Testing the performance of Asynchronous Advantage Actor Critic within environments featuring 2D and 3D graphics

Bachelor's Project Thesis

Vlad Muscoi, s4718267, v.n.muscoi@student.rug.nl,
Supervisors: J.D. Cardenas Cartagena, M.Sc.

Abstract: In this paper, we will use the popular games Super Mario Bros and Minecraft to test the performance of Asynchronous Advantage Actor-Critic (A3C), an algorithm that uses parallel agents to gather additional samples at the same time, within 2D and 3D environments respectively. Although high variance has been observed during training, the model when run within the Mario environment was able to converge to a policy capable of completing its task of finishing the level. Minecraft provided insight into the sample inefficiency of the algorithm, showing much slower learning for a much wider state space. The algorithm was not able to converge to a policy that would solve the task within the Minecraft environment. The code used within this project can be found on github:https://github.com/vluvl/A3C_with_Mario_and_Minecraft

1 Introduction

One of the challenges of applying reinforcement learning agents to real-world problems is the high dimensionality of state and action spaces required to define such tasks. To overcome this, we can use computer simulations to model the tasks and environments that we want the agents to learn. This has several advantages, such as speeding up the learning process and allowing us to monitor and evaluate the agent's performance more easily. A rich source of simulated environments that are designed for training agents to achieve a goal are video games, where humans are the agents that have to learn to master the game mechanics (Goecks et al., 2021).

Video games are a popular tool for Reinforcement Learning experiment due to their inherit nature. Video games offer rich and complex interactions between agents and their environments. These interactions involve decision-making, perception, and strategic planning. Researchers can study how agents learn to navigate these intricate scenarios. Additionally, the environments are under the full control of the user, allowing researches to manipulate game parameters, create specific scenarios or observe agent behaviour without any real world risk. Video games span various genres, from classic arcade games to first-person shooters and real-time strategy games. Each genre presents unique challenges, such as exploration, resource management, and adversarial interactions. Researchers can explore different aspects of reinforcement learning in these diverse contexts. Despite these advantages, challenges remain. Researchers grapple with issues like exploration-exploitation trade-offs, sample efficiency, generalization, multi-agent learning, and handling imperfect information. Addressing these challenges drives ongoing research in the field such as development of Deep Reinforcement Learning approaches (Shao et al., 2019).

Bridging the gap between the theoretical advantages of reinforcement learning in video games and the practical applications of these concepts, the Asynchronous Advantage Actor-Critic (A3C) emerges as a potent solution. This technique leverages the controlled, risk-free environments provided by video games to address some of the most pressing challenges in the field. A3C is a model-free Reinforcement learning technique that employs multiple parallel agents to explore the environment and learn from their experience (Mnih et al., 2016). This method consists of two main components: The Ac-

tor and the Critic. The Actor is responsible for choosing actions to be performed by the agent, while the Critic evaluates the actions chosen by the Actor and provides feedback to the Actor. The algorithm has two main components: the global model and the worker models. The global model is created in the main function of the algorithm while the worker models are created in different threads spawned by the main function. The global model does not act in an environment itself, but it is used as a "bank" of information that worker agents can updated and get updates from. In A3C, the parallel actor-critic agents are trained asynchronously: once one of the multiple agents finishes an episode, it updates the global model and then updates itself with the up to date global model parameters.

Two games were selected to use as the environments within which the performance of A3C has been tasted: Super Mario Bros and Minecraft.

Mario is a popular choice among reinforcement learning researchers, and has the advantage of being lightweight and allowing parallel execution of multiple instances to collect more samples. In a report done by Liao et al. (2012), the authors explore the application of the Q-Learning algorithm to design an automatic agent capable of playing Mario. The main challenge lies in handling the complex game environment while ensuring real-time responsiveness. The A3C algorithm could be a significant upgrade from Q-Learning in the scenario of playing Mario, due to its ability to handle temporal differences more effectively. A3C operates by running multiple instances of the environment in parallel, which aligns well with the lightweight and parallelizable nature of the Mario game environment. This parallelism allows for a more diverse range of experiences to be collected, leading to a more robust policy.

Minecraft is a more complex and modern game, with a 3 dimensional environment, unlike Mario which is 2D. The agent also has a first person perspective, while in Mario the agent is visible from the 3rd person view. Minecraft is more often used in researching sample efficient deep reinforcement learning algorithms due to its high dimensional complexity.

Due to the higher complexity of Minecraft over Mario, much more advanced tools have been developed to enable using the game as an environment for reinforcement learning. Microsoft took the initiative and developed project Malmo (Johnson et al., 2016) which provides basic machine learning functionality. It offers a simple environment within which toy tasks can be defined that are often restricted to 2D movement, discrete positions, or artificially confined maps unrepresentative of the intrinsic complexity that human players typically face. The MineRL framework uses Malmo as its base to expand its functionality and provide more control over the environment and the agent.

The main focus of the MineRL competition is to explore sample efficient methods for training reinforcement learning agents (Guss et al., 2019). This project uses the MineRL framework. Within this competition, several notable algorithms have been developed such as the one developed by Amiranashvili et al. (2020) which utilized expert generated behaviour to teach itself by example how to complete a task. This algorithm achieved second place in the 2019 edition of the MineRL competition. A step up in the imitation learning branch of models used within the MineRL competition, achieving first place the next competition iteration, is the Sample-efficient Hierarchical AI (SEIHAI) (Mao et al., 2021) which fully capitalizes on the strengths of human demonstrations and the inherent structure of the task while employing a hierarchical approach to break down the task into several sequentially dependent sub-tasks. For each sub-task, it trains a suitable agent using a combination of reinforcement learning and imitation learning.

