Curriculum learning for increasing the performance of a reinforcement learning agent in a static first-person shooter game

MARCUS ADAMSSON
Curriculum learning for increasing the performance of a reinforcement learning agent in a static first-person shooter game

MARCUS ADAMSSON

Master in Computer Science
Date: October 4, 2018
Supervisor: Joel Brynielsson
Industrial Supervisor: Stefan Freyr Gudmundsson
Examiner: Olov Engwall
Swedish title: Användning av läroplanering för att öka prestandan hos en agent som lärs upp med förstärkt inlärning i ett förstapersonsskjutspel med en statisk spelare
School of Electrical Engineering and Computer Science
Abstract

In this thesis, we trained a reinforcement learning agent using one of the most recent policy gradient methods, proximal policy optimization, in a first-person shooter game with a static player. We investigated how curriculum learning can be used to increase performance of a reinforcement learning agent. Two reinforcement learning agents were trained in two different environments. The first environment was constructed without curriculum learning and the second environment was with curriculum learning. After training the agents, the agents were placed in the same environment where we compared them based on their performance. The performance was measured by the achieved cumulative reward. The result showed that there is a difference in performance between the agents. It was concluded that curriculum learning can be used to increase the performance of a reinforcement learning agent in a first-person shooter game with a static player.
Sammanfattning

Acknowledgments

I would first like to express my greatest gratitude to my academic supervisor Joel Brynielsson and my industrial supervisor Stefan Freyr Gudmundsson. Their feedback and continuous support helped me with the research and writing of this thesis. I want to thank my examiner, Olov Engwall at KTH for examining my thesis work. I also want to thank Torbjörn Söderman at King for the support and making this thesis possible.

Furthermore, I would like to thank all my colleagues in the Artificial Intelligence team and in the Reload team at King. I want to direct a particular "thank you" to Narinder Basran and Alex Nodet for helping me out with Unity and Google Cloud Platform. I also want to direct a particular "thank you" to Davide Anghileri and Rachael Ann for reading and giving me feedback on my thesis and for all the fun we have had during this period.

Finally I want to express my gratitude to my friends, family and my girlfriend Johanna Dahl, for the encouragement and support through my years of study. This accomplishment would not have been possible without them.

Stockholm, September 23, 2018

Marcus Adamsson
# Contents

## 1 Introduction
1.1 Purpose ................................................. 2  
1.2 Ethics and sustainability ................................. 2  
1.3 Research question ......................................... 3  
1.4 Thesis outline .............................................. 3

## 2 Background
2.1 Curriculum learning ....................................... 4  
2.2 Related work ................................................ 5

## 3 Theory
3.1 Reinforcement learning ...................................... 9  
  3.1.1 Components of reinforcement learning ................. 10  
3.2 Proximal policy optimization .............................. 13  
  3.2.1 Actor-critic methods .................................. 13  
  3.2.2 Policy gradient methods ............................... 15  
  3.2.3 Clipped surrogate objective ......................... 16  
  3.2.4 Algorithm ............................................. 16  
3.3 Feed-forward artificial neural networks ............... 17  
3.4 Convolutional neural network ............................ 18  
3.5 Unity and ML-Agents SDK ................................. 19

## 4 Method
4.1 Computer settings ......................................... 21  
4.2 Game, training and test environment .................... 21  
  4.2.1 Game environment .................................. 21  
  4.2.2 Training environment ................................. 23  
  4.2.3 Test environment ...................................... 25  
  4.2.4 Reward system ....................................... 25  
4.3 Implementation ............................................ 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1 Neural network architecture</td>
<td>26</td>
</tr>
<tr>
<td>4.3.2 Learning settings</td>
<td>27</td>
</tr>
<tr>
<td>4.4 ML-Agents implementation</td>
<td>28</td>
</tr>
<tr>
<td>4.5 Evaluation methods</td>
<td>30</td>
</tr>
<tr>
<td>5 Results</td>
<td>32</td>
</tr>
<tr>
<td>5.1 Training the agents</td>
<td>32</td>
</tr>
<tr>
<td>5.2 Evaluating the agents</td>
<td>34</td>
</tr>
<tr>
<td>6 Discussion, conclusions and future research</td>
<td>37</td>
</tr>
<tr>
<td>6.1 Result analysis</td>
<td>37</td>
</tr>
<tr>
<td>6.2 Method discussion</td>
<td>39</td>
</tr>
<tr>
<td>6.3 Conclusions and future research</td>
<td>39</td>
</tr>
<tr>
<td>Bibliography</td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years, Artificial Intelligence (AI) has developed much in different areas such as image-classification, self-driving cars and games [1]. In some of these fields AI has surpassed human level. Games have been a popular domain for early AI attempts as they are formal, highly constrained, complex and consist of decision-making environments. In interactive problems, it is not feasible to obtain examples of a described behavior in all different situations where an agent has to act. In uncharted situations, it must learn from its own experience.

Reinforcement Learning (RL) is part of machine learning inspired by behavioural psychology [2]. Supervised and unsupervised machine learning algorithms are not suited to solve all different problems. RL can be used to solve sequential decision making problems. The agent in these problems needs to continually select an action in the environment that will have an impact on what it will see next. An agent is not trained based on predefined training samples. It is instead placed directly into an environment and allowed to learn by trial and error. RL models learns by by receive rewards and punishments on every action taken. RL are able to train agents to respond to unforeseen environments. It can be applied to many different problems such as robotics, personalized recommendations, self-driving cars, chemistry and games [3]. Training a RL model to learn a complex task could take a lot of time. Therefore, decreasing the training time could have a significant impact on the final result.

State of the art RL has previously been used to achieve human performance in observable environments, e.g., in Atari games and Doom [4] [5] [6]. Algorithms such as Q-learning, Asynchronous Advantage
Actor-Critic (A3C) and Proximal Policy Optimization (PPO) have been used to train the agents [7]. RL in a First-Person Shooter (FPS) game where there is a long term goal is not an easy task, due to a sparse reward system and partially observable environment. Curriculum Learning (CL) with A3C has previously been applied to Doom to increase performance.

RL could be valuable for games in early development where there can be very limited game-play data to work with. Game companies can use AI bots for difficulty balancing prior to public launch.

1.1 Purpose

The purpose of this thesis is to expand and apply RL techniques to games under development where there is limited player data to work with. RL agents can be used to reduce the time spent by humans testing newly created levels and games. Furthermore, another purpose is to investigate available RL frameworks that can be applied on different RL problems and not only games.

1.2 Ethics and sustainability

There are some ethical considerations that need to be addressed. An ethical aspect is violence in shooting games. Donald Trump claimed that exposure to simulated violence in video games begets violent tendencies in real life [8]. It is a recent news [9] that Steam, a publishing marketplace run by Valve Corporation of Bellevue, would not publish any games developed by Acid. This was after Steam faced online calls for a boycott. Acid developed a game named Active Shooter where the player is played by the point of view of an attacker, aiming a weapon down a school corridor. The game ran into controversy after several recent school shootings [9]. However, decades of research within the field have not found any significant connection between playing violent video games and behaving violently in real life [8].

