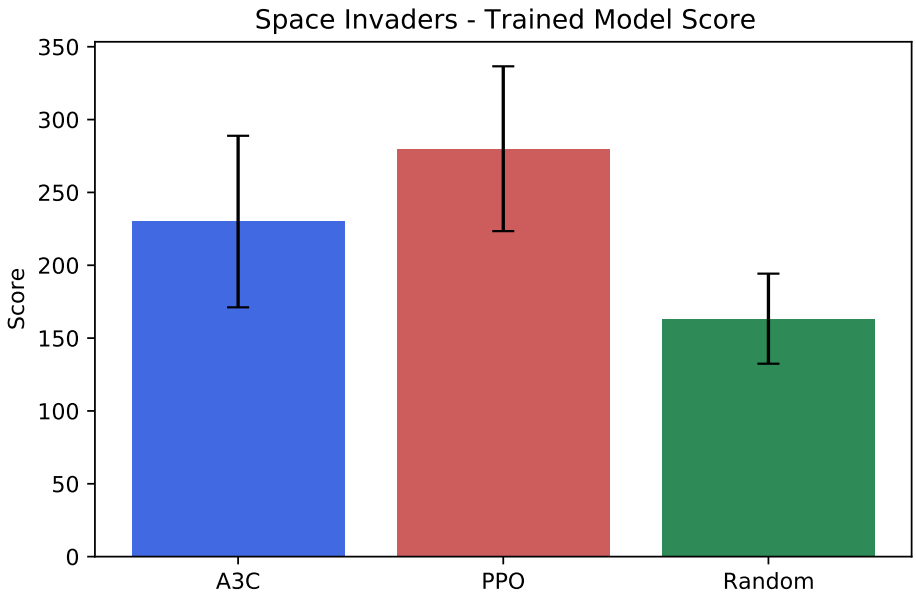

Figure 5.13: *Final test scores for Space Invaders. Averaged over three runs, error bars represent standard deviation.*

5.2.4 Space Invaders

Again, PPO did better than A3C in Space Invaders, as Figure 5.13 shows. Neither algorithms performed very well compared to the random agent. They would often die quickly, failing to dodge the enemies, or shooting their own shelter so they were defenceless.

5.2.5 Super Mario Bros.

Both agents performed well compared to the random agent in Super Mario Bros., though variance was high because a large amount of score comes from reaching the flagpole at the end of Level 1-1 and progressing onto Level 1-2. Reaching the flagpole gives around 10,000 score alone, and also the further along the level more opportunities to get score arise. If the model did not reach the flagpole, they usually got stuck at some point and died. At the end of Level 1-2, the agent has to direct Mario through a pipe to reach the flagpole, and all agents got stuck here if they made it to Level 1-2 at all. This is perhaps because this did not happen in Level 1-1, so they did not know what to do.

5.2.6 Conclusion

PPO out-performed A3C in all games except for Super Mario Bros., where A3C got a significantly higher average score because it managed to reach the flagpole, gaining lots of score. Both algorithms failed to beat the random agent in Gradius. We can conclude that PPO is the better algorithm when it comes to performance.

5.3 Evaluation

We set out to compare the algorithms in three aspects: their learning curves, their final high scores and their time taken to train. We also looked at their cumulative reward over time, however this

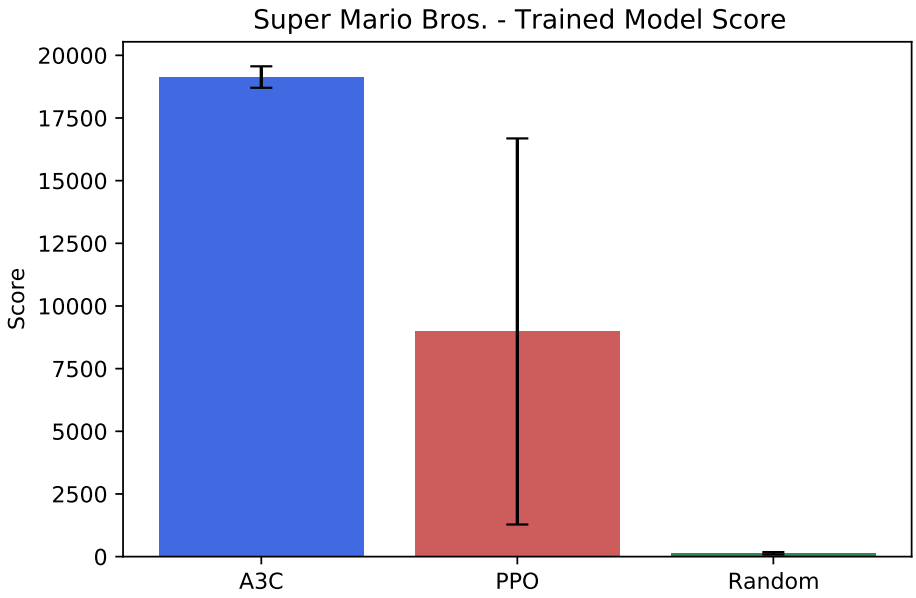

Figure 5.14: *Final test scores for Super Mario Bros.. Averaged over three runs, error bars represent standard deviation.*

doesn't tell us very much about actual performance. Looking at learning curves, we found that PPO often had a higher standard deviation, with results varying more than A3C and being less stable overall. It tended to learn quickly at the start then level off, while A3C steadily learned over time and reached higher scores in most games. When it came to time to train, A3C was much faster, being on average 17 minutes faster to reach 2 million timesteps. We then showed the cumulative reward over time, with PPO's being smoother than A3C's which dropped at times and changed in large increments at others. However, in the final high score comparison, PPO beat A3C in all games but Super Mario Bros., though both algorithms did fail to beat the random agent in Gradius. Overall, we conclude that PPO is the better algorithm when it comes to pure final performance. However, A3C has a better speed and stability, with less variation in score.

We believe this benchmark overall is successful, and offers opportunity for extension be it through more algorithms or more games. NES games are a great platform for benchmarking reinforcement learning algorithms, with each game providing a challenge for an agent that allows for rich comparisons to be made between algorithms. An aspect it could be improved in is the time allowed to train the models - 2 million timesteps is quite low in comparison to similar benchmark work, and would allow agents more time to explore the environments and perfect their policies. More powerful hardware could also be used - A3C and PPO scale well across multiple machines or GPUs, as seen in Stooke and Abbeel (2019). This not only increases the performance of the agents but lowers the time needed to train.

Another way the benchmark could be improved is change - most of our agents did not get far into the game in training. This is problematic as it does not show how they react to change - for example a change in level layout or different enemies. This could also be demonstrated by testing their ability to generalise, for example training an agent in one game or level and testing it in another.

Hyper-parameter tuning could also be performed, to improve agent performance further. Mahmood et al. (2018) showed that PPO is sensitive to hyper-parameter tuning, and we just used the same hyper-parameters for each run. Changing these could drastically alter performance, for

better or for worse. An example could be learning rate - agents may get stuck during training in places they would not if the learning rate was different, prompting the agent to look for better solutions.

5.4 Chapter Summary

This chapter presented the results of the benchmarks. For each game, we showed the learning curves, which saw A3C get better scores during training in most games, and PPO showing some high variance in scores. We then showed how long each algorithm took to reach 2 million timesteps per game, where A3C was on average 17 minutes faster. We also showed the cumulative reward over time, looking at how PPO's was a much smoother progression than A3C. However, the final test scores had PPO doing better in nearly every game. From this we concluded that PPO is the better reinforcement learning algorithm for performance, however A3C offers better speed, stability and higher scores during training, but this does not translate into better final scores.

6 Conclusion

This concluding chapter provides a summary of the Benchmarking Reinforcement Learning Algorithms in NES Games project, offers some ideas for future work and reflects on the project as a whole.

6.1 Summary

This project set out to benchmark reinforcement learning algorithms in NES games. Reinforcement learning is a machine learning paradigm where an agent interacts with an environment based on the consequences of its actions, wanting to increase a numerical reward over time. Reinforcement learning has seen strong results in video games, with agents being able to learn how to play a video game without any input and eventually beat human players. We wanted to benchmark two state-of-the-art reinforcement learning algorithms, A3C and PPO, and compare them using a set of Nintendo Entertainment System games to analyse their advantages and disadvantages. To do this we used Gym Retro, a Python platform that supports reinforcement learning in various retro video games. We used the NES games Arkanoid, Gradius, Pac-Man, Space Invaders and Super Mario Bros. to set out a wide variety of tasks for the reinforcement learning algorithms to try. After training three agents per algorithm for each game, we ran the trained models in each games to see what high scores they could accomplish. We also looked at the agents' learning and reward curves and how long they took to complete 2 million timesteps in training. Overall, PPO was able to get the highest scores in more games, however A3C trained faster and had less variance in its training scores.

On the whole, we conclude that PPO is the more successful algorithm in terms of pure performance, however A3C is still worth using if speed and stability are important. This is in line with similar work's benchmarking results, and also OpenAI et al. (2019) choosing PPO to defeat human champions at Dota 2, an eSports game, because of its training efficiency and effectiveness. Both A3C and PPO are incredibly powerful reinforcement learning algorithms that can have great results when used in video games, but we found PPO to be superior in most tasks.