This paper delves into the comparative analysis of the Asynchronous Advantage Actor-Critic algorithm's performance in two distinct gaming environments: the classic 2D platformer Mario and the expansive 3D sandbox game Minecraft. By evaluating A3C in these contrasting contexts, the study aims to shed light on the algorithm's adaptability and efficiency across different dimensions and complexities of gaming spaces. Minecraft presents a significantly more intricate and extensive array of potential states for the agent, providing a rigorous test for A3C's state management capabilities. The game's open-ended environment and the sheer number of possible interactions offer a challenging arena to assess the scalability and robustness of the algorithm.

Conversely, Mario, with its more predictable and confined set of states, serves as an ideal base-

line to gauge the fundamental efficiency of A3C. The controlled and less complex nature of Mario's world allows for a clearer evaluation of the algorithm's performance without the confounding variables present in a more complex environment like Minecraft. This stark contrast between the two games will enable a thorough investigation into how A3C handles the transition from a simple to a complex state space, and whether its performance scales proportionally with the increased complexity.

Furthermore, the paper will explore the impact of scaling up the number of worker agents responsible for collecting experience samples. This aspect is crucial as it could reveal insights into the parallelization capabilities of A3C and how it affects learning efficiency and convergence rates. The hypothesis is that an increase in worker agents may lead to faster accumulation of diverse experiences, potentially accelerating the learning process and improving the overall performance of the algorithm.

2 Theoretical Framework

2.1 Asynchronous Advantage Actor-Critic

The Asynchronous Advantage Actor-Critic algorithm (A3C) is a policy gradient algorithm that maintains a policy $\pi(a_t|s_t;\theta)$ (the "actor") and an estimate value function $V(s_t|\theta_v)$ (the "critic") where π is the policy, a_t and s_t are the action and state at time t and θ and θ_v are the parameters of the policy and of the value function respectively. The policy and value functions are updated after every t_{max} actions or when a terminal state is reached.

A soft-max function is constructed using the output of the Actor layer. This soft-max function provides a probabilistic decision-making process, where the probability of selecting an action is proportional to the exponential of its value. This allows for a balance between choosing the action with the highest expected utility (exploitation) and trying out less certain actions (exploration).

The estimate of the advantage function is given by:

$$\delta = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v) \quad (2.1)$$

where δ is the advantage function, k can vary from state to state and is upper-bounded by t_{max}, r is the reward at time-step $t+1$, $\gamma \in (0,1)$ is the learning rate, $V(s_t,\theta_v)$ is the value-function at state t of the critic. The advantage function is used to gauge how much better an action is compared to an average action by computing the difference between the expected cumulative reward and the value of the current state. It is used to perform the updates of the policy and value-function.

The pseudo-code from Mnih et al. (2016) has been provided in Algorithm 2.1. In the original paper, entropy is used to enhance the exploration period, but following empirical results detailed in the appendix A, the step where entropy is added to θ' is removed from the algorithm used in this paper to improve training stability.

2.2 Algorithm overview

In the following section the algorithm and its parallelization will be explained in detail.

The A3C algorithm is made of two components: the global model and worker models. The global model is the main function that spawns multiple worker models on different threads so that the workers can learn in parallel. An environment is created for the global model, but it is not used to execute actions and gather samples. That environment is used to extract state and action sizes to initialize the neural network dimensions of both global and worker models. The global model is then set to share its memory so that other models can update it. Lastly, the global model creates an array of processes with a pointer to itself and starts them on different threads. The different processes execute the local models which are initiated much like the global model, but with the main difference that they do not give access to their internal parameters.

The pseudo-code provided in Algorithm 2.1 details the functionality of a worker agent. It assumes the availability of the global shared parameters of the global model and a global counter. It also assumes that the agent has thread specific parameters

Algorithm 2.1 Modified Asynchronous Advantage Actor-Critic - pseudo-code for each actor-learner thread. (Mnih et al., 2016)

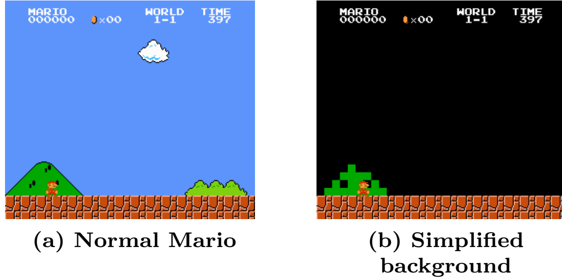
//Assume global shared parameter vectors θ and θ_v and global shared counter T = 0
//Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradient: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
  Synchronize thread-specific parameters: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

