UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Disertație

REINFORCEMENT LEARNING IN VIDEO GAMES

Absolvent
Smarandache Mihnea

Coordonator științific
Păduraru Ciprian Ionuț

București, septembrie 2023

Abstract

Video games have always been a domain where technological innovations were always important and necessary for its advancement. One of the more recent advancements in technology was the brining of the fields of machine learning and deep learning into the mainstream. These two could prove to be real game changers in the industry of video game development, assisting in the development of AI systems. The role of this paper is to implement simple AI behaviours commonly used in video games using Reinforcement Learning, and compare the results with a more classical approach.

Contents

Chapter 1

Introduction

The video games industry is one that has been around since 1970 and over the years it became one of the most successful forms of entertainment. As the years went by, the complexity of the games also increased, requiring more time to develop and more manpower. In the early days, mainstream titles would have smaller teams, usually comprised of 10 to 20 people, however, as time went by, so did the teams increase in size, with studios having over hundreds of people working on a single game.

One important aspect of video games is the AI. It is a tough challenge for a developer to create and implement an AI to perform actions in such a way that feels *natural* and not scripted, so that the player's immersion is not broken. As the video games evolved, so did the AI, becoming more complex and having more ways to interact with the player and the environment.

A solution for this problem would be to implement Reinforcement Learning techniques to train the AI to perform the required tasks. Some advantages of this approach would be that the developer would need to create a set of rules for the AI, and then reward or punish the it for its taken actions. Another advantage would be that it would be simple to implement this behaviour, if the actions that the AI perform are similar to the actions that the player performs since they have already been implemented. However, this approach also has some negative aspects. For starters, this training could take a long period of time, depending on the complexity of this problem and the allotted resources for the training process. Another disadvantage would be that the even if the developers do not have to manually implement the AI logic, they still haev to create punishments and rewards for the training agent, and also have to balance these rewards to reach an acceptable result, which could take time due to trial-and-error.

Chapter 2

Preliminary

2.1 Reinforcement Learning & PPO

According to [4], Reinforcement Learning falls within the domain of machine learning and is primarily concerned with the process of making sequential decisions; it is a method of training an agent to learn a specific behaviour by interacting and observing the environment that it exists inside of. With each action performed by the agent which affects the environment, it can be rewarded or punished so that with each iteration it can adapt its behaviour to be slightly more similar to the desired behaviour.

As mentioned by [4], the fundamental Reinforcement Learning problem can be characterized as a discrete-time stochastic control process. In this process, an agent engages with its environment in the following manner: The agent commences in a specific state within the environment, $s_0 \in \mathcal{S}$, and collects an initial observation, $\omega_0 \in \Omega$. As time progresses, at each time step t, the agent must make a decision regarding its action, $a_t \in \mathcal{A}$. This leads to three outcomes:

1. The agent acquires a reward $r_t \in \mathcal{R}$,
2. The state undergoes a transition to $s_{t+1} \in \mathcal{S}$
3. The agent acquires an observation $\omega_{t+1} \in \Omega$

Because of this, Reinforcement Learning can be modeled as a Markov Decision Process (MDP). According to [2], a MDP is a 4-tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where:

$\mathcal{S}$: is the state space

$\mathcal{A}$: is the action space

T : is the transition function; it is defined as $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and represents the probability that an action $a \in \mathcal{A}$ will lead from state $s \in \mathcal{S}$ to state $s' \in gS$

R : is the reward function; it is defined as $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$, and represents the reward obtained when moving from state state $s \in \mathcal{S}$ to state $s' \in gS$ because of action $a \in \mathcal{A}$

Another notion of Reinforcement Learning is the policy, which shows how an agent selects the action that it will take. It can be defined as $\pi : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, where $\pi(a, s)$ is the probability of action a being chosen while in state s.

To solve a Reinforcement Learning problem there are two ways, as defined by [16]: value-based methods, which try to find an optimal Q-function (a function that determines what is returned by taking a particular action in a particular state) and policy-based methods, which search the policy-space in order to find an optimal policy. The ones that are important for this paper are the policy-based methods, since those are the ones that will be used. Policy gradient methods learn a parameterized policy and optimize it directly. The policy can be defined as

$$\pi_\theta(a|s) = Pr\{a_t = a|s_t = s, \theta_t = \theta\} \tag{2.1}$$

where $\theta \in \mathbb{R}^{d'}$ is the policy's parameter vector.

According to [14] The operation of policy gradient methods involves the computation of a policy gradient estimator, which is then incorporated into a stochastic gradient ascent algorithm. Usually, the gradient estimator has the form

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.2}$$

where π_θ represents a stochastic policy and $\hat{A}_t$ represents an estimator of the advantage function at timestep t.

[14] defines $\hat{\mathbb{E}}_t$ as the expectation which is used in an algorithm that alternates between optimization and sampling, and which denotes the empirical average over a finite batch of samples. Implementations utilizing automatic differentiation software operate by creating an objective function, the gradient of which serves as the policy gradient estimator. According to [14], the estimator $\hat{g}$ is derived through the differentiation of the objective

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.3}$$

Proximal policy optimization (PPO) works by sampling the data obtained from interacting with the environment and then optimizes the objective function with stochastic gradient ascent.

[15] defines the surrogate objective as:

$$L(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \tag{2.4}$$

with $r_t(\theta)$ being defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{2.5}$$

$r_t(\theta)$ being the probability ratio. It can be seen that the probability function is clipped in order for the surrogate objective to be modified.

The PPO algorithm that uses fixed-length trajectory segments as defined in [14] can be seen in Alogrithm 1

Algorithm 1 PPO, Actor-Critic Style

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

At each iteration, each of the N actors collect T timesteps of data, then the surrogate loss is constructed on the NT timesteps of data and optimized with a minibatch SGD for K epochs.

2.2 Unity & ML-Agents Toolkit

The environment where the solution will be implemented will be a Unity project consisting of a small level that has different structures an objects, and which contains tanks that can move, shoot, and fight eachother. The project also contains an AI implementation that is based on behaviour trees and genetic algorithms and is described in more detail in [12]. The project can be found at https://github.com/AGAPIA/BTreeGeneticFramework. The level that is contained in the mentioned project, and that will be used for training, can be seen in Figure 2.1.

As defined by [5], the ML-Agents Toolkit is an open-source project designed to allow researchers and developers to build simulated environments within the Unity Editor. Through a Python API, users can engage with these environments. The toolkit contains the ML-Agents SDK, encompassing all the required features for defining environments in the Unity Editor, along with essential C# scripts for constructing a learning pipeline. The toolkit also provides implementation in PyTorch for most state-of-the-art reinforcement learning algorithms, including PPO, and several example environments to quickly test the toolkit.

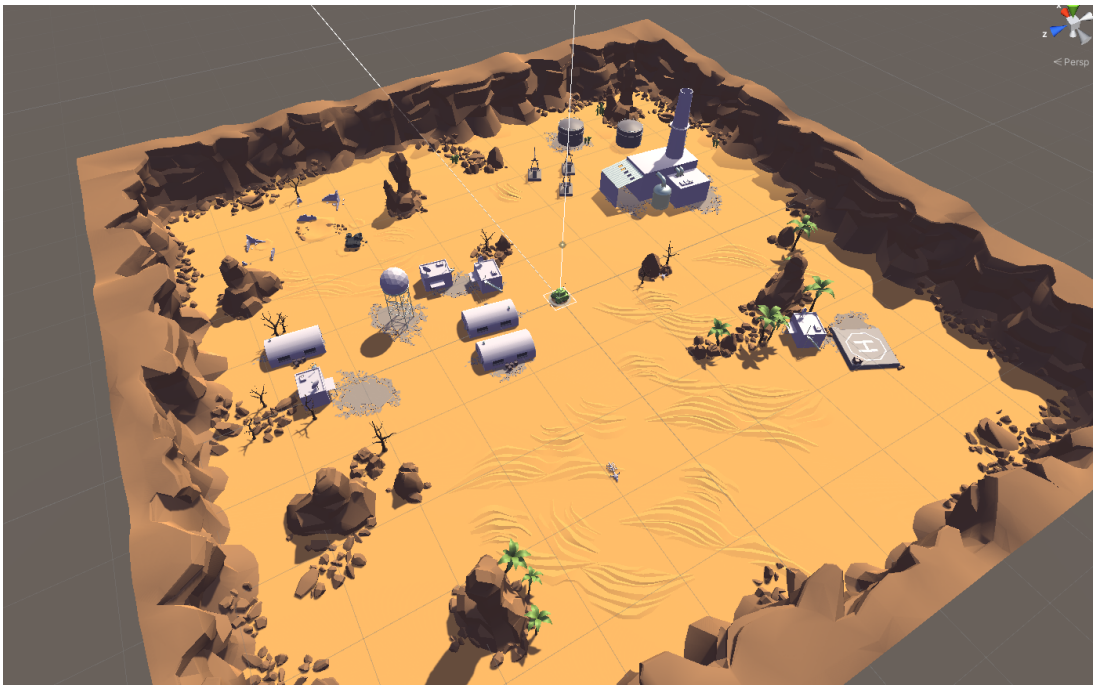

Figure 2.1: The training environment

The training hyperparameters for PPO can be configured via a *.yaml* file, and the configurable parameters, according to [7], are:

- Gamma (γ): the discount factor for future rewards
- Lambda (λ): the lambda parameter used when calculating the Generalized Advantage Estimate
- Beta (β): the strength of the entropy regularization
- Epsilon (ϵ): the acceptable threshold of divergence between the old and new policies during gradient descent updating
- Buffer Size: how many experiences should be collected before we do any learning or updating of the model
- Batch Size: the number of experiences used for one iteration of a gradient descent update
- Number of Epochs: the number of passes through the experience buffer during gradient descent
- Learning Rate: the strength of each gradient descent update step

In addition, ML-Agents also provides support for using Recurrent Neural Networks (RNN), in the form of LSTMs (Long short-term memory). This option can be used by setting a flag in the trainer's configuration. It provides two configuration parameters:

- Sequence Length: how long the sequences of experiences must be while training
- Memory Size: the size of the memory the agent must keep

There are 3 entities provided by the ML-Agents SDK:

- Sensors, which are used to collect the observations from the environment
- Agents, which are components to add the Agent functionality to a Unity GameObject, which enables them to collect observations, decide which action to take by using the observations and execute them, and receive rewards
- Academy, which can be defined as a singleton instance that manages agent training and decision making [7]

The observations that are collected, the actions that should be taken by the agent, and the rewards received by the agent can be implemented manually using the provided *Agent* class.

The ML-Agents Toolkit also has a Python package, *mlagents*, which according to [8], offers a collection of reinforcement and imitation learning algorithms intended for utilization in Unity environments. This package will be used to effectively run the training process, by communicating with the Unity application via a gRPC communication protocol.

Chapter 3

Related Works

3.1 Deep Reinforcement Learning for Navigation in AAA Video Games

The paper ([1]) is made by employees working at *Ubisoft La Forge* and presents multiple ways in which NPC navigation is done in video games. The classic approach is represented by methods such as Waypoint Graphs and NavMeshes, while the deep learning approach is represented by a model-free Reinforcement Learning method using Soft Actor-Critic as the learning algorithm.

The experiment in the paper is trying to teach an agent how navigate from one point to another one using the RL method mentioned earlier, and making comparisons between different types of observations used.

The observations made by the agent are its absolute position, velocity and acceleration, the goal's relative and absolute position, a 3D occupancy map that is made using box casts, and a 2D depth map that is made using raycasts. The action space is continuous, and correspond to movment and jump actions. The agent is rewarded if it moves closer to the objective between two consecutive timesteps. Also, a constant penalty is applied to force the agent to reach the objective as quicky as possible.

In the results it is revealed that removing the absolute positions of the agent and goal, increase the agent's performance, that removing either the 3D occupancy map, or 2D depth map reduce the agent's performance, while removing both make the agent unable to reach the goal, and that using LSTMs in the agent's network improve its performance if training environment is larger, and decrease it if it's smaller. Also, a comparison is made between the approach with NavMeshes and the one with RL, with the one with RL being better in a dynamic environment.

3.2 On Efficient Reinforcement Learning for Full-length Game of StarCraft II

The paper ([6]) details the implementation of an agent that is supposed to play full length *StarCraft II* matches against the AI provided by the game and win. It is also tested against a smaller version of the AlphaStar AI created by DeepMind, named mini-AlphaStar.

StarCraft II is a complex RTS game that provides a challange for Reinforcement Learning approaches. The game can be regarded as an imperfect information one, due to the fact that the player can only observe what is shown by the camera, and not the entire map. In addition to that, there is also a fog of war present which further complicates the problem. Another issue is the huge action space, due to the fact that there is a large number of units and buildings.

To win the match, the agent has multiple objectives such as gathering resources fast, or killing enemy units. For this, the agent's architecture is composed of multiple sub-policies that are meant to solve a smaller part of the bigger problem, and a controller. Each of the sub-policies has a local state, while the controller has a global state. The controller selects what is the active policy so that it can take actions specific to that policy. The policies are changed in a iterative process at a given number of time steps.

To define the action space, a data mining operation is performed on game replays so that a sequence of actions, a macro action, can be extracted. These marco actions can be the placing of a building, sending of workers to mine resources, etc. These extracted marco actions are then sorted in order of their frequency and only the most frequent ones are kept.

For learning, the algorithm that is used is PPO. A curriculum learning method is used to train the agent. This means that the agent is pitted against easier versions of the game's AI, and as training progresses, the game's AI difficulty is increased. This is used because starting to learn from scratch against a difficult AI may prove challenging, and the agent may not be able to learn anything.

There are multiple types of reward functions that can be used. The first one is the outcome of the match, if the agent won or lost against the AI. This is not a good reward function since it is sparse and does not tell the agent anything about the actions taken during the match. Another type of reward is the score provided at the end of the match, which gives more information about the agent's performance but still is not the optimal choice. The selected reward function is a custom one, that contains reward functions for each of the sub-policies and thus provides the best learning experience for each of the agent's desired behaviours.

The results obtained are that the trained agent had a much higher winrate against the mini-AlphaStar AI, that is trained for the same amount of time, managing to obtain

a winrate of 93% against the hardest non cheating AI in the game. Against the hardest cheating AI in the game it obtains a winrate of 86%.

3.3 Lucy-SKG: Learning to Play Rocket League Efficiently Using Deep Reinforcement Learning

The paper ([9]) presents the implementation of a Reinforcement Learning-based model that is able to play the video game *Rocket League* and also beat the two best ranking bots developed for this game, Necto and Nexto. The proposed solution uses sample-efficient Reinforcement Learning techniques, being faster to train than the previously mentioned bots and a novel technique named Kinesthetic Reward Combination (KRC). The agent uses PPO as the learning algorithm.

Due to the fact that the game has a high complexity, modelling the reward function is also difficult. For example, the scoring a *Goal* is the agent's objective and provides a big reward, however it is rarely obtained, and the agent may not understand how it got to receive that reward. The solution to this is the using of a reward function that is defined as a linear combination of different reward function components that are also weighted. This provides a continuous reward function.

To help with the development of *Lucy-SKG*, the authors developed a Python library: *rlgym-reward-analysis*. It is used to visualize the reward functions as contour plots and extract reward values from the frames that are located in replay files.

The observation space is split into three parts: the first part contains information about the current player, the second part is a key-value byte array that represents a sequence of important objects for the agent to be mindful about, and the third part is a key padding mask which enables the agent to be trained in a vectorized environment.

The results are measured as the number of one-goal matches that are won. Lucy-SKG is pitted against Necto and Nexto which are both trained at 1 billion steps. Lucy-SKG manages to achieve a win percentage of 100% against both bots. However, the agent is not yet able to outperform humans at this game.

3.4 Gotta Learn Fast: A New Benchmark for Generalization in RL

The paper ([10]) is made by employees working at OpenAI and presents multiple ways to train an agent to play the video games *Sonic The Hedgehog*™, *Sonic The Hedgehog*™*2* and *Sonic 3 & Knuckles* using thier Gym Retro framework.

The experiment in the paper consists of using transfer learning and other Reinforcement Learning methods that train the agent from scratch, to learn how to play and finish

various levels from the game. The RL algorithms that were used are: Rainbow (a variant of DQN), JERK (Just Enough Retained Knowledge), PPO, Joint PPO, Joint Rainbow.