6.2 Future work

This work could be improved upon in a few ways. More algorithms could be compared, specifically non-policy gradient algorithms. There are two main types of reinforcement learning algorithms: policy gradient algorithms and Q-learning algorithms. Q-learning methods learn the action value function, where a value is assigned to each action given a state. (Watkins and Dayan 1992). Both being policy gradient algorithms, A3C and PPO are quite similar, so it could be beneficial to benchmark a wider variety of algorithms.

The benchmark itself could also be worked on. Different games could be used, or a wider range of games. NES games are quite simple, and while complex gameplay can be achieved, more complex games such as those for the Super Nintendo Entertainment System (SNES) could be used to provide a greater challenge to the agents. Bhonker et al. (2017) provides a comparison between Atari 2600 and SNES games, noting that SNES games have a larger action space and

more background noise than Atari games. SNES games can also be more unpredictable than Atari games and can exhibit more random behaviour. While NES games are more complex than Atari 2600 games, they are still not as complex as SNES games (SNES is the successor to the NES) and so these comparisons still stand. Using SNES games, or any other more complicated game, could give a larger variety of tasks to benchmark the reinforcement learning algorithms' performances.

An interesting path to take to further this work could be to split the benchmark into "train" and "test" environments, like the work done in Nichol et al. (2018). This paper presents a benchmark for generalisation in reinforcement learning, using Sonic the Hedgehog games. It measures transfer learning and cross-task generalisation by splitting between training and testing environments. This split gave higher results than learning from scratch without a split. This approach could be adopted for this NES game benchmark, evaluating if a train/test split improves the results of the algorithms tested here.

6.3 Reflection

I feel this project went well. I have always been interested in video games, particularly Nintendo games, and also artificial intelligence so getting to combine the two was very rewarding. While it was difficult getting my head around some of the more theoretical ideas of reinforcement learning, the coding and experimenting was enjoyable. It was interesting researching related works for the topic and finding all the ways artificial intelligence can be used in video games. I wish I could have had some more time to work on it more fully and tackle some of the ideas proposed for future work, however I am pleased with the project overall.

	A3C				PPO				Random			
	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average
Arkanoid	3800	3700	2540	3346 ± 571.9	3850	2480	5260	**3896 ± 1176.2**	480	270	270	340 ± 98.9
Gradius	8900	8100	9600	8866 ± 612.8	11900	8500	11700	10700 ± 1557.8	10000	10700	13700	**11466.7 ± 1604.9**
Pac-Man	1070	1460	2040	1523 ± 398.5	2210	2440	1490	**2016 ± 404.7**	150	210	220	193 ± 31.0
Space Invaders	310	170	210	230 ± 58.9	360	240	240	**280 ± 56.6**	120	180	190	163 ± 31.0
Super Mario Bros.	18600	19150	19650	**19133 ± 428.8**	7000	700	19250	8983 ± 7701.8	100	200	100	133 ± 47.1