Another ethical aspect is the outcome of adding automatic playtesting to games. This can reduce the number of human testers and thus reduce the number of jobs. However, automatic playtesting could add value to a game in terms of quality by automatically testing newly created levels. The human testers could instead spend their time
developing and improving the game by analyzing the results from the tests. Adding automatic playtesting removes repetitive work by letting a bot do most of the testing. This contributes to sustainability because it provides more decent work and removes repetitive work.

1.3 Research question

• Can curriculum learning be used to increase the performance of a reinforcement learning agent in a static first-person shooter game?

1.4 Thesis outline

The rest of this thesis is structured in the following manner. Chapter 2 introduces related work. Chapter 3 captures and explains the components in RL as well as the theory behind PPO. Chapter 4 explains the methodology that consists of the environments, CL configuration and PPO configuration. Chapter 5 summarizes the most important findings when executing the methodology. Chapter 6 discusses the result and the used methodology. In Chapter 7 we conclude the result and propose future research.
Chapter 2

Background

This chapter covers CL and previous research within RL. Previous research include how state of the art RL algorithms have been used to play the Atari games and Doom.

2.1 Curriculum learning

CL is inspired by the learning process of humans and animals. When a human tries to learn a complex task the learning process is usually by starting with a easier example of the task and gradually increase the difficulty [10]. An example is when a human is trying to learn to ride a bicycle. In the beginning of the training, a pair training wheels are usually used. The wheels are used to assist humans until they have developed a usable sense of balance on the bicycle. Later in the learning process, the training wheels are removed from the bicycle. The training process starts with an easier aspects of a task, and then gradually take more complex examples into consideration.

Imagine that you are training a RL agent to push a block into a place to jump over a wall and reach the goal, as shown in Figure 2.1 [10]. The policy in the beginning of the training will be random. The RL agent will probably run in circle and most likely never jump over the wall to reach the goal and receive the reward. CL can be used by starting with a simpler task and gradually increase the difficulty by increasing the height of the wall.
2.2 Related work

This section gives an overview of related work that has been done within the field of RL and games. Common RL algorithms within the field are Deep Q-learning, A3C and PPO. They have been implemented to play the Atari games, Doom and Battlefield 1. In these games RL agents were trained using raw pixels as observations of the environments. The related work will be summarized in chronological order to give a historical overview.

Mnih et al. [5] in Playing Atari with Deep Reinforcement Learning used Deep Q-learning with experience replay to train a RL agent to play the Atari games. The Atari games consists of a wide range of games such as shooting games and sport titles, as shown in Figure 2.2.

The RL agent was provided with a set of legal game actions. At each time-step it selects an action. The selected action modifies the state of the game and the game score. The agent will only observe an image that consists of raw pixel values that represent the current game screen. It also receives a reward that is a representation of the change in the game score. The goal of Q-learning is by using a state-action function, to interact with the game such that the selected actions maximizes future rewards. A state-action function calculates the
expected future reward by being at a state and taking an action. The input to the neural network was the 4 last frames as a gray-scale representation with a size of 84x84. The network architecture consisted of a Convolutional Neural Network (CNN). Seven different Atari games were tested without modifying any network architecture or hyperparameter. The goal was to create a single neural network agent that performed well on a wide range of different games. The RL agent was provided with the same information as a human player in each game; all possible actions, video input, reward and terminal signals. This work demonstrated the ability to master difficult control policies for Atari 2600 computer games by only using raw pixels as input to the RL model. However, the environment in the Atari games are fully observable whereas FPS games are partially observable.

Kempka et al. [4] in ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning used Deep Q-learning with experience replay to train a RL agent to play Doom. A software called ViZDoom was used. The software allows developing bots that can play the game with the game screen as input. ViZDoom can run custom scenarios that include creating maps, programming the environment, defining terminal conditions and rewards. This creates the possibility to do a lot of experiments. The learning procedure is similar as in Playing Atari with Deep Reinforcement Learning. However, the Atari games have a 2D environment while Doom has a 3D environment, as shown in Figure 2.3.

![Figure 2.3: Screenshot from Doom.](image)

Two RL agents were trained in two different environments. The first environment had a basic move and shoot setup. The second environment had a complex maze-navigation setup. One experiment
consisted of navigating a 3D maze and collecting some objects while avoiding others. The result showed that the Deep RL algorithm was able to perform well in a first-person perspective 3D-environment. The network architecture consisted of a CNN. The input to the CNN was an RGB image with a size of 60x45. The output of the network corresponds to all available actions, as shown in Figure 2.4. The conclusion was that visual RL is possible in 3D virtual environment using ViZDoom. The work shows that it is feasible to train a agent to play an FPS game using RL.

Wu and Tian [6] in Training agent for first-person shooter game with actor-critic curriculum learning used A3C with CL to train a RL agent to play Doom. A3C is actor-critic method and it uses a value function and a policy function. The value function gives the expected reward of the current state and the policy function gives a probability distribution on the available actions in the current state. The policy function is updated such that actions that lead to high reward are encouraged and actions that lead to low reward are discouraged. A basic setup of reward signals in an FPS is +1 for a kill and -1 for dying. To complement the reward system intermediate rewards were added to increase the exploration. Ammunition and health kits were placed around the map. The RL agent received a small amount of reward for picking up ammunition and health while being punished for losing ammunition or health. CL was designed by varying health, movement speed and the number of enemies in each episode. It was used to progressively increase the difficulty of a level. This work demonstrates an approach to train a agent to play an FPS game using deep RL and CL. The agent built its own tactics during the game. The work shows how CL can be
used with RL to train an agent to play an FPS game.

Harmer et al. [12] in *Imitation Learning with Concurrent Actions in 3D Games* used a batched version of A3C algorithm (A2C) with some modifications to train several RL agents to play Battlefield 1. The algorithm used both Imitation Learning (IL) and Temporal Difference (TD) RL. IL works by imitating the behaviour of a human player. It is a supervised learning technique that maximizes the likelihood of selecting the same actions as selected by the human expert at the same state. IL can be used to speed up training by letting a human expert demonstrate the desired behavior of the agent. The network is updated to match that behaviour. Furthermore, the human data was generated by recording human play. Additionally to IL a Multi-Action (MA) policy was used to enable selecting multiple actions at each time step. Using IL with a MA policy improved training time by a factor of 4 and the performance by a factor of 2.5 over single actions selection TD RL. It would be interesting to use IL and MA on our problem but Unity does not support this at the moment. However, it would be interesting to apply this approach to our problem in the future.
Chapter 3

Theory

This chapter describes the main components in RL. The main components are action space, state space, reward signal, policy, and value functions. Furthermore, it explains PPO that is one of the most recent policy gradient method.

3.1 Reinforcement learning

RL is used to train an agent how to act in an environment [2]. An agent is a general term that in this work is a neural network. The agent is the learner and decision maker in the environment. It is provided an action space $A$ and a state space $S$. The agent interacts and manipulates the environment at each time step $t \in \{0, 1, ..., T_{max}\}$. At each interaction, a new representation of the current state is formed by selecting action $a_t \in A$ in state $s_t \in S$, as shown in Figure 3.1.

