UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Laur Sisask

Learning Competitive Minecraft Minigames with Reinforcement Learning

Bachelor's Thesis (9 ECTS)

Supervisor: Tambet Matiisen, MSc

Tartu 2022

Learning Competitive Minecraft Minigames with Reinforcement Learning

Abstract:
In recent years, deep reinforcement learning methods have successfully been used to play complex games like Go, StarCraft II, and Dota 2 at a professional level. In this thesis, reinforcement learning methods are used to train artificial agents in the game of Minecraft. Various competitive 1v1 Minecraft minigames from one of the most popular Minecraft servers Hypixel are selected. Deep neural networks are trained to play each of these games using proximal policy optimization algorithms and self-play. In all the games, artificial agents were able to play the game at least on a beginner level. In one game, the agent reached the level of expert human players.

Keywords: reinforcement learning, artificial neural networks, self-play

CERCS: P176 Artificial intelligence

Minecrafti võistlusmängude õppimine stiimulõppega

Lühikokkuvõte:
Viimaste aastate jooksul on stiimulõppe meetodeid edukalt kasutatud, et luua tehisintellekt mängimaks keerulisi mänge nagu Go, StarCraft II ja Dota 2. Selles töös rakendatakse stiimulõppe meetodeid, et treenida Minecraftile tehismängijad. Ühest populaarsemast Minecrafti serverist Hypixel valitakse välja erinevad üks-ühele minimängud, millele kõigile treenitakse tehismängijad. Treenimiseks kasutatakse piiritletud käitumisviisi optimeerimise (inglise keeles *proximal policy optimization*) algoritmi ja iseenda vastu mängimist (inglise keeles *self-play*). Kõik treenitud mängijad olid võimelised mängima valitud minimänge vähemalt algaja tasemel. Ühes mängus jõudis tehismängija ekspertmängijate tasemele.

Võtmesõnad: stiimulõpe, tehisnärvivõrgud, iseenda vastu mängimine

CERCS: P176 Tehisintellekt

Contents

1 Introduction

Minecraft[1] is a multiplayer sandbox game released in 2011. While Minecraft itself is an open game and does not limit what players can and should do, a number of smaller games within Minecraft with more restrictions and more concrete objectives have been developed. These games are commonly referred to as minigames and are played by over 1.1 million players every day.[2]

Most research towards Minecraft in reinforcement learning has been towards minigames that human players do not normally play (Kanitscheider et al., 2021; Matiisen et al., 2017; Perez-Liebana et al., 2019). The goal of this thesis is to train deep learning models to play the same minigames that real people normally play.

Building agents for games that are played by real people is important for two reasons. First, it paves the way for pairing reinforcement learning (RL) agents with human players. RL agents could stand in for human players when there are not enough players to play a certain game in a server. While it is possible to write handwritten algorithms to play minigames, it can be difficult to come up with an algorithm in games that require more sophisticated strategies. Additionally, agents with handwritten policies often have a monotonous game style and strategy that makes it not as fun to play against them.

Second, training agents to play same games as humans makes it possible to compare the performance of RL agents to humans. In competitive minigames, a game ends with one player losing and the other winning, making it trivial to compare the skills of an artificial agent to a real person.

In this thesis, a framework for running Minecraft minigames as reinforcement learning environments is developed. Artificial neural networks are then trained to play three competitive minigames from a popular Minecraft server using the developed framework. Depending on the game, the skills of the trained neural networks range from a beginner to an expert player. The neural networks are trained purely on visual input from the game, using proximal policy optimization algorithms (Schulman et al., 2017) and self-play (Tesauro, 1995).

This thesis is structured as follows. In chapter 2, the reinforcement learning problem and different algorithms to solve the reinforcement learning problem are presented. In chapter 3, the Minecraft minigames are described, and a framework to run these minigames as reinforcement learning environments is introduced. Neural networks are then trained to play minigames which is the focus of chapter 4. In the final chapter, the trained neural networks are evaluated, and the results are presented.

[1] https://www.minecraft.net/

[2] Information about daily player count retrieved from https://hypixel.net/jobs/

2 Background

This chapter gives an overview of reinforcement learning methods used in other parts of the thesis. First, the reinforcement learning problem is formulated. Then, two main branches of reinforcement learning algorithms are presented. Finally, existing research into self-play, curriculum learning, and reinforcement learning applied to Minecraft is reviewed.

2.1 Reinforcement Learning Problem

In broad terms, the goal of reinforcement learning is to find a way to act optimally in an environment. This section presents how environments are represented in reinforcement learning and what acting optimally means.

In reinforcement learning, environments are usually formulated as Markov Decision Processes (commonly referred to as MDPs). Watkins (1989) defines the MDP as a four-element tuple that consists of the following parts:

- Set of states $\mathcal{S}$.
- Set of actions $\mathcal{A}$.
- Reward function $\mathcal{R}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. $\mathcal{R}(s, a)$ is the expected received reward when the agent takes action a while in state s.
- The transition function $T\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. $T(s, a, s')$ is the probability of transitioning to state s' when taking action a while in state s.

While an MDP captures the environment, it does not say anything about acting in the environment. An agent is anything that interacts with an environment (Russell & Norvig, 2010, p. 34). The agent starts in some state $s \in \mathcal{S}$, takes action $a \in \mathcal{A}$, receives reward $r \in \mathbb{R}$ and a new state $s' \in \mathcal{S}$ (Sutton & Barto, 2018, p. 48). This process then repeats until the agent ends up in a terminal state (Sutton & Barto, 2018, p. 54). One sequence of these interactions is called an episode (Sutton & Barto, 2018, p. 54).

To formally describe how an agent chooses actions and what is the optimal way to act, the concepts of policy and return are introduced.

The following information is from Watkins (1989). Policy is a two-variable function $\pi\colon S \times A \to [0, 1]$, so that $\pi(s, a)$ is the probability that the agent takes action a when it is in state s. Return at time t is defined as the discounted sum of received rewards from time t until the end of the episode:

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

In here, $\gamma \in [0, 1]$ is a hyperparameter known as the discount factor. It controls the worth between immediately received rewards and those received in the more distant future. State-value function $v_\pi(s)$ is defined as the expected return when starting in state s and following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2}$$

$\mathbb{E}_\pi[X]$ denotes the expected value of random variable X while following policy π. S_t refers to the state at time t.

A policy is an optimal policy (denoted with π^*) if the value of its state-value function is greater or equal to state-value function values among all policies in all states (Sutton & Barto, 2018, p. 62):

$$v_{\pi*}(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \tag{3}$$

The goal of reinforcement learning is to find the optimal policy. To find the optimal policy, there exist two branches of reinforcement learning algorithms: value-based methods and policy gradient methods (Sutton et al., 2000). The next two sections give an overview of both value-based and policy gradient methods.

2.2 Value-based Methods

In value-based methods, an action-value function is learned (Sutton & Barto, 2018, p. 321). Action-value function $q_\pi(s, a)$ is the expected return when the agent starts in state s, takes action a, and follows policy π from there onwards (Sutton & Barto, 2018, p. 58):

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{4}$$

Here, S_t refers to the state at time t and A_t to the action taken at time t.

Typically, the value function is learned from experience using differentiable function approximators such as neural networks. In the most simple case, a number of timesteps are rolled out and the return is calculated for every timestep. Stochastic gradient descent is then used to train the value function approximator using mean squared error as a loss function (Sutton & Barto, 2018, p. 202):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} \left[G_t - \hat{q}_\pi(S_t, A_t, \boldsymbol{w})\right]^2 \tag{5}$$

Here, $\hat{q}$ is the function approximator with parameters $\boldsymbol{w}$, α is the learning rate, and G_t is the true return at time t. Other methods which do not use true returns also exist. For example, the SARSA algorithm uses $R_t + \gamma \hat{q}_\pi(S_{t+1}, A_{t+1})$ in place of G_t (Rummery & Niranjan, 1994). Methods that make use of an estimate of a future state to estimate the target value (like SARSA) are called temporal-difference learning methods, while

methods that do not do that are called Monte Carlo methods (Sutton & Barto, 2018, p. 119).