Table 1: *High scores achieved by a trained agent in each game.*

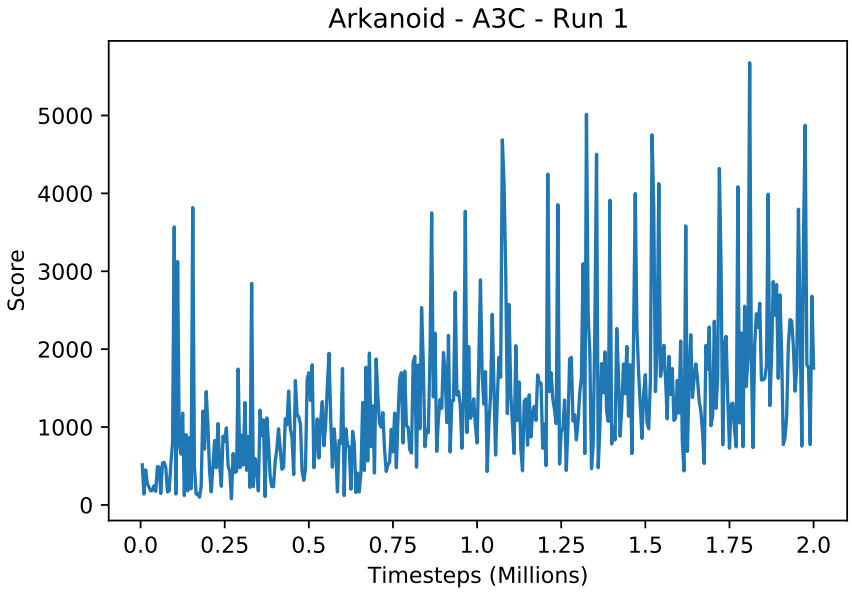

Figure 1: *A3C Training scores for Run 1 of Arkanoid.*

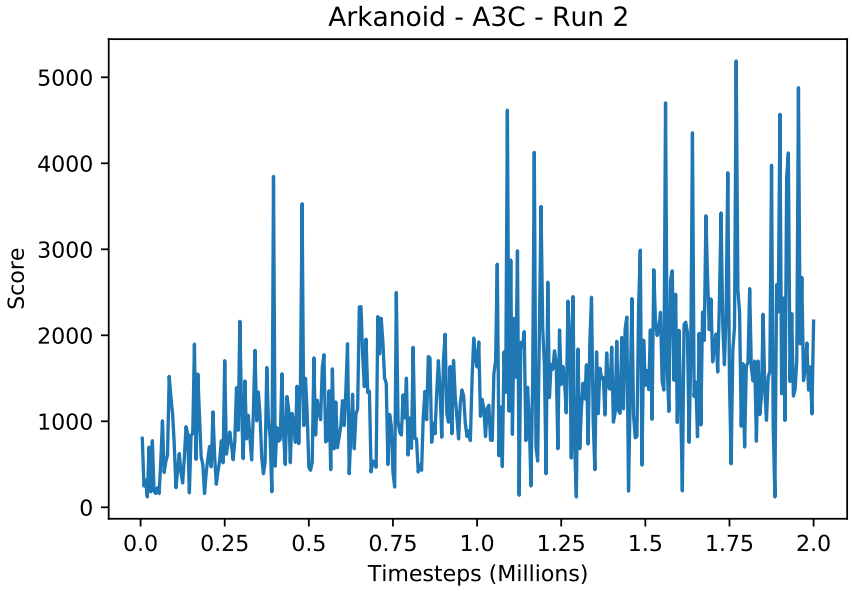

Figure 2: *A3C Training scores for Run 2 of Arkanoid.*

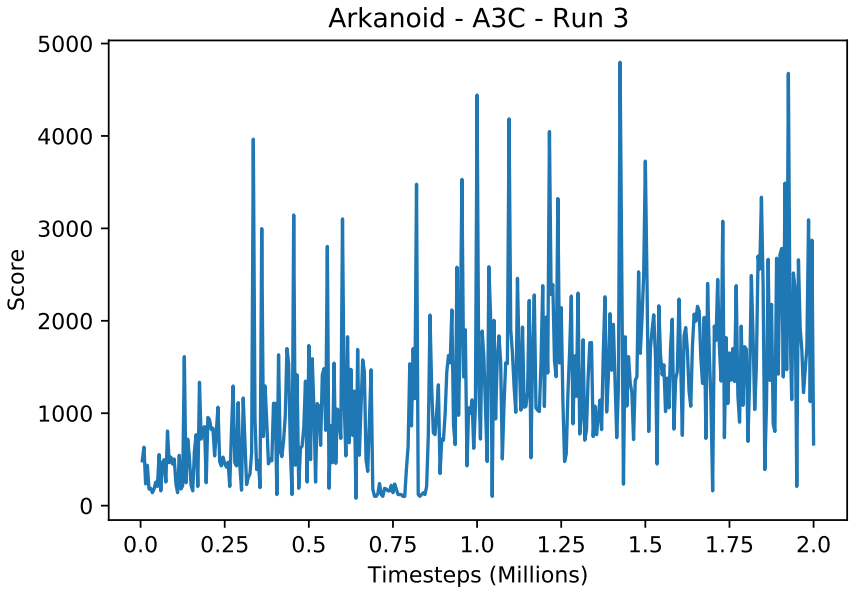

Figure 3: *A3C Training scores for Run 3 of Arkanoid.*

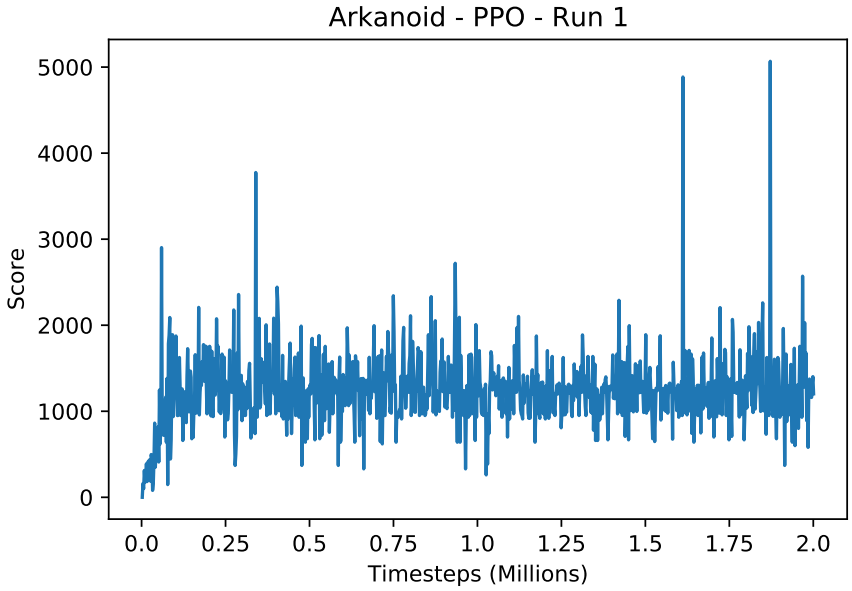

Figure 4: *PPO Training scores for Run 1 of Arkanoid.*

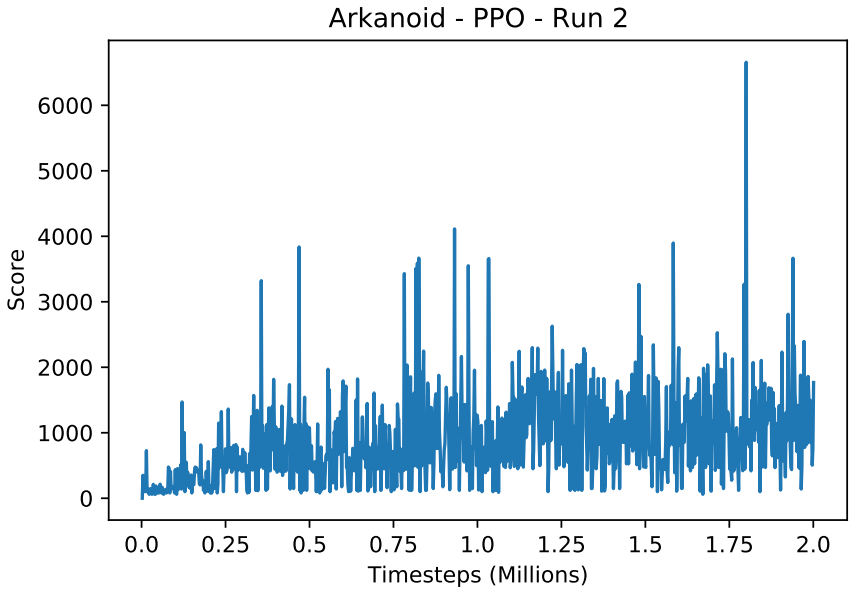

Figure 5: *PPO Training scores for Run 2 of Arkanoid.*

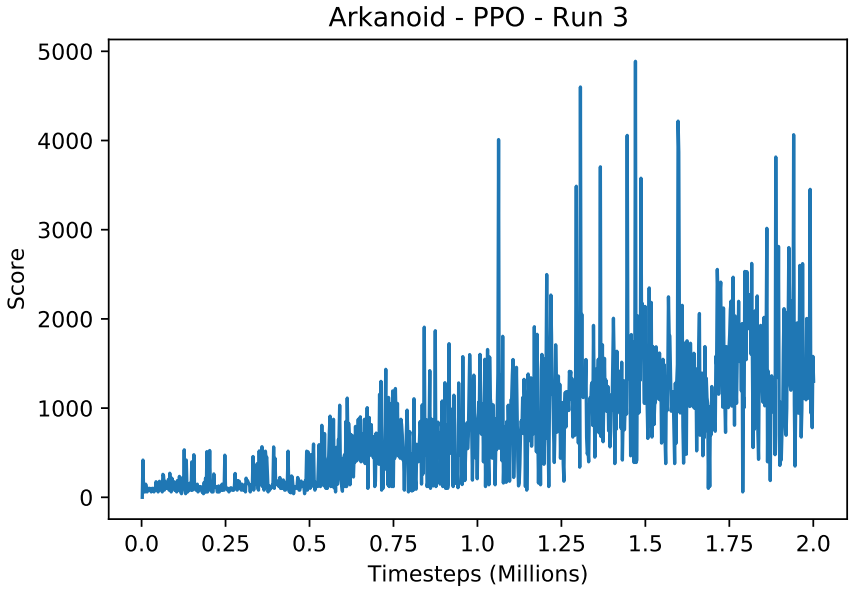

Figure 6: *PPO Training scores for Run 3 of Arkanoid.*

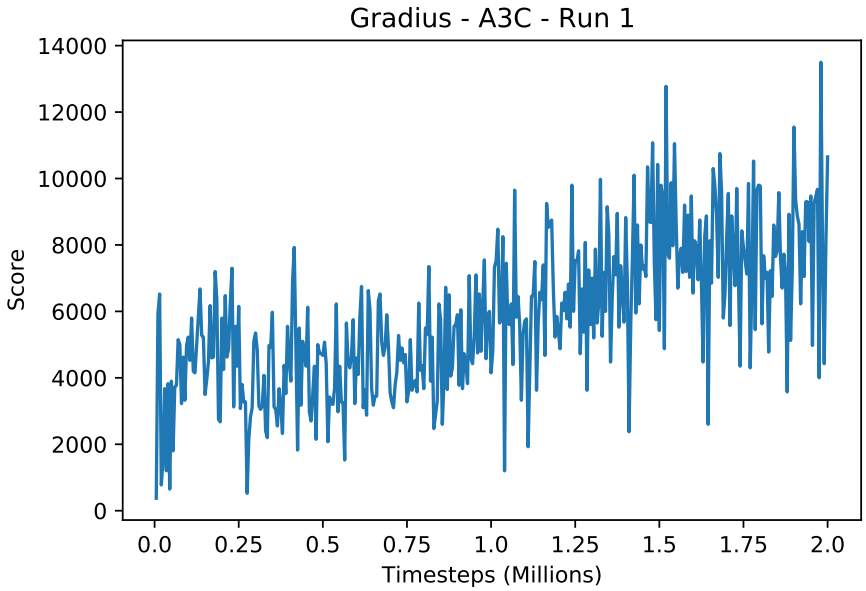

Figure 7: *A3C Training scores for Run 1 of Gradius.*

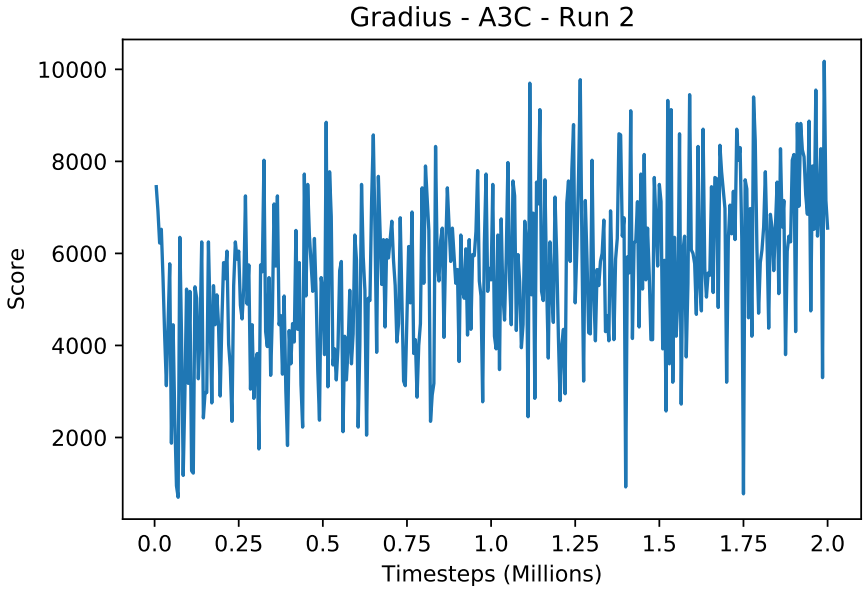

Figure 8: *A3C Training scores for Run 2 of Gradius.*

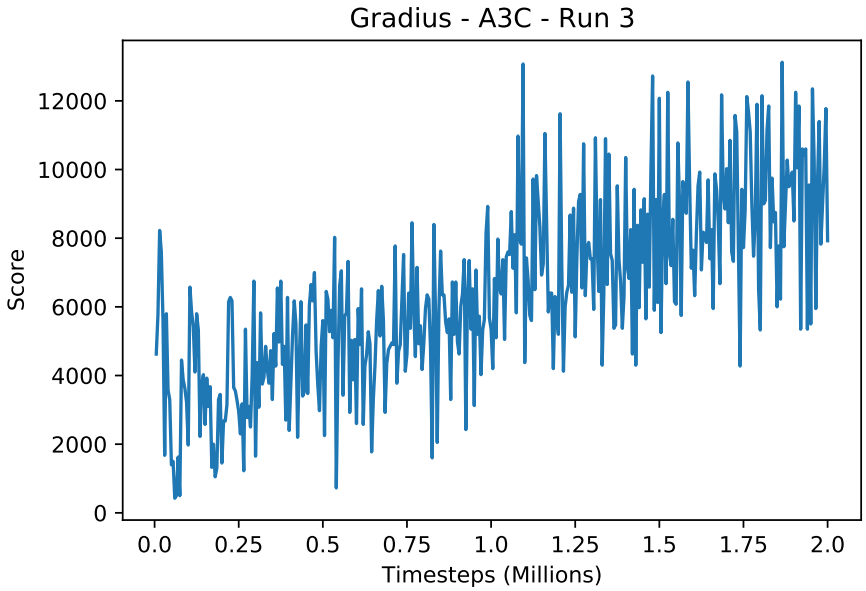

Figure 9: *A3C Training scores for Run 3 of Gradius.*

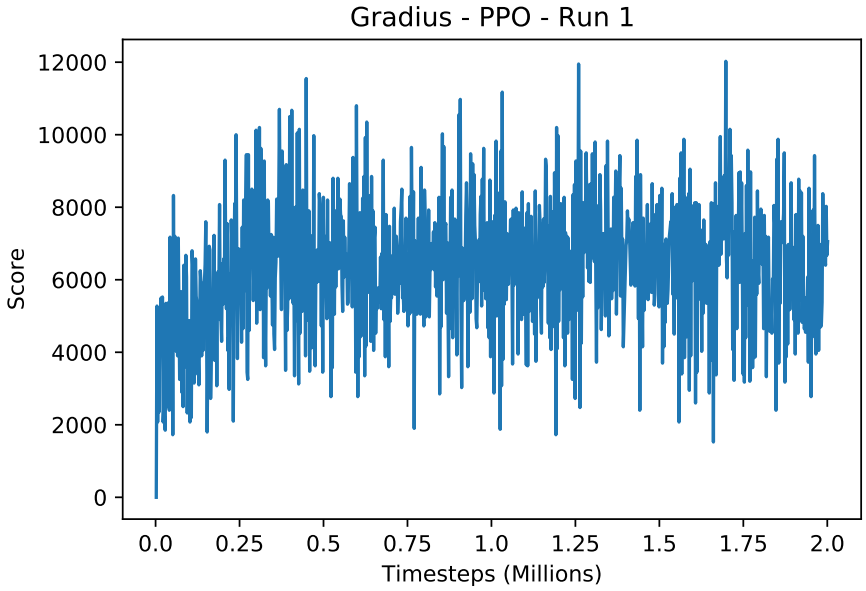

Figure 10: *PPO Training scores for Run 1 of Gradius.*

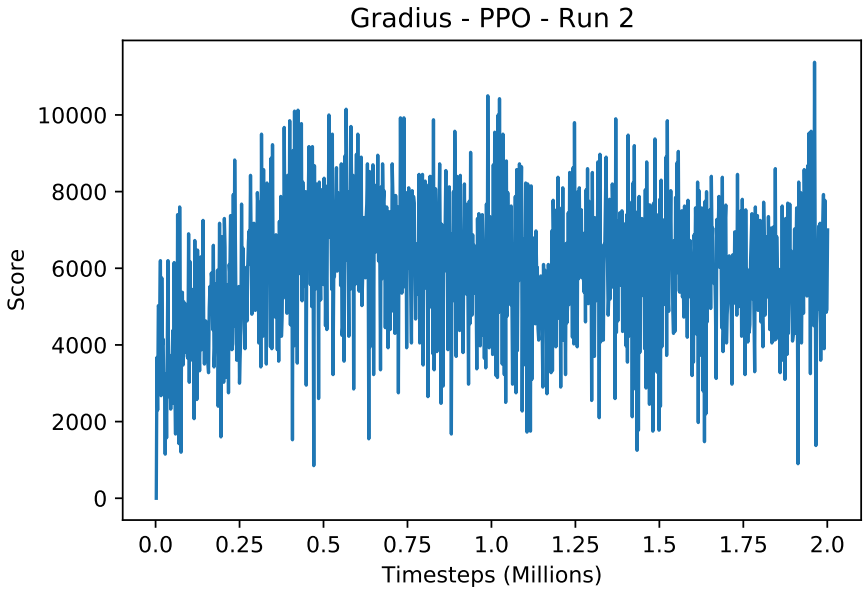

Figure 11: *PPO Training scores for Run 2 of Gradius.*

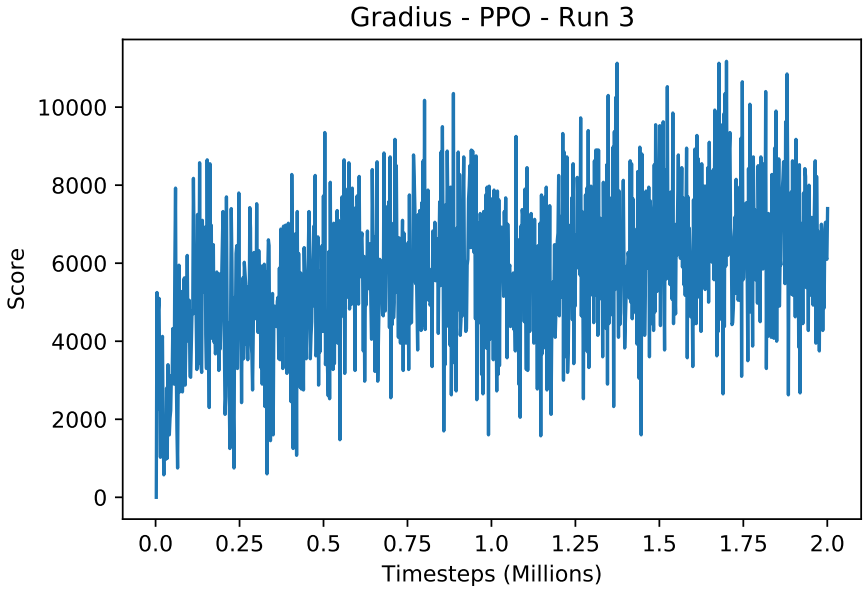

Figure 12: *PPO Training scores for Run 3 of Gradius.*

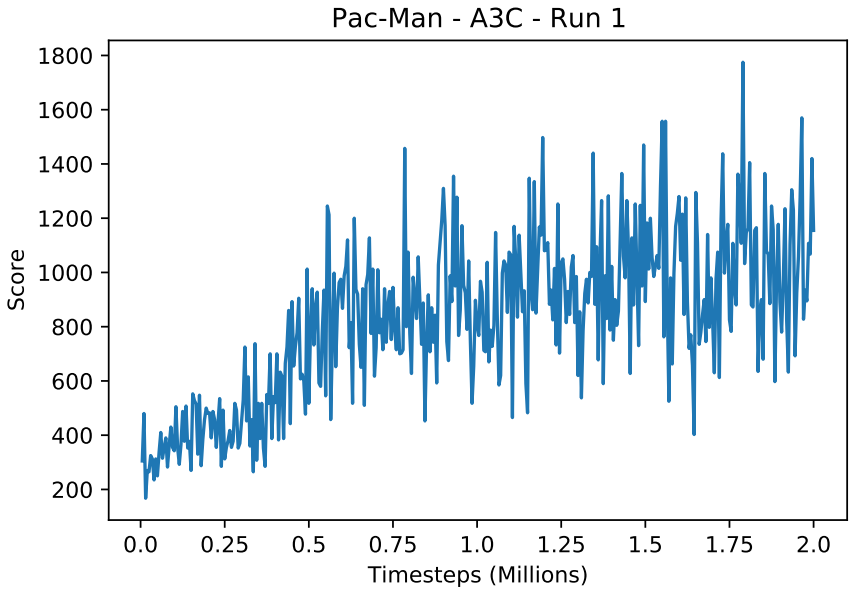

Figure 13: *A3C Training scores for Run 1 of Pac-Man.*

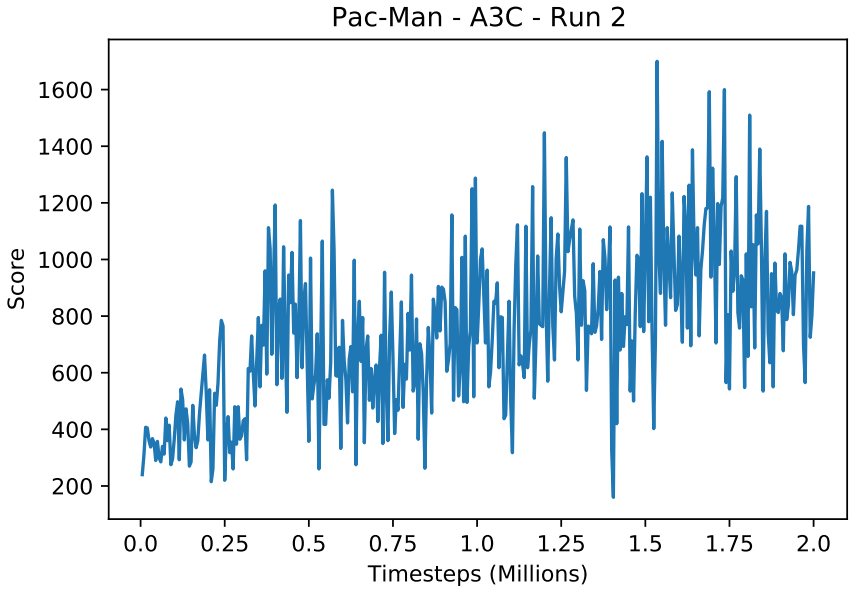

Figure 14: *A3C Training scores for Run 2 of Pac-Man.*

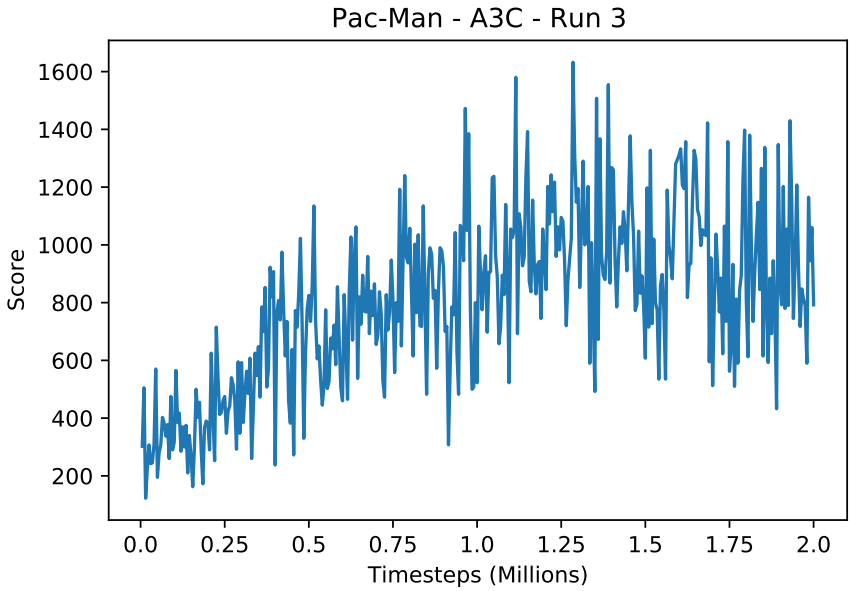

Figure 15: *A3C Training scores for Run 3 of Pac-Man.*

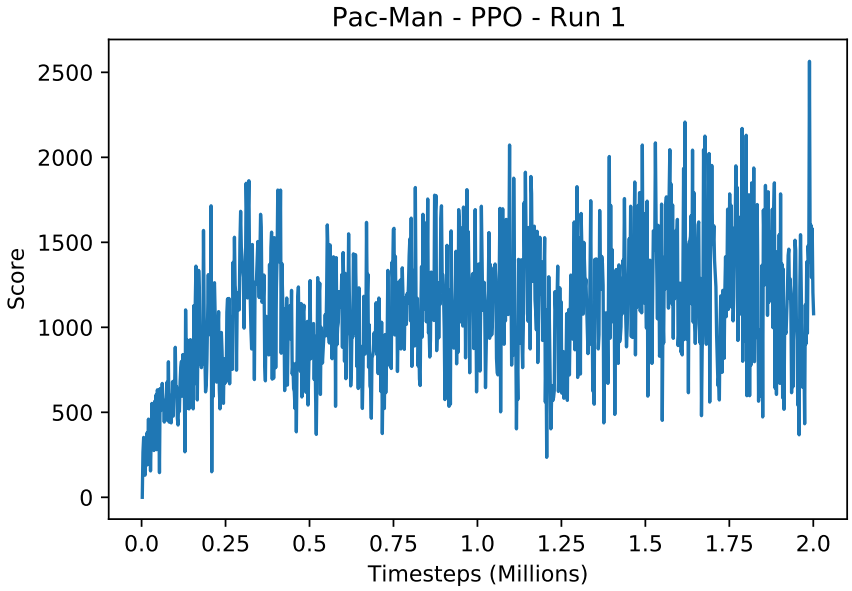

Figure 16: *PPO Training scores for Run 1 of Pac-Man.*