![Figure 3.1: At each time-step $t$, the agent observes $s_t$ and selects an action $a_t$. The environment responds with a reward signal $r_{t+1}$ and a new state $s_{t+1}$.](image)

9
At each time step a numerical reward signal is produced by the environment \[2\]. The goal in RL is to maximize the reward over time by letting the agent select actions from the action space. At each time step the agent receives a representation of the current state \(s_t \in S\) and on that basis selects action \(a_t \in A\). Selecting action \(a_t\) at state \(s_t\) causes the environment to produce a new state \(s_{t+1}\) and a numerical reward \(r_{t+1}\). The learning process is based on how to map observations to actions using numerical reward signals.

All RL agents have explicit goals and can choose actions that will have an influence on their environment. They use their own experience to improve performance over time and monitor the result of an action to react appropriately.

### 3.1.1 Components of reinforcement learning

This section captures the different components in RL. Pong is a two-dimensional table tennis sport game, as shown in Figure 3.2. It is a simple game that will be used as a running example throughout this section to explain how the different components are used.

![Figure 3.2: The game Pong.](image)

**Action space**

An action space can either be of discrete or continuous type depending on the problem \[13\]. A discrete action space is the most basic one since it contains a finite number of actions. Each action has a static position
in the action space such that it easily can be selected and identified. At each time step one action from the action space is selected by the agent.

In Pong there are two different actions. The action space consists of moving the paddle in two different directions. Thus the action space will be defined as $A = \{\text{Up, Down}\}$.

In an autonomous vehicle problem it would not be sufficient to have a discrete action space. The action space for a car is how much to turn the wheels or press the gas pedal. Thus it is not enough to describe it with just turning left/right or pressing the pedal or not. In a continuous action space each action is associated with a degree of how much to do of a certain action, for example, how much to turn the wheels.

State space

At each time step an agent receives information that includes the current representation of the environment. The environment representation at time step $t$ is described as $s_t \in S$, where the state space is either discrete or continuous.

Previous research, as explained in Section 2.2, uses frame pixels as observations and thus it provides the agent with the same information as provided to a human [5][14]. It would not be fair to provide the agent with information that is unknown to the player, e.g. position of enemies that are out of sight. Thus the observable state may consist of a small part of what is actually occurring within the environment. If an agent is not provided with sufficient and relevant information it may perform poorly.

In Pong, the state space can be described using different approaches. Mnih et al. [5] used a representation of greyscaled images with dimensionality of $84 \times 84$. The state space can also be described as a continuous state including information such as position of the ball, velocity of the ball and positions of the paddles.

Reward signal

The numerical reward signal is used to reward or punish the agent based on the outcome of the selected action [2]. At each time step a scalar feedback signal $r_t \in R$ is sent to the agent. A reward signal can either be a positive or a negative value that corresponds to good or bad
behavior respectively. The agent’s goal is to maximize the cumulative reward during an episode. The reward sequence can be described by a function $G_t$. The simplest case is where the expected return is the sum of the future rewards formulated by

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{t+T_{\text{max}}} = \sum_{i=0}^{T_{\text{max}}-1} r_{t+i+1} \quad (3.1)$$

where $T_{\text{max}}$ denotes the final time step.

An extension of Equation 3.1 is obtained by adding a discount term in front of each reward value $[2]$. This formula weights immediate reward higher than reward that is further in the future. Thus it is formulated as

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T_{\text{max}}-1} r_{t+T_{\text{max}}} = \sum_{i=0}^{T_{\text{max}}-1} \gamma^i r_{t+i+1} \quad (3.2)$$

where $\gamma$ is the discount factor in the range $0 \leq \gamma \leq 1 [2]$. Rewards that are further in the future are evaluated towards zero given $\gamma < 1$. The purpose of using a discount factor is to favour immediate rewards rather than rewards that are potentially received in the far future. In Pong positive reward can be received when the player scores a goal and negative reward when the player drops a goal.

**Policy**

At each time step $t$, the agent has to decide which action $a_t$ to select when observing state $s_t [2]$. The policy can be formulated as a stochastic function of selecting each action $a \in A$ for every state $s \in S$. When an agent follows a policy $\pi$ at time step, $t$ then $\pi(a_t|s_t)$ denotes the probability of selecting $a_t$ in state $s_t$. Thus, policy $\pi(a_t|s_t)$ defines a probability distribution over $a_t \in A$ for each $s_t \in S$. During training the policy $\pi$ is updated as a result of processed experience.

**Value functions**

The state-value function is often denoted as $V_\pi(s)$, where $\pi$ is the policy and $s$ is the state $[2]$. $V_\pi(s)$ estimates how good it is to be in a given state. It estimates the expected discounted reward for an agent in state
Thus it can be formulated as:

\[ V_\pi(s) = E\left[ \sum_{i=0}^{T_{\text{max}}} \gamma^i r_{t+i+1} | s_t = s \right]. \]  (3.3)

In addition to the state-value function we define a function that estimates how good it is to take action \(a\) in state \(s\) under policy \(\pi\). The action-value function is denoted as \(Q_\pi(a, s)\). Thus the function estimates the expected reward starting from state \(s\) taking action \(a\) and following policy \(\pi\). It is formulated as:

\[ Q_\pi(s, a) = E\left[ \sum_{i=0}^{T_{\text{max}}} \gamma^i r_{t+i+1} | s_t = s, a_t = a \right]. \]  (3.4)

### 3.2 Proximal policy optimization

The RL algorithms can be divided into three different groups: actor-only, critic-only and actor-critic methods \[15\]. The critic is a synonym for a value function and the actor is a synonym for a policy function.

Value methods consist of trying to update the policy based on the estimation of the state-value function Equation 3.3. They are used to find a deterministic policy whereas the optimal policy is often stochastic. However, policy methods are directly trying to optimize the policy function instead of the state-value function.

PPO uses a third approach to solve a RL problem. The third approach is called actor-critic methods. A actor-critic method combines a state-value function and a policy function. A3C is an actor-critic method that was used to train a RL agent in Doom \[6\]. The actor part is a policy that is optimized and the critic is the state-value function which is being learned.

#### 3.2.1 Actor-critic methods

Actor-critic methods use a state-value function Equation 3.3 and a policy function \(\pi(a_t|s_t)\) to solve a RL problem \[16\][15]. An agent that is in state \(s\) trying to choose action \(a\) to maximize the discounted future reward. The actor is trying to learn a policy \(\pi(a_t|s_t)\) i.e. which action to take at a given state by receiving feedback from the critic. The advantage function in actor-critic methods is formulated as

\[ A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t), \]
whereas the function evaluates the advantage of performing action $a_t$ in state $s_t$ \cite{mnih2015human}. If the advantage function is greater than zero, then the agent performed better than was expected on average \cite{mnih2015human}. The advantage function allows the agent to determine not only how good an action is, but also how much better it is than the expected value. The neural network can by using an advantage function focus on parts where the predictions are lacking.

Mnih et al. \cite{mnih2015human} popularized a policy gradient implementation where the advantage estimator is formulated as

$$A_\pi(s_t, a_t) = k - \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V_\pi(s_{t+k}) - V_\pi(s_t),$$  \hspace{1cm} (3.5)$$

where $k$ is the time horizon, $r$ is the reward and $\gamma$ is the discount factor. The time horizon defines the length of a trajectory. The goal of a policy algorithm is to maximize the expected reward of the agent over trajectories. The time horizon should be long enough such that the agent receives some meaningful reward within it \cite{mnih2015human}.