The agent's observation are 24-bit RGB images that have the resolution 320x224, and which represent the images of the game shown by the emulator. The action space consists of single or multiple button presses, that are valid in the context of the game, or an empty action, which means no button presses. The agent is rewarded based on how far it managed to get in the level, which is measured by its position on the X axis. Since the player's start position is on the left side of the level, and its end is on the right side, this is a valid method of measuring progress. However, there are certain instances where the agent has to move in the left direction to progress. Another reward is granted if the agent manages to finish the level. This reward starts from a given value and then decreases over time.

In the results, the RL mehtods are greatly outperformed by a human, with the human being more than twice as better as the trained agents. The best RL algorithm was Joint PPO, followed by Rainbow & Joint Rainbow, JERK, and finally, PPO. Another conclusion was that the transfer learning method was not much better than the method where the agent is trained from scratch.

3.5 Creating Pro-Level AI for a Real-Time Fighting Game Using Deep Reinforcement Learning

The paper ([11]) describes the implementation of an agent in a 1v1 fighting game, *Blade and Soul*, using a self-play curriculum and an actor-critic off-policy learning method. The trained agents are then pitted against professional gamers to test them.

An inovation brought by this paper is the usage of a self-play curriculum. Three types of agents are trained: aggressive, defensive and balanced, each with different rewards and penalties based on their desired behaviour. When training, network parameters for agents are saved at certain intervals and are then added to the pool of opponents. Each opponent in this pool has a probability to be chosen to fight the agent that is currently training which is higher if the opponent has a newer network configuration. Thus, agents can train against multiple types of opponents while also using specific behaviours.

Another used technique is that of data skipping. This consists of maintaining the actions chosen by the agent for a given number of timesteps and not changing the action at each timestep, and discarding the usage of a "no-op" or empty action, which means that the agent does nothing, it does not move or use any skill. By using this technique of data skip, the agent's winrate during the training process increases significantly compared to the win rate of the agent that does not use this technique.

When fighting against a human opponent, another limitation is added to the trained

agents, which is a delay for taking actions, so that the AI does not have an unfair advantage ofer the human player. The results were that the 3 types of trained agents managed to achieve a combined average win rate of 62% against professional gamers, which means that the trained AI was on par with advanced human players, even with the introduced handicap in the form of an action delay.

3.6 Counter-Strike Deathmatch with Large-Scale Behavioural Cloning

The paper ([13]) presents the implementation of an AI agent that is trained to play the game *Counter-Strike; Global Offensive* (CSGO) from pixel input using behavioural cloning. Compared to other FPS games, like Doom, which can be run on just the CPU, CSGO has much higher system requirements, thus the idea of using behavioural cloning came about. The authors would spectate online matches and record them so that they will be used for training. Then, the authors would record clean footage of them playing the game, which allows for the clean labelling of actions and also allow the agent to specialise to a high-skill policy. The agent will be trained on an aim training map, and on the famous *Dust 2* map, in deathmatches.

The agent's observation space is represented as a 280x150 pixels image, which is obtained in the following way: the game is rendered at a resolution of 1024x768 pixels; afterwards it is croppped, removing UI elements, to an image with a resolution of 824x498 pixels; finally it is downscaled to the resolution of 280x150 pixels. This is done so that the agent can be run at a *playable* framerate on an average consumer GPU.

Modelling the action space is a more complex task: in game several actions need to be considered such as character movement, camera movement and other actions such as shooting or reloading. For this, a discrete action space was used. Also, to use multiple actions at the same time, such as moving the character and the camera at the same time, independent losses were used for the actions: for actions done by using keys and mouse clicks a binary cross entropy loss was used, and for the mouse axis a multinomial cross entropy loss was used.

There are 3 criterias for which the agent is evaluated: the score obtained in game, how much the agent performs like a human, which is measured by observing its map coverage, and the agent's ability to adapt to new maps. The obtained results show that the agent is able to perform like a casual gamer on the aim training map, and on the *Dust 2* map it is able to beat the in-game AI on the *Easy* and *Medium* difficulties. The map coverage is similar to the coverage in the training dataset, however it is not as good as the in-game AI. The agent is also able to adapat what it has learned to new maps, with the results being worse than the ones obtained on *Dust 2*, however quickly improving if the agent is

trained on the new map too.

Chapter 4

Implementation

4.1 Reaching a static target

4.1.1 The Problem

A common task for an AI in a game is to reach a a given destination. According to [1] the common way to achieve this goal is by using *navigation meshes* (NavMeshes). These are represented as graphs, and their nodes represent surfaces that can be traversed. Afterwards, algorithms such as $A*$ can be used to find the fastest way for an agent to go from point A to point B. However, these *NavMeshes* require to be *baked* in advance, so updating them in real time can prove to be a challenge depending on several factors such as:

- the game engine used
- the complexity of the game environment
- the computational cost associated with recreating in real time these meshes

A proposed solution for this problem would be to use AI agents that have been trained using deep learning methods, in such a manner that they would be independent from changes to the environment.

4.1.2 Implementing the solution

First, the action space for the agent is defined: due to the fact that we only need to move the agent to a given position, the action space consists of moving the agent forward or backward, and rotating it. Because the agent is supposed to be controlled by a controller, its movement input is defined as a real number in the $[-1, 1]$ interval for both X and Y axes. However, a discrete action space will be used instead of a continuous one to reduce the difficulty of learning, and also because there is no need for the agent to have

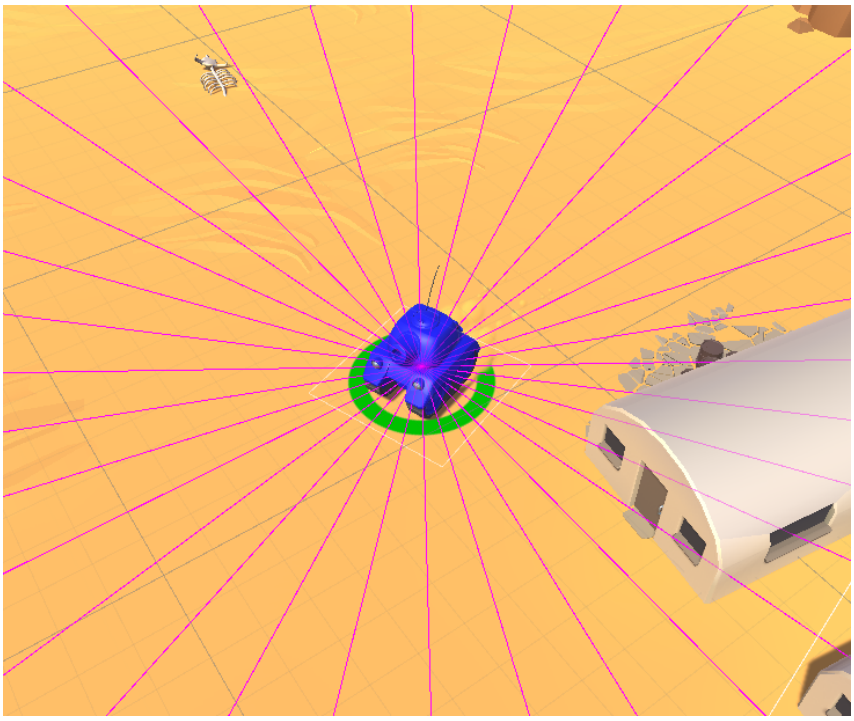

Figure 4.1: The raycasts (in magenta) that are *shot* by the agent

movements that are so precise. For each axis the action space will be represented by the discrete space: $\{-1, -0.5, 0, 0.5, 1\}$.

To make decisions, the agent will need to make several observations about its surroundings. Firstly, it will need to know how close it is to certain objects in the environment. To accomplish this, several *raycasts* will be used to measure the distance from the agent, similar to a *LIDAR*. The raycasts start from the center of the agent and are spread in such a way that the angle between 2 consecutive rays is equal for any 2 consecutive rays. The observation will contain the distance until the rays hit an object. Also, the index of the layer of the hit object will be included in the observations, so that the agent will differentiate between regular environment objects and more important objects, such as other players, enemies, etc. For this implementation, 32 rays were used. A visual representation can be seen in Figure 4.1

Other observations that are made are the agent's position in space and also that of the target. This observation is included so that the agent can learn how movement brings it closer or further to the target. The next observation is the agent's forward vector so it can know in which direction it is moving. The optimal direction that the agent should take is also observed and obtained by computing the vector difference between the target's position and the agent's position. To know how much to adjust its trajectory, the angle between the agent's forward vector and the target is observed; the angle is signed so that the agent can learn to adjust its trajectory to the left or to the right. The distance to the

target is added to the observations so that the agent can learn that when the distance is getting smaller it is rewarded. The agent's normalized velocity vector is observed to show in which direction it is moving based on the given input. The angle between the agent's velocity vector and its forward vector is observed to tell if the agent is moving forwards or backwards. Finally, the agent's velocity magnitude is observed so the agent can know if it is moving or standing still.

In summary the following observations are being made:

- distance for each raycast until it hits an object
- layer index of object hit by the raycast
- spatial postion of the agent
- spatial position of the target
- agent's forward vector
- optimal direction of the agent
- signed angle between the agent's forward vector and the target
- distance from the target
- agent's normalized velocity vector
- angle between the agent's velocity vector and its forward vector
- agent's velocity magnitude

In order for the agent to be able to learn to solve a specific problem, in this case, to reach a target, it must be rewarded or punished according to the actions that were taken. It is important to be mindful of the rewards that are given to the agent because only giving rewards and not punishing it can lead the agent to learn a behaviour that will maximize its reward, but will not be able to solve the problem, as it will be described shortly.

To begin, the first reward that was implemented, was the reward for reaching the objective, which is the goal of the agent and should also be a substantial reward. The other rewards that ware added were based on the direction of the movement and how close to the target it was, the reward increasing in value if the agent was closer to the objective (4.1) and a reward if the raycasts are hitting the destination object, which also increases in value if the agent is closer to the target (4.2).

$$R_{\text{distance to target}} = \frac{(1 - \frac{\alpha}{180}) \cdot r}{d} \tag{4.1}$$

where:

α : is the angle between the agent's current direction and the optimal direction

r : is the reward that is obtained if the agent has the optimal direction

d : is the distance from the agent to the objective

$$R_{\text{ray distance}} = \frac{r}{d_i} \tag{4.2}$$

where:

r : is the reward obtained if the distance between the agent and the objective is minimum

d_i : is the obtained distance by the raycast i from the agent to the object

However, these three rewards are not enough for the agent: through learning experiments it was observed that the agent was performing poorly: it would get stuck trying to move through a wall, it would never reach the objective and just spin around, or in some cases, it would just stand still.

To solve these problems, several punishments were implemented to correct the behavior of the agent. To prevent the agent for not moving, a constant penalty was added, to incentivise the agent to move towards the reward, so that it will receive a reward. Also, to comabat standing still, if the agent does not move, it receives an additional penalty, and if it does not move for more than 100 time steps, it receives a huge penalty and the episode ends.

To stop the agent from trying to pass thorugh walls, a penalty is added if the raycasts that hit walls or other objects in the environment, have the distance to the hit object be a smaller than a given number. This penalty also increases the closer the agent gets to a wall (4.3).

$$R_{\text{wall penalty}} = \frac{p}{d_i}, \text{if } d_i < d \tag{4.3}$$

where:

p : is the penalty obtained if the distance between the agent and the wall/environmental object is minimum

d_i : is the obtained distance by the raycast i from the agent to the object

d : is the maximum distance for which the penalty is applied

Another penalty was added if the agent is moving away from objective, and as before, it increases the further away it gets from the objective (4.4).

$$R_{\text{moving away}} = \frac{\alpha}{180} \cdot p \tag{4.4}$$

where:

p : is the penalty obtained if the distance between the agent and the wall/environmental object is minimum

α : is the angle between the agent's current direction and the optimal direction

Through training, two undesirable behaviours were observed: it was observed that the agent would sometimes make sudden jerky movments, trying to change its direction and immediatly returning to its previous trajectory, and that the agent would learn to drive backwards. To fix the first problem, a penalty was added if the agent would change its direction (4.5), and to fix the second one, a penalty was added if the tank was moving backwards (4.6).

$$R_{\text{direction change}} = \frac{\alpha}{180} \cdot p \tag{4.5}$$

where:

p : is the penalty obtained for changing the movement direction

α : is the angle between the agent's current direction and its previous direction

$$R_{\text{backwards}} = \frac{\alpha}{180} \cdot p, \text{if } \alpha > 90 \tag{4.6}$$

where:

p : is the penalty obtained for moving backwards

α : is the angle between the agent's current direction and its forward vector

In summary, the used rewards and penalites and their values can be seen in Table 4.1. The final reward function can be expressed as:

$$R = \mathbb{1}_{\text{target}} + \alpha + R_{\text{distance to target}} + R_{\text{ray distance}} + R_{\text{wall penalty}} + R_{\text{moving away}} + R_{\text{direction change}} + R_{\text{backwards}} \tag{4.7}$$

where:

R : is the total reward

$\mathbb{1}_{\text{target}}$: is the reward obtained for reaching the target

α : is the constant penalty

Name	Value	Notes
Reach Objective Reward	10	
Move Towards Objective Reward	0.001	is scaled by the distance between agent and objective
Raycast Touches Objective Reward	0.001	is scaled by the distance between agent and objective
Constant Penalty	-0.005	is applied at each time step
Not Moving Penalty	-0.05	
Not Moving For 100 Steps Penalty	-50	
Moving Towards Wall Penalty	-0.002	is scaled by the distance between agent and object
Moving Away From Objective Penalty	-0.025	is scaled by the distance between agent and objective
Sudden Movement Penalty	-0.01	is scaled by the angle between the agent's current direction and its previous one
Moving Backwards Penalty	-0.005	

Table 4.1: Rewards and Penalites

4.1.3 Training

Each training session consisted of 10^7 steps, and 30 agents were trained in parallel. The training times were between 3-4 hours. The agents are supposed to learn to reach an objective which appears in one of 17 predefined positions on the map. Once the agent reaches the objective, it is moved in another location chosen randomly. The objective is reprezented as a cuboid, as it can be seen in Figure 4.2, for better visualization of the training process. Each separate training instance uses the same seed for the random number generator, so that the objectives will appear in the same order for each of the training instances. Each training episode has a limit of 5000 steps.

In the initial training sessions, the agents were unable to learn to reach the objective, becoming stuck rotating in a circle. A proposed solution was to keep the objective in the same position until the agent reaches it 30 times. After reaching the objective 30 times, the objective would start appearing in the random predefined locations. This was done to possibly kickstart the agent's learning process, but the approach failed, the agent being unable to learn to reach the objective. Another approach was to increase the neural network's size, however this approach prove unsuccessful as well. The approach that worked was to remove the agent's postion and the objective's position from the observations.

The agent's training process was done using multiple neural network configurations, including different number of network layers (1, 3, 5, 7) and different number of units per layer (128, 256, 512).

The hyperparameters used for the training can be seen in Table 4.2.