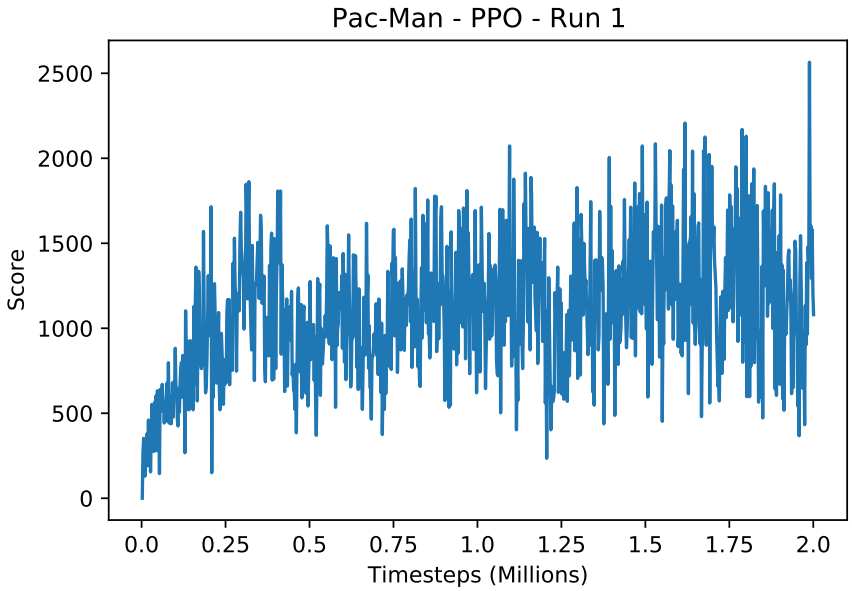

Figure 17: *PPO Training scores for Run 2 of Pac-Man.*

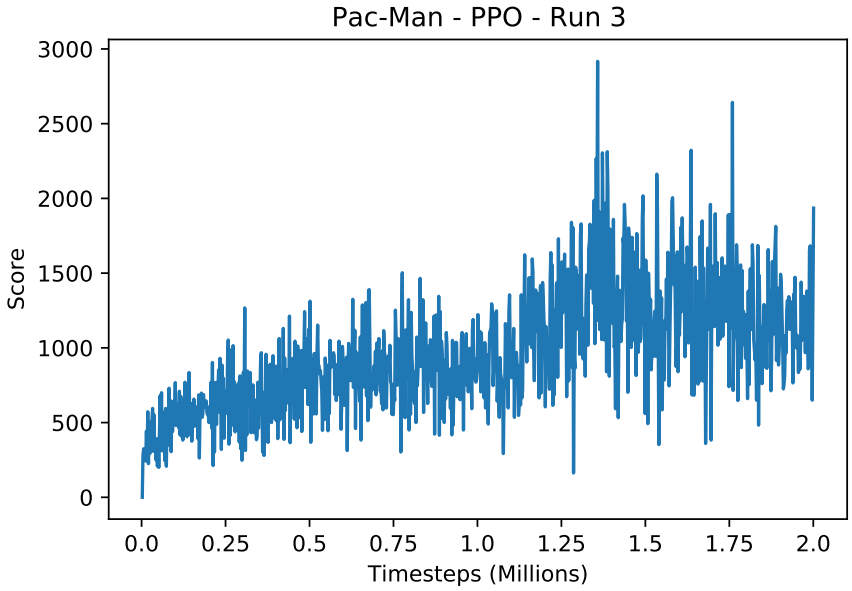

Figure 18: *PPO Training scores for Run 3 of Pac-Man.*

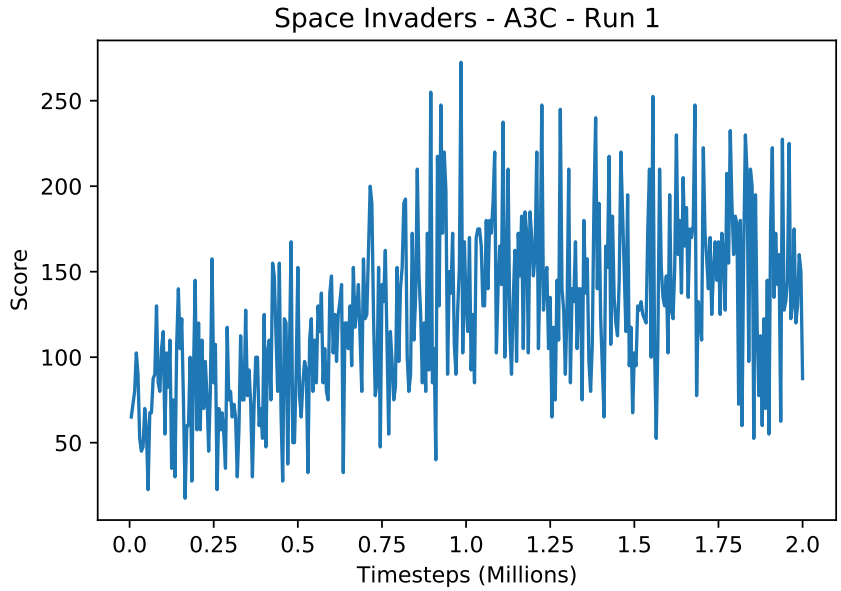

Figure 19: *A3C Training scores for Run 1 of Space Invaders.*

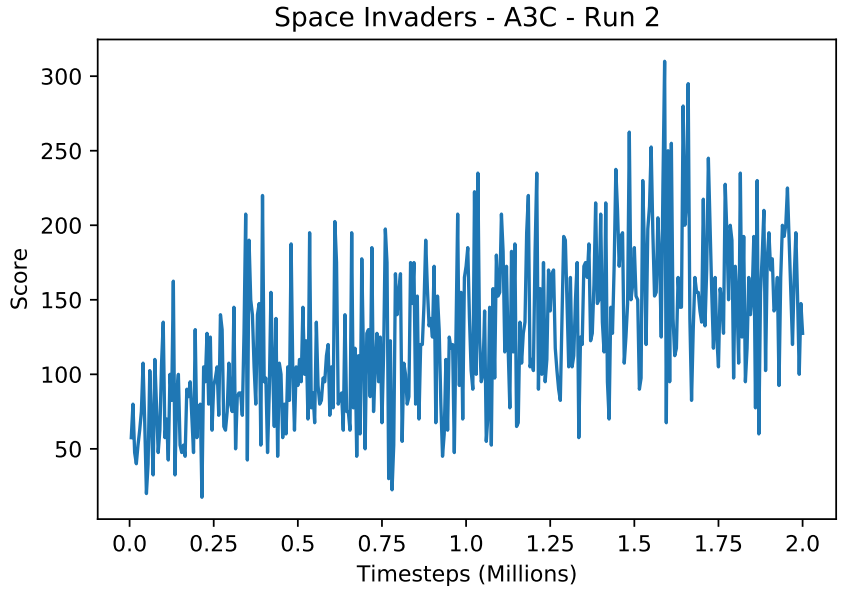

Figure 20: *A3C Training scores for Run 2 of Space Invaders.*

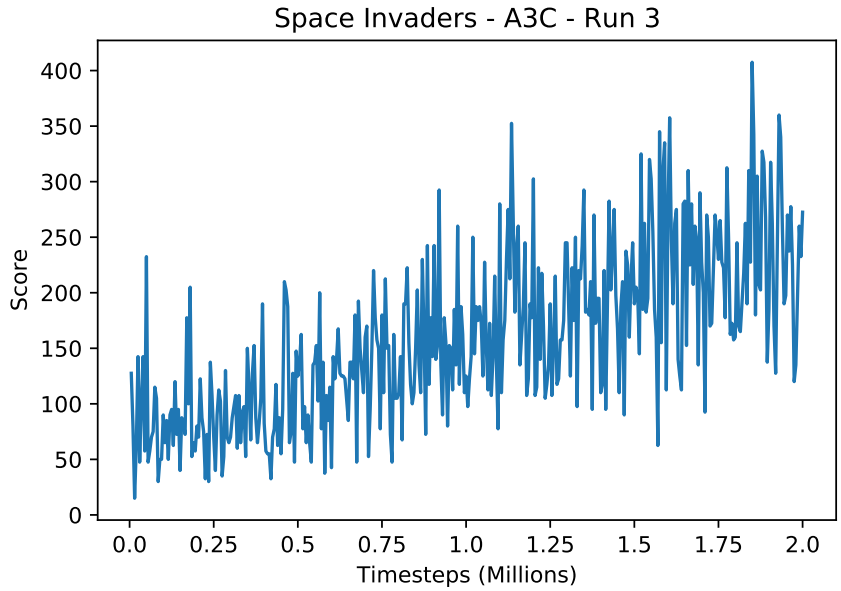

Figure 21: *A3C Training scores for Run 3 of Space Invaders.*

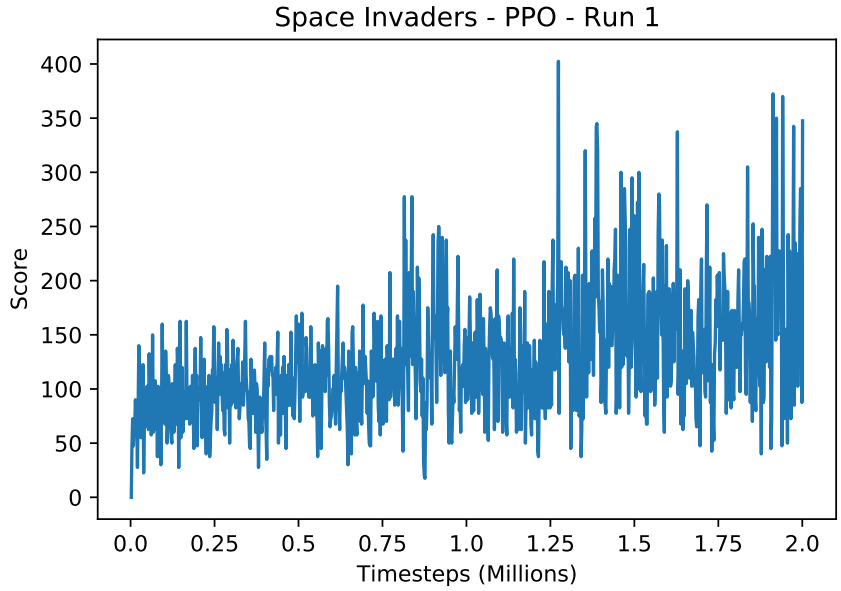

Figure 22: *PPO Training scores for Run 1 of Space Invaders.*

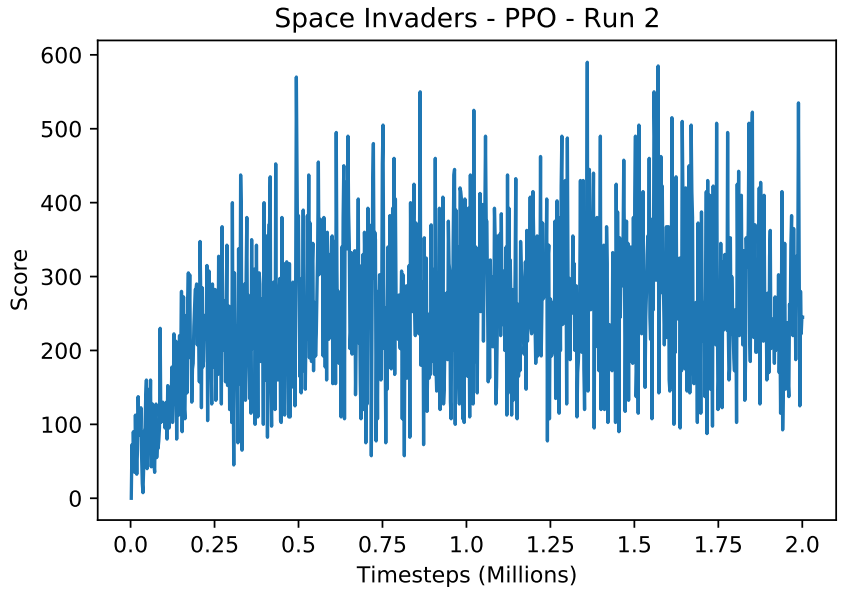

Figure 23: *PPO Training scores for Run 2 of Space Invaders.*

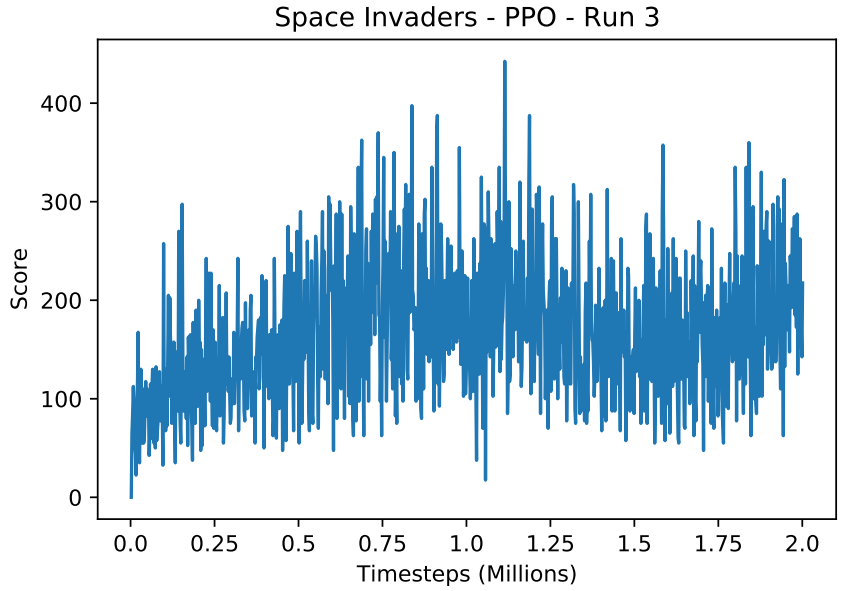

Figure 24: *PPO Training scores for Run 3 of Space Invaders.*

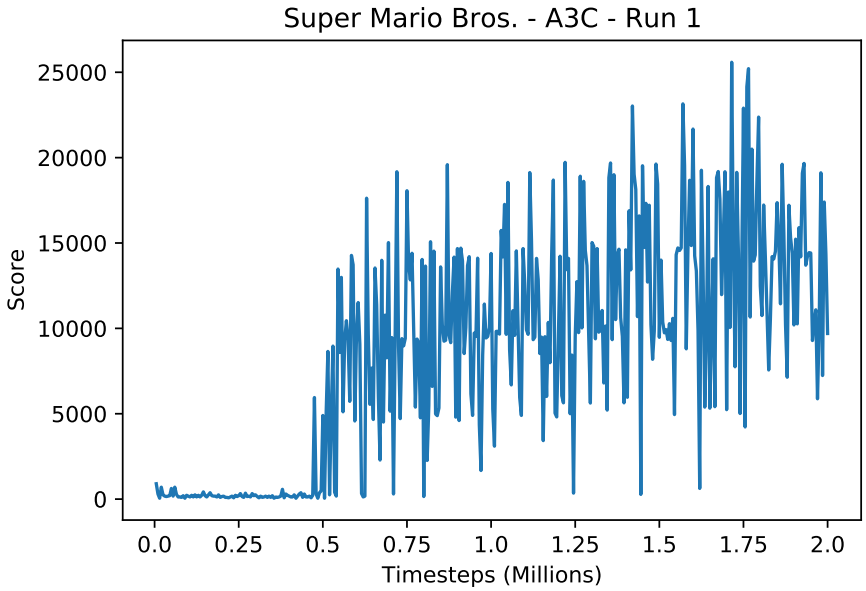

Figure 25: *A3C Training scores for Run 1 of Super Mario Bros.*

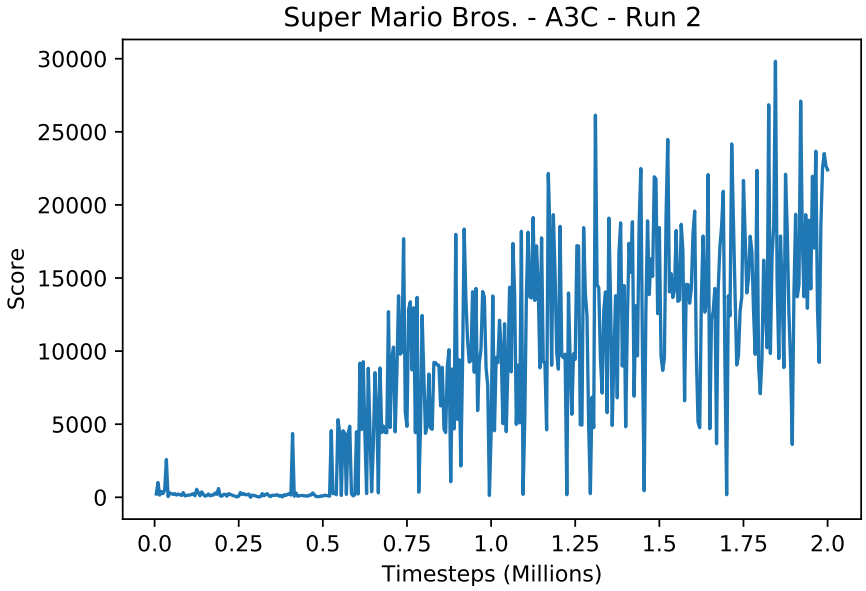

Figure 26: *A3C Training scores for Run 2 of Super Mario Bros.*

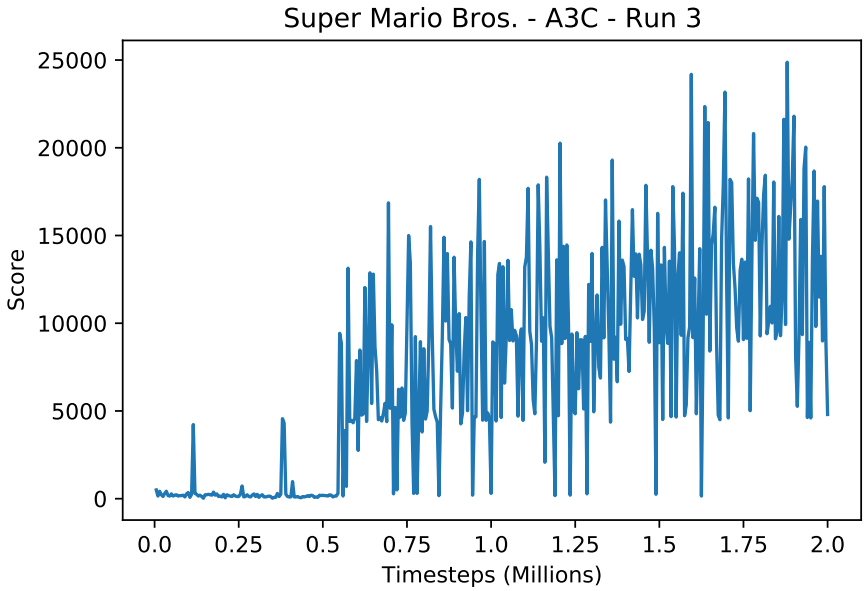

Figure 27: *A3C Training scores for Run 3 of Super Mario Bros.*

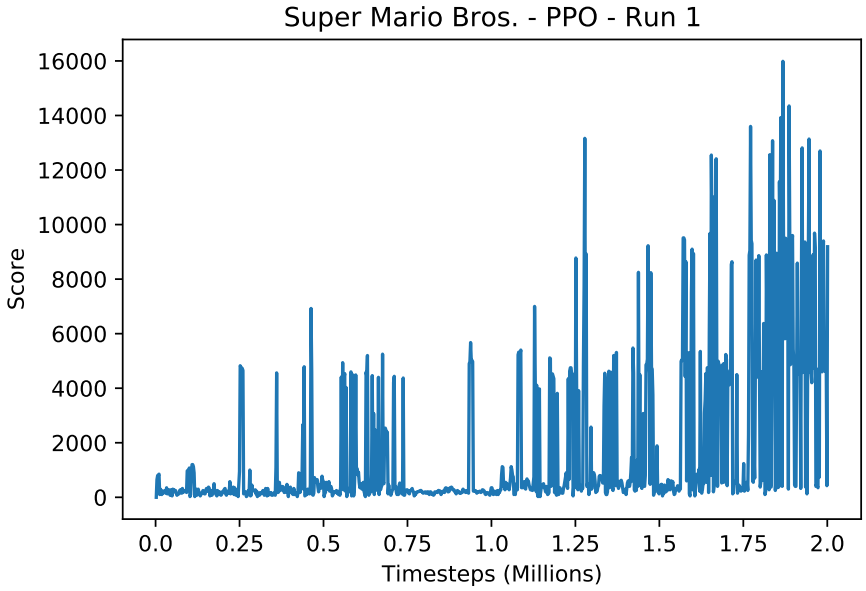

Figure 28: *PPO Training scores for Run 1 of Super Mario Bros.*

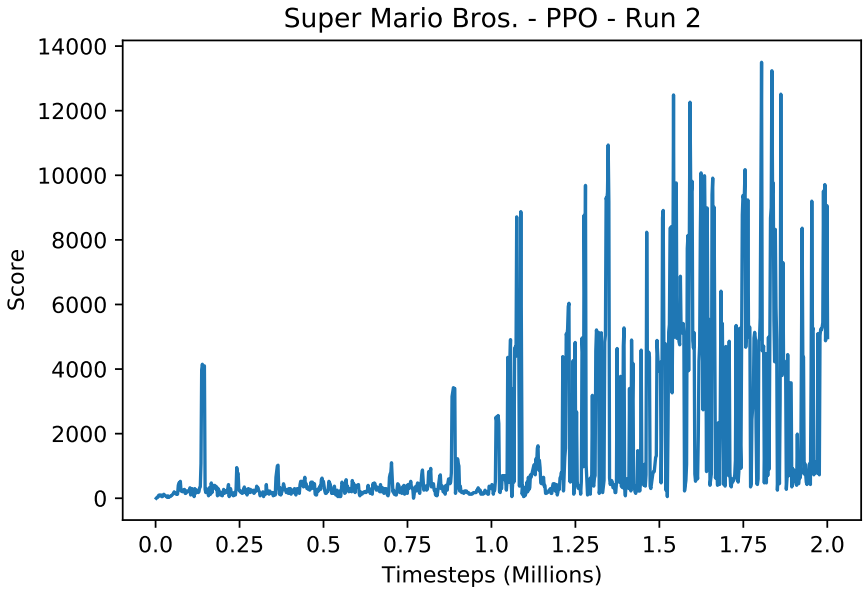

Figure 29: *PPO Training scores for Run 2 of Super Mario Bros.*

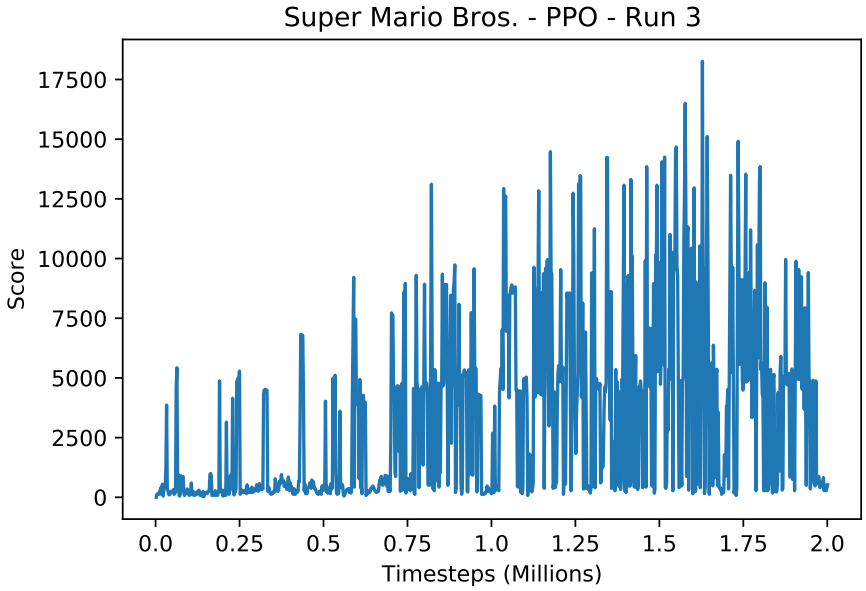

Figure 30: *PPO Training scores for Run 3 of Super Mario Bros.*

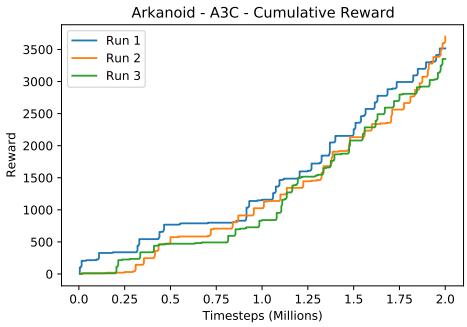

Figure 31: *A3C Cumulative Reward for Arkanoid.*

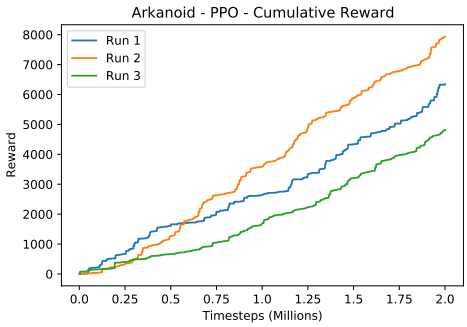