The learned action-value function is used to construct the policy. As previously stated, the goal is to find a policy that maximizes the value function. A natural way to make the policy maximize value function is to make the policy pick the action that yields the most value, that is, the action with highest action-value estimate (Sutton & Barto, 2018, p. 79):

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \hat{q}_\pi(s, a, \boldsymbol{w}) \tag{6}$$

Such policies are called greedy policies. During training, as the action-value estimate becomes more accurate, the policy is improved.

The deterministic nature of greedy policies poses one problem — the policy can get stuck choosing a suboptimal action because it never explores other options. This problem is known as the exploration–exploitation dilemma — in order to maximize return, the agent should pick actions that yield the most return but to find such actions, it needs to take other actions that are currently not considered optimal (Sutton & Barto, 2018, p. 3). To solve that problem, non-deterministic policies such as ε-greedy are used. Sutton and Barto (2018, p. 100) define a policy to be ε-greedy if it picks the action with highest value estimate with probability $1 - \varepsilon$ and with probability ε a random action. By giving every action a small chance of being chosen, the agent continues to explore the environment and will not converge on a suboptimal policy as easily.

2.3 Policy Gradient Methods

The following is based on Sutton et al. (2000). Compared to value-based methods, policy gradient methods have an explicit policy instead of constructing one based on the learned value function. That is, there is some differentiable function $\pi(a \mid s, \boldsymbol{\theta})$ that outputs the probability of taking action a, given state s and parameters $\boldsymbol{\theta}$. The parameters of that function are optimized to maximize the value of the start state of the episode using stochastic gradient ascent.

Policy gradient methods are based on the policy gradient theorem which states that the gradient of the value function in some arbitrary state s with respect to policy's parameters $\boldsymbol{\theta}$ is proportional to (Sutton & Barto, 2018, p. 326):

$$\frac{\partial v_\pi(s)}{\partial \boldsymbol{\theta}} \propto \sum_{s \in \mathcal{S}} \mu_\pi(s) \sum_{a \in \mathcal{A}} q_\pi(s, a) \frac{\partial \pi(a \mid s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{7}$$

Here, $\mu_\pi(s)$ refers to the probability of being in state s when following policy π. Based on the policy gradient theorem, several algorithms such as REINFORCE (Williams, 1992), asynchronous advantage actor-critic (A3C) (Mnih et al., 2016), and proximal policy optimization (PPO) (Schulman et al., 2017) have been developed.

REINFORCE uses the following rule for updating the policy (Sutton & Barto, 2018, p. 328):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \, \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \tag{8}$$

Here, α is the learning rate, G_t the return, A_t the action taken at time t, S_t the state the agent was in at time t, and $\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})$ denotes the gradient of the policy with respect to its parameters $\boldsymbol{\theta}$.

The following is based on Mnih et al. (2016). Actor-critic algorithms learn a state-value function in addition to the policy. The learned value state function is denoted as $\hat{v}$ and its parameters as $\boldsymbol{w}$. The asynchronous advantage actor-critic uses the following rule for updating the policy:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[R_t + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})] \frac{\nabla_{\boldsymbol{\theta}} \, \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \tag{9}$$

The term $R_t + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$ replaces the return G_t from REINFORCE. The $R_t + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$ part of the expression is the bootstrapped estimate of true return G_t. $\hat{v}(S_t, \boldsymbol{w})$ term is subtracted to keep the variance of the gradients small.

The following is based on Schulman et al. (2017). Proximal policy optimization algorithms improve the performance of policy gradient algorithms by constraining the size of policy updates and using the same experience several times during the training phase. Constraining the size of policy updates means that the probabilities of taking some action in some state before and after the iteration are similar.

To give an update rule for PPO, the probability ratio function is first defined. Probability ratio $r_t(\boldsymbol{\theta})$ is defined as:

$$r_t(\boldsymbol{\theta}) = \frac{\pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta}_{\text{old}})} \tag{10}$$

$\boldsymbol{\theta}_{\text{old}}$ refers to the policy's parameter before the start of the current iteration. The update rule for PPO is:[3]

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \left[\min(r_t(\boldsymbol{\theta}) \hat{A}_t, \text{clip}(r_t(\boldsymbol{\theta}), 1 - \varepsilon, 1 + \varepsilon) \hat{A}_t) \right] \tag{11}$$

Here, $\hat{A}_t$ is to an advantage estimator. Advantage estimator is a more general term for the expression that the gradient was multiplied by in the advantage actor-critic algorithm. For advantage actor-critic, $\hat{A}_t = R_t + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$. The concept of advantage estimator is introduced because other expressions besides the one from advantage actor-critic may be used instead.

[3]This is the rule for the "clip" version of PPO. The PPO paper also introduces another algorithm that is not discussed in this thesis.

The advantage is multiplied by the probability ratio to allow reusing the same training data for several training epochs. Policy gradient theorem requires that the data used to calculate the gradient is collected by following the same policy that is optimized. However, if the policy is trained for several epochs, then during later epochs, the policy may have shifted, and that can make the gradients inaccurate. To take into account the fact that the policy which is updated is not necessarily the same as the one that was used to collect the data, it is necessary to scale the advantages with the probability ratio. This approach is essentially importance sampling, a technique used in most off-policy reinforcement learning methods (Sutton & Barto, 2018, p. 104).

Similar to value-based methods, policy gradient methods roll out some number of timesteps and then use that experience to update the policy according to the update rules (Sutton & Barto, 2018, p. 332). This process is repeated for a number of iterations (Sutton & Barto, 2018, p. 332).

Unlike value-based methods, policy gradient methods use explicit policies that are stochastic by construction. While that can balance exploration and exploitation during learning, policy gradient methods still often converge on a suboptimal policy early on and require other techniques to balance exploration and exploitation (Mnih et al., 2016). One such technique is adding an entropy bonus to the training objective (Mnih et al., 2016). Using REINFORCE as an example, the full gradient with entropy bonus would then be:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = G_t \frac{\nabla_{\boldsymbol{\theta}}\, \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} + \beta\, \nabla_{\boldsymbol{\theta}}\, H(\pi(S_t, \boldsymbol{\theta})) \tag{12}$$

Here, J is the training objective, H is entropy, and β is a hyperparameter. Higher values of β lead to more random policies.

2.4 Generalized Advantage Estimation

As mentioned in the previous section, there exist several ways of estimating the advantage. Schulman et al. (2016) define advantage as the difference between the value of the action taken and the value of the state the action was taken in:

$$A(s, a) = q_\pi(s, a) - v_\pi(s) \tag{13}$$

In practice, $q_\pi(s, a)$ and $v_\pi(s)$ are not known, so advantage is estimated using various methods, like one-step return in asynchronous advantage actor-critic algorithm. An alternative is to use k-step advantage estimates that is a generalization of one-step returns. Schulman et al. (2016) define the k-step advantage estimate as:

$$\hat{A}_t^{(k)} = \sum_{i=1}^{k} \gamma^{i-1} R_{t+i} + \gamma^k \hat{v}(S_{t+k}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w}) \tag{14}$$

The difference between the different k-step estimates is the estimator's bias and variance. Lower values of k lead to more biased estimates with low variance while high values of k lead to unbiased estimates with high variance. To obtain estimators that have low variance but are not too biased, Generalized Advantage Estimation (Schulman et al., 2016) is used. Generalized Advantage Estimation (GAE) calculates the advantage estimate using an exponentially-weighted sum of all different k-step estimates. Formally, it is defined as follows:

$$\hat{A}_t^{\text{GAE}} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots) \tag{15}$$

Here, $\lambda \in [0, 1]$ is a hyperparameter that is used to control bias and variance. Higher values of λ lead to estimates with lower bias but higher variance.

2.5 Related Work

Training agents to play Minecraft with reinforcement learning methods has been explored by several researchers. Similar to work done in this thesis, most research has been directed towards Minecraft minigames where there exists a clear notion of reward. Kanitscheider et al. (2021) trained agents to obtain specified items in Minecraft. Guss et al. (2020) developed a platform for training Minecraft agents to accomplish tasks such as navigating to given location, building a house, obtaining a diamond, and chopping down a tree. In 2019, 2020, and 2021, they also hosted a competition[4] for building models to solve these tasks.