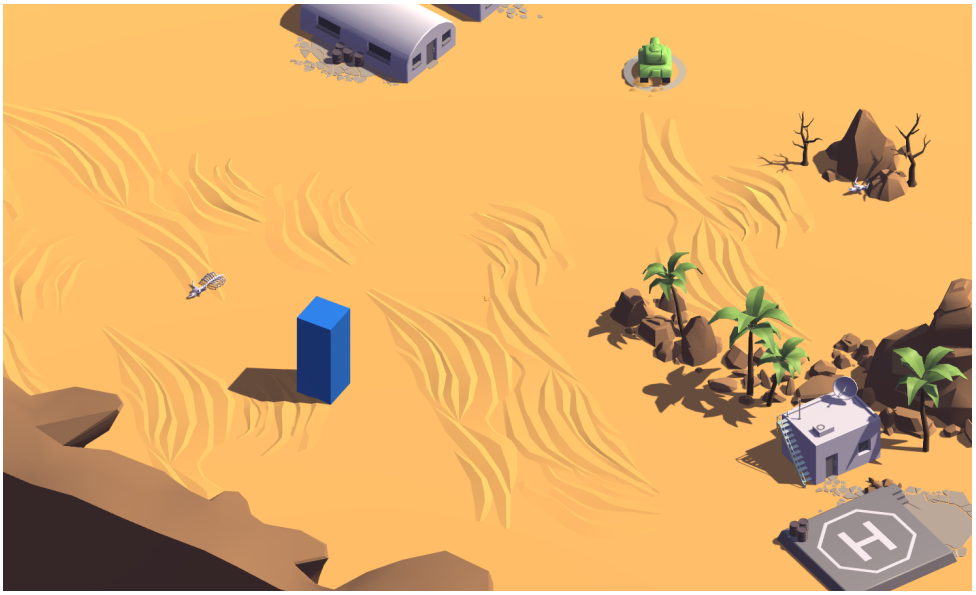

Figure 4.2: Photo of agent (green) and the target that it is supposed to reach (blue)

Hyperparameter	**Value**
Gamma	0.9
Lambda	0.95
Beta	0.005
Epsilon	0.2
Buffer Size	8192
Batch Size	256
Number of Epochs	3
Learning Rate	0.0003

Table 4.2: Training hyperparameters

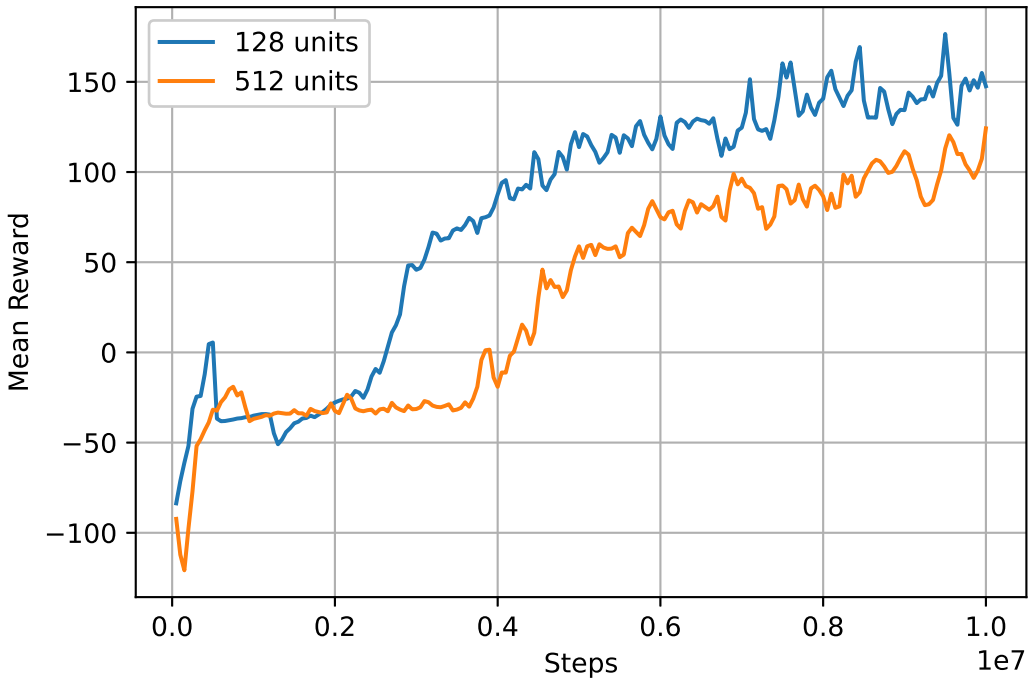

Figure 4.3: Training results for reaching a static target with a network with 1 hidden layer

The results for training the agent with 1 network layer can be seen in Figure 4.3, and in Table 4.3. In this case, the configuration with 256 units was not able to be trained properly, remaining stuck moving in a circle. Between the two configurations that were successfully trained, it can be seen that the one with 128 units, performed better during the training process than the one with 512 units, obtaining, at the end, a mean reward that is higher by 18.79%.

The results for training the agent with 3 network layers can be seen in Figure 4.4, and in Table 4.3. From these results, it can be seen that the configurations with 128 and 256 units, performed very similarily during the training, however, the one with 512 units started to lag behind these two at around $4 \cdot 10^6$ steps. Again, the configuration with the least units had the best training results, obtaining a result that is 2.61% better than the one with with 256 units and better by 29% than the one with 512 units.

Results for training the agent with 5 network layers can be seen in Figure 4.5, and in Table 4.3. From the training results, it can be seen that in the later stages, the configuration with 256 units falls behind the other two. Unlike the results with 1 layer and 3 layers, the best performing configuration is the one with 512 units per layer, being better by 8.72% than the configuration with 128 units, and by 35.45% than the configuration with 256 units.

Results for training the agent with 7 network layers can be seen in Figure 4.6, and in

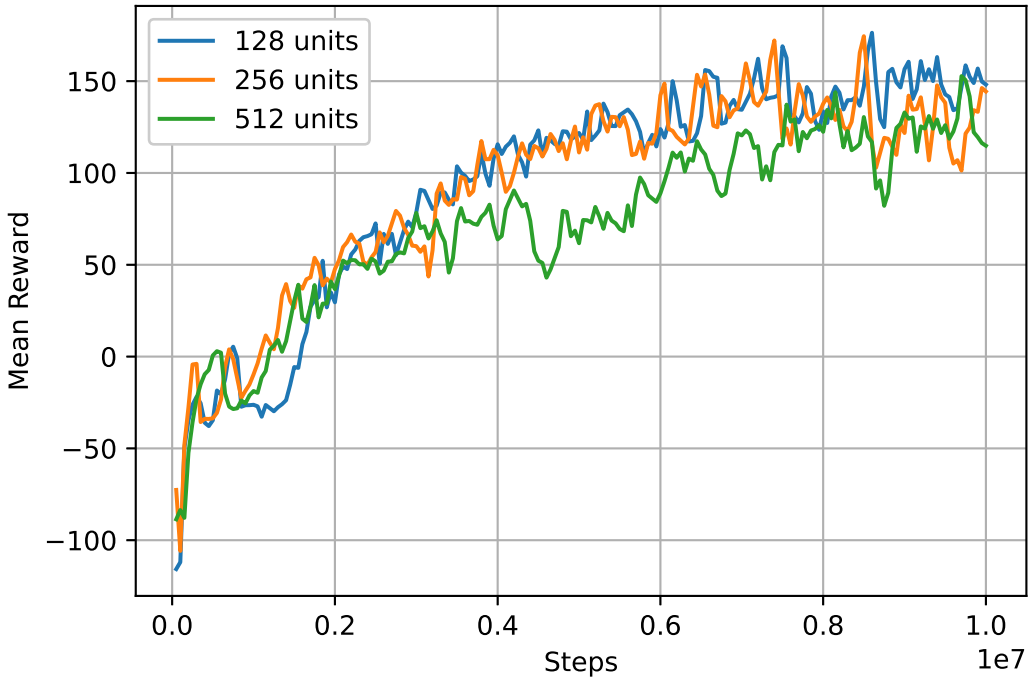

Figure 4.4: Training results for reaching a static target with a network with 3 hidden layers

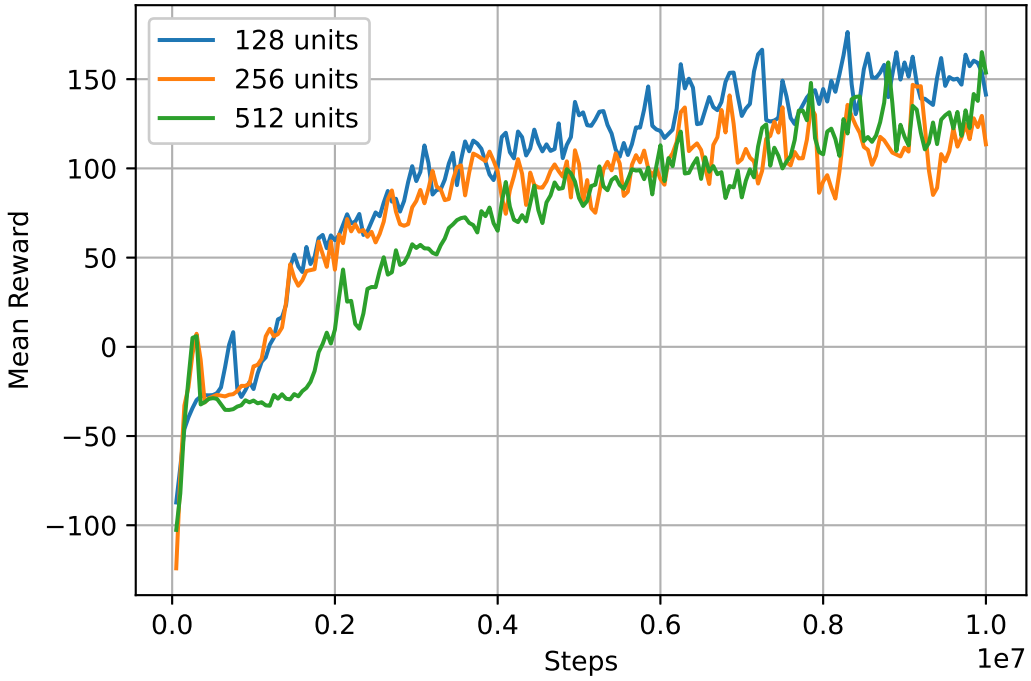

Figure 4.5: Training results for reaching a static target with a network with 5 hidden layers

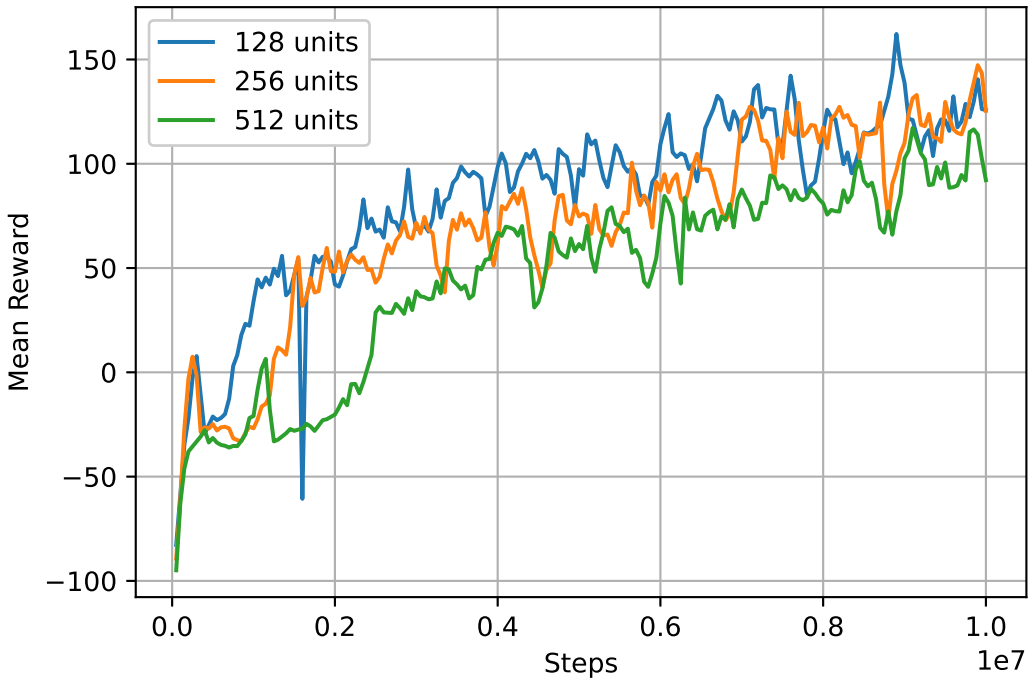

Figure 4.6: Training results for reaching a static target with a network with 7 hidden layers

Table 4.3. From the training results it can be seen that the configuration with 128 units per layer performs better than the other two, but at the final stages of the training, it is caught up to by the configuration with 256 units. In the end, the configurations with 128 and 256 units per layer had virtually identical results at the end of the training, and were better by $\sim 36.3\%$ than the one with 512 units.

All the training results can be seen in Table 4.3 and in Figure 4.7. It can be observed that network configurations with a smaller number of units per layer obtained better training results than the others. This could be due to the fact that the problem is not a very complex one, and because a smaller network can be trained faster. The configurations with 1, 3, or 5 layers obtained similar results, however, when 7 layers are used, there is a noticeable decrease in the agent's performance. These being said, from the training results it seems that using a network configuration with less layers and less units per layer offers almost the best result, thus it is the most practical one to be used, since it uses the least computing resources. Using larger networks should be done by training the agent for longer periods of time. However, it is possible when using a smaller network to obtain an unusable model, as was the case in this experiment using a network configuration with 1 hidden layer and 256 units per layer.

Network Configuration	Final Mean Reward
1 layer, 128 units	147.568
1 layer, 256 units	DNF
1 layer, 512 units	124.221
3 layers, 128 units	148.136
3 layers, 256 units	144.366
3 layers, 512 units	114.828
5 layers, 128 units	141.369
5 layers, 256 units	113.478
5 layers, 512 units	153.709
7 layers, 128 units	125.727
7 layers, 256 units	125.441
7 layers, 512 units	92.128

Table 4.3: Final training results for reaching a static target

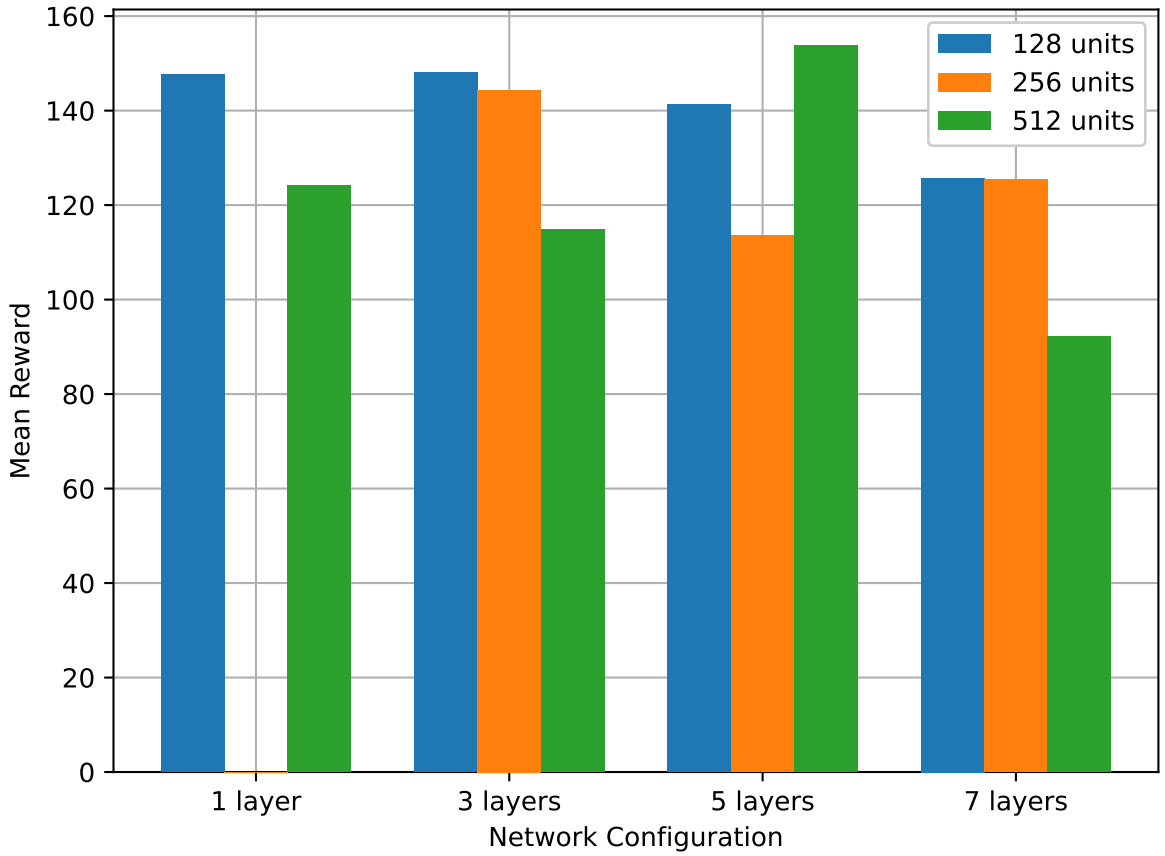

Figure 4.7: Training results for all network configurations for reaching a static target

4.2 Reaching a moving target

4.2.1 The Problem

Another standard behavior of an AI in video games is to be able to chase a target, so that it can, maybe, perform a specific action once the object controlled by the AI is close enough to the target . A simple way to achieve this behavior would be to use a *NavMesh* and periodically change the agent's destination. Compared to the problem described at 4.1.1, the addition here is that the agent has to recompute its route at specified intervals.

Again, the proposed solution would be to use AI agents that are trained using deep learning methods.

4.2.2 Implementing the solution

The basic implementation for this problem is the same as the one described at 4.1.2, with the same observations, rewards and penalties being used. The addition would be to add an observation that is the direction in which the target is moving. This can be used by the agent to predict in which direction the target is moving and to eventually intercept it along its trajectory. Another idea would be to add memory to the agent, in the form of LSTM, so that it can achieve this result in a more *natural* way.