Figure 32: *PPO Cumulative Reward for Arkanoid.*

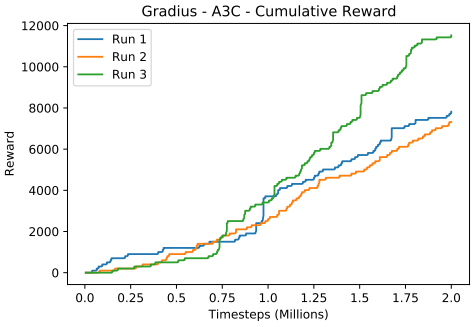

Figure 33: *A3C Cumulative Reward for Gradius.*

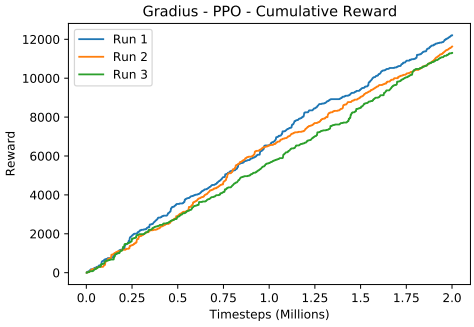

Figure 34: *PPO Cumulative Reward for Gradius.*

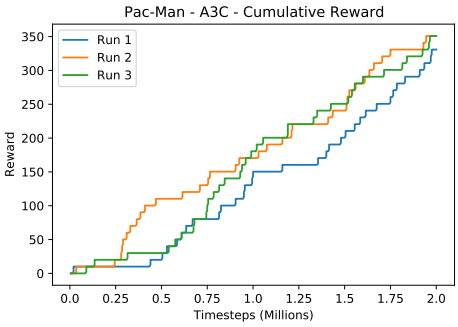

Figure 35: *A3C Cumulative Reward for Pac-Man.*

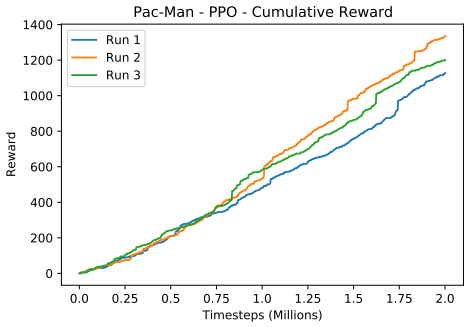

Figure 36: *PPO Cumulative Reward for Pac-Man.*

Bibliography

Aragon-Camarasa, G. (2020), 'Github - mario-bm (gerardo aragon-camarasa)'. https://github.com/cvas-ug/mario-bm Last accessed: 2021-04-17.

Argerich, M. F. (2020), 'Understanding learning rates and how it improves performance in deep learning'. https://towardsdatascience.com/entropy-regularization-in-reinforcement-learning-a6fa6d7598df Accessed 19/4/21.

Barati, E. and Chen, X. (2019), An actor-critic-attention mechanism for deep reinforcement learning in multi-view environments, *in* 'The 28th International Joint Conference on Artificial Intelligence'.

Bellemare, M. G., Naddaf, Y., Veness, J. and Bowling, M. (2013), 'The arcade learning environment: An evaluation platform for general agents', *Journal of Artificial Intelligence Research* **47**, 253–279. http://dx.doi.org/10.1613/jair.3912.

Bergdahl, J., Gordillo, C., Tollmar, K. and Gisslén, L. (2020), Augmenting automated game testing with deep reinforcement learning, *in* '2020 IEEE Conference on Games (CoG)', pp. 600–603.

Bhonker, N., Rozenberg, S. and Hubara, I. (2017), 'Playing snes in the retro learning environment'.

Chrabaszcz, P., Loshchilov, I. and Hutter, F. (2018), 'Back to basics: Benchmarking canonical evolution strategies for playing atari'.