All the sources mentioned in the previous paragraph focused on single-player minigames. In addition to single-player, multiplayer games have also been researched. In 2017, Microsoft hosted The Malmo Collaborative AI Challenge[5]. In that challenge, contestants had to build an AI for a game where two players have to trap a Minecraft pig into a corner. Similarly, Perez-Liebana et al. (2019) organized a contest in which participants had to train agents for various collaborative and competitive minigames, including a build battle game where two teams were competing to build a given cuboid structure within a time limit.

Self-play has been used to train agents in various multi-agent competitive environments. Bansal et al. (2018) trained agents to play player-vs-player games in MuJoCo environments. Silver et al. (2016), Vinyals et al. (2019), and OpenAI et al. (2019) applied self-play along learning from human experience to play the games Go, StarCraft II, and Dota 2, respectively. The self-play related approaches in this thesis draw on the work from all these four papers.

In complex environments, curriculum learning can be used to speed up training. Curriculum learning refers to letting the model first learn more simple tasks and then

[4]https://minerl.io/

[5]https://www.microsoft.com/en-us/research/academic-program/collaborative-ai-challenge/

increase the difficulty as training progresses. In this thesis, two different versions of curriculum learning are used. Firstly, by using self-play it is ensured that the agent plays against an opponent with similar skills, allowing the agents to continuously make progress. The same approach has been previously used by Bansal et al. (2018).

The second form of curriculum learning is to increase the complexity of the environments as training progresses. Similar techniques have been previously used by Bansal et al. (2018) in MuJoCo simulator environments and in Minecraft by Matiisen et al. (2017) and Kanitscheider et al. (2021), albeit in a single-player setting.

In this thesis, a new framework to run Minecraft as a reinforcement learning environment is developed. There are also existing frameworks with similar purposes, with the most notable ones being Malmo (Johnson et al., 2016) and MineRL (Guss et al., 2019). A new framework is developed because the existing ones were not flexible for defining all the minigames.

3 Environments

Hypixel features over 50 different minigames where players can compete against each other. In this section, three minigames are selected and explored in detail. The games are presented from the reinforcement learning environment viewpoint — every minigame is considered to be a reinforcement learning environment with some specified observation structure and action structure. Additionally, the technical implementation of these environments is presented.

3.1 Minigame Descriptions

The following minigames from Hypixel were picked: Sumo, Classic Duels, The Bridge. All of these games are part of the Hypixel Duels suite — a collection of 1v1 player-vs-player games which usually last 1–15 minutes. Games from the Duels suite were picked because those are complex enough that writing a handwritten policy for them would be unfeasible but they are still simple enough that they hopefully could be learned without using prior human data. The descriptions of minigames are as follows.

Sumo Two players are teleported to a round platform. Players do not have any equipment but they are able to hit each other with fists. Hitting the other player causes them no damage but deals the same amount of knockback as usual in Minecraft. A player wins the game when the other player falls off the platform and loses when they themselves fall off the platform. If both players are still on the platform after a game has lasted 2 minutes, the game ends in a draw. There are 5 premade unique maps on which the game can be played. Two Sumo maps are shown in figure 1.

Figure 1. Sumo maps.[6]

Classic Duels Two players are teleported to an arena and are given iron armor, a sword, a fishing rod, a bow, and 5 arrows each. The players can damage each other according to

[6]Images retrieved from https://hypixel.net/threads/duels-update-v1-4-new-modes-game-improvements-divisions-and-more.4456969/

normal game mechanics. A player wins the game if the other player dies and loses if they themselves die. If both players are still alive after 8 minutes, the game ends in a draw. There are 10 premade unique maps on which the game can be played. Two Classic Duels maps are depicted in figure 2.

Figure 2. Classic Duels maps.[7]

The Bridge Two players are teleported to a map that consists of two identical islands and a 1 block wide bridge between those islands. Both players are assigned an island that they have to protect. The players start out on that island. On both islands, there is a hole. If a player jumps into the hole on their opponent's island, they score a point and the players are teleported back to their respective islands for a new round. The player that reaches four points first, wins the game. If neither of the players reach four points in 15 minutes, the game ends in a draw. The players are given leather armor, iron sword, diamond pickaxe, clay blocks, bow, arrows, and golden apples. The same PvP mechanics apply as in vanilla Minecraft. If a player dies, they are teleported back to their island. Unlike in Sumo and Classic Duels, in this game, players are able to break and place blocks. There are 17 premade unique maps on which the game can be played. Two The Bridge maps are shown in figure 3.

Figure 3. The Bridge maps.[8]

[7]Images retrieved from https://hypixel.net/threads/duels-update-v1-3-the-bridge-duels-new-maps-map-terrain-and-more.1854874/

3.2 System Architecture

The environments for minigames were implemented from scratch so that they could be played locally. While it would be possible to train the agents by playing in Hypixel directly, it is not feasible for the following reasons:

- The training system would have to be resilient against network failures and other issues like getting kicked out of the server.
- It would require running the game at its normal speed (20 ticks per second), slowing down training.
- Switching to the next game can take a long time (around 10 seconds) that can slow down training.

The system for running the environments consists of three components:

1. Minecraft server which orchestrates the games and manages game lifecycle
2. Minecraft client which takes actions (moving the player, hitting, etc)
3. A program that sends actions to the Minecraft client and receives back observation data from the client

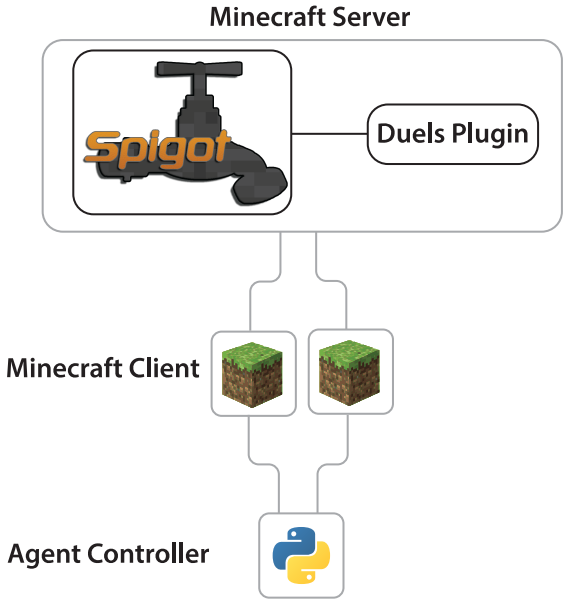


Figure 4. Overview of the Minecraft multi-agent reinforcement learning framework.

[8]Images retrieved from https://hypixel.net/threads/the-bridge-v0-2-new-maps-kit-editor-and-more.1613764/

The overall architecture of the system is presented in figure 4. The program that sends the actions can be any Python program. In the next sections, the Minecraft client and Minecraft server are explained in more detail.

3.3 Minecraft Client

A vanilla Minecraft client with a few modifications was used for taking actions in the environments. The following modifications were made to the vanilla Minecraft client:

- It was possible for other programs to control the client, i.e., another program could move the mouse and keyboard and receive the values of pixels on the screen.
- The game was optimized to consume fewer resources in order to accelerate it to a higher tick rate and allow for several clients to run in parallel on one machine.

The following performance optimizations were done:

- The game was changed to render only once after every timestep.
- Animations (such as water flowing) were turned off.
- Block light updates were disabled as the game was run with maximum brightness where the brightness of a block never changes.
- Particles were removed from the game.

These optimizations allowed the game to run at 400 ticks per second compared to the default of 20 ticks per second.

The Minecraft client and an external program that controls the agents communicated over the network. More specifically, all data was sent via TCP and was encoded using Google Protocol Buffers.

3.4 Minecraft Server

The Minecraft server was responsible for managing the game lifecycle. A modified version of Spigot[9] was used for the server. The modified version included support for changing the tick rate of the Minecraft server in configuration files. That modification was made to accelerate the game for training.