4.2.3 Training

The training is similar to the process described at 4.1.3, with the same number of agents being trained at the same time, same number of steps, same hyperparameters (Table 4.2), etc.

Trying to add memory to the agent was a failure, the agent being unable to learn to chase the target, and getting stuck moving in a circle. This behaviour happened even when changing the recurrent network's hyperparameters *sequence length* and *memory size*.

Training this agent was done in 2 separate ways: one where the target's direction is not in the observations, and one where it is. The first round of training is done without the observation that was mentioned previously.

Results for training the agent with a single network layer and without the target's direction obsrevation can be seen in Figure 4.8 and Table 4.4. From these results it can be seen that all 3 configurations performed virtually identically, and at the end of the training had obtained the same mean reward.

Results for training the agent with 3 network layers and without the target's direction obsrevation can be seen in Figure 4.9 and Table 4.4. In these results, it seems that the configuration with 128 units obtains better results during training than the configuration

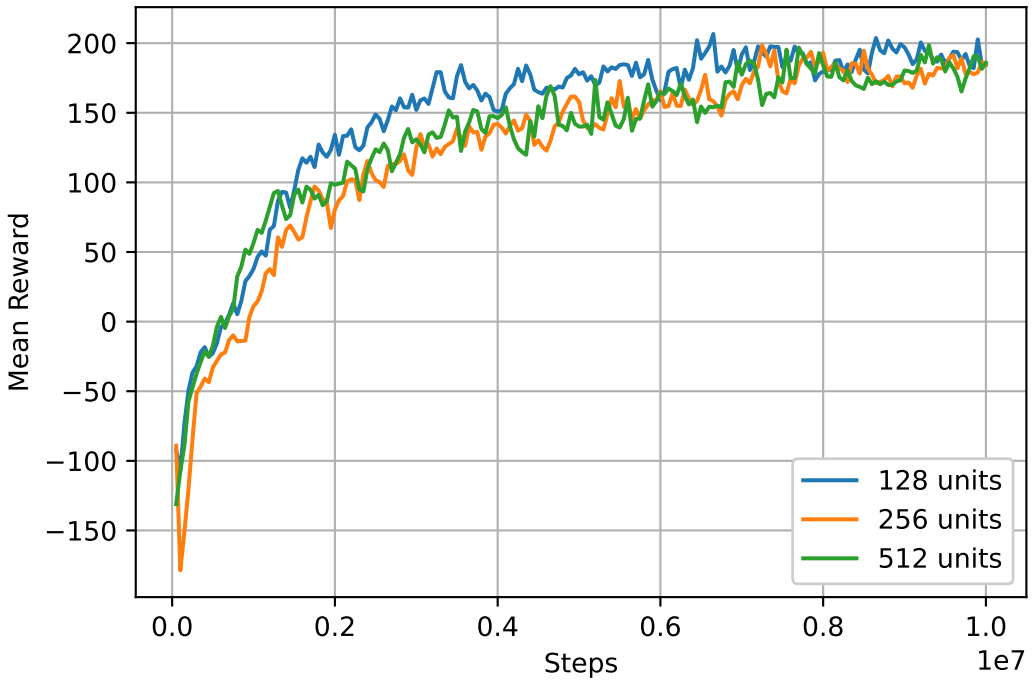

Figure 4.8: Training results for reaching a moving target with a network with 1 hidden layer

with 256 units, which in turn obtains better results than the one with 512 units. At the end of the training, the configuration with 128 units is better by 1.02% than the one with 256 units and better by 28.58% than the one with 512 units.

Results for training the agent with 5 network layers and without the target's direction obsrevation can be seen in Figure 4.10 and Table 4.4. The results during training are similar to the configuration with 3 layers, with the configuration with 128 units being better than the other two. The final result is that the configuration with 128 units is better by 8.87% than the one with 256 units and better by 17.9% than the one with 512 units.

The second round of training is done by adding the target's movement direction to the agent's observations.

The results for training the agent with a single network layer and without the target's direction obsrevation can be seen in Figure 4.11 and Table 4.4. As in the previous cases, the best performed configuration during training is the one with the least units, i.e. 128 units. This one is better by 39.47% than the configuration with 256 units, and by 6.88% than the one with 512 units.

Training the agent with a network with 3 layers obtains the results that can be seen in Figure 4.12 and Table 4.4. This time, all 3 configuration performed very similarily during

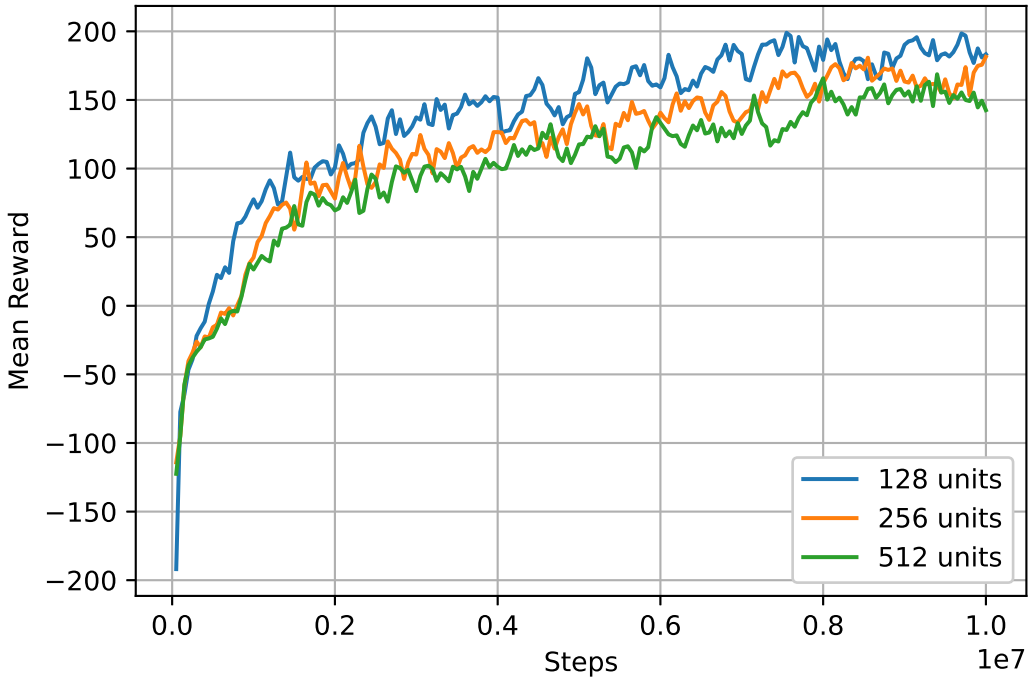

Figure 4.9: Training results for reaching a moving target with a network with 3 hidden layers

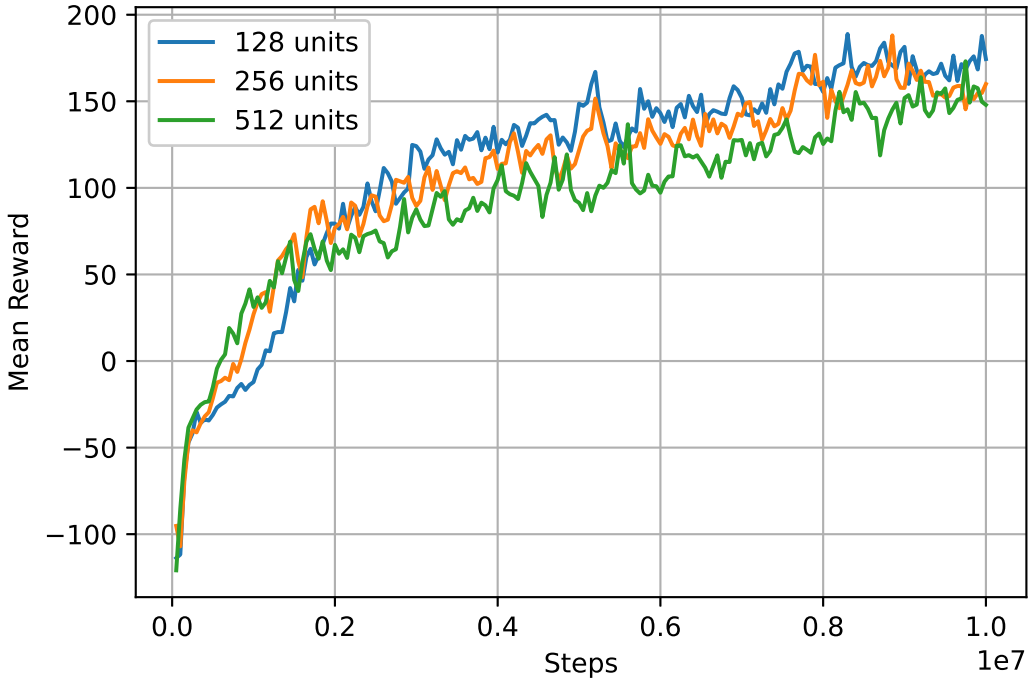

Figure 4.10: Training results for reaching a moving target with a network with 5 hidden layers

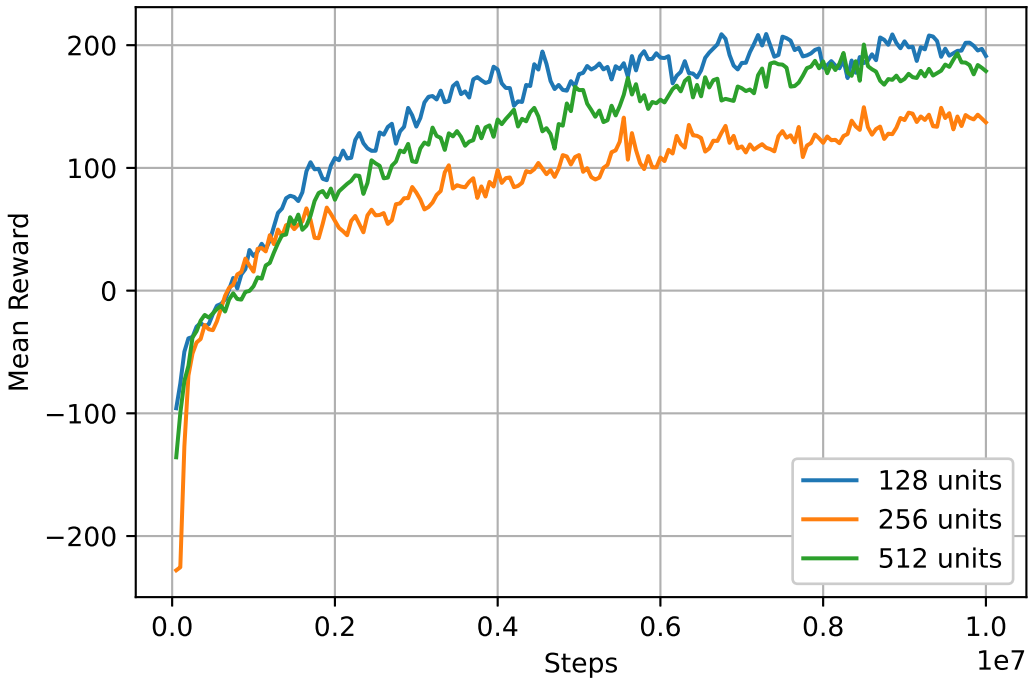

Figure 4.11: Training results for reaching a moving target with a network with 1 hidden layer and with observation of target's direction

training. The best configuration here was the one with 512 units which was better by 4.99% than the one with 128 units, and by 7.37% than the one with 256 units.

Results for training the agent with 5 network layers and without the target's direction obsrevation can be seen in Figure 4.13 and Table 4.4. It can be observed that the configuration with 128 units outperforms the other two during training, the ohter two performing mostly identically until the final training steps, where the one with 512 units slightly overtakes the one with 128 units. The configuration with 512 units is better by 1.26% than the one with 128 units and better by 17.93% than the one with 256 units.

Looking at the final results in Figure 4.14 and Table 4.4 we can observe that the best netowrk configurations have either 128 units per layer or 512 units per layer. When increasing the number of layers and not using the target's movement direction as an observation, the agent's performance slightly decreased during the training. Adding the target's movement direction as an observation increases the performance of the agent, compared to the configuration with the same number of layers and without this observation.

In conclusion, from these training results, it seems that adding the target's moving direction to the observations improves the agent's performance, the best results are achieved with either 128 or 512 units per hidden layer, and the best number of hidden layers is 3.

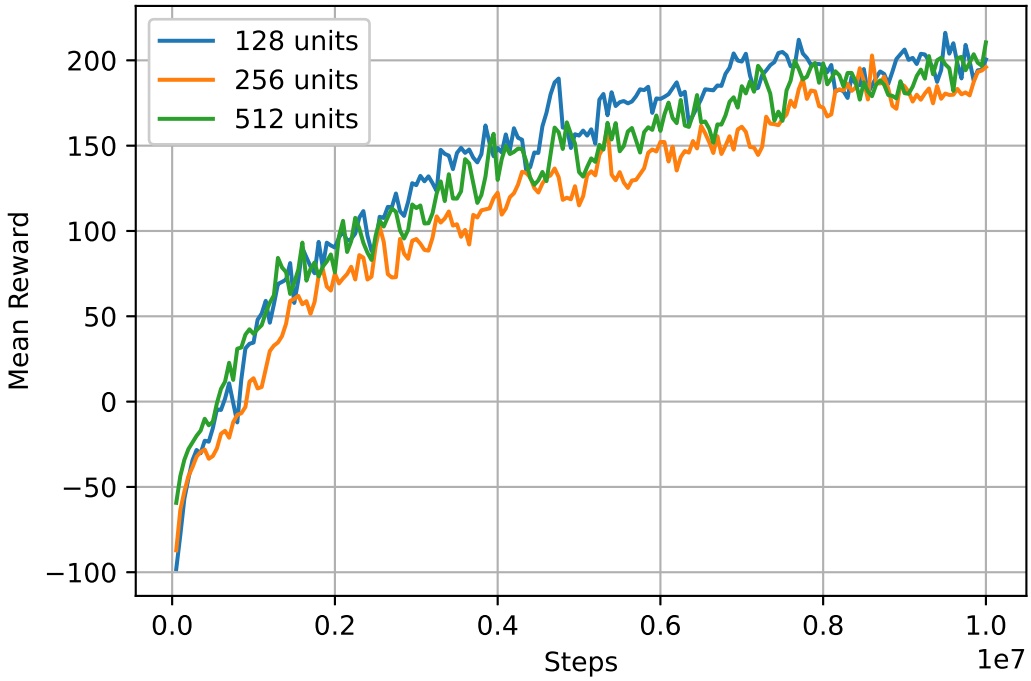

Figure 4.12: Training results for reaching a moving target with a network with 3 hidden layers and with observation of target's direction

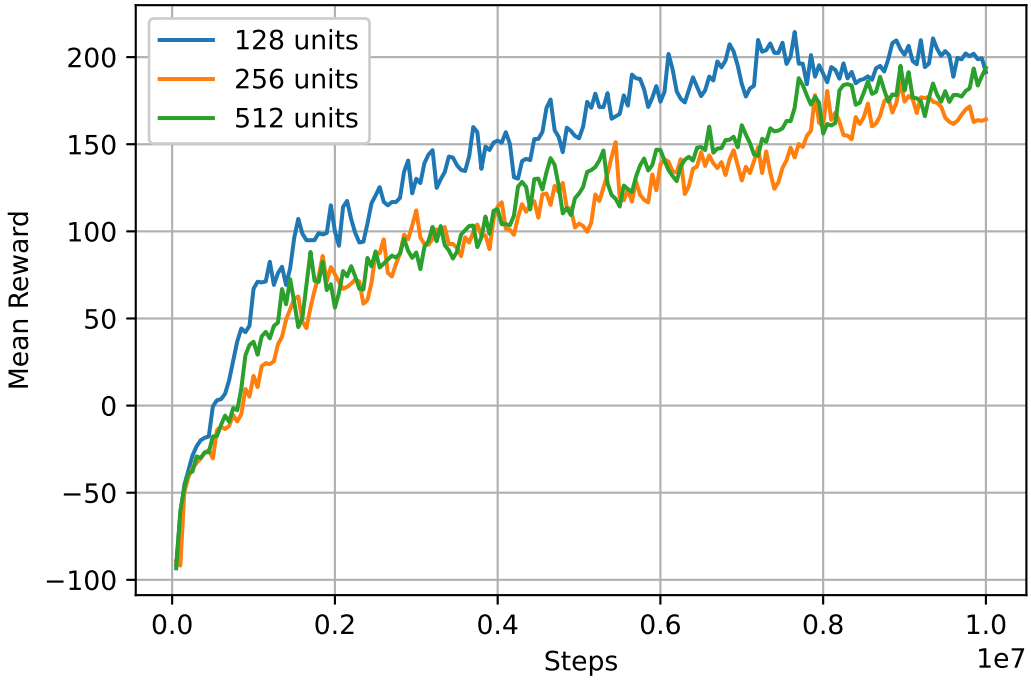

Figure 4.13: Training results for reaching a moving target with a network with 5 hidden layers and with observation of target's direction

Network Configuration	Observed target's direction	Final Mean Reward
1 layer, 128 units	No	184.547
1 layer, 128 units	Yes	191.184
1 layer, 256 units	No	184.563
1 layer, 256 units	Yes	137.075
1 layer, 512 units	No	185.936
1 layer, 512 units	Yes	178.876
3 layers, 128 units	No	183.288
3 layers, 128 units	Yes	200.502
3 layers, 256 units	No	181.435
3 layers, 256 units	Yes	196.064
3 layers, 512 units	No	142.54
3 layers, 512 units	Yes	210.524
5 layers, 128 units	No	174.404
5 layers, 128 units	Yes	191.399
5 layers, 256 units	No	159.966
5 layers, 256 units	Yes	164.351
5 layers, 512 units	No	147.922
5 layers, 512 units	Yes	193.828

Table 4.4: Final training results for reaching a moving target

The difference between using 128 units instead of 512 is a $\sim 5\%$ decline in the agent's performance. If there are no reasons to reduce resource usage, the configuration with 3 hidden layers and 512 units per layer should be used.