A neural network using an actor-critic method outputs a value and a policy. The structure of an actor-critic model is shown in Figure 3.3 \cite{mnih2015human} \cite{mnih2015human}.

![Figure 3.3: The basic framework of actor-critic model \cite{mnih2015human}.](image)

The value function gives the expected reward of the current state $s$, and the policy gives the probability distribution of all the actions $A$ in state $s$.

Furthermore, in actor-critic methods a value loss function and a policy loss function are defined as the functions to be minimized \cite{mnih2015human} \cite{mnih2015human}. The value loss, also called TD error, is formulated as

$$L_{value} = \left( \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V_\pi(s_{t+k}) - V_\pi(s_t) \right)^2$$  \hspace{1cm} (3.6)$$
The policy loss is formulated as
\[
L_{\text{policy}} = -\log(\pi(a_t|s_t)) A_{\pi}(s_t, a_t),
\] (3.7)
where \( \pi(a_t|s_t) \) is the probability of selecting action \( a_t \) in state \( s_t \). A term measuring the entropy can be added to the policy loss function to prevent the agent from getting stuck at a local optima [16]. The entropy is a measurement of how spread the action probabilities are. It can be added to increase exploration. This procedure is called entropy regularization. Thus, the policy loss (3.7), including an entropy term, is formulated as
\[
L_{\text{policy}} = -\log(\pi(a_t|s_t)) A_{\pi}(s_t, a_t) - \beta H(\pi),
\] (3.8)
where \( \beta \) is the magnitude of the regularization and \( H(\pi) \) is the Shannon entropy value formulated by
\[
H(\pi(a_t|s_t)) = -\sum_{i=1}^{M} P(a_t|s_t) \log P(a_i|s_t),
\] (3.9)
where \( M \) is the number of actions and \( \pi(a_t|s_t) = [P(a_1|s_t), ..., P(a_M|s_t)] \)

### 3.2.2 Policy gradient methods

Policy gradient (PG) methods are used to learn a parameterized policy [2]. The parameterized policy can perform action selection without involving the value function. The value function might be needed through the learning process of the policy but it is not required for the action selection. The vector of policy parameters is denoted as \( \theta \in \mathbb{R}^d \).

The probability of taking action \( a \) given state \( s \) and policy parameters \( \theta \) at time \( t \) is formulated as \( \pi(a|s, \theta) = P\{a_t = a|s_t = s, \theta_t = \theta\} \). PG methods are used to minimize a function e.g. a loss function such as Equation 3.6 and Equation 3.8.

Thus, gradient ascent is used to move in the direction to increase the objective function. This is done by adjusting the policy parameters in the direction of the gradient [7]. The most commonly used gradient estimator is formulated as
\[
\hat{g} = \hat{E}_t \left[ \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) \hat{A}_t \right],
\] (3.10)
where \( \pi_{\theta} \) is a stochastic policy, \( \nabla_{\theta} \) is the gradient and \( \hat{A}_t \) is an estimator of the advantage function at timestep \( t \) [7]. Equation 3.10 formulates an expectation of an average over a finite batch of samples. If the
advantage function is positive i.e. the agent received a lot of reward from the environment then the gradient should be adjusted to favor that specific behaviour. If the advantage function is negative then the gradient should be adjusted to prevent that type of behaviour. The policy is updated by calculating the objective function and adjusting $\theta$ in the direction of the gradient.

### 3.2.3 Clipped surrogate objective

PPO uses a clipped surrogate objective function to prevent large policy updates \[7\][18]. It is called "clipped surrogate" as it discourages big change between the old and the new policy by clipping the objective function. The clipped surrogate objective is defined as

$$L_{CLIP}(\theta) = \hat{E}_t \left[ \min \left( r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$$  \hspace{1cm} (3.11)

where $\epsilon$ is a hyperparameter and $\hat{A}_t$ is the advantage function. The hyperparameter $r_t(\theta)$ is the ratio between the old and the new policy defined as

$$r_t(\theta) = \frac{\pi_\theta}{\pi_{\theta,old}}$$

If $r_t(\theta) \hat{A}_t$ gets to large then it is clipped. This prevents $r_t(\theta)$ to go outside the interval $[1 - \epsilon, 1 + \epsilon]$. PG updates a maximum of $1 + \epsilon$ if the advantage function is positive and $1 - \epsilon$ if the advantage function is negative. This prevents large policy updates.

### 3.2.4 Algorithm

The PPO implementation uses an actor-critic method. The policy loss used in PPO is a clipped surrogate objective combined with a value loss function. It is defined as

$$L_t^{CLIP+VF+S}(\theta) = \hat{E}_t \left[ L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + \beta H(\pi) \right]$$  \hspace{1cm} (3.12)

where $c_1$ and $\beta$ are coefficients that describe the magnitude of the value loss function $L_t^{VF}(\theta)$ and the entropy $H(\pi)$.

The actor part can be implemented with different policy optimization methods such as trust region policy optimization, Vanilla PG and PPO.
PPO uses one or several actors. Each actor is placed in a separate environment such that it collect its own observations. At each iteration of the algorithm, each actor collects data in parallel for \( T \) time steps \[7]. A loss function \( L \) is constructed with the collected data and is optimized using minibatch stochastic gradient descent (SGD) for \( K \) steps, as shown in Algorithm 1.

**Algorithm 1 PPO, Actor-Critic Style**

```plaintext
for iteration=1,2,... do
    for actor=1,2,...,N do
        Run policy \( \pi_{\theta_{old}} \) in environment for \( T \) timesteps
        Compute advantage estimates \( A_1, ..., A_T \)
    end for
    Optimize surrogate \( L \) wrt \( \theta \), with \( K \) epochs and minibatch size \( M \leq NT \)
    \( \theta_{old} \leftarrow \theta \)
end for
```

### 3.3 Feed-forward artificial neural networks

Feed-forward neural networks are models used within machine learning, also known as multilayer perceptron or deep feed-forward networks \[20\]. The neural network tries to approximate some function \( f^*(x) = y \) which maps an input value \( x \) to an output \( y \). It learns a policy that yields the best function approximator. Thus, a neural network defines a mapping as \( y = f(x; \theta) \), where \( \theta \) are the weights of the neural network. The network estimates the value of the parameters \( \theta \) such that it results in the best approximator. The model receives input \( x \) and processes it through the neural network and outputs \( y \). There exists no feedback connections in a feed-forward neural network. A network with feedback connections is called recurrent neural network.

A neural network consists of an input layer, one or more hidden layer(s) and an output layer \[20\]. The layers in a neural network are fully connected when each neuron in a layer is connected to every neuron in the previous layer, as shown in Figure 3.4.

The connections between the neurons are associated with a weight. The weights are used in the mathematical calculation that approximates the function \( f^*(x) = y \).
3.4 Convolutional neural network

A CNN is similar to a neural network. They are built on neurons that have learnable weights and biases [21]. A CNN architecture makes the assumption that the inputs are images and this allows encoding specific properties into the architecture.