for the Critic (θ'_v) and the Actor (θ') that are part of the local model. For every start of an episode, the algorithm will reset the parameters of the local model and then update its parameters with those of the global model. It will then get the first state s_t and start iterating through actions a_t chosen by the policy $\pi(a_t|s_t;\theta')$. For each action taken, it will observe the new state s_{t+1} and receive a reward r_t. Once the step counter marks the end of the episode or a terminal state has been reached, the first term used in the calculation of the advantage function sum R is being determined depending on if the last state was a terminal state or not. If the last state was not a terminal state, the first term will be the value-function for the given state, otherwise it will be 0. The algorithm will then go through each iteration starting from the last towards the first of that episode. During this, the advantage function term R is being calculated by adding the current reward r_i to the previous R value influenced by the learning rate γ. The gradients for the Actor and the Critic parameters are being computed and accumulated using their respective functions. For the Actor, to the accumulation gradient $d\theta$ is added the gradient of the logarithm of the policy $\pi(a_t|s_t;\theta')$ and multiplied by difference of the advantage function term R and the value-function of the current iteration $V(s_i;\theta'_v)$. For the Critic, to the accumulation gradient $d\theta_v$ is added half of the squared difference of the advantage function term R and the value-function of the current iteration $V(s_i;\theta'_v)$ divided by θ'_v. Once the gradients have been calculated, the parameters will be sent to the global model to be updated.

2.3 Environment Frameworks

2.3.1 Mario

Mario has a well-known python package that integrates the NES game into a gym environment (Kauten, 2018). These environments are made such that only reward-able game-play frames are sent to the agent: cut-scenes, loading screens or other non game-play frames are not available. The framework offers different levels of image simplifications to help reduce the visual clutter of the game.



(a) Normal Mario (b) Simplified background

Figure 2.1: A comparison between normal Mario gameplay and a simplified version

For this paper, the first layer of simplifications has been applied, where the images have been down-sampled and the backgrounds of the levels have been drastically simplified. An example has been provided in Figure 2.1

The reward function assumes the objective of the game is to move as far right as possible (increase the agent's x value), as fast as possible, without dying. This is done using three variables: $V + C + D$. V is the difference in the value of the x coordinate of the agent between states: $V = x' - x$ where x and x' are the x coordinates old and new state respectively. The x coordinate is increasing towards the right of the screen, thus moving to the right translates to a positive V. C is the difference in the game clock between states: $C = c - c'$ where c and c' are the clock values before and after the state respectively. The variable C can be regarded as a penalty because it rewards the agent with a negative amount the more time it spends in an episode. Lastly, D is the death penalty that is earned when the agent dies in a state. The rewards obtained per state can be much higher or much lower than average, so they are clipped into the rage (-15,15).

The "SIMPLE_MOVEMENT" action scheme was used, which provides seven combinations of controller input that represent the actions that the agent can take. The actions are tailored to facilitate movement towards the right while allowing the agent to experiment with other less ideal actions such as jumping in place or going to the left. The combinations of inputs are:

- NOOP (no operation, the agent does nothing)
- Right
- Right + A (jump)
- Right + B (accelerate)
- Right + A + B
- A
- Left

2.3.2 Minecraft

Minecraft has been adapted to be used as an environment for gym via the MineRL framework (Guss et al., 2019). MineRL uses project Malmö as a base, developing upon it to offer a more sophisticated framework. Project Malmö is an AI experimentation platform designed to support fundamental research in artificial intelligence (Johnson et al., 2016) utilizing toy tasks, often restricted to two-dimensional movement or artificially confined maps. MineRL aims to take project Malmö and its mission a step forward by enabling a wider range of experiments to be done within Minecraft by removing all of the restrictions imposed by Malmö to allow the environment to feature the full domain encountered by humans while playing the game.

MineRL version 1.0 uses Minecraft 1.16 and offers a selection of predefined tasks such as "MINERL_OBTAIN_DIAMOND_SHOVEL_V0" which tasks an agent with learning to acquire the required items to craft the diamond shovel. The reward functions for these tasks are unique. For the before-mentioned task, the reward is distributed when the agent acquires one of each items in a predefined track of items that are needed to progress. For example, the agent first needs to get a block of wood, process that block into wooden planks, and then use the planks to craft a crafting bench which can be used to craft a wooden pickaxe with more wooden planks and sticks. Each of the items mentioned before has a reward attached to it for when it is first acquired. The reward function is not only limited to item acquisition, other tasks such as exploration or building can be programmed to give reward to the agent.

The framework has a complex set of 24 distinct actions that are made available to the agent such as movement in four directions, jumping, changing the pitch and yaw of the camera, opening the player inventory, using an item and interacting with a block. As mentioned before, MineRL aims to mimic the

human experience of the game. As such, the majority of human actions have been represented in the action space. For the tests performed in this paper, the action space has been limited to 8 actions. The actions are:

- NOOP
- Attacking
- Moving forward
- Jump while moving forward
- Move camera up
- Move camera down
- Move camera to the left
- Move camera to the right

It also offers extra information that is available to the player such as current health and inventory. The task descriptions within the framework are python scripts that create specific "environment specifications" that dictate how the game is created, how the agent view is rendered and how rewards are distributed. For this experiment, a custom specification has been created that will be detailed in section 3.2.

3 Methods

3.1 Mario

Mario is a well-known 2d platformer video game often used for testing and training reinforcement learning agents due to its light yet complex gameplay from an RL standpoint. The game works as follows: the player needs to traverse a two-dimensional scrolling level while avoiding enemies and obstacles that can harm the player and send it back to the start. The goal of the game is to reach the flag situated at the end of the level while avoiding danger.