The game logic was implemented as a Spigot plugin. The plugin was responsible for moving players into games and managing the game lifecycle. For example, it teleported players to correct locations once the game started and notified the players of the outcome of the game when one player died. The code for the plugin can be found at https://github.com/laursisask/agi/tree/master/duels-hub-plugin.

[9]https://www.spigotmc.org/wiki/about-spigot/

3.5 Comparison with Malmo

The created Minecraft reinforcement learning framework is similar to the existing Malmo framework. This section gives some insights into why it was decided to build a new framework from scratch instead of using an existing one. All the arguments in this section also apply to the MineRL framework (Guss et al., 2019), as that is built on top of Malmo.

The first reason is that Malmo is not flexible enough. Although it allows constructing a wide variety of Minecraft environments, it is still limited by predefined rules. If there is some behavior that cannot be defined using those rules, the environment cannot be constructed. For example, it is not possible to specify that hitting the other player deals knockback but not damage, as is desired in the Sumo minigame.

The second drawback of Malmo is its performance. Malmo has not been optimized for multi-agent tasks which has led to several inefficiencies that can significantly slow down training. For instance, in multi-agent environments, all concurrently running games start their own Minecraft server which all use up a lot of CPU resources. Having the players join the same server would save CPU. From experiments, it was observed that a single Minecraft server takes similar amount of CPU, regardless of how many concurrent minigame sessions there are. Another related issue is that in Malmo, one of the agents rejoins the server after every episode, adding an unnecessary delay between episodes.

Table 1. Differences in performance between Malmo and the framework developed in this thesis. The relative TPS (ticks per second) shows how much faster is a system than a single vanilla Minecraft client.

System	TPS per client	Number of clients	Overall TPS	Relative TPS
Vanilla Client	20	1	20	1x
Malmo	40	6	240	12x
Created framework	400	10	4000	200x

Malmo also does not implement any of the performance optimizations that were described in section 3.3. The lack of these optimizations in Malmo leads to poorer performance when compared to the created framework. On a computer with NVIDIA RTX 2070 GPU and AMD Ryzen 5950X CPU, when running a single Classic Duels environment with random policy, Malmo is able to reach a tick rate of 40 while the framework developed in this thesis is able to reach a tick rate of 400. Furthermore, it can scale up to 10 replicas without causing tick rate to drop more than 10%. The corresponding figure for Malmo is 6 replicas. The differences in performance between Malmo and the developed framework are captured in table 1.

3.6 Action Structure

In the introduced environments, the Minecraft client is controlled in the same way as when played by a real person. Conceptually, the agent is able to specify which keys to press and how much to move the mouse on every timestep. Actions consist of several components that all control some part of the player's behavior. For example, one component controls the forward movement and another one the left-right movement. The components are as follows:

- Whether the agent should move forward, do nothing, or move backward.
- Whether the agent should move left, do nothing, or move right.
- Whether the agent should attack (left-click), use (right-click), or do nothing.
- Whether the agent should start sprinting (press left control).
- Whether the agent should jump (press space).
- The change in pitch (rotation around the horizontal axis).
- The change in yaw (rotation around the vertical axis).
- The index of the hotbar slot which should be selected. This action is used to switch the item that the player holding. The value of this component is given by an integer from 0 to 8.

On every timestep, the value of all these components is set and the client presses the keys and moves the mouse according to these values.

3.7 Observation Structure

Observation structure is shared between all the minigames — it is the raw visual input from the game, similar to how humans perceive Minecraft. The game is played at a resolution of 336x336 but the image is downsampled to 84x84 and converted to grayscale. Such preprocessing approach was taken from Mnih et al. (2013). An example observation is shown in figure 5.

(a) Normal output from the game

(b) Grayscaled output

(c) Grayscaled and downsampled output

Figure 5. Example of an observation from the environment and comparison with how the input usually from the game looks like.

A new image is received on every timestep, i.e., when an episode starts and after every time the agent takes an action. For better performance, downsampling and grayscaling of the image were already implemented in the Minecraft client to decrease the network bandwidth between the Minecraft client and the program which controls the client.

3.8 Dynamic Elements

The environments are mostly static, meaning that the maps look identical across episodes. However, there are two elements in every environment that can change across episodes. These are player skins and emblems.

In Minecraft, players can change their appearance, commonly referred to as skin. Initially, the environments were implemented without considering skins, i.e., all agents had default Minecraft skins. This led to a discrepancy between Hypixel and the local environment, as most human players do not use default skins. While the agents were able to play against most players, there were cases where the agent started acting suboptimally due to the opponent having a previously unseen skin. To address that problem, the environments were changed to rotate the skin of a player every time it joined a new game.

Figure 6. Examples of skins.

The new skin for a player was sampled out of a dataset which contained 10 000 skins. The dataset was obtained by scraping the NameMC skin collection random skin page[10] until 10 000 unique skins had been observed. Examples of skins are shown in figure 6.

Another dynamic element is the emblem. An emblem is a structure made of blocks that is located high in the air above the game arena. In Hypixel, players can unlock emblems and choose one emblem that is shown in games they have joined. Some examples of emblems are shown in figure 7.

Figure 7. Examples of emblems.

Like with skins, emblems were not initially implemented as part of the environments which led to a discrepancy between Hypixel and the local environment. Agents that had been trained without seeing emblems were able to generalize in most games, even if there was an emblem. However, sometimes they started acting suboptimally, either just randomly or mistaking the emblem for the opponent. To counter that, emblems were implemented in local environments. The emblems were changed every 80 seconds in game-time, and for each emblem spot, there was a 40% chance that an emblem was displayed. The same emblems were used locally as those available in Hypixel.

[10]https://namemc.com/minecraft-skins/random

4 Experiments

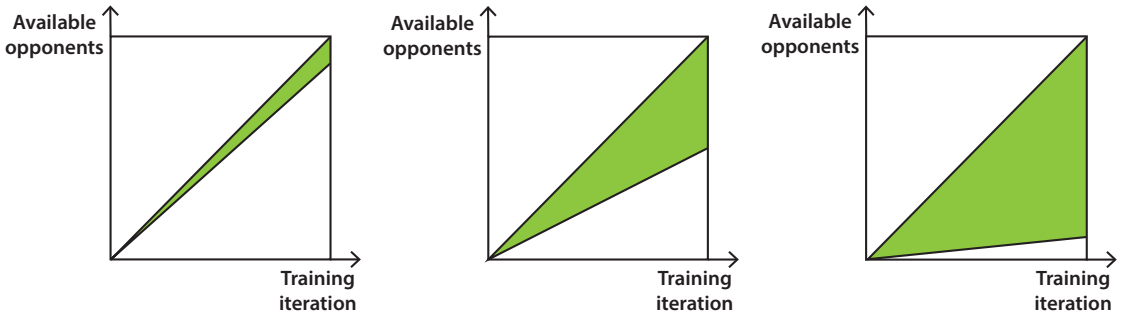
In this chapter, a scheme for training neural networks to play the minigames Sumo, Classic, Duels, and The Bridge is proposed.

4.1 Self-Play

All the minigames were competitive multi-agent games. There are two high-level ways to train agents for such multi-agent environments: supervised learning and self-play. With supervised learning, data from human players is used to train the model. More specifically, the model is trained to imitate humans. With self-play, the agent learns the game by playing against previous versions of themselves (Bansal et al., 2018). The learning is still done the same way as in single-agent reinforcement learning.

In self-play, it is important that the agent plays against a pool of opponents and not only itself to prevent the agent from overfitting to specific policies (Bansal et al., 2018). If the agent only played against themselves, they would learn to play against themselves but not necessarily generalize to opponents with different policies.

As there was no public human data available for Hypixel Duels games, the games were learned using self-play. Self-play was also selected because it provides a natural curriculum for learning. That is, by pitting the agent against earlier versions of themselves, the agent always plays against a player with similar skills which makes it easy for agents to learn new behaviors and improve their abilities (Bansal et al., 2018).