DeMarco, N. (2017), 'Turn to channel 3: Worse than 'e.t.,' 'pac-man' is a hard pill to swallow on atari 2600'. https://nepascene.com/2017/08/turn-channel-3-worse-et-pac-man-hard-pill-swallow-atari-2600 Last accessed: 2021-04-14.

Kingma, D. P. and Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Konami (1986), *Gradius Instruction Manual*. https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NABRE.pdf.

Liang, Y., Machado, M. C., Talvitie, E. and Bowling, M. (2016), 'State of the art control of atari games using shallow reinforcement learning'.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. and Bowling, M. (2017), 'Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents'.

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W. and Bergstra, J. (2018), 'Benchmarking reinforcement learning algorithms on real-world robots'.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. and Kavukcuoglu, K. (2016), 'Asynchronous methods for deep reinforcement learning'.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013), 'Playing atari with deep reinforcement learning'.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**, 529–33.

Namco (1983), *Pac-Man NES Instruction Manual*. https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NABME.pdf.

Nguyen, V. (2019-2020), 'Github - uvipen (viet nguyen)'. https://github.com/uvipen Last accessed: 2021-04-15.

Nichol, A., Pfau, V., Hesse, C., Klimov, O. and Schulman, J. (2018), 'Gotta learn fast: A new benchmark for generalization in rl'.