A CNN architecture have neurons arranged in three dimensions: width, height, and depth [21]. Instead of having each input connect to each output, as in a fully connected neural network, it uses filters that are shared between the inputs. The filters are moved over step by step over the inputs, as shown in 3.5.

Figure 3.5: An example of a CNN architecture. For each input the filter covers a small region and are moved step by step over the entire input [22].
CNN are built on three main types of layers: convolutional layers, pooling layers, and fully-connected layers. A CNN is a sequence of layers and every layer transforms one volume of activations to another through a differentiable transfer function. The pooling layer is normally inserted between convolutional layers. It is also referred to as a downsampling layer [23]. The fully connected layer computes the class scores which results in a vector as large as the number of classes [21].

3.5 Unity and ML-Agents SDK

Unity is a cross-platform game engine developed by Unity Technologies. It is used to develop video games for PC, consoles, mobile devices, and websites. Scenes are created inside Unity that consists of 3D objects with different types of components attached to them, such as script, audio or a camera. A level of a game consist of one or multiple scenes.

The Unity development team have introduced ML-Agents SDK into Unity [10]. ML-Agents SDK allow researchers and developers to build games and simulations using the Unity Editor into environments where deep RL agents and other machine learning methods can be used through a Python API. ML-Agents SDK operates as shown in Figure 3.6.
Figure 3.6: An overview of how ML-Agents SDK operates inside Unity [10].

Each agent can have unique actions, set of states and observations within the environment [10]. It can receive unique rewards from events within the environment. The action each agent takes is decided by the brain that is connected to the agent. This creates the possibility of creating several agents inside an environment and linking them to the same brain.

Each brain is responsible for deciding which action each of the agents that is connected to the brain should take [10]. The brain also defines a specific state and action space. The action space and state space can be either discrete or continuous.

The Academy object within a scene in Unity contains all brains as their children within the environment [10]. A scene in Unity is a unique level of the game. In each scene, you add your environments, obstacles and decorations, essentially designing and building your game in pieces. Each environment contains a single Academy, which defines the scope of the environment, for example frame to skip, target frame rate during training, max steps, and training configurations.
Chapter 4

Method

This chapter explains the experiments in detail. It covers implementation details, game environment, training environment and test environment. Furthermore, the chapter describes the ML-Agents implementation and evaluation methods.

4.1 Computer settings

Google Cloud was used to create and run several virtual machine (VM) instances. On each VM instance different parameter settings and configurations were tested.

We used a Macbook Pro (Retina, 15 inch Early 2017) to run the experiments used to generate the result. It is equipped with a 3,1 GHz Intel Core i7. Furthermore it has 16GB of RAM.

4.2 Game, training and test environment

This section covers the game environment. Furthermore, it explains how we created the training and test environment.

4.2.1 Game environment

The agents in this thesis are static players in an FPS game in a 3D-environment. The goal in the environment is to defend a fortification against enemies. One type of enemy was used and they are spawned at random locations in the environment. The player wins if the fortification is defended and loses if the enemies manage to breach it. A level
of the game consist of $x$ enemies, where $x \in [1, 30]$. Each enemy will have $y$ health and $z$ movement speed, where $y \in [0, 1]$ and $z \in [1, 100]$. The enemies spawn inside a spawning circle with a radius of $r$, where $r \in [1, 6]$.

The environment is in 3D and a view from above is shown in Figure 4.1. The static player is standing in the fortification and the spawning circles are placed in-front of it. The circles in the environment represents the spawning circles and the square represents the static player. The enemies moves towards the fortification and when they are close enough they will start to hit it.

![Figure 4.1: The game environment.](image)

In an FPS game there are several parameters that have an influence on the environment difficulty. We selected 4 parameters that determine the difficulty for the RL agent. These parameters are the number of enemies, enemies’ health, enemies’ movement speed and spawning radius.

Increasing the number of enemies in a level will increase the difficulty. It is much harder to face several enemies than only one, at least for a human. However, at the beginning of the training in our environment it can be more difficult for the RL agent to face one enemy
than several. The policy is random in the beginning of the training. Thus, the RL agent needs a policy that adjusts the aim to an enemy and shoots to receive a positive reward signal from the environment. If there is only one enemy in the environment then the probability of adjusting the aim to that enemy and shooting is low. Having several enemies in the environment will increase the probability that the RL agent receives positive reward from the environment.

Another parameter is the spawning radius, i.e., positions in the environment where an enemy can appear. A larger spawning radius will increase the positions where the enemies can spawn. Thus, the agent has to do larger movements in order to kill all enemies. Enemy health and movement speed were also selected as parameters to control the difficulty of the environment. An enemy that is killed by one bullet is easier to encounter than an enemy that needs several hits to be killed. It is easier to target an enemy with a lower movement speed than an enemy with higher movement speed.

### 4.2.2 Training environment

Two RL agents were trained in two different environments. RL agent 1 was placed in environment 1 and RL agent 2 was placed in environment 2. They will be referred to as agent 1 and agent 2 for the remainder part of this thesis. The two agents were trained in the following two environments:

1. **Environment without CL.** This environment has the same difficulty throughout the whole training phase. Agent 1 was trained in this environment.

2. **Environment with CL.** This environment consists of nine different lessons. The environment is updated at the end of every lesson. Agent 2 was trained in this environment.

The two environments will be referred to as **EnvBase** and **EnvCL** throughout the remainder of this thesis.

We trained the agents in two different environments. The difficulty is the same within **EnvBase** throughout the training process. The difficulty in **EnvCL** is increasing. **EnvBase** is described in Table 4.1.
Table 4.1: Enemy health, enemy movement speed, spawning radius and number of enemies in EnvBase.

<table>
<thead>
<tr>
<th>Enemy health</th>
<th>Enemy movement speed</th>
<th>Spawning radius</th>
<th>Number of enemies</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.0</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

EnvCL is a modification of EnvBase. The modification is an adjustment where CL is applied.

Unity provides a documentation [24] of how to use CL to train the RL agents. In EnvCL we choose to vary these 4 parameters. The whole training session consists of these parameters starting with an easy setup and progressively increase in difficulty. We used a progress parameter to decide when the environment should increment to the next lesson. A step is a training epoch in the PPO algorithm. The progress parameter is defined as:

\[
\text{progress} = \frac{\text{current step}}{\text{max step}}
\]

The progress parameter in each lesson was defined such that the RL agent spend an equal amount of steps in each lesson. The environment will change to the next lesson once the progress parameter within the current lesson is reached. For example, if we train the RL agent for 1000k steps and we defined 10 lessons then the agent would spend 100k steps in each lesson. EnvCL consists of 9 different lessons as shown in Table 4.2. Lesson 8 does not have any progress parameter since it is the last lesson.
Lesson | Enemy health point | Enemy movement speed | Spawning radius | Number of enemies | Progress
--- | --- | --- | --- | --- | ---
0 | 10 | 0.1 | 3 | 30 | 0.1
1 | 20 | 0.2 | 3 | 20 | 0.2
2 | 30 | 0.3 | 4 | 10 | 0.3
3 | 40 | 0.4 | 4 | 6 | 0.4
4 | 50 | 0.5 | 4 | 3 | 0.5
5 | 60 | 0.6 | 5 | 3 | 0.6
6 | 70 | 0.7 | 6 | 3 | 0.7
7 | 80 | 0.8 | 6 | 9 | 0.8
8 | 90 | 0.9 | 6 | 9 |

Table 4.2: CL configuration for EnvCL.