4.3 Shooting a moving target

4.3.1 The Problem

Another common use of AIs in gaming is to provide an opponent for the player, in this case to fight against. This brings a whole new dimension to the problem besides having the AI move in the given worldspace. Now, the AI must find the target, then be able to shoot at it, and also try to defend itself against the target, here by not getting hit by it. The AI must also be *defeatable*, since providing players with an opponent that can always take the best course of action and being superior in every way to the human player does not constitute a fun experience. By not being *perfect*, the human has a chance to defeat the AI, thus having a pleasant gaming experience. Balancing an AI to provide a good experience for a human player is out of the scope of this paper, the goal here being to manage to train an agent that can fight and eventualy win against an opponent. However, the agent should act in a way so that it could be actually used in a game. This would consist of acting more *humanlike*, and less like a predictable computer program.

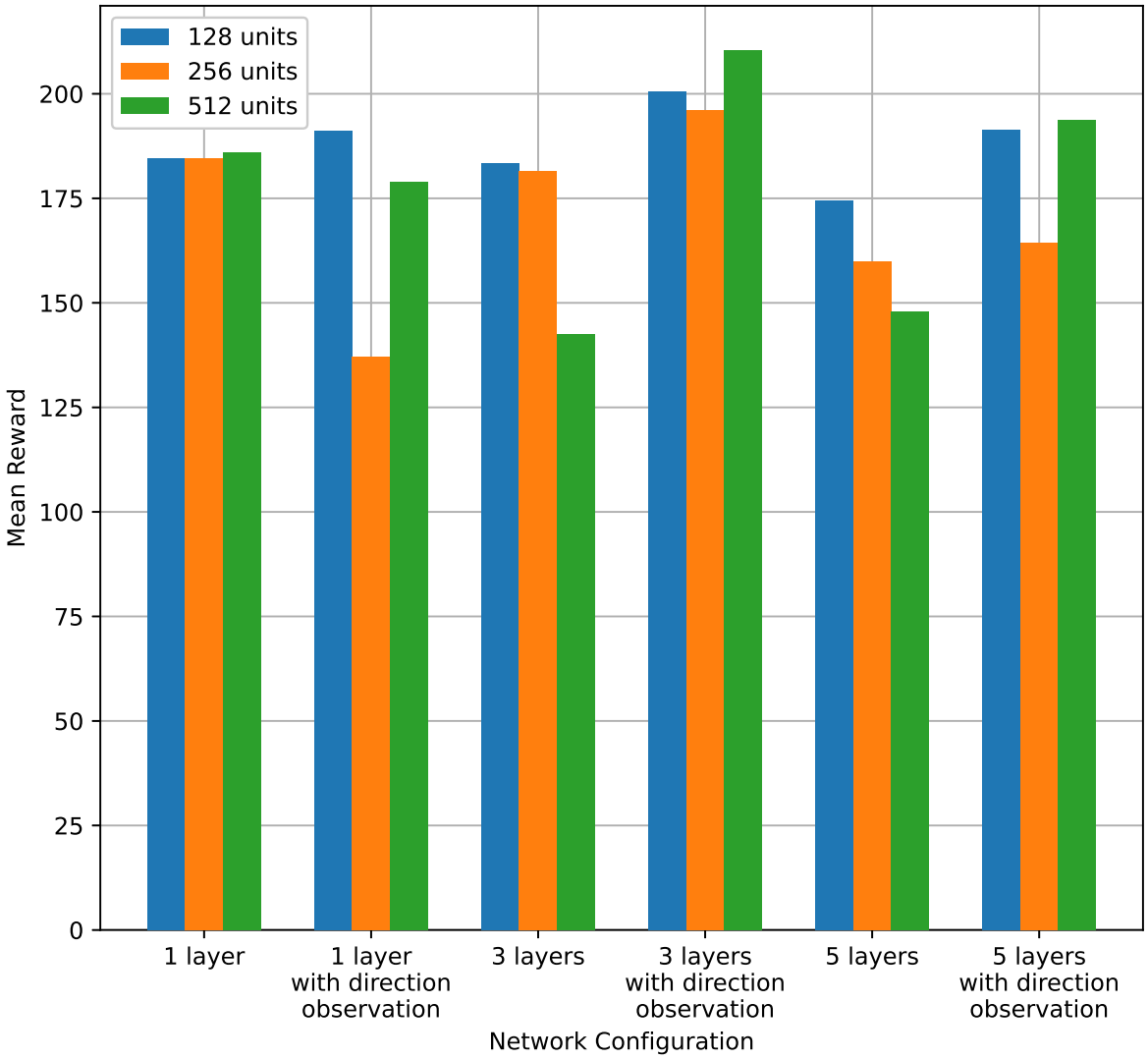

Figure 4.14: Training results for all network configurations for reaching a moving target

4.3.2 Implementing the solution

The implementation builds on what was described in Sections 4.1.2 and 4.2.2. The addition is that the agent does not need to reach a certain target, but to shoot it. For this, several aspects need to be changed to implement the bullet logic.

For starters, the agent's action space was increased from 2 branches to 4 branches. The first 2 branches will still handle the agent's movement, the third branch can have two values: 0 or 1, and this tells the agent if he should fire a bullet, and the fourth branch sets the bullet's shooting force, which can have 3 values: low, medium, or high.

Another change that was made, was adding new observations, most of them being related to the bullet. The first one is a flag that tells if the bullet is fired. The second one is the bullet's trajectory, which can be used by the agent to learn to shoot on target. The third observation is about the bullet's speed so that the agent can learn with what force the bullet has been shot. The final new observation, which is not related to the bullet, is the angle between the target's front vector and the vector from the target to the agent. This is used so that the agent can learn when it is in the target's crosshairs, and to possibly learn how to flank the target.

In summary, the following observations have been added:

- bullet is fired or not
- bullet's trajectory vector
- bullet's speed
- angle between the target's front vector and vector from target to the agent

Finally, new rewards and punishments were implemented. Firstly, the agent is no longer rewarded if it touches the target, it has to hit it with a bullet. To incentivise firing a bullet, a small reward is received when firing a bullet. Also, a penalty is added if the bullet misses the target, and disappears because it has travelled a certain distance, or it hit an environmental object. The other bullet reward that was added is a reward that is based on the bullet's trajectory and how close it is to the optimal trajectory to the target and is scaled by the distance from the bullet to the target, with the reward increasing the closer the bullet gets to the target. This reward is defined using Formula 4.8:

$$R_{\text{bullet trajectory}} = \frac{(1 - \frac{\alpha}{180}) \cdot r}{d} \cdot 2 \tag{4.8}$$

where:

α : is the angle between the bullet's current trajectory and the optimal trajectory

r : is the reward that is obtained if the bullet has the optimal trajectory

d : is the distance from the bullet to the target

Two punishments were added to help the agent learn as if it were fighting a real opponent. The first one makes the agent stay at a given distance from the target, so that the agent will not try to stay glued to the target while trying to shoot it. This punishment is computed using Fomrula 4.9:

$$R_{\text{desired distance}} = \begin{cases} \frac{d-d'}{50} \cdot p & \text{for } d \geq d' \\ \frac{p}{d} & \text{else} \end{cases} \tag{4.9}$$

where:

d : is the distance from the agent to the target, that is clamped in the interval $[0, 50]$

d' : is the desired distance from the agent to the target

p : is the penalty for the agent not being at desired distance d' from the target

The second punishment is applied when the agent is in front of the target, which means that the target can attack it. This punishment is added to try and teach the agent that it should flank its target so that it can attack it while being safe from being shot at. The punishment is computed using Formula 4.10

$$R_{\text{in target's crosshair}} = (1 - \frac{\alpha}{180}) \cdot p \tag{4.10}$$

where:

α : is the angle between the target's forward vector and the vector from the target to the agent

p : is the penalty for being in front of the target

In addition to the rewards and penalties already being used, described in Table 4.1, the new ones have the following values:

4.3.3 Training

The agent's training was done in 2 ways: a naive way, where the agent is only rewarded if it manages to shoot the target, without regards with its positioning, and a more *tactical* way where the agent is punished for not keeping distance between it and the target, and also for being in front of the target. These two approaches are documented at 4.3.3 and 4.3.3 respectively.

Name	Value	Notes
Shoot Target Reward	10	
Fire Bullet Reward	0.1	
Bullet's Trajectory is Optimal Reward	0.001	is scaled by the distance between bullet and target
Miss Target Penalty	-0.1	
Agent Not At Desired Distance Penalty	-0.005	is scaled by the difference of the desired distance and distance between agent and object
Agent In Front Of Target Penalty	-0.05	is scaled by the angle between the target's front vector and the vector from the target to the agent

Table 4.5: New Rewards and Penalites for agent that is shooting a moving target

Naive approach

As mentioned above, this approach only cares about the agent shooting the target, and not necessarily about the agent's positioning in relation to the target. This means that the penalties described in Formulae 4.9 and 4.10 are not applied. However, the reward concerning the bullet's trajectory described in Formula 4.8 is used. The observations regarding the bullet (if the bullet is fired, the bullet's trajectory vector and its speed) are used, while the observation regarding the angle between the target's front vector and vector from target to the agent is unused.

It is also worth noting that during the first trainings, the only observation regarding the bullet that was used is the one that tells if the bullet is fired.

Results for training the agent with a single network layer can be seen in Figure 4.15 and Table 4.6. As with the previous agents, the best performing configuration during training is the one with 128 units per layer, while the other two lag behind it. The configuration with 128 units was better by 32.12% than the one with 256 units, and better by 10% than the one with 512 units.

The results for training the agent with 3 network layers are in Figure 4.16 and Table 4.6. Here, the configuration with 128 units clearly overtakes the other two during most of the training process. The configuration with 128 units was better by 16.75% than the one with 256 units, and better by 41.25% than the one with 512 units at the end of the training.

Training the agent with a network with 5 layers obtains the results that can be seen in Figure 4.17 and Table 4.6. The situation is similar to the previous ones, meaning that the configuration with 128 units still performs the best. It is better by 1.76% than the one with 256 units and better by 29.38% than the one with 512 units per layer.

Results for training the agent with 7 network layers can be seen in Figure 4.18 and

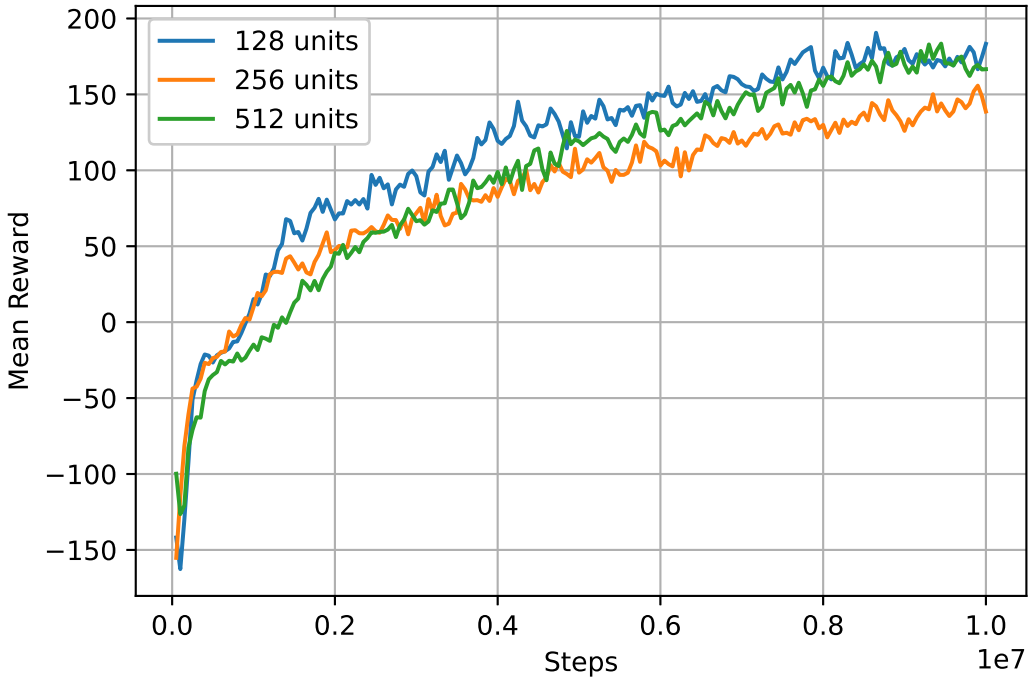

Figure 4.15: Training results for shooting a moving target with a network with 1 hidden layer

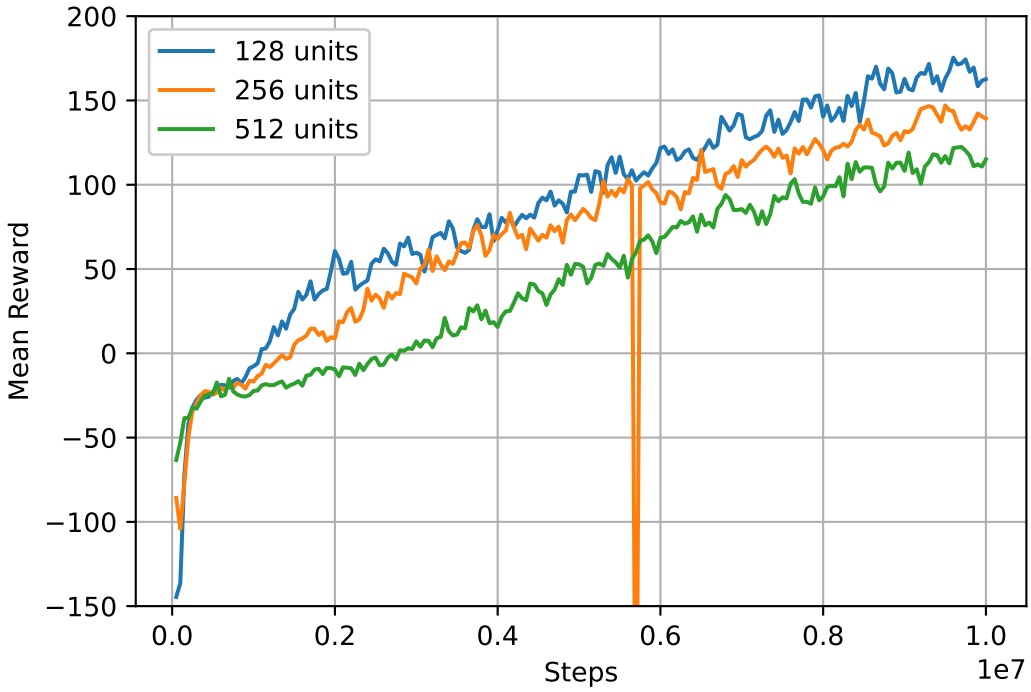

Figure 4.16: Training results for shooting a moving target with a network with 3 hidden layers

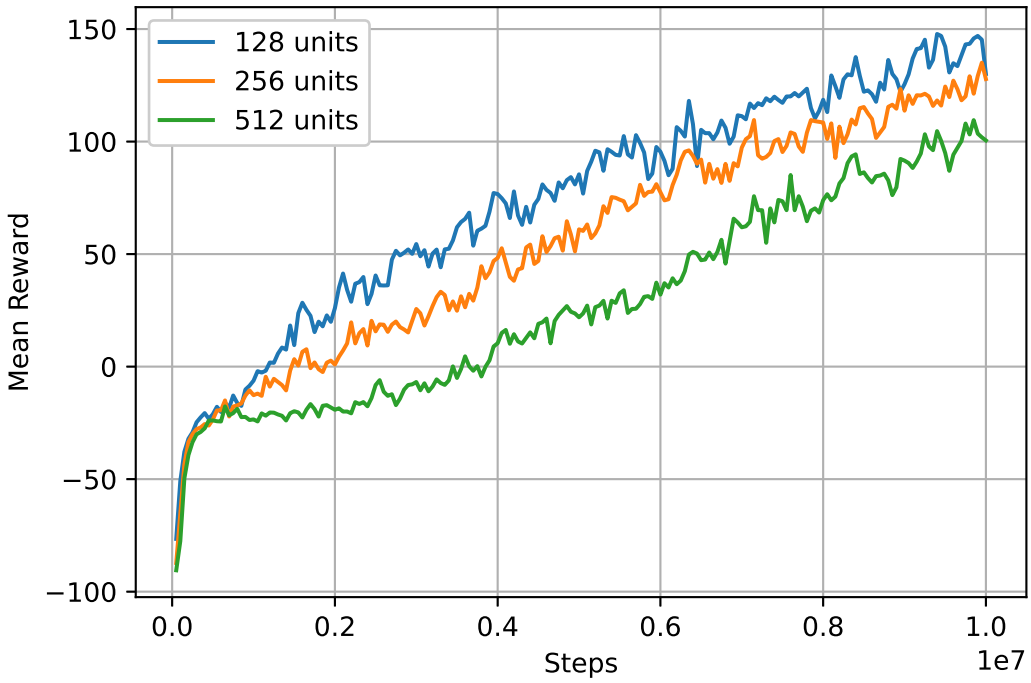

Figure 4.17: Training results for shooting a moving target with a network with 5 hidden layers

Table 4.6. Again, as in the previous cases, the configuration with the smallest number of units per layer performs better than the other ones during training. The configuration with 128 units performs better by 10.76% than the one with 256 units, and better by 29.09% than the one with 512 units per layer.