Nintendo (1985), *Super Mario Bros. Instruction Manual*. https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NAAAE.pdf.

OpenAI (2016), 'Gym documentation'. https://gym.openai.com/docs/ Accessed 19/4/21.

OpenAI (2017), 'Proximal policy optimization'. https://openai.com/blog/openai-baselines-ppo/ Accessed 19/4/21.

OpenAI (2018), 'Gym retro', https://openai.com/blog/gym-retro/. (Accessed on 04/14/2021).

OpenAI (2019*a*), 'Game integration — gym retro documentation', https://retro.readthedocs.io/en/latest/integration.html. (Accessed on 04/14/2021).

OpenAI (2019*b*), 'random_agent.py'. https://github.com/openai/retro/blob/master/retro/examples/random_agent.py Last accessed: 2021-04-15.

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., d. O. Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F. and Zhang, S. (2019), 'Dota 2 with large scale deep reinforcement learning'.

Peters, J. (2010), 'Policy gradient methods'. http://www.scholarpedia.org/article/Policy_gradient_methods Accessed 19/4/21.

Quaedvlieg, L. (2020), 'Introduction to reinforcement learning'. https://github.com/justLars7D1/Reinforcement-Learning-Book.

Schulman, J., Moritz, P., Levine, S., Jordan, M. and Abbeel, P. (2018), 'High-dimensional continuous control using generalized advantage estimation'.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017), 'Proximal policy optimization algorithms'.

Shao, K., Tang, Z., Zhu, Y., Li, N. and Zhao, D. (2019), 'A survey of deep reinforcement learning in video games'.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. and Riedmiller, M. (2014), Deterministic Policy Gradient Algorithms, *in* 'ICML', Beijing, China. https://hal.inria.fr/hal-00938992.