For the scope of this project, Mario has been used to develop and test implementations of A3C to have a stable and operational algorithm that could be scaled up to a more difficult task. The final iteration of the algorithm is a modified version of the one developed by uvipen (2021) on their GitHub repository that uses the Mnih et al. (2016) paper as a base. The algorithm as implemented by uvipen creates a configurable number of parallel processes that run the training algorithm and a test process that is meant to show the progress of those parallel processes. Uvipen designed his algorithm with simplicity in mind, making simplifications to, according to him, unimportant parts such as image preprocessing and weight initialization while strictly following Mnih et al. (2016) paper. In terms of image pre-processing, the input image is resized to a resolution of 84 by 84 and converted to grayscale.

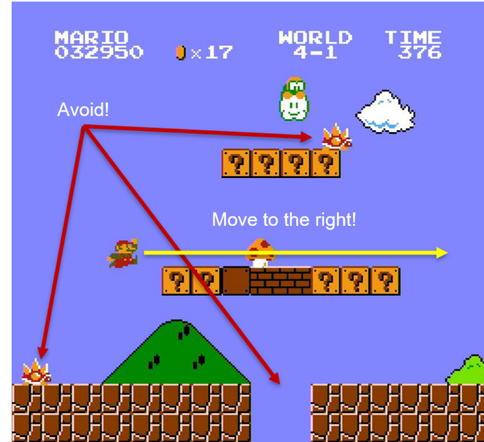


Figure 3.1: Summary image of what the agent is encouraged and discouraged to do in Mario.

The state and action spaces for Mario are simple. After the pre-processing mentioned before, frames are represented by an 84 by 84 pixels, grayscale image depicting the current rendered frame of the game. The custom frameskip function is declared to concatenate together 4 frames which will represent the states. The actions that can be taken by the agent are limited to the SIMPLE_MOVEMENT configuration defined by default within the Mario gym environment.

The reward for the Mario environment, according to the official documentation, is distributed in terms of three separate variables: the difference between the position of the agent between states, the difference in the game clock between frames and the death of the agent. A custom reward function has been added to reward the agent if the flag has been reached within a run and to penalize if not.

3.2 MineRL

The code had to be adapted to work with the MineRL framework. Utilizing slightly altered versions of the wrapper functions developed to make MineRL usable with Stable Baselines (Raffin et al., 2021) and other small modifications to the code, compatibility between the A3C algorithm and MineRL was established. Minecraft is far more complex compared to Mario:

- The world is rendered in three-dimensional space, allowing for much more freedom in movement.
- The perspective is a lot more limiting in terms of information gathering, being in first person, the agent will not be aware of something unless it is in its field of view.
- The goal of the game is also conceptually different compared to the objective of Mario, in that there is no set objective in Minecraft.
- Being a survival sandbox game, the main underlying goal is that the player can survive, but they can do so as they will.

Due to the freedom of choice that the game offers, a simple task has been devised for this experiment: **chopping and collection wood**. The environment used in this experiment is derived from the MineRLObtainDiamondShovel-v0 environment in which the agent has to collect items in a sequential order to obtain the Diamond Shovel item starting from nothing. The goal and methods of this environment have been simplified greatly: the agent does not have an end goal of collecting one specific item, but the objective is to collect as much as possible of one item (any wood block). The player is spawned on a randomly generated world with a diamond axe in its inventory. The goal of the player is to collect as much wood as possible by finding a tree, walking up to it, chopping a block of wood and then collecting it from the ground by walking over or near it. The diamond axe has been provided to aid the agent in chopping the trees faster.

To further limit the randomness of the environment, the base code of MineRL was altered to allow for a constant seed to be set. Previously, a seed could be provided only for the random seed generator, which would result in a "random" generated world each time the agent would need to reset the environment. The passage of in-game time was also an issue because at night the agent cannot see its surroundings that well unless it is near a source of light and also hostile creatures can spawn, which can kill the agent and cause it to lose its diamond axe. To prevent this and also to allow the agent to better learn from its starting environment, after a set number of iterations, the world will be re-generated, allowing for past interactions to be reverted and for the day to start anew. However, there are other methods in which the agent can get stuck or die such as drowning, taking fatal fall damage or getting stuck in a hole. These events are unaccounted for because the agent can easily avoid or escape such situations.

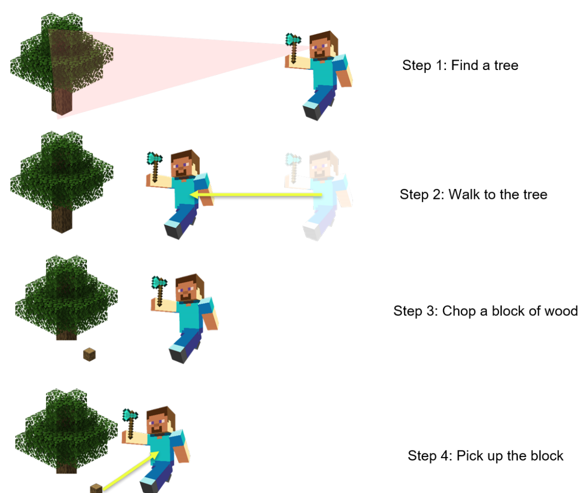


Figure 3.2: 4 step cycle of what an agent would need to do to gather wood in Minecraft.