### 4.2.3 Test environment

After training agent 1 and agent 2, we placed the agents in a test environment to evaluate their performance. The agents were evaluated for 300k steps in the test environment. The test environment has the same configuration as EnvBase, as shown in Table 4.3. The test environment will be referred to as EnvTest throughout the remainder of this thesis.

<table>
<thead>
<tr>
<th>Enemy health point</th>
<th>Enemy movement speed</th>
<th>Spawning radius</th>
<th>Number of enemies</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.0</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.3: Test environment.

### 4.2.4 Reward system

In these two environments a reward system was created. The reward system is described in Table 4.4. We designed the reward system to encourage and discourage specific types of behaviours. We want to encourage the agents to kill enemies and finish the level, and thus the agents receive a positive reward for these events. We want to discourage a behaviour where the agents waste ammunition and thus the agents receive a negative reward for each ammunition the agents uses. The agents should prefer shooting enemies that are closer to the fortification to prevent them from hitting it. To encourage this behaviour, the
agents receive a negative reward for each health point the fortification loses.

Table 4.4: The reward system used in the environments.

<table>
<thead>
<tr>
<th>Amount of reward</th>
<th>Event description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1.0</td>
<td>if the agent kills an enemy</td>
</tr>
<tr>
<td>+1.0</td>
<td>if the agent successfully defends the fortification</td>
</tr>
<tr>
<td>-0.01</td>
<td>for each health point the fortification loses</td>
</tr>
<tr>
<td>-0.05</td>
<td>if the agent uses ammunition</td>
</tr>
<tr>
<td>-0.01</td>
<td>at each time step</td>
</tr>
</tbody>
</table>

4.3 Implementation

We used PPO to train two RL agents implemented by Unity [25][26]. The main motivation for using PPO is based on the paper Proximal Policy Optimization Algorithms where Schulman et al. [7] tested PPO on a collection of benchmark tasks. Result showed that PPO outperforms other online policy gradient methods. Another reason is that Unity provides a PPO implementation. Lastly, PPO provides better performance and does not need much hyper-parameter tuning.

The PPO implementation stores training statistics during training. These statistics were used to get an overview of the training and to evaluate the RL agents.

4.3.1 Neural network architecture

The network used in the experiments consist of a CNN with 4 convolutional layers similar to the CNN used by Harmer et al. [12] as described in Table 4.5. The input of the CNN is an RGB image with a size of 128x128. The network outputs a vector with the same size as the number of actions. This vector contains a probability distribution over the actions. The network also outputs a value estimation of being at that given state and taking the selected action.
Table 4.5: Network architecture used in the experiments.

<table>
<thead>
<tr>
<th>Layer</th>
<th>N</th>
<th>Kernel</th>
<th>Stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv. 1</td>
<td>32</td>
<td>5x5</td>
<td>2</td>
</tr>
<tr>
<td>Conv. 2</td>
<td>32</td>
<td>3x3</td>
<td>2</td>
</tr>
<tr>
<td>Conv. 3</td>
<td>64</td>
<td>3x3</td>
<td>2</td>
</tr>
<tr>
<td>Conv. 4</td>
<td>64</td>
<td>3x3</td>
<td>1</td>
</tr>
<tr>
<td>Fully connected layer</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fully connected layer</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Policy</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 Learning settings

This section describes the hyperparameters used in the experiments. Unity provides a documentation that includes typical range for each hyperparameter in the PPO algorithm and how they are used [27]. We started with the default setup that is provided by Unity, as shown in Table 4.6. In the default setup, we modified max steps and summary freq. The max steps variable corresponds to how many steps of the simulation are run during the training. The summary freq corresponds to when to measure the training statistics.

Table 4.6: Default parameter setup.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Max steps</td>
<td>27e5</td>
</tr>
<tr>
<td>Batch-size</td>
<td>1024</td>
</tr>
<tr>
<td>Beta</td>
<td>5e-3</td>
</tr>
<tr>
<td>Buffer-size</td>
<td>10240</td>
</tr>
<tr>
<td>Epsilon</td>
<td>0.2</td>
</tr>
<tr>
<td>Gamma</td>
<td>0.99</td>
</tr>
<tr>
<td>Number of hidden units</td>
<td>128</td>
</tr>
<tr>
<td>Lambda</td>
<td>0.95</td>
</tr>
<tr>
<td>Learning rate</td>
<td>3e-4</td>
</tr>
<tr>
<td>Number of epoch</td>
<td>3</td>
</tr>
<tr>
<td>Number of hidden layers</td>
<td>2</td>
</tr>
<tr>
<td>Time horizon</td>
<td>64</td>
</tr>
<tr>
<td>Summary freq</td>
<td>10000</td>
</tr>
</tbody>
</table>
With the default hyperparameter setup, we used the VMs to try different learning rates, number of hidden units, and number of hidden layers, as shown in Table 4.7 and Table 4.8. We tried different values of these parameters because previous papers such as [4] [12] [5] different values on these parameters. We used EnvCL to try the different hyperparameters.

Table 4.7: Learning rate.

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>1.0e-5</th>
<th>1.0e-4</th>
<th>3.0e-4</th>
<th>1.0e-3</th>
</tr>
</thead>
</table>

Table 4.8: Number of hidden layers and number of hidden units.

<table>
<thead>
<tr>
<th>Number of hidden units</th>
<th>128</th>
<th>128</th>
<th>256</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of hidden layers</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

After trying different hyperparameters, we ended up with the hyperparameters shown in Table 4.9. These hyperparameters were used to train and evaluate agent 1 and agent 2.

Table 4.9: Global parameter used in the experiments.

<table>
<thead>
<tr>
<th>Training steps</th>
<th>27e5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch-size</td>
<td>1024</td>
</tr>
<tr>
<td>Beta</td>
<td>5e-3</td>
</tr>
<tr>
<td>Buffer-size</td>
<td>10240</td>
</tr>
<tr>
<td>Epsilon</td>
<td>0.2</td>
</tr>
<tr>
<td>Gamma</td>
<td>0.99</td>
</tr>
<tr>
<td>Number of hidden units</td>
<td>256</td>
</tr>
<tr>
<td>Lambda</td>
<td>0.95</td>
</tr>
<tr>
<td>Learning rate</td>
<td>3e-4</td>
</tr>
<tr>
<td>Number of epoch</td>
<td>3</td>
</tr>
<tr>
<td>Number of layers</td>
<td>2</td>
</tr>
<tr>
<td>Time horizon</td>
<td>64</td>
</tr>
<tr>
<td>Summary freq</td>
<td>10000</td>
</tr>
</tbody>
</table>

### 4.4 ML-Agents implementation

We created an agent, a brain and an academy object inside Unity. These game objects are used in the learning environment as described in Sec-
The static player in the game will be tied to the agent object. The decision frequency variable controls when the RL agent should collect a new observation and select a new action. For example, if the decision frequency is set to 3 then the RL agent will collect a new action every third frame. The agent repeats the last action on the skipped frames. Timestep and frame are two terms that will be used interchangeably in this work. We used a decision frequency value of 1. The agent's observation is a camera in Unity that captures a first-person view of the player. The environment resets to its initial point when all enemies are killed or when the fortification is breached. However, we gave the fortification enough hit points such that the environment will reset before it is breached by the enemies. We used this configuration to know if an episode ended earlier than we know that it was because all enemies were killed and not because the fortification was breached.