(a) $p = 0.1$ (b) $p = 0.5$ (c) $p = 0.9$

Figure 8. Diagrams showing out of which models the opponent is sampled from for different values of p. For every training iteration, the part that is green represents the set of models from which an opponent is sampled.

During training, the opponent was uniformly sampled out of the last p fraction of models. For example, if p is 0.5, then the opponent is sampled from half of the previous models and if p is 1, then the model is sampled uniformly from all previous models. Such

sampling scheme was adopted from Bansal et al. (2018). An illustration of how p affects which models can be sampled is shown in figure 8.

The choice in p is ultimately a trade-off between the skill of the opponents and the diversity of their behaviors. With high p, the sampled opponents may be from a long time ago and thus less skilled than the current version of the agent. If the agent is easily able to defeat them, such a high p can inhibit learning because the agent does not need to improve their abilities to win the game. However, higher p leads to more diverse opponents and games which ensures that the agent can play against a diverse set of policies and not only itself.

In all three minigames, $p = 0.5$ was used. In Sumo, $p = 0.9$ was tried but it led to learning being very slow since most games ended with the current model almost immediately knocking the opponent off the platform.

4.2 Reward Function

Since reinforcement learning algorithms maximize the return and return is just a discounted sum of rewards, choosing a correct reward function is crucial. If the reward function is not aligned with the end goal (e.g., winning the game), the agent may learn to maximize the return without learning to complete the task.

In competitive minigames, the end goal is for the player to win the game. Naturally, the reward could be given as follows: positive reward for winning the game and negative reward for losing the game or when the game ends in a draw. While such a reward function is consistent with the objective, it has one other problem — it is too sparse to learn from. While an agent should theoretically learn to play the game, it can take a very long time to do so using only sparse rewards.

To make learning faster, the sparse reward was complemented with game-specific exploration reward that was used to make the agent develop basic motor functions, such as hitting the other player. This framework was adopted from Bansal et al. (2018). The reward function used in each minigame was as follows.

Sumo Player received a reward of +10 when they won the game and -10 when they lost or the game ended in a draw. Players were also given a reward x every time they hit the other player and a reward of $-x$ when the other player hit them. Initially, $x = 10$, but it was linearly annealed to 0 during the first 3000 iterations of training. Additionally, after the first 1000 iterations, the players got a negative reward every time they attempted to hit the other player (regardless of whether they hit the player or not). That reward was added because otherwise the agents attempted to hit the other player on every step which made them look less realistic. The negative reward was present during iterations 1000–2000. It was first linearly decreased from 0 to -0.5 and then linearly increased back to 0 during iterations 2000–3000.

Classic Duels Player received a reward of +10 when they won the game and -10 when they lost or the game ended in a draw. Reward was also given every time a player dealt damage to the other player and or the player got damaged by the other player. The reward given for damaging was equal to the amount of damage done and for receiving damage equal to the negative of the amount of damage done.

The Bridge Player received a reward of +200 when they scored a point (jumped to opponent's hole) and a reward of -200 when the opponent scored a point. To encourage exploration, players initially also received reward for moving based on if they moved closer to the opponent's hole or away from it. Similar to Classic Duels, reward was also given for dealing and receiving damage. Finally, players got a negative reward for falling to void or jumping to their own hole. The reward function for The Bridge is summarized in table 2.

Table 2. Reward function for The Bridge. Exploration rewards were multiplied by a coefficient that was linearly annealed from 1 to 0 during first 10 000 iterations.[11]

When given?	Amount	Is exploration reward?
Jumping to opponent's hole	200	No
Opponent jumps to player's own hole	-200	No
Falling to void	-20	Yes
Jumping to own hole	-20	Yes
Moving closer to opponent's hole	$0.5 \times$ distance moved closer	Yes
Moving away from opponent's hole	$-0.5 \times$ distance moved away	Yes
Damaging the opponent	$3 \times$ damage dealt	Yes
Receiving damage	$-3 \times$ damage received	Yes

4.3 Model Architecture

The core of an artificial agent is the model — a function that predicts which actions to take next, given the current state. Additionally, when using proximal policy optimization, the model also learns a state-value function because state-value estimates are used by PPO during the training phase.

The same model architecture was used in all games. A neural network that consisted of several convolutional layers followed by an LSTM layer was chosen. The architecture

[11] The Bridge was actually trained only for 3500 iterations but the exploration reward was still annealed with a rate that it would reach 0 on the 10 000th iteration.

for the convolutional layer stack was adopted from Mnih et al. (2015). It consisted of three convolutional layers, each one followed by ReLU nonlinearity. The first one had 32 filters of size 8x8 and stride 4. The second layer had 64 filters of size 4x4 and stride 2, and the final convolutional layer had 64 filters of size 3x3 with stride 1.

The convolutional layer stack was followed by a linear layer with 512 cells and ReLU activation function. That layer was followed by 256 LSTM cells. Finally, there was a linear layer that was used to generate a value for each of the policy parameters and predict the state-value function.

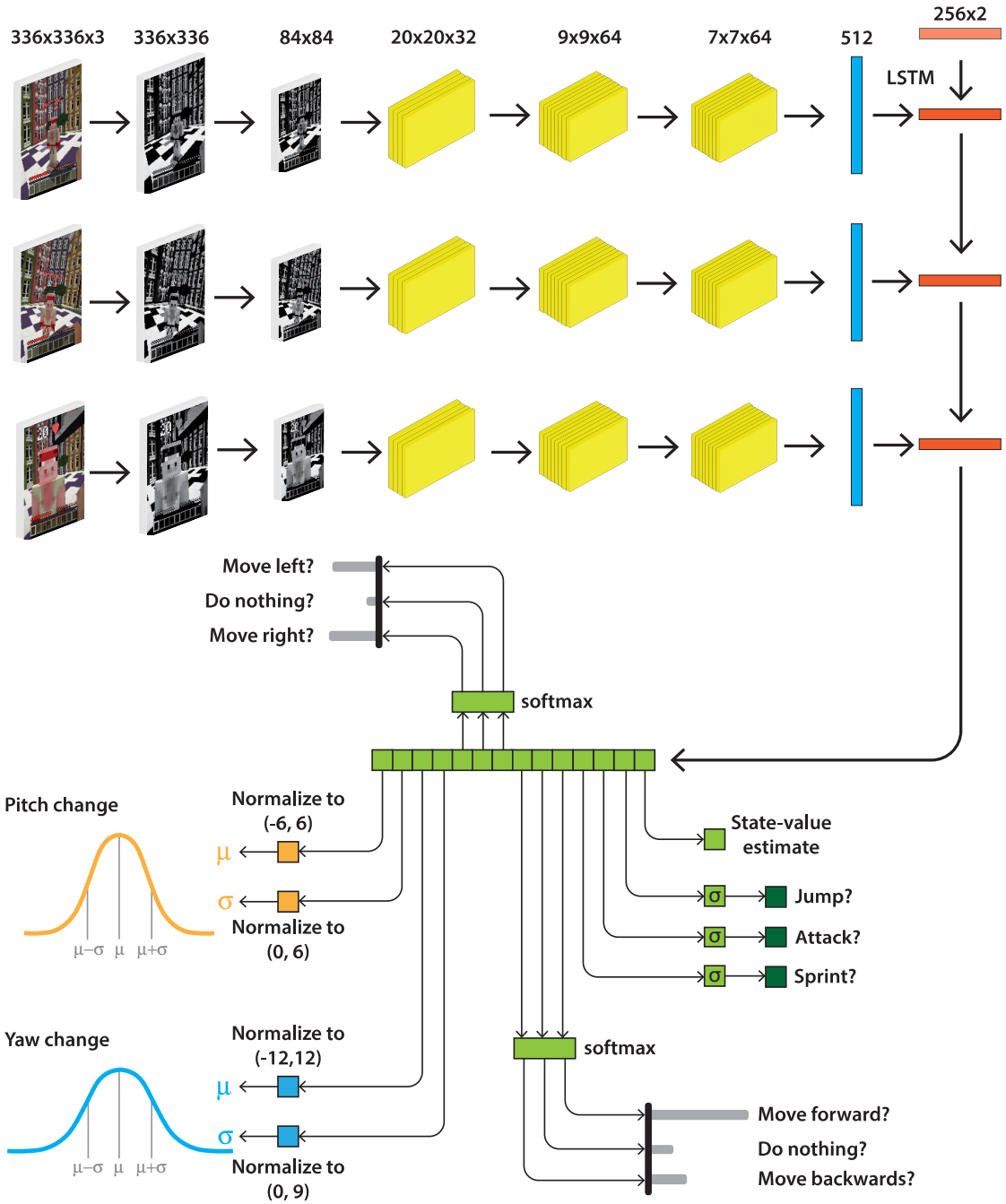


Figure 9. Model architecture. The policy construction part of the model is for the Sumo minigame. Other minigames had slightly more complicated processes, as the agents were also able to switch the item in hand and right-click.