Regarding the state and action spaces of MineRL, the states are being generated similarly to Mario: four concatenated 84 by 84 grayscale images of rendered game-play. The action space is much larger compared to Mario, but, using the wrappers provided in the Stable Baselines tutorial, actions have been limited to a set of eight actions.

In Minecraft, to break a block, the player needs to continually "attack" that block. This adds an additional layer of complexity that can be avoided by telling our agent to continually attack while performing any other action regardless of what block is being hit. This however creates the risk that when the agent is continually looking down, a hole can

be made that traps the agent. To avoid this, a limit of 45 degrees has been added to the negative pitch movement of the camera. This ensures that the agent can harvest blocks that are slightly lower than its eye level without digging a hole while allowing it to look up and harvest higher blocks of wood.

As mentioned before, the reward is earned according to the amount of wood the agent has collected during an episode. A secondary, smaller reward, can be acquired if the agent picks up sticks which can be obtained by breaking leaf blocks that grow around trees. This smaller reward has been added to encourage the agent to remain close to trees while looking for wood knowing that, to harvest higher blocks of wood, some leaf blocks should be destroyed beforehand.

3.3 General Model structure

After the input image stack produced by the environment is preprocessed, it is fed into a four layer convolution neural network (CNN) that ends with a Long Short-Term Memory (LSTM) layer that connects to the Linear Actor and Critic layers. The optimizer used is Adam.

Parallelization is done by creating a global model in the main function of the algorithm while the actual training will be done by worker processes. Multiple threads will be launched with a reference to this global model. The worker threads will create local models that take part in the learning process, while the global model will receive asynchronous updates from each thread whenever an episode of a worker agent is over. At the same time, the worker agents will update themselves with the global model's parameters whenever they finish their episode, after which a brand new episode will start.

3.4 Computational Resources

The training of the A3C models on Mario and Minecraft has been done with the help of the Hábrók high-performance computing cluster of the University of Groningen. Both environments were executed on the GPU clusters. Three experiments were conducted within the Mario environment which had access to 6 CPU cores with 2GB of memory for each core. The experiments used 4 and 12 worker agents. Two experiments have been done within the Minecraft environment which had access to 8 CPU cores with 10GB of memory for each core. The algorithm in this case used 4 and 6 worker agents.

4 Results

4.1 Mario

In the scheduled time, the Mario model with 12 worker agents trained for close to 40 000 episodes and had 33 032 trainable parameters per agent. The LSTM layer has not been accounted for when summing the number of parameters, check appendix C for more information. The graph in Figure 4.1 shows the rolling mean reward earned by a worker agent over its training period with a window size of 20 elements (Check appendix B for more information on rolling algorithms). From the graph, it can be observed that, from around episode 5 000, the agent had hit a plateau earning between 450 and 600 mean reward per episode. From graph 4.1 we can also see that the rewards earned by the agent suffered from high standard deviation during training.

Taking a closer look at the initial training period, we can see a certain stepping stone that the agent had to achieve to progress further into the level and earn higher rewards. Figure 4.3 depicts the first 11 thousand training episodes of the 12 and 6 worker agents models respectively. The section of the level where the agents get stuck temporarily is a hole in the ground that Mario can fall into if the player jumps too soon. We can observe where the models had to adapt to that gameplay charge around the 2 000 episode mark for the 12 agent model and around the 4 500 episode for 6 agent model. Once the agent learned how to pass that obstacle, the model was able to reach the plateau by completing the level multiple times.

Another metric that was being tracked throughout the experiments in the Mario environment was the amount of level clears the agent was able to achieve up to an episode. A level clear was counted when the agent reached the end flag. Because Mario was tested with three different numbers of working agents, we can compare the speed at which the agents within the different experiments began com-

pleting the level more often than losing it. From figure 4.2 we can observe that a higher number of agents has a positive impact on the time it took for a model to reliably complete the level. For 4 agents, the model was able to complete the level more often after about 5 000 episodes while the model with 6 agents was able to start clearing the level from episode 4 000. As expected, the model with 12 agents was able to clear the level more often much sooner than the other two models at just after 2 000 episodes were over.

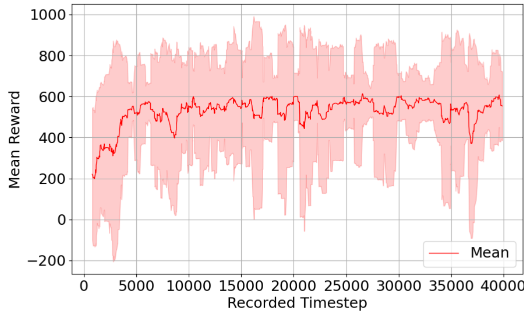


Figure 4.1: Mario with 12 agents: Graph of the reward per episode earned by the agent and the standard deviation. Rolling window size of 20.

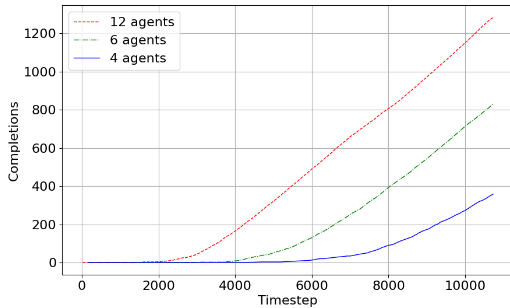


Figure 4.2: Mario with 12, 6 and 4 worker agents: Graph of the amount of completed runs up to an episode for the first 11 thousand episodes.