The final results for all configurations can be seen in Figure 4.19 and Table 4.6. From here we can deduce that using a smaller network with fewer units per layer gives the best results during training. Increasing the number of layers decreased the performance of the agent in every single case by up to $\sim 40\%$. Thus, it can be clearly infered that using a network architecture with 1 layer and 128 units per layer should give the best performance of the agent, and also use the least amount of computing resources.

In the second part of the trainings, the agent is trained using a configuration of 128 units per network layer, since this yielded the best results in the previous phase, and different observations regarding the bullet. The first observation combination uses all three bullet observations: if the bullet is fired, its trajectory vector and its speed; the second observation combination contains only two bullet observations: the bullet's speed and if the bullet is fired; and the final combination contains only the observation that tells if the bullet was fired. This comparison is made to see if adding more observations would increase the agent's performance, or decrease it, similar to what happened when

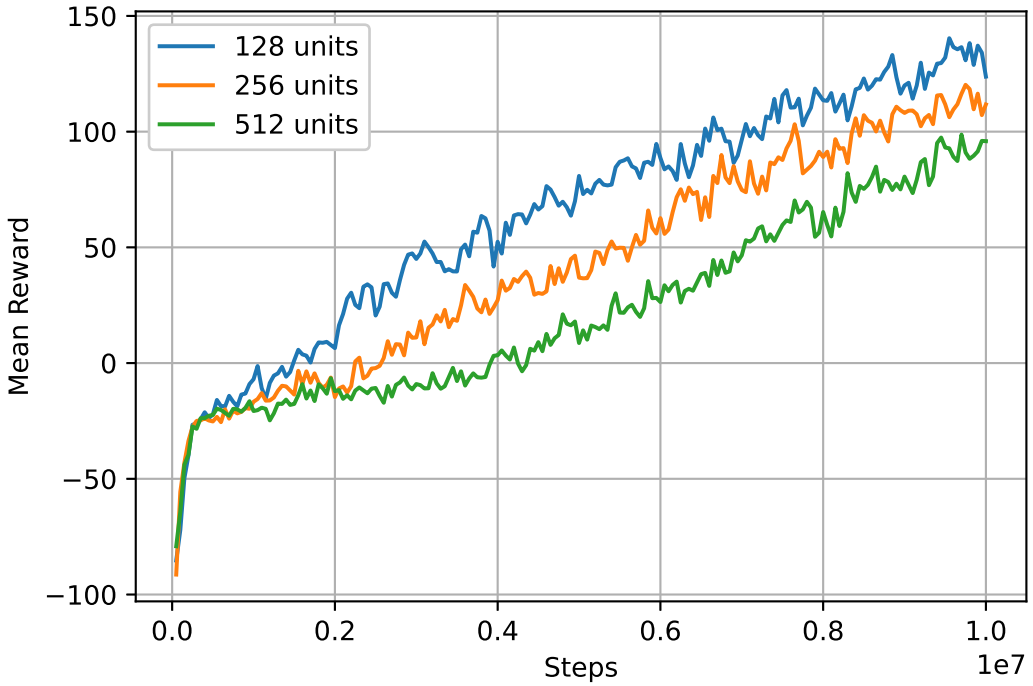

Figure 4.18: Training results for shooting a moving target with a network with 7 hidden layers

Network Configuration	**Final Mean Reward**
1 layer, 128 units	183.278
1 layer, 256 units	138.719
1 layer, 512 units	166.612
3 layers, 128 units	162.629
3 layers, 256 units	139.293
3 layers, 512 units	115.129
5 layers, 128 units	130.032
5 layers, 256 units	127.781
5 layers, 512 units	100.497
7 layers, 128 units	123.777
7 layers, 256 units	111.752
7 layers, 512 units	95.882

Table 4.6: Final training results for shooting a moving target

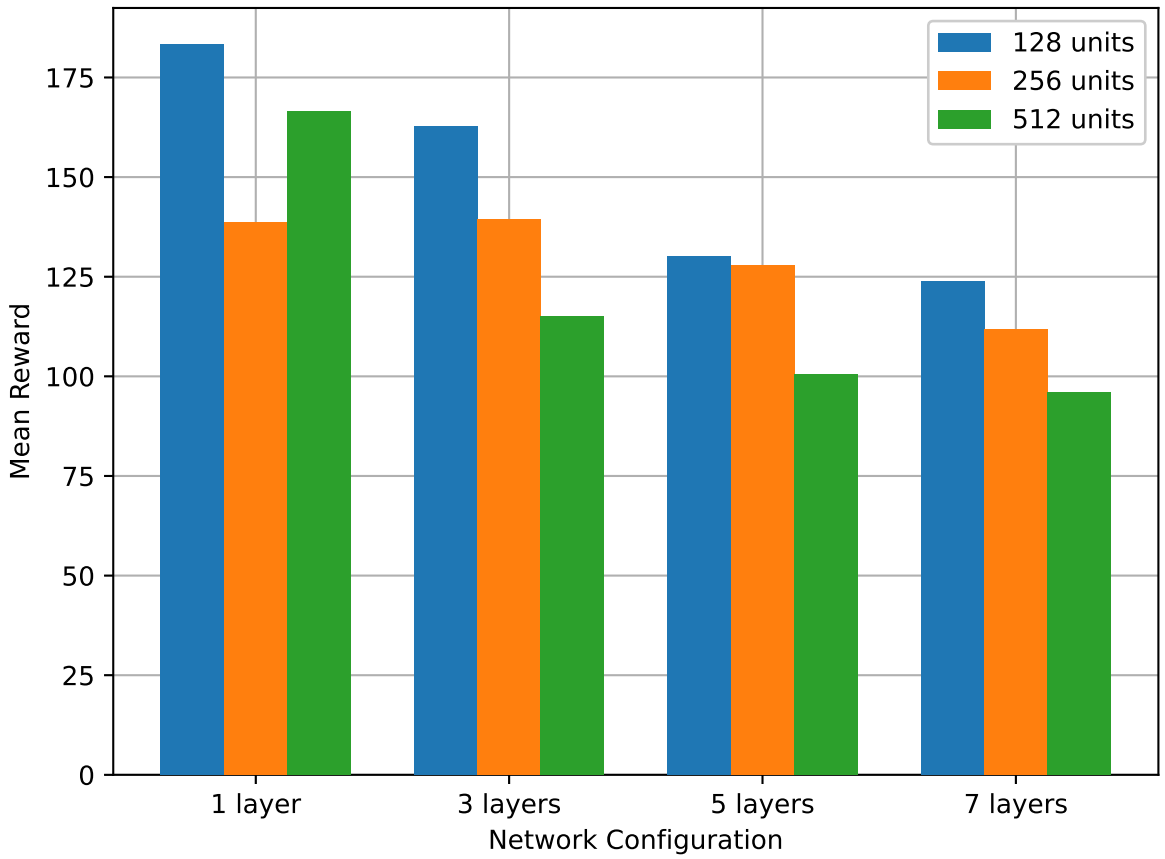

Figure 4.19: Training results for all network configurations for shooting a moving target

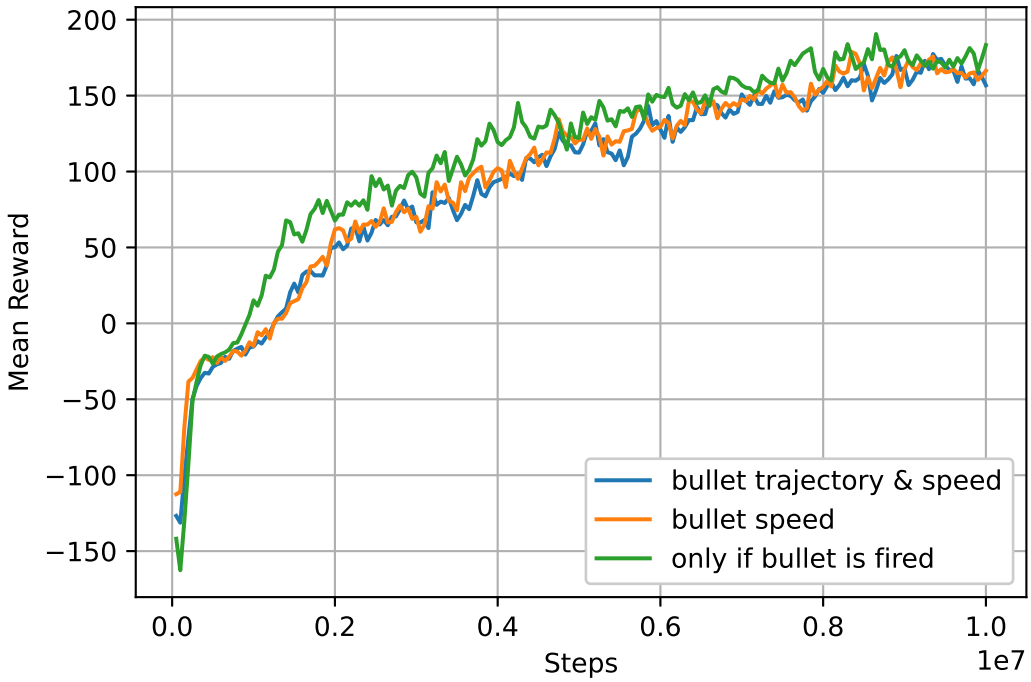

Figure 4.20: Training results for shooting a moving target with a network with 1 hidden layer, 128 units, and different observations

adding the target's movement direction to the observations in Section 4.2.3.

Results for training the agent with a single network layer can be seen in Figure 4.20 and Table 4.7. All 3 observation combinations yield very similar results, with the variant that observes only if the bullet is fired being slightly better then the other two. It is better by 16.91% than the observation combination with both bullet trajectory and speed, and better by 10.12% than the combination with the bullet's speed.

The results for training the agent with 3 network layers are in Figure 4.21 and Table 4.7. Here, it is similar to the previous situation, with all 3 combinations achieving similar training results. Here, the combination with the bullet's speed is the best one, being better by 0.41% than the combination that observes only if the bullet is fired, which means these two combinations obtain virtually identical results, and better by 7.3% than the combination with both bullet trajectory and speed.

Training the agent with a network with 5 layers obtains the results that can be seen in Figure 4.22 and Table 4.7. Again, training results are very similar for all 3 combinations. Here, the best one is the combination with both bullet trajectory and speed being better by 11.69% than the combination with the bullet's speed, and better by 18.01% the the one that only observes if the bullet was fired.

Results for training the agent with 7 network layers can be seen in Figure 4.23 and Table 4.7. All 3 combinations perform virtually identically having a maximum difference

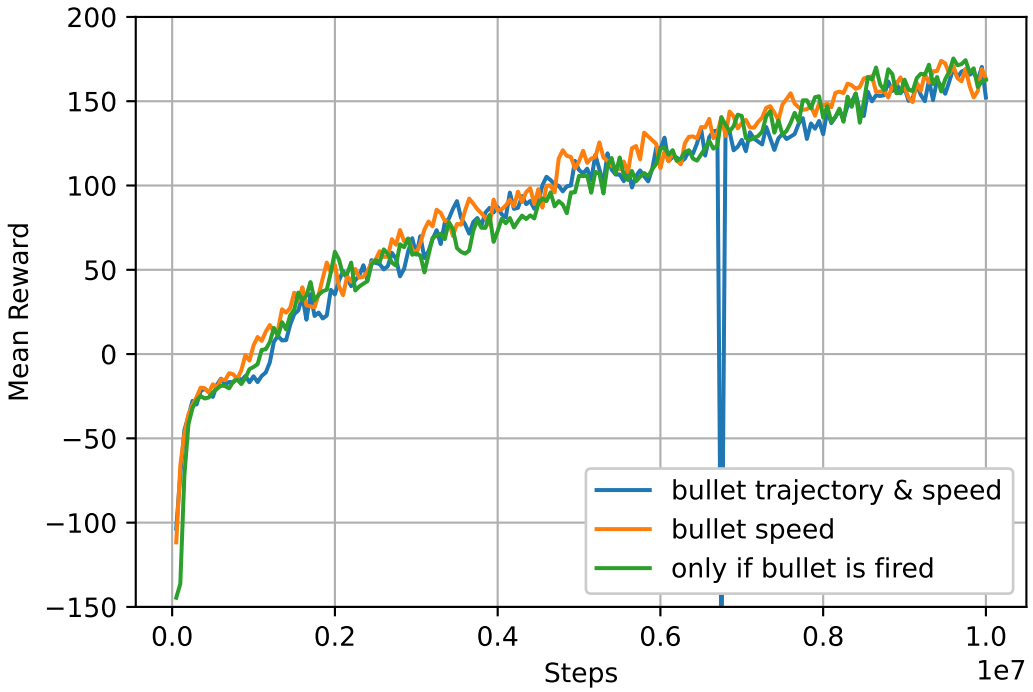

Figure 4.21: Training results for shooting a moving target with a network with 3 hidden layers, 128 units, and different observations

Figure 4.22: Training results for shooting a moving target with a network with 5 hidden layers, 128 units, and different observations

Figure 4.23: Training results for shooting a moving target with a network with 7 hidden layers, 128 units, and different observations

in agent performance of $\sim 3\%$.

The final results for all configurations can be seen in Figure 4.24 and Table 4.7. As with the first training stage, increasing the number of layers leads to a poorer performing agent. Also, there were no huge differences when introducing the new bullet observations, with the new observation combinations performing the same or slightly worse then when observing only if the bullet was fired. The only exception is when using a network with 5 layers, where adding the bullet's trajectory and speed to the observations obtains a performance that is 18.01% better than if not including them. However, the best performing configuration still remains the one with 1 layer, and where the only observation about the bullet is if it was fired.

In conclusion, when comparing the two approaches of adding or not multiple bullet observations, it seems that the best results are obtained when observing only if the bullet was fired. When the number of layers increseas, it looks like there is a tendency to obtain better results when adding more bullet observations. However, these results are still worse than when using a network architecture with 1 hidden layer, 128 units per layer, and the only bullet observation is if it was fired.