One of the output values from the neural network was used as the state-value function estimate. The rest of the output was used to generate a probability distribution over action space. Actions were split into several components, and specific outputs from the neural network were used to generate a distribution for each component. For example, one component controlled forward movement for which a distribution was generated by applying the softmax function to three specific values from the model's output. The three values denoted the probabilities of moving backward, doing nothing, and moving forward.

For boolean components, a Bernoulli distribution was generated by applying the sigmoid function to one of the outputs from the network. For other discrete components, softmax function was used. For continuous components, a normal distribution was constructed, with mean and standard deviation predicted by the neural network. The values of mean and standard deviation were constrained to be in a specific range using the sigmoid function. For instance, to limit the mean to be in range (-6, 6), the sigmoid function was applied to the output from the network, the result was multiplied by 12 and subtracted by 6.

To sample an action, a value was sampled from each of the components' distributions. For the sake of simplicity, the values for the components were sampled independently, i.e., the sampled value of one component did not affect how values for other components were sampled. The architecture of the network and the process for generating a policy in a given state is depicted in figure 9.

4.4 Curriculum Learning

Curriculum learning is a technique for training machine learning models in which the model is first presented with simpler cases and the difficulty is gradually increased. In reinforcement learning, curriculum learning is often implemented on the environment level — the complexity and difficulty of the environment are increased as the agent learns various skills (Matiisen et al., 2017).

Curriculum learning was used for training agents in all minigames. In Sumo, the agents initially played the game on a single map and new maps were added after every 100 iterations. For the first 200 iterations, all games were on one specific map and after 200 iterations, a new map was introduced every 100 iterations.

In Classic Duels, the size of the game arena started out significantly smaller than a full-sized map. Namely, the full-sized maps are of size 136x136, but initially the game arena was constrained to a 7x7 area in the middle. This was done by placing transparent barrier blocks around the area, making it impossible for players to go outside it. Such an approach was adopted because on larger arena, the players spent most of the time just randomly running around, rarely seeing the opponent which made them progress extremely slowly.

To know when to increase the size of the arena, the training system kept track of the fraction of games that ended in a draw. When during the last 10 iterations, the fraction of such games dropped below 2%, the size of the arena was increased by 2% of its original size. Figure 10 shows three different arena sizes: full-sized (136x136), 50% (68x68), and the initial one when training starts (7x7).

In Classic Duels, the players were not spawned on the positions they would naturally spawn in the real Classic Duels minigame. The spawn positions were uniformly sampled from the range starting with center of the arena and ending with the natural spawn position. Figure 10 shows that range for an arena of size 0.5. It was observed that if players spawned on their natural positions, as the arena size increased, the players' PvP skills started to deteriorate because more and more time was spent on locating the opponent rather than fighting them. By making the players occasionally spawn closer to each other, they seemed to retain their fighting skills better.

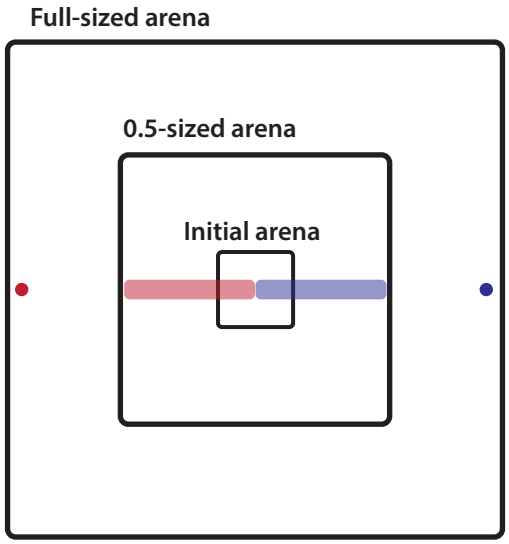


Figure 10. Classic Duels game arena scheme. The red and blue dots represent the locations where the players would spawn when playing the game in Hypixel. The red and blue areas show the locations where the player's spawn location is sampled from when the players are playing on an arena that is 50% of the original size.

In The Bridge, a similar technique to Classic Duels was applied when it comes to player spawn positions. Namely, the player spawn positions were not fixed and were uniformly sampled from some range. That range started out as only consisting of the central position and linearly increased to anywhere between the central position and the original spawn position during the first 2000 iterations. Without applying that method, the agents quickly converged to a suboptimal policy where they moved only near the block they spawned on. This was likely caused by the fact that it was difficult for the

agent to get started, as in most games, the agent fell into the void without making it to the bridge.

4.5 Training Objective

As the same model was used for generating a policy and approximating the value function, they were trained together. The clip version of proximal policy algorithms (Schulman et al., 2017) was used. Generalized Advantage Estimation (Schulman et al., 2016) was used to calculate the advantages. The policy objective was complemented with state-value estimate objective which was to minimize the mean square error. n-step return (Sutton & Barto, 2018, p. 143) was used to calculate state-value estimate targets. The n-step return is defined as follows:

$$G_{t:t+n} = \sum_{i=1}^{n} \gamma^{i-1} R_{t+i} + \gamma^n \hat{v}(S_{t+n}, \boldsymbol{\theta}) \tag{16}$$

Here, $\hat{v}$ denotes the value function approximator, $\boldsymbol{\theta}$ the parameters of the neural network and γ the discount factor.

In addition to PPO and state-value estimate objective, the training objective included a third component — entropy objective. Entropy regularization[12] was applied, as without it the agents usually converged to a suboptimal policy early on. The final gradient with respect to the parameters was

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = & \nabla_{\boldsymbol{\theta}} \left[\min(r_t(\boldsymbol{\theta}) \hat{A}_t^{\text{GAE}}, \text{clip}(r_t(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon) \hat{A}_t^{\text{GAE}}) \right] - \\ & - 2\, b\, [\hat{v}(S_t, \boldsymbol{\theta}) - G_{t:t+n}] \nabla_{\boldsymbol{\theta}} \hat{v}(S_t, \boldsymbol{\theta}) + \\ & + \beta\, \nabla_{\boldsymbol{\theta}} H(\pi(S_t, \boldsymbol{\theta})) \end{aligned} \tag{17}$$

Here, b is a hyperparameter known as value function coefficient. The first term on the right side of the equation corresponds to the PPO objective, second one to the state-value estimate objective, and the third one to the entropy objective.

4.6 Training

Input Preprocessing The input image was normalized to be in the range [-1, 1] by subtracting 128 from all pixels and dividing by 128. The reward was divided by the current estimate of its standard deviation, calculated using data from all observed rewards. The mean was not subtracted, in order to not change the dynamics of the reward function.

[12] As the policy consisted of several components, it was not trivial to calculate the entropy. Entropy was separately calculated for each of the components and the arithmetic average of these entropies was used in place of true entropy.

The values of advantage estimators were normalized per batch. That is, their mean in the current batch was subtracted from the advantages and they were divided by their standard deviation in the current batch.

Gradient Estimation The truncated backpropagation through time algorithm (Sutskever, 2013) was used for computing the gradients. The input was processed one step at a time, and every k_1 timesteps, backpropagation was performed for k_2 timesteps. In all experiments, $k_1 = k_2 = 20$.