4.2 Minecraft

In the scheduled time, the models with 4 and 6 worker agents trained for over 4 000 episodes in

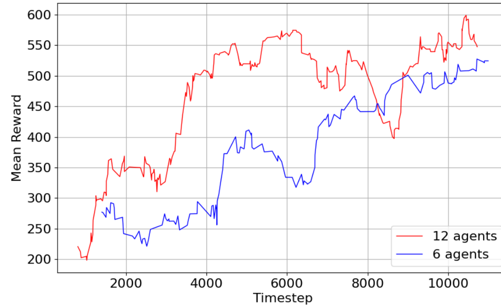


Figure 4.3: Mario with 6 and 12 agents: Graph of mean reward over the first 11 thousand episodes.

the Minecraft environment, the model having 33 545 trainable parameters per agent. The Minecraft models have slightly more parameters compared to Mario due to there being one additional action available to the Minecraft agent. The graph in Figure 4.4 show the rolling window mean, with a window of 50 elements, of the rewards earned by the models over their training period.

From the graph, it can be observed that the agent has not reached a ceiling like in the Mario environment, thus learning could have been continued. Looking closer at the 4.4 graph, we can see that the model with 4 working agents was able to acquire an average reward of 30, which translates to an average of three wood blocks per episode and the model with 6 working agents was able to acquire an average reward of 45, which translates to an average of four and a half wood blocks per episode towards the end of their training period.

Similarly to the Mario environment, the effect of high variance during training persisted in the Minecraft experiments. Figure 4.5 shows the standard deviation of the 6 agent model during its training period. We can observe that the deviation amplifies as the reward increases.

5 Conclusions and Discussion

5.1 A3C

The algorithm performed as expected in our experiments, being capable of learning tasks such as finishing a level in a video game in a relatively fast

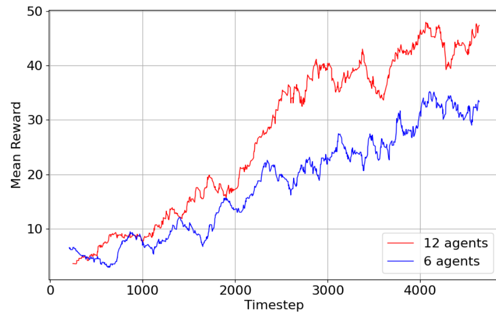


Figure 4.4: Minecraft with 4 and 6 agents: Graph of the reward per episode earned by the agent in 4500 episodes. Rolling window of 50.

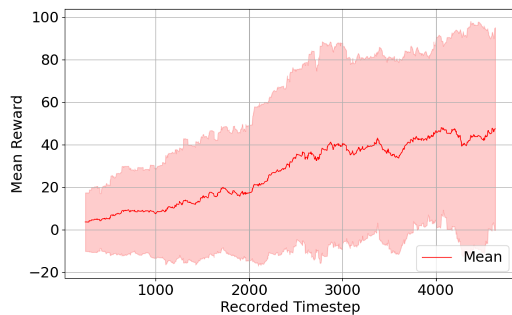


Figure 4.5: Minecraft with 6 agents: Graph of the reward per episode earned by the agent in 4500 episodes and the standard deviation. Rolling window of 50.

time. For example, during the training within the Mario environment, the agent was able to begin consistently finishing the level after about 2 000 episodes as shown in Figure 4.2.

One of the main issues with the current implementation is the high variance of the model during training. We can observe this in Figure 4.1 where the area around the line, which denotes the standard deviation of a window, covers a large surface of the graph. This means that the algorithm achieved vastly different reward amounts in each episode. The cause of this variance during training was due to the exploration/exploitation strategy in play. Even if the agent has found a suitable policy, it still attempts to look for something better. These attempts caused some episodes to end early and award a much lower end reward. Unfortunately, due to time constraints, an evaluation algorithm that tested the policy without exploration in mind could not be developed, thus the performance of the models was not tested after training.

The deviation in the Minecraft experiment depicted in Figure 4.5 can be explained as the agent having less chances of cutting a block of wood than actually choosing a suitable action. This is caused by the design choice of starting a new episode from the last state of the previous episode, thus nearby trees could have been cut already. This explains the cone shape that the standard deviation area has created around the mean reward line. At the beginning of the training period, the agent collected less wood, so it was closer to a total reward of 0 per episode. Due to this, episodes where the agent was not near wood are similar to episodes where the agent was near wood during the initial stages of training. As the agent got more sophisticated, the reward per episode while the agent was near wood increased, but the reward when the agent has no wood to collect was as low as before, causing a higher gap in rewards between episodes. Also, the agent was still learning and it had not hit a plateau like in the Mario experiments, so wrong actions were still being made in order to gather information.

All worker agents employed the same exploration/exploitation procedure: sampling from a softmax distribution. For environments with very large state spaces, this was not a big issue as small differences in actions would lead the agents in very different states, but for environments with smaller state spaces such as Mario, worker agents tended to do the same actions. This can be improved by having different exploration policies for each worker agent to remove possible correlations in later iterations of the environment.

5.2 The impact of more Worker Agents

Through this project, multiple experiments with different amounts of worker agents were conducted. One of the main advantages of A3C is its parallelization, thus testing the influence of increasing the number of parallel working agents has on the task being learned is of interest.