Stadie, B. C., Levine, S. and Abbeel, P. (2015), 'Incentivizing exploration in reinforcement learning with deep predictive models'.

Stooke, A. and Abbeel, P. (2019), 'Accelerated methods for deep reinforcement learning'.

Sutton, R. S. and Barto, A. G. (2015), *Reinforcement Learning: An Introduction*, 2 edn, The MIT Press. http://incompleteideas.net/book/RLbook2020.pdf.

Taito (1985), *Space Invaders Famicom Instruction Manual*. http://www.replacementdocs.com/download.php?view.7730.

Taito (1986), *Arkanoid NES Instruction Manual*. https://www.gamesdatabase.org/Media/SYSTEM/Nintendo_NES//Manual/formated/Arkanoid_-_1987_-_Taito_Corporation.pdf.

Vega, I. (2019), 'Nintendo and the video game crash of 1983 - "now you're playing with power"', https://www.nintendo.co.uk/Corporate/Nintendo-History/Nintendo-Entertainment-System/Nintendo-Entertainment-System-627024.html. (Accessed on 04/14/2021).

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N. (2016), 'Dueling network architectures for deep reinforcement learning'.

Watkins, C. J. C. H. and Dayan, P. (1992), 'Technical note q-learning.', *Mach. Learn.* **8**, 279–292. http://dblp.uni-trier.de/db/journals/ml/ml8.html#WatkinsD92.

Weng, L. (2018), 'Policy gradient algorithms'. https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html Accessed 19/4/21.

Yu, C., Liu, J. and Nemati, S. (2020), 'Reinforcement learning in healthcare: A survey'.

Zulkifli, H. (2018), 'Understanding learning rates and how it improves performance in deep learning'. https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 Accessed 19/4/21.