Optimizer Adam optimizer (Kingma & Ba, 2015) with a fixed learning rate was used. The hyperparameters of the optimizer are given in appendix 1. Gradient clipping was used — if the L2 norm of the gradient was bigger than l, then the gradient was scaled to have an L2 norm of l. In all experiments, l was chosen to be 1.

Hardware and Schedule The training was done on a machine with AMD Ryzen 5950X and NVIDIA RTX 2070 SUPER. 6 Minecraft clients running at 400 ticks per second were run in parallel. The training for each of the minigames took a total of 7 days in wall-clock time. The values of all hyperparameters for the training are given in appendix 1.

Implementation The training code was written in Python, using PyTorch automatic differentiation library. The code is located here: https://github.com/laursisask/agi/tree/master/duels-training-python.

5 Results

In this section, the neural networks are evaluated and the resulting policies are discussed.

5.1 Sumo

In order to compare the abilities of the trained Sumo agent with human players, the agent was deployed to Hypixel where it played the game for around a thousand episodes. The agent joined games using the official Hypixel matchmaking system, similar to real players. It is not known whether the queueing system pairs people randomly or also takes into account their statistics from previously played games.

The overall evaluation results for Sumo are presented in table 3. In Sumo, the neural network powered agents were able to play the game on a similar level as skilled real players.

Table 3. Evaluation results for Sumo.

Games played	1345
Unique opponents	1065
Wins	954
Draws	0
Losses	391
Win rate	71%

The win rate achieved by the artificial agent is affected by the players they play against. To give more meaning to the achieved win rate of 71%, the statistics about opponents were collected. Hypixel does not have a public ELO rating system to put players on a skill level but it does have simpler statistics such as how many games each player has played and how many of these they have won.

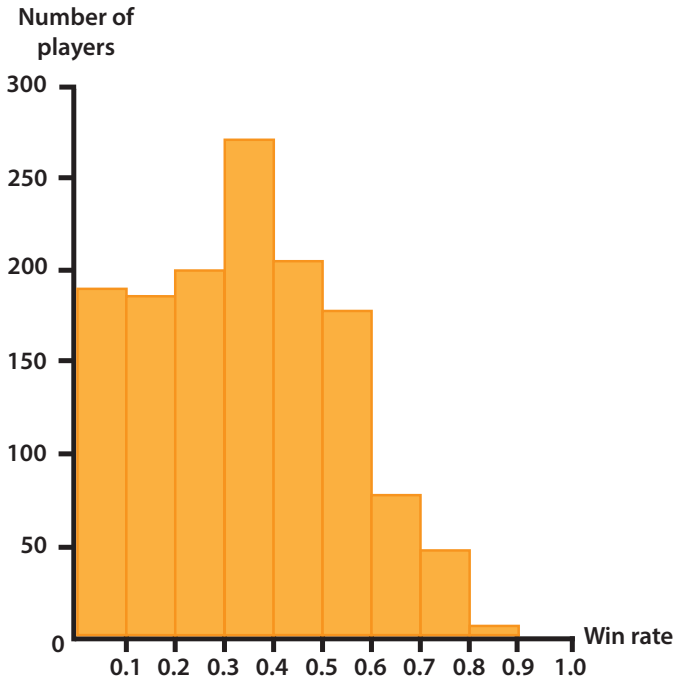


Figure 11. Distribution of global Sumo win rate among players that the agent played against. The global win rate of a player is defined as the number of Sumo games the player has won divided by the number of Sumo games they have played in total.

Figure 11 shows the distribution of global Sumo win rates of players that the agent played against. From the figure, it is clear that only a small portion of players rank above Sumo agent when it comes to win rate. In fact, only around 9% of the players have a higher win rate than the trained agent. Although it must be noted that this win rate is calculated using all the Sumo games the player has ever played but the same is not done for the artificial agent, as the artificial agent learned the game before joining any games in Hypixel, unlike human players who learn to play the game by playing in Hypixel.

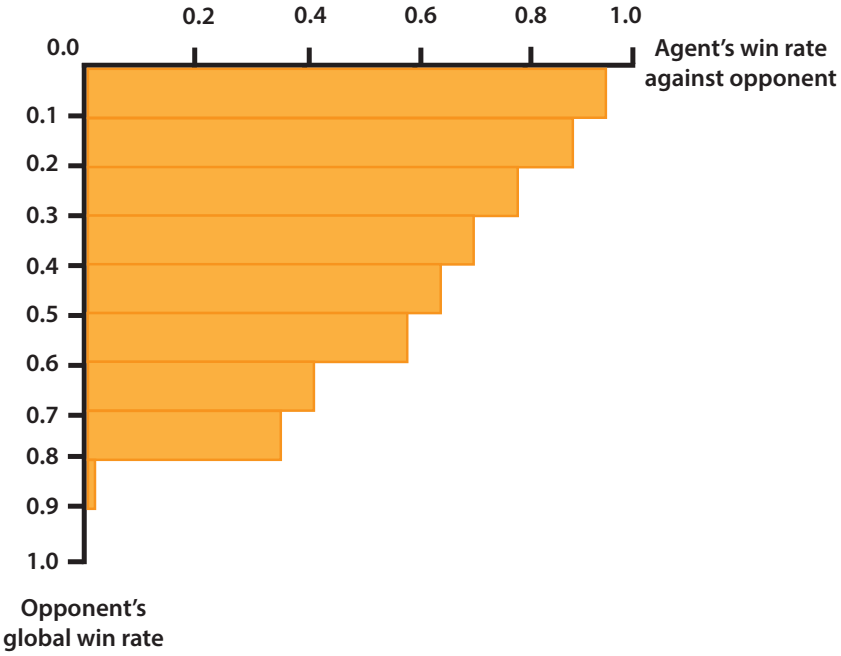


Figure 12. Artificial agent's win rate against opponents with given global win rate.

Figure 12 shows how well the neural networks perform against players with various win rates. This figure conveys why win rate is an appropriate metric for measuring skills. As the global win rate of the opponent increases, the agent's success decreases significantly.

To sum up, the Sumo agent makes a formidable opponent to almost all human players. Furthermore, the author of the thesis finds that the play style of the agent is natural and similar to that of real people. A video compilation of Sumo games that the agent player during evaluation can be found at https://youtu.be/v7sY5LFBMQ4.

5.2 Classic Duels

In Classic Duels, the artificial agent performed much worse against human players than it did in Sumo. It played in Hypixel for around 100 episodes and only won a few games. The main skill that the agent lacked was locating their opponent. In most games, the opponent was easily able to sneak behind the agent and easily kill them, thus winning the game.

Classic Duels turned out to be more difficult than Sumo, likely because it is a more complex environment. In Sumo, the size of the play area is around 11x11 while in Classic Duels it is 136x136. The action space is also larger, as the player can choose between different items like sword, fishing rod, bow, arrows, golden apple. It was observed that the agents only learned to use the sword and occasionally selected other items randomly.

This leads to the conclusion that with limited computation power, the setup presented in this thesis is not sufficient for mastering Classic Duels. It may be possible to improve the agent's performance by applying more computation or by combining self-play with other methods, like supervised pre-training using human data.

The Classic Duels' agent play style was less natural than that of its Sumo counterpart. It was using the sword at all times, even if the opponent was not visible. Further, when the opponent was not visible or was far away, the agent was just moving randomly. While that behaviour is acceptable, as it may be used to locate the opponent, it was definitely not optimal and did not look natural. A video compilation of a Classic Duels agent playing against itself can be found at https://youtu.be/_z3-7UqRfok.

As can be seen from the video, the most challenging part seems to be finding the other player. Once the other player is in the agent's sight and close enough, the agent actually performs relatively well. One possible reason why locating the other player is difficult, is that the agent has only access to a 84x84 grayscaled image. From that image it can be difficult to extract information about the opponent's location, especially if they are far away. If that is the case, then using some other state representation may aid in improving performance.

5.3 The Bridge

In The Bridge, the agent performed the worst, barely grasping the goal of the game. As The Bridge is a much more complex game compared to Sumo and Classic Duels, such result was expected. The Bridge requires a lot more coordination and long-term planning than Classic Duels and Sumo. The reward in The Bridge is also the most delayed, making it difficult to be learned using reinforcement learning methods.