Network Configuration	Bullet Observations	Final Mean Reward
1 layer, 128 units	Bullet's trajectory and speed	156.766
1 layer, 128 units	Bullet's speed	166.427
1 layer, 128 units	Only if bullet was fired	183.278
3 layers, 128 units	Bullet's trajectory and speed	152.188
3 layers, 128 units	Bullet's speed	163.311
3 layers, 128 units	Only if bullet was fired	162.629
5 layers, 128 units	Bullet's trajectory and speed	153.46
5 layers, 128 units	Bullet's speed	137.391
5 layers, 128 units	Only if bullet was fired	130.032
7 layers, 128 units	Bullet's trajectory and speed	125.061
7 layers, 128 units	Bullet's speed	127.677
7 layers, 128 units	Only if bullet was fired	123.777

Table 4.7: Final training results for shooting a moving target with different bullet observations

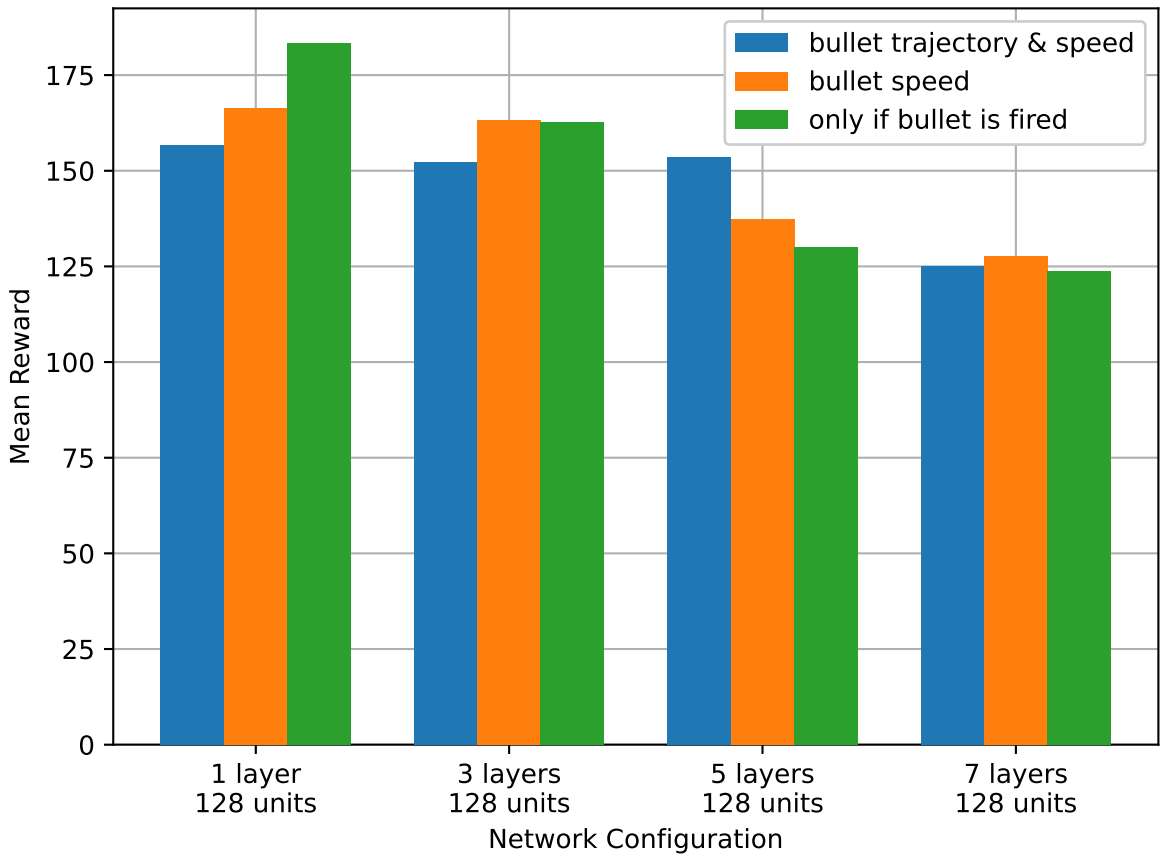

Figure 4.24: Training results for network configurations with 128 units per layer and different bullet observations, for reaching a moving target

Figure 4.25: Training results for shooting a moving target with a network with 1 hidden layer

Tactical approach

This approach tries to mimic a more realistic fighting scenario, where the target would shoot back at the agent. Thus, through the newly added punishments, described in Formulae 4.9 and 4.10, the agent should learn to keep a certain distance from the target, and also to flank it. Here, the only bullet observation that is made, is if the bullet was fired or not. This decision was made based on the training results obtained when training the agent in a more *naive* way, in the above experiment. Also, introducing more observations would require the agent to train for a larger number of steps to efficiently use them, if possible, and since the number of training steps still remains 10^7, and new penalties were added, this would lead to a poorer performing agent.

Results for training the agent with a single network layer can be seen in Figure 4.25 and Table 4.8. The configurations with 256 and 512 units per layer had similar perofrmances during the training, however the one with 128 units began to fall behind these two at around $6 \cdot 10^6$ steps. The best performing configuration was the one with 256 units.

The results for training the agent with 3 network layers are in Figure 4.26 and Table 4.8. The configurations with 128 and 256 units per layer performed virtually identically during the training, with the best one being the configuration with 128 units, while the

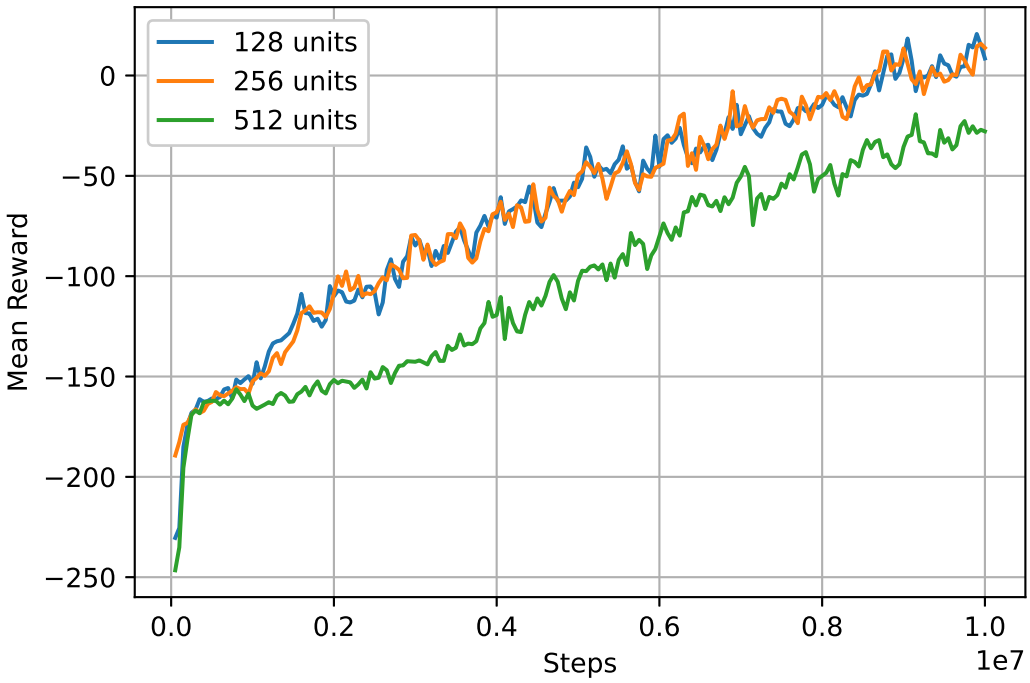

Figure 4.26: Training results for shooting a moving target with a network with 3 hidden layers

one with 512 units managed to fall behind them from the beginning.

Results for training the agent with 5 network layers can be seen in Figure 4.27 and Table 4.8. During the training, the configuration with 128 units per layer performs better than the one with 256 units, while the one with 512 falls behind these two by a large amount.

Training the agent with a network with 5 layers obtains the results that can be seen in Figure 4.28 and Table 4.8. The results are similar to the ones obtained with an architecture with 5 hidden layers with the configuration with 128 units per layer performing better than the one with 256 units, and the one with 512 units greatly falling behind these two.

In summary, for this type of agent, a smaller network architecture should be used, since it obtains better results, with networks with over 3 hidden layers having a heavily readuced performance. Also, configurations with over 128 units per layer should be used, since the ones with 128 units perform worse in comparison to the others, but only when the number of hidden layers is small.

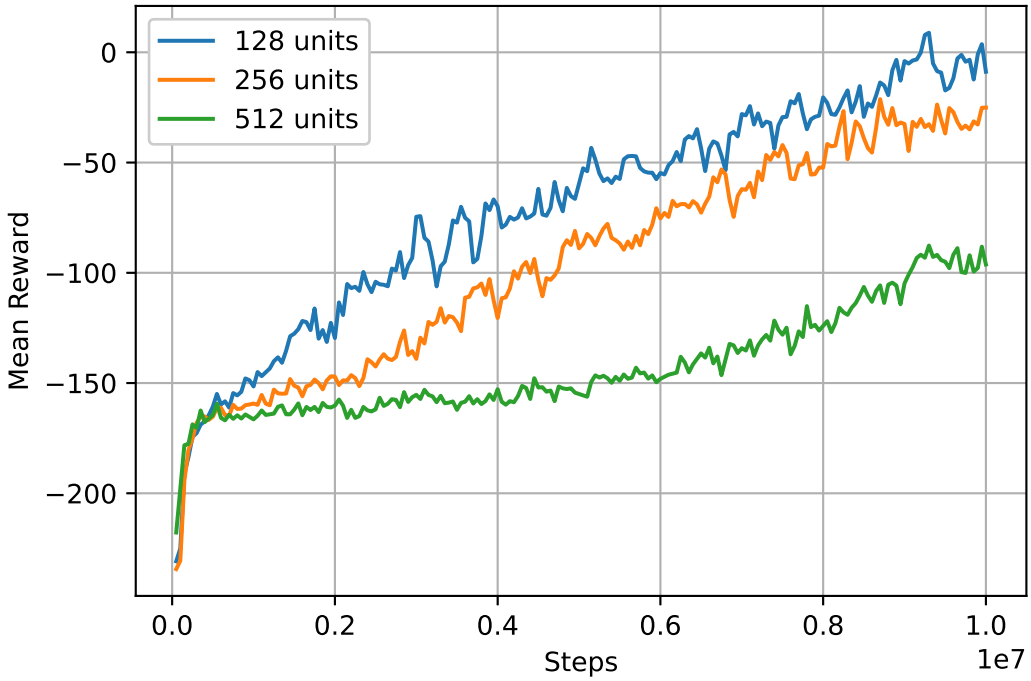

Figure 4.27: Training results for shooting a moving target with a network with 5 hidden layers

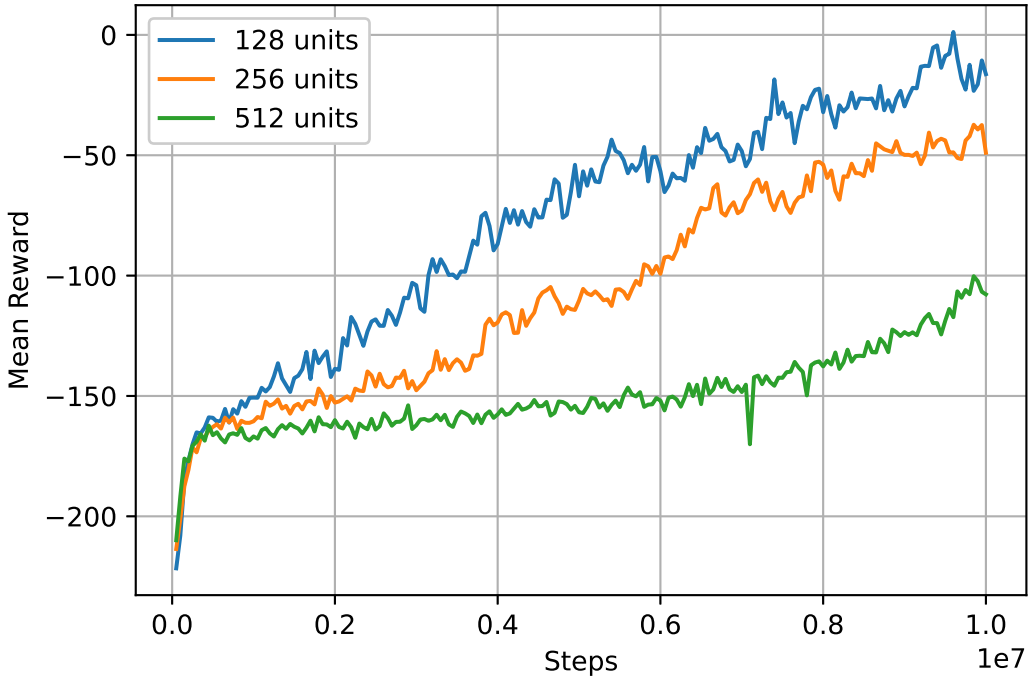

Figure 4.28: Training results for shooting a moving target with a network with 7 hidden layers

Network Configuration	Final Mean Reward
1 layer, 128 units	-4.727
1 layer, 256 units	40.452
1 layer, 512 units	25.225
3 layers, 128 units	8.479
3 layers, 256 units	13.789
3 layers, 512 units	-27.845
5 layers, 128 units	-8.788
5 layers, 256 units	-25.076
5 layers, 512 units	-96.232
7 layers, 128 units	-16.319
7 layers, 256 units	-48.976
7 layers, 512 units	-107.791

Table 4.8: Final training results for shooting a moving target

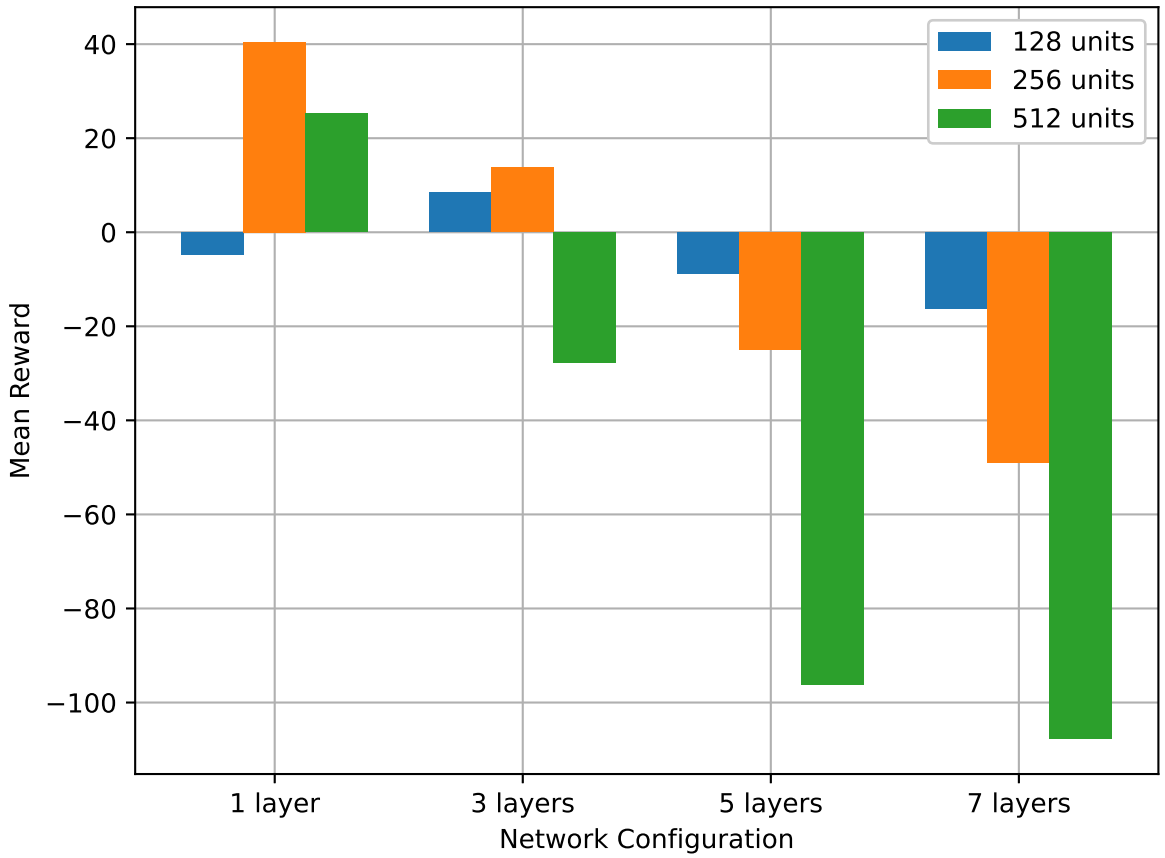

Figure 4.29: Training results for all network configurations for shooting a moving target

Chapter 5

Test Results

To understand how well the trained agent performs, training results are not enough since they only provide information about how well rewarded the agent is, and not about how it will perform in a real environment. For this, several tests were set up for the 3 types of problems that were tackled in this paper.