<table>
<thead>
<tr>
<th>Agent setup inside Unity</th>
<th>Decision frequency: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent observation: First-person view of the player</td>
<td></td>
</tr>
</tbody>
</table>

The academy controls the environment. The setup of the academy is described in Table 4.11. The quality level defines the rendering quality of the environment. This variable can be in the range of $[0, 5]$. The time scale parameter controls the environment speed. A higher value will increase the training speed. However, we found that a too large time scale value results in a badly trained model for our problem. The time scale variable can be in the range of $[1, 100]$. The max steps variable defines how many steps the environment runs before it resets. The target frame rate defines the number of frames per second. We used the same quality level value and target frame rate value as used in many of the example environments implemented by Unity.

The brain setup we used inside Unity is described in Table 4.5. The observation is used as an input to the PPO algorithm. The observation is a RGB image with the size 128x128. The motivation for using this image size is based on the research by Harmer et al. [12]. The vector action space defines the different actions.
Table 4.11: Academy setup inside Unity.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max steps</td>
<td>3000</td>
</tr>
<tr>
<td>Quality level</td>
<td>0</td>
</tr>
<tr>
<td>Time scale</td>
<td>10</td>
</tr>
<tr>
<td>Target frame rate</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 4.12: Brain setup inside Unity.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual observation</td>
<td>First-person view of the player with a resolution of 128x128</td>
</tr>
<tr>
<td>Vector action space</td>
<td>Discrete action space with a size of 5. The actions are shooting and moving the aim left, right, up and down</td>
</tr>
</tbody>
</table>

The environment resets to its initial point when the fortification is breached, when all enemies are killed, or when the max episode length is reached. To reset the environment to its initial point, several actions are taken, described in Table 4.13. This is used to make sure that all training episodes have the same environment configuration in the beginning of an episode.

Table 4.13: The actions to reset the environment.

- Adjust gun pivot position to initial point
- Refill ammunition on the gun
- Kill all existing enemies on the map
- Reset the waves to their starting point
- Restore health on fortification
- Set number of kills to 0

4.5 Evaluation methods

This section covers the methods we used to compare agent 1 and agent 2. The agents were compared by episode length, cumulative reward,
and standard deviation of the reward by using the training statistics stored when executing the PPO implementation [27]. We trained the agents for 2700k steps. After the training we measured their performance in the test environment. During training and testing, we measured the training statistics at every 10k step defined by the summary freq variable in Table 4.9.

The cumulative reward is the sum of rewards received within an episode as defined in Equation 3.1. The motivation for using cumulative reward is that it indicates how well the agents performed in an episode. The environment produce reward according to Table 4.4. Thus, the sum of reward will be a good measurement of the performance throughout the episode. The highest received cumulative reward was stored for both agent 1 and agent 2. The cumulative reward resets to 0 after 3k steps i.e when the max episode length is reached.

Furthermore, we measured the standard deviation of the reward. When we measured standard deviation of the reward we calculated the mean of 10 measurements at 3 times during training. We took the mean of the standard deviation in the beginning of the training in the range (10k-100k), in the middle of the training in the range (1450k-1550k), and in the end of the training in the range (2600k-2700k). For example, we measured the standard deviation at step (10k, 20k, .., 100k) and then we calculated the mean of these measurements. We measured the standard deviation to get an indication of how spread the rewards were during training.

We used a static episode length where the environment resets to its initial point when the max length is reached. The agent receives a small negative reward at each time step. This is used to motivate the agent to take faster decisions to kill an enemy. The enemies are moving towards the fortification. The agents can, by quickly killing enemies, prevent them from attacking the fortification. Thus, a shorter episode length indicates how fast the agents killed all enemies.
Chapter 5

Results

In this chapter we present the result from training and testing agent 1 and agent 2. The agents were compared by cumulative reward, standard deviation of the reward and episode length.

5.1 Training the agents

The agents were executed with the hyperparameters from Table 4.9. Figure 5.1 and Figure 5.2 show the cumulative reward received by agent 1 and agent 2 in EnvBase and EnvCL respectively. The cumulative reward received by agent 1 increases in a slow manner. After 2700k steps agent 1 received a cumulative reward of -31.2. The cumulative reward received by agent 2 is higher than the cumulative reward received by agent 1. Furthermore, the cumulative reward received by agent 2 has higher standard deviation. The vertical black dashed lines in Figure 5.2 indicates where a new lesson starts. The 8 lines define the 9 different lessons in EnvCL. Agent 2 spends 300k steps in each lesson. The lessons are defined by Table 4.2. After 2700k steps agent 2 received a cumulative reward of -7.3. The first cumulative reward measurement is made at step 10k, as shown by Figure 5.1 and Figure 5.2.
Figure 5.1: The cumulative reward received by agent 1 in EnvBase.

Figure 5.2: The cumulative reward received by agent 2 in EnvCL.

Figure 5.3 and Figure 5.4 show the episode lengths in EnvBase and EnvCL. The episode length measured in EnvBase is constant with a value of 3000 throughout the training session. Thus, the episode length in EnvBase is always the same as the max episode length.

Furthermore, the episode length in EnvCL starts to decrease at step 400k, as shown in Figure 5.4. The episode length of the last measurement in EnvCL was 1510. The vertical black dashed lines in Figure 5.4 indicates where a new lesson starts. The 8 lines define the 9 different lessons in EnvCL. The episode length is always less than the max episode length in the last 7 lessons.

Figure 5.3: The episode length while training agent 1 in EnvBase.

Figure 5.4: The episode length while training agent 2 in EnvCL.
Table 5.1 shows the mean of 10 standard deviation of reward measurements received by the agent at 3 different times during training in EnvBase as described in Section 4.5. The standard deviation is highest at the first measurement and lowest in the last measurement.

Table 5.1: Standard deviation of reward received by agent 1 in EnvBase during training.

<table>
<thead>
<tr>
<th>Step range</th>
<th>Standard deviation of reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k-100k</td>
<td>4.5</td>
</tr>
<tr>
<td>145k-155k</td>
<td>1.2</td>
</tr>
<tr>
<td>2600k-2700k</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 5.2 shows mean of 10 measurements of the standard deviation of the reward at 3 different times during training in EnvCL. The standard deviation is highest at the first measurement and lowest in the last measurement. The standard deviation decreases throughout the training session.

Table 5.2: Standard deviation of reward received by agent 2 in EnvCL during training.