The agent successfully learned to cross the bridge and move to the opponent's island. It was also able to cross any obstacles by putting blocks under itself. In some maps, when the player made it to the other island, it sometimes jumped to the hole it was supposed to. Although it must be noted that moving to the hole was already a real challenge and often took it more than 30 seconds, while for a human it usually takes no more than 5 seconds. The agent also learned to hit the opponent with sword when the opponent was in its sight.

Overall, while the agent did not make much of an opponent to most human players, it was still able to play the game and achieve the objective on some occasions. It was not even attempted to deploy the agent to Hypixel, as it was clear from self-play footage that the agent stands no chance against human players. Similar to Classic Duels, it may be possible to train a better agent with more compute or by combining self-play with supervised learning. A video compilation of The Bridge agent playing against itself can be found at https://youtu.be/IORCrF6bBbU.

5.4 Conclusion

The goal of this thesis was to train artificial neural networks to play competitive Minecraft minigames, a domain that has been largely unexplored by researchers thus far. By selecting three minigames and training neural networks to play each of these, it was demonstrated that it is in fact possible to train neural networks to play competitive Minecraft minigames. Further, it was shown that doing so is possible without injecting any prior knowledge of the environment into the models. It was discovered that training agents that are on par with humans is possible with self-play in simple minigames but not in more complex games, at least with computing power that was available to the author.

In chapter 3, three competitive Minecraft minigames were selected and a framework to run these games as reinforcement learning environments was developed. The environments matched closely their equivalents in Hypixel, making it possible to train models offline, that is, outside Hypixel, and later run the same models in Hypixel. In addition to supporting the minigames, the developed framework also ran faster than existing alternatives, making training and experimentation easier.

In chapter 4, a scheme for training neural networks using reinforcement learning for the selected minigames was presented. As the environments were complex and multi-agent, it was necessary to apply a variety of special techniques to successfully train the models. Self-play was used to provide the learning agent with an appropriate adversary. To make the agents acquire skills that are necessary for winning, exploration rewards were used. To speed up the training process even more, game-specific curriculums were used that gradually increased the complexity of the environments.

In chapter 5, the performance of the trained agents was evaluated. Sumo agent was deployed to a Hypixel where it played over a thousand episodes of the game. The agent made formidable opponent to most players and its play style was similar to that of humans. In Classic Duels and The Bridge, the agents were able to play the game on a beginner level. This leads to the conclusion that the scheme presented in this thesis can be used to train agents to play competitive Minecraft minigames, but the agents can only challenge real players in less complex games.

5.5 Future Work

There are several directions for future work.

Expanding to other minigames The framework used in this thesis could be used to train agents to play other Minecraft minigames. There exist hundreds of minigames in Hypixel and other Minecraft servers and are played by people every day.

Combining with supervised learning All skills developed by agents in this thesis were discovered on their own, solely from the reward. One way to bootstrap learning

would be to combine self-play with supervised learning and first train a neural network to imitate human players. It was observed in Classic Duels and The Bridge that the policies learned using self-play are more primitive than those employed by humans, hinting that they may benefit from supervised pre-training.

Faster learning To reduce the time it takes to train agents, neural networks could be trained on non-visual input. Representing the world in some other space besides 2D image may be useful for two reasons. First, it can speed up the environments significantly, as 80% of the time for running an environment is spent on rendering. Second, while a 2D image of a 3D world is easily understood by humans, there may be other representations that neural networks can learn more efficiently. It was observed in Classic Duels that the most challenging part was finding the other player and one possible reason is that extracting that information from a heavily downsampled and grayscaled image is difficult, hinting that using some other state representation may be helpful there.

References

Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., & Mordatch, I. (2018). Emergent Complexity via Multi-Agent Competition. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

Guss, W. H., Castro, M. Y., Devlin, S., Houghton, B., Kuno, N. S., Loomis, C., Milani, S., Mohanty, S. P., Nakata, K., Salakhutdinov, R., Schulman, J., Shiroshita, S., Topin, N., Ummadisingu, A., & Vinyals, O. (2020). The MineRL 2020 Competition on Sample Efficient Reinforcement Learning using Human Priors. *NeurIPS Competition Track*.

Guss, W. H., Houghton, B., Topin, N., Wang, P., Codel, C., Veloso, M., & Salakhutdinov, R. (2019). MineRL: A Large-Scale Dataset of Minecraft Demonstrations. In S. Kraus (Ed.), *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence* (pp. 2442–2448). International Joint Conferences on Artificial Intelligence Organization.

Johnson, M., Hofmann, K., Hutton, T., & Bignell, D. (2016). The Malmo Platform for Artificial Intelligence Experimentation. In S. Kambhampati (Ed.), *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (p. 4246). AAAI Press.

Kanitscheider, I., Huizinga, J., Farhi, D., Guss, W. H., Houghton, B., Sampedro, R., Zhokhov, P., Baker, B., Ecoffet, A., Tang, J., Klimov, O., & Clune, J. (2021). *Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft*. arXiv: 2106.14876 [cs.LG].

Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Y. Bengio & Y. LeCun (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Matiisen, T., Oliver, A., Cohsen, T., & Schulman, J. (2017). *Teacher-Student Curriculum Learning*. arXiv: 1707.00183 [cs.LG].

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of The 33rd International Conference on Machine Learning, PMLR* (pp. 1928–1937).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning*. arXiv: 1312.5602 [cs.LG].

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

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., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., … Zhang, S. (2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. arXiv: 1912.06680 [cs.LG].

Perez-Liebana, D., Hoffmann, K., Mohanty, S. P., Kuno, N., Kramer, A., Devlin, S., Gaina, R. D., & Ionita, D. (2019). *The Multi-Agent Reinforcement Learning in MalmÖ (MARLÖ) Competition*. arXiv: 1901.08129 [cs.AI].

Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* (Technical Report CUED/F-INFENG/TR 166). Engineering Department, Cambridge University.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., & Abbeel, P. (2016). High-Dimensional Continuous Control Using Generalized Advantage Estimation. In Y. Bengio & Y. LeCun (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–489.

Sutskever, I. (2013). *Training Recurrent Neural Networks* (Doctoral dissertation). University of Toronto.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. Solla, T. Leen, & K. Müller (Eds.), *Advances in Neural Information Processing Systems*. MIT Press. https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, *38*(3), 58–68.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., ... Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*, 350–354.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation). University of Cambridge.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Appendix

I. Experimental Details

Table 4. Values of hyperparameters during training.

Parameter	Value
Batch size	1280
Truncated backpropagation steps	20
Clip range (PPO parameter)	0.1
Number of epochs (PPO parameter)	4
Learning rate	$5 \cdot 10^{-4}$
β_1 from Adam optimizer (used for computing running average of gradients)	0.9
β_2 from Adam optimizer (used for computing running average of the square of the gradient)	0.999
Maximum gradient L2 norm	1
Entropy regularization coefficient	0.01 (Sumo) 0.001 (Classic Duels) 0.005 (The Bridge)
Value function coefficient	0.5
Number of steps before bootstrapping return (n in n-step returns)	5
Discount factor (γ)	0.99
Generalized Advantage Estimation parameter (λ)	0.95
Number of steps collected data per iteration	65536
Number of iterations	6800 (Sumo) 3500 (Classic Duels) 3500 (The Bridge)

The hyperparameters for training are given in table 4.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Laur Sisask**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to
   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the
2. expiry of the term of copyright,
   Learning Competitive Minecraft Minigames with Reinforcement Learning,
   (title of thesis)
   supervised by Tambet Matiisen.
   (supervisor's name)
3. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web
4. environment of the University of Tartu, including via the DSpace digital archives, under the Creative
5. Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce,
6. distribute the work and communicate it to the public and prohibits the creation of derivative works and
7. any commercial use of the work until the expiry of the term of copyright.
8. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
9. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights
10. or rights arising from the personal data protection legislation.

Laur Sisask
10/05/2022