For the Mario experiments, the algorithm that ran the model kept track of whenever an agent fin-

ished a level. Each agent had a counter of level completions that would increment by one whenever that agent finished a level. As a reminder, finishing a level consisted of the player touching the flag at the end of the level. In figure 4.2 the level completion counter for each time-step is plotted. It is immediately clear that models with fewer agents, such as the one with 4 agents depicted as the continuous blue line, began completing the level much later than the models employing more worker agents, such as the model with 12 worker agents represented by the red dotted line.

Another impact of having more agents can be seen in figure 4.3 where the 12 agent model was able to get over an obstacle that would require the agent to learn a slightly different strategy faster than the 6 agent model.

The same trend can be observed in figure 4.4: after about 1 500 episodes the model with 6 worker agents began earning more mean reward per episode compared to the model with 4 agents.

The main downside of multiple worker agents is the higher resource consumption caused by running more neural networks and especially more environments. In the case of Minecraft, the environments are very heavy because, not only it needs to render a 3d environment, but the game is also written using the Java programming language which is notorious for its memory consumption. A new working agent running the Minecraft environment would require, on average, 7 GB more ram. Thus much more advanced parallelization techniques should be used, such as running the worker agents on different machines or GPUs. For lighter environments such as Mario, resource consumption is not as bad, thus allowing models to utilize more resources to gather additional samples in order to train faster.

5.3 State Dimensionality difference

Mario and Minecraft are fundamentally different games, featuring a distinct set of actions and states from each other. Mario is a two dimensional game where the player can be seen from a third person perspective and only the first level has been used while training. Minecraft is a three dimensional game where the player is able to look around its environment from a first person perspective and even if the same world is generated each time, tiny movements in the camera or the player can yield a totally different state. From this comparison alone it should be clear that the state dimensionality of Minecraft is vastly greater than that of Mario. The action space sizes of both Mario and Minecraft have been modified to be similar to allow a comparison to be made between the state spaces.

Looking back at figures 4.1 and 4.5 we can observe a striking difference between the two experiments. The Mario model was able to find a policy that allowed it to complete the level and earn maximum reward relatively soon in its training period. If we think in the context of what frames of the game will be used as states for training, we can better understand the size of the state space. The episodes after the favorable policy has been found are filled with different attempts at exploring other solutions which is the cause of the high variance that can be observed from the standard deviation depicted in figure 4.1. Minecraft on the other hand has not hit a plateau during its learning period. The graph in figure 4.4 depicts the experiments in the Minecraft environment with 6 and 4 working agents.

As mentioned before, Mario is a 2 dimensional game with the main playable character visible at all times on the screen. The background of the level is the same, objects are placed in the same places and enemies move the same way in each episode, thus even if the agent does different movements, as long as it arrives in a similar position across different episodes, the states will be very similar or even the same. Minecraft on the other hand is a 3 dimensional game, allowing the player to move on both horizontally and vertically within the environment world. The camera perspective of the player is first person meaning that whatever the player can see in its field of view will consist a state for the agent. Due to this increased movement freedom, the agent can end up seeing vastly different states even after similar movements. Minecraft also has dynamic elements that cannot be controlled with a seed such as weather and other non-playable character spawns such as animals. The agent will end up needing to learn how to recognize objects within its vision, such as wooden blocks, rather than recognizing in which part of the level it is situated, for example before a hole.

The state inefficiency, combined with the heavy environment, made learning the environment in Minecraft difficult for A3C, but not impossible. Given enough samples, the model should be able

to be trained to complete the task of the environment.

5.4 MineRL

Minecraft is a vast and complex game that offers a unique experience for each player. The player is able to freely choose their actions and objectives. In Reinforcement Learning terms, this freedom is currently problematic since it implies a high state-action space dimensionality. As we have seen from the results discussed earlier, even after a lot of restrictions have been set in place, such as camera movement restrictions and re-generations of the environment, learning was still very slow, mainly due to the state inefficiency of the algorithm. The MineRL competition, which the MineRL framework and dataset have been made for, aims to research sample efficient methods for Reinforcement Learning. The main focus of exploration is research into imitation learning, as participants in the competition would have access to a subset of the large dataset provided by MineRL which contains human game-play recordings. Imitation learning is a combination between classic reinforcement learning and learning from expert behavior. The models are tasked with exploring the expert provided samples and test out policies within an environment.

5.5 Future work

As we have discussed in the methods section, we had to impose a relatively high amount of restrictions on the agent in order to avoid long training times. Details such as camera angle and the ability to destroy blocks have been simplified to aid the agent in its learning phase. Some of these restrictions constitute skills within themselves, such as breaking a block. In order to break a block, a player would need to continuously look at a specific block while holding down the harvest/attack button. If the player lifts that button, the block will regenerate and will need to start breaking it from full. An agent that would be able to get over this stepping stone and execute a more complex task afterward quickly would be an interesting research direction.

As we have mentioned before in subsection 5.4, MineRL is made to be used as a testing playground for researchers looking into sample efficient reinforcement learning methods in a competition style format. Sample efficiency is indeed an important aspect of reinforcement learning that should be investigated further.

Lastly, Minecraft has a hierarchical progression system: you need better tools to obtain better crafting ingredients for even better tools. Wood block is only but the first step in this progression ladder. Having an agent that can accomplish multiple steps is an interesting idea.