<table>
<thead>
<tr>
<th>Step range</th>
<th>Std of reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k-100k</td>
<td>4.1</td>
</tr>
<tr>
<td>145k-155k</td>
<td>1.5</td>
</tr>
<tr>
<td>2600k-2700k</td>
<td>1.4</td>
</tr>
</tbody>
</table>

5.2 Evaluating the agents

After finishing the training of agent 1 and agent 2 for 2700k steps, they were placed in EnvTest. The agents were executed with the hyperparameters from Table 4.9 in EnvTest. The agents were tested for 300k steps where the cumulative reward received by the agents was stored at every 10k step. The cumulative reward received by the agents’ in EnvTest is shown in Figure 5.5. The straight blue line shows the cumulative reward received by agent 1. The red dashed line shows the cumulative reward received by agent 2. There is a gap between the lines.
Figure 5.5: The cumulative reward received by agent 1 and agent 2 in EnvTest.

Figure 5.6 shows the episode length of agent 1 and agent 2 during the evaluation of their performance in EnvTest. The blue straight line is the episode length achieved by agent 1 and the red dashed line is the episode length achieved by agent 2. The episode length achieved by agent 1 is constant with a value of 3000 steps. Agent 1 reaches the max episode length in every measurement. However, the episode length achieved by agent 2 is almost constant with a value of 1500 steps.

Figure 5.6: The episode length in EnvTest achieved by agent 1 and agent 2.
The highest cumulative reward received by agent 1 and agent 2 in EnvTest during the evaluation of their performance is shown in Table 5.3:

<table>
<thead>
<tr>
<th>Agent</th>
<th>Highest cumulative reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-30.0</td>
</tr>
<tr>
<td>2</td>
<td>-7.0</td>
</tr>
</tbody>
</table>

Table 5.3: The highest cumulative reward received by agent 1 and agent 2 in EnvTest during the evaluation of their performance.
Chapter 6
Discussion, conclusions and future research

This chapter begins with a discussion of the result of the performance achieved by the agents in the training environments and test environment. Furthermore, the chapter discusses the method used in this thesis. Lastly,

6.1 Result analysis

A RL agent relies on reward signals from the environment to evaluate and update its current policy. The training process is based on a trial and error process. To train a RL agent to learn to complete a complex task could take a lot of time. Thus, placing the RL agent in an appropriate environment could increase its performance. Therefore, placing the RL agent in the most suitable environment is important. In this thesis we investigate how CL can be used to increase the performance of a RL agent in an FPS game with a static player. We trained two agents, agent 1 and agent 2, in two different environments, EnvBase and EnvCL. After finishing the training, we placed them in the same environment to measure how their performance differs. We used one type of enemy. Using a harder/easier enemy could give different results.

We investigated how CL can be used to increase performance of a RL agent in an FPS game with a static player. The results show that there is a significant difference in performance between agent 1 and agent 2 when evaluating the agents in EnvTest as shown in Figure 5.5.
The highest cumulative reward received during an episode by agent 1 was -30.0 and for agent 2 it was -7.0. This result can be explained by the fact that agent 2 was placed in an easier environment in the beginning of the training and it developed a policy that performed well in more difficult environments too. Thus, using CL increased the performance of agent 2. The difference in cumulative reward between the agents can be explained by the fact that agent 2 killed all enemies before the episode finished. This can be noted in Figure 5.6 where the episode length of agent 2 is less than the max episode length.

The agents received a small negative reward each time step. Thus, agent 1 received a small negative reward until the max episode length is reached.

The cumulative reward received by agent 1 during the training increases in a slow and steady pace, as shown in Figure 5.1. The cumulative reward stabilizes around -30.0. The cumulative reward received by agent 2 in EnvCL during training increases towards -7 as shown in Figure 5.2. This result can be explained by the fact that EnvBase was too difficult in the beginning of the training and it takes more training steps to receive enough positive reward to encourage specific behaviours such as moving the aim towards an enemy and shooting at an enemy. Agent 1 might need to train for more training steps and process more experience in EnvBase to receive a higher cumulative reward.

However, EnvCL starts with an easy configuration of the environment and progressively increase in difficulty. There are many enemies with low movement speed and low health in the first lessons. The enemies are moving slowly and this increases the number of steps it takes for the enemies to get close to the fortification and start hitting it. The policy is random in the beginning of the training. Thus, a RL agent needs a policy that adjusts the aim to an enemy and shoots to receive a positive reward signal from the environment. If there is only one enemy in the environment then the probability of adjusting the aim to that enemy and shooting is really low. However, having several enemies in the environment will increase the probability that the RL agent receives positive reward for killing an enemy from the environment. Therefore, increasing the number of enemies in the environment could decrease the difficulty for a RL agent.

The episode length while training agent 1 is constant throughout the training session, as shown in Figure 5.3. This result shows that
agent 1 didn’t manage to kill all enemies during training. However, the episode length while training agent 2 starts to decrease at step 300k. This can be explained by the fact that the enemies started with low health and low movement speed. Thus it was easier for agent 2 to an enemy and receive positive reward at the beginning of the training in EnvCL. Agent 2 manages to kill all the enemies when the lessons are getting more difficult, as shown in Table 4.2.

6.2 Method discussion

During the training of agent 1 and agent 2, there were some limitations, such as time frame of the project, hyper parameter settings, Unity configuration settings and training time. All these factors could have an impact on the result, resulting in higher or lower values for the RL agents’ performance.

The variable factor that had the most impact on the performance of the agents was the training time. We used a time scale variable of 100 in beginning of the project which resulted in a badly trained RL agent. We had to decrease this variable. This increased the training time by a factor of 5. It took 2 days to run 3000k steps on a local machine and 4 days on a VM. However, this does not affect the evaluation of the agents in EnvBase.

Another thing that might have an impact on agent 2 training in EnvCL is the CL setup. We used a progress parameter to define when the environment should increment to the next lesson. Using this setting might result in agent 2 spending too many steps in a too easy environment. A threshold parameter could be used instead. This parameter defines how much cumulative reward the agent should receive in an episode before the environment increments to the next lesson. However, using this approach could be difficult because the maximum achieved reward within the environment is different in each lesson, since we chose to vary the number of enemies.

6.3 Conclusions and future research

This study has investigated how CL can be used to increase the performance of an RL agent in an FPS game with a static player. In conclusion, the agent that was trained in a CL environment outperformed
the agent that was not trained in a CL environment, based on their performance. The result showed that the RL agent performance increases while training in an environment with CL.

In this study a ML SDK was used to train RL agents. ML SDK is developed by Unity. The toolkit was used to train RL agents to play an FPS game. Furthermore, there are a lot of areas and possibilities where the ML toolkit can be used. It can be used to train RL agents in different fields such as self driving cars, robotics and other types games. With Unity it is easy to train and simulate RL agents during training and after training.

The next step in this research is to increase the agents’ action space and the maximum steps during training. The action space could be expanded to include more complex actions such as throwing grenades, switching weapons and reloading the gun. IL could be used at the beginning of the training to demonstrate how and when to use these actions. IL works by imitating the behaviour of a human player. It is a supervised learning technique that maximizes the likelihood of selecting the same actions as selected by the human expert at the same state. Another step would be to use an MA policy that enables selecting several actions at each time step instead of just one.
Bibliography