References

Amiranashvili, A., Dorka, N., Burgard, W., Koltun, V., & Brox, T. (2020, July). *Scaling Imitation Learning in Minecraft.* arXiv. Retrieved 2023-12-21, from http://arxiv.org/abs/2007.02701 (arXiv:2007.02701 [cs, stat])

Goecks, V. G., Waytowich, N., Asher, D. E., Park, S. J., Mittrick, M., Richardson, J., . . . Kott, A. (2021, October). *On games and simulators as a platform for development of artificial intelligence for command and control.* arXiv. Retrieved 2024-05-30, from http://arxiv.org/abs/2110.11305 (arXiv:2110.11305 [cs])

Guss, W. H., Houghton, B., Topin, N., Wang, P., Codel, C., Veloso, M., & Salakhutdinov, R. (2019, July). *MineRL: A Large-Scale Dataset of Minecraft Demonstrations.* arXiv. Retrieved 2023-12-21, from http://arxiv.org/abs/1907.13440 (arXiv:1907.13440 [cs, stat])

Johnson, M., Hofmann, K., Hutton, T., Bignell, D., & Hofmann, K. (2016, July). The malmo platform for artificial intelligence experimentation. In *25th international joint conference on artificial intelligence (ijcai-16).* AAAI - Association for the Advancement of Artificial Intelligence. Retrieved from https://www.microsoft.com/en-us/research/publication/malmo-platform-artificial-intelligence-experimentation/

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

Liao, Y., Yi, K., & Yang, Z. (2012, December). CS229 Final Report Reinforcement Learning to Play Mario.

Mao, H., Wang, C., Hao, X., Mao, Y., Lu, Y., Wu, C., … Tang, P. (2021, November). *SEIHAI: A Sample-efficient Hierarchical AI for the MineRL Competition.* arXiv. Retrieved 2023-12-21, from http://arxiv.org/abs/2111.08857 (arXiv:2111.08857 [cs, eess])

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., … Kavukcuoglu, K. (2016, June). *Asynchronous Methods for Deep Reinforcement Learning.* arXiv. Retrieved 2023-12-21, from http://arxiv.org/abs/1602.01783 (arXiv:1602.01783 [cs])

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

Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. (2019, December). *A Survey of Deep Reinforcement Learning in Video Games.* arXiv. Retrieved 2024-05-24, from http://arxiv.org/abs/1912.10944 (arXiv:1912.10944 [cs])

uvipen. (2021, January). *Super-mario-bros-A3C-pytorch.* Retrieved from https://github.com/uvipen/Super-mario-bros-A3C-pytorch

A Removal of entropy

The decision to remove the entropy from the loss function was made due to the strange empirical results gathered while testing the algorithm in the Mario environment. In Mnih et al. (2016) paper, it is mentioned that adding the entropy of the policy to the objective function can improve exploration and discourage sub optimal convergence, but for the algorithm used in this paper, the entropy is not used due to the model converging to a very undesirable set of action that would persist throughout the training process, effectively stopping any further exploration of the action space.

After the removal of the entropy, the agent was able to continue learning and reach a good policy.

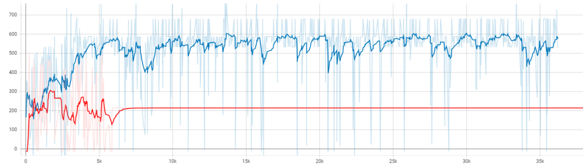


Figure A.1: Graph showing the difference between the algorithm with (Red) and without (Blue) adding the entropy to the objective function.

B Rolling Algorithm

The data collected from the experiments has been processed using the rolling algorithm from the pandas python package. Within this appendix, the algorithm and other terms related to it will be explained. The pandas dataframe.rolling() function provides the feature of rolling window calculations. A window is a chosen subset of the whole data that we can use in other calculations or processes. A window has a fixed size and it "rolls" over the whole data set computing a chosen function for that window. Algorithm B.1 is a basic example of how the rolling window algorithm works.

The algorithm consists of a dataFrame hash-map containing sequential data. This data is the used with the .rolling() function and the .max() function to find the maximum number withing the given window size and store that number into a separate hash-map. We can see that the first 2 elements have been ignored because the window starts calculating the maximum only starts calculating once it has three elements available. It iterates through the dataFrame and looks at the last window_size elements, applying the .max() function only on those three.

Algorithm B.1 Rolling window calculations

```
import pandas as pd

# create a DataFrame
dataFrame = pd.DataFrame('value': [1, 2, 3, 4, 5, 6, 7, 8, 9])

# use rolling() to calculate the rolling maximum

window_size = 3
rolling_max = dataFrame['value'].rolling(window_size).max()


# display the rolling_max
print(rolling_max)

'''
Output

0 NaN
1 NaN
2 3.0
3 4.0
4 5.0
5 6.0
6 7.0
7 8.0
8 9.0
Name: value, dtype: float64
'''
```

C Parameter counting

To find the number of trainable parameters, the summary() function from the torchsummary package. Torch-summary provides information complementary to what is provided by print(model) in PyTorch, similar to Tensorflow's model.summary() API to view the visualization of the model, which is helpful while debugging a network.

The function does not count parameters of hid-

den layers. The models used within this paper had an LSTM layer with hidden parameters, thus the number of training parameters reported within the results section (33 032 for Mario and 33 545 for Minecraft) are only the visible parameters.