5.1 Reaching a static target

For reaching a static target, the test consists of an agent that has to touch 40 targets that are static (i. e. not moving) and that appear in a predefined order. Once the agent touches the target, it disappears and a new one appears in a new location, according to the predefined order. For each test, the same predefined order of targets is used so that the test course is the same for each run.

Initially, this test was done using only agents that were trained to reach static targets, however a more robust experiment would be to also run the test using agents that were trained to reach moving targets, and compare the results at the end.

5.1.1 Agent trained to reach static targets

The results for testing the agent that was trained to reach static targets can be seen in Table 5.1 and Figure 5.1. It can be observed that, unlike the training results, using a smaller network architecture does not yield better times, with the architecture with 5 hidden layers and 128 units per layer being the one that achieved the best time, 170.9822 seconds. For the architecure with a single hidden layer, the configuration with 128 units was faster by 22.66 seconds (11.01%) than the one with 512 units per layer. Since the configuration with 256 units per layer was unable to be trained, because it learned just to move in a circle, it was unable to finish the test course.

For the architecture with 3 hidden layers, the results were much more similar, with the configurations that had 256 and 512 units per layer, having virtually identical times, with

Network Configuration	**Time to complete (s)**
1 layer, 128 units	183.1415
1 layer, 256 units	DNF
1 layer, 512 units	205.8087
3 layers, 128 units	182.1884
3 layers, 256 units	180.4801
3 layers, 512 units	180.2796
5 layers, 128 units	170.9822
5 layers, 256 units	186.4627
5 layers, 512 units	205.8265
7 layers, 128 units	188.0436
7 layers, 256 units	182.1618
7 layers, 512 units	219.4408

Table 5.1: Test results for reaching a static target using agent that was trained to reach static targets

the difference between them being 0.2 seconds, and the configuration with 128 units being slower than these two by 2 seconds ($\sim$ 1% slower). These results are somewhat surprising since there was a differnce of 29% in the training performance of the configuration with 128 units compared to the one with 512 units.

The architecture with 5 hidden layers has the configuration that is the best performing in this test, the one with 128 units per layer, which manages to achieve a time of 170.9822 seconds. The other two configurations perform worse, the one with 256 units being slower by 15.4805 seconds ($\sim$ 8.3% slower), and the one with 512 units being slower by 34.8443 seconds ($\sim$ 16.92% slower). This shows that a smaller number of layers does not necessarily lead to better results, however, using less units per hidden layer usually means that the agent will perform better.

The final network architecture, the one with 7 layers has the configuration with 256 units be the fastest one in the test, clocking in at 182.1618 seconds. The configuration with 128 units was slower by 5.8818 seconds (3.22%), and the one with 512 units was slower by 37.279 seconds (20.46%).

In summary, from this test we can see that increasing the number of hidden layers used in the architecture does not necessarily decrease the agent's performance, when using up to 5 layers. Also, using a smaller number of units per layer tends to lead to better agent performance. This could be due to the fact that, according to [3], increasing the width of the network can lead to overfitting.

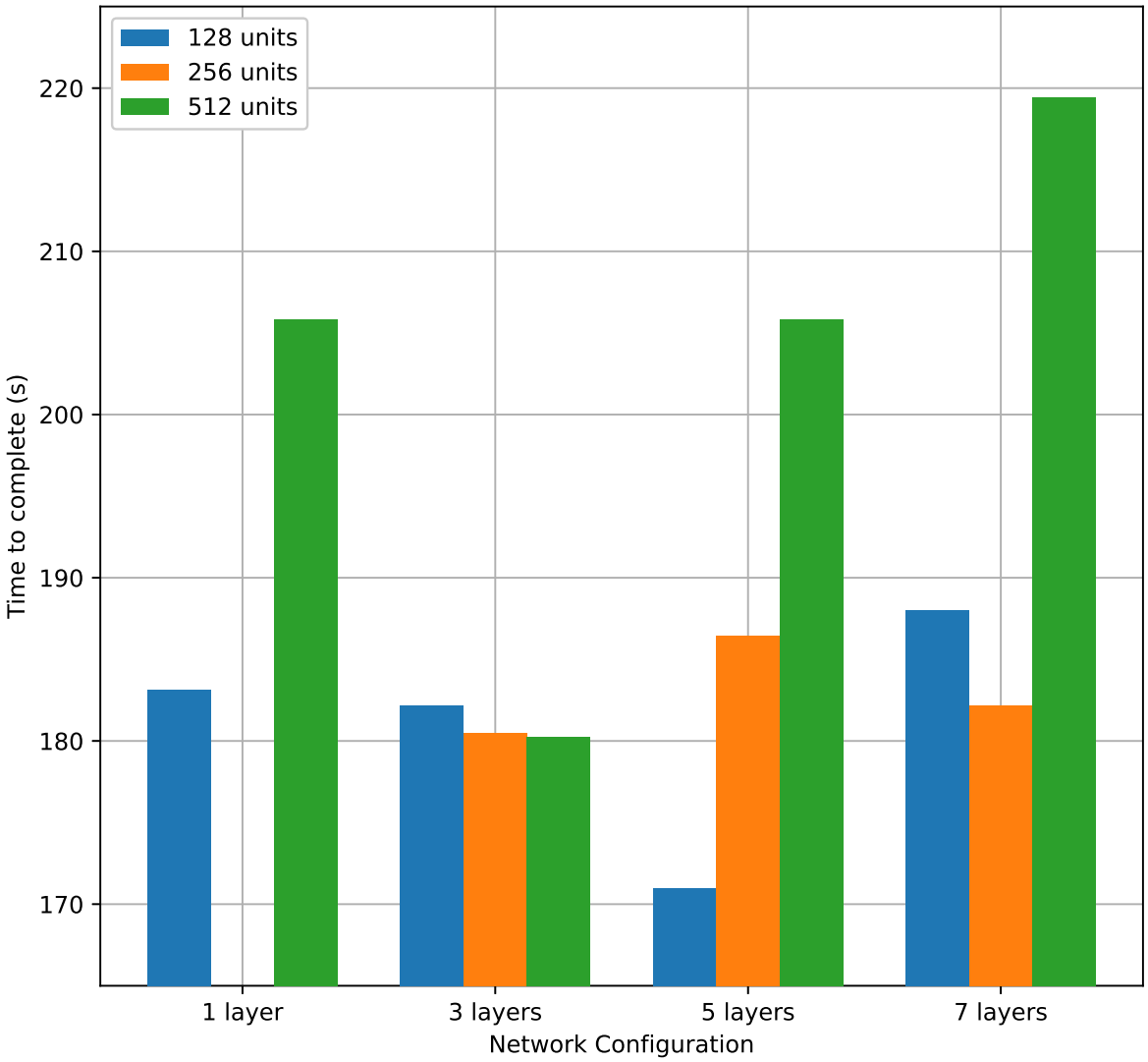

Figure 5.1: Test results for reaching a static target using agent that was trained to reach static targets

5.1.2 Agent trained to reach moving targets

The second part of the test is measuring how well an agent that was trained to reach moving targets can also reach static targets, and if it could also beat the agent that was trained to reach static targets. The obtained results are in Table 5.2 and Figure 5.2. Just like in the training Section 4.2.3, half of the tested agents will also have the target's direction as an observation (which is the vector $(0, 0, 0)$ since the target is not moving).

Looking at the results for the architecture with just a single hidden layer, it apperas that the best performing configurations are the ones with 128 units per layer, with the one without the target's direction observation being slightly faster, by 3.5389 seconds (1.8%). The configuration with 256 units per layer and with the target's direction observation takes 464.985 seconds to finish the test, more than twice the amount of time it took with 128 units per layer, due to the fact that it becomes stuck trying to move through a narrow path and can't decide to go through, because it tries to find an alternative path that does not exist. For the configurations with 512 units, the one where the target's direction is observed is faster than the other one by 7.2391 seconds (3.36%).

For the architecture with 3 hidden layers, the agents that observed the target's direction were faster than the others that did not. The fastest configuration was the one with 512 units per layer with the time of 188.5515 seconds. The second best was the one with 128 units which was slower by 3.6596 seconds (1.9%). The agents that had 256 and 512 units per layer and did not observe the target's direction, had very slow times, of 491.0297 seconds and 602.9445 seconds respectively. This was due to the fact that the agent became somehow confused and did not manage to get around the level's geometry.

The architecture with 5 hidden layers has both the slowest and fastest configuration, for this type of agent and test, at the same time. The fastest configuration was the one with 128 units and with the target's direction observation, and it finished the test course in 187.1074 seconds. The slowest configuration was the one with 512 units and without the target's direction observation, and it managed to finish the course in 576.1487 seconds. This was due to the fact that the agent was not able to correctly go around a wall and kept trying to go the wrong way for a long period of time. The configuration with 256 units and no target direction observation also managed to score a bad time of 363.128 seconds, becoming stuck in the same place as the previoulsy mentioned one.

In summary, here too, having a smaller number of units per layer seems to increase the performance of the agent, with almost all architectures having their best times be achieved by configurations with 128 units per layer. As for including the target's direction observation, it seems to positively impact the agent only when using a larger number of layers.

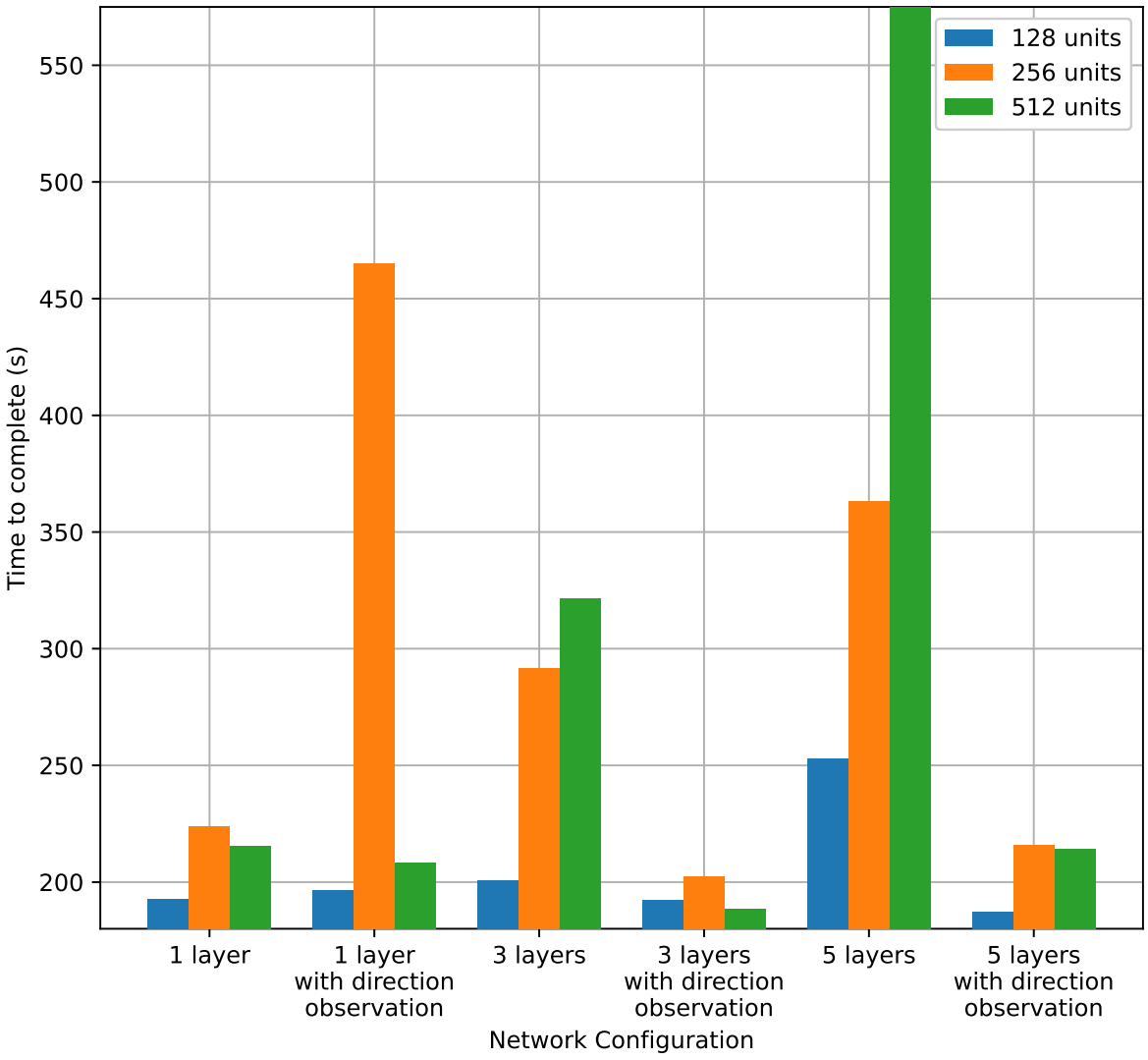

Figure 5.2: Test results for reaching a static target using agent that was trained to reach moving targets

Network Configuration	Observed target's direction	Time to complete (s)
1 layer, 128 units	No	192.7527
1 layer, 128 units	Yes	196.2916
1 layer, 256 units	No	223.6499
1 layer, 256 units	Yes	464.985
1 layer, 512 units	No	215.4364
1 layer, 512 units	Yes	208.1973
3 layers, 128 units	No	200.7101
3 layers, 128 units	Yes	192.2111
3 layers, 256 units	No	291.4421
3 layers, 256 units	Yes	202.1844
3 layers, 512 units	No	321.6053
3 layers, 512 units	Yes	188.5515
5 layers, 128 units	No	252.7738
5 layers, 128 units	Yes	187.1074
5 layers, 256 units	No	363.128
5 layers, 256 units	Yes	215.8631
5 layers, 512 units	No	576.1487
5 layers, 512 units	Yes	214.0343

Table 5.2: Test results for reaching a static target using agent that was trained to reach moving targets

5.1.3 Results comparison

Comparing the obtained results from the two approaches, the better one was when using an agent that was trained to reach static targets. The best time for this agent was 170.9822 seconds, while the best time obtained by the agent that was trained to reach moving targets was 187.1074 seconds, which is 16.1252 seconds slower (8.61%). It is worth noting that both of these results came from architectures that had 5 hidden network layers and 128 units per layer.

The fact that the best result for reaching a static target was achieved by an agent that was trained to reach static targets is not surprising, since it was trained for this exact specific scenario.

5.2 Reaching a moving target

For reaching a moving target, the test consists of an agent that has to touch 100 moving targets that appear in a predefined order and that have predefined paths that they will take. When a target appears it moves between predefined points using a predefined route, which is different for each of the 100 times the target appears. Once the agent reaches the target, the target appears at a different location, according to the predefined order,

and the agent is reset to the middle of the level, so that it will start from the same place each times it chases a new target. This is done to make the test course be identical for each tested agent.

Just like in the previous section (5.1), the test will be run for both agents that were trained to reach moving targets and agents that were trained to reach static targets, and afterwards the results will be compared.

5.2.1 Agent trained to reach moving targets

The first ones to be tested will be the agents trained to reach moving targets. Both agents that do not observe the target's movement direction and agents that do are tested. The obtained results are in Table 5.3 and Figure 5.3.

For the architecture with a single hidden layer, the agents that did not observe the target's direction performed better than those that did, with the fastest configuration being the one with 512 units per layer, achieving a time of 452.052 seconds. It is followed by the configuration with 128 units, that was slower by 30.4473 seconds (6.73%). The configuration with 256 units and that had the target's movement direction observation performed the worst out of all agents with a time of 688.592 seconds, over 3 minutes slower than the fastest configuration due to the fact that it took suboptimal paths when chasing the target and trying to avoid the objects in the environment in a weird manner.

Looking at the results for the architecture with 3 hidden layers, the agents that had the target's direction observation performed slightly better than the agents that did not have it. The best configuration is the one with 512 units per layer, which finished the course in 467.162 seconds. The configurations 128 and 256 units, with or without the extra observation, obtained similar results that were in the range of 484–491 seconds, which means that the difference between them was of at most $\sim 1\%$. The configuration with 512 units that did not have the extra observation performed much worse, achieving a time of 602.9445 seconds, due to fact that it tried to avoid the objects placed in the level at a much greater distance than it should have.

The architecture with 5 hidden layers had very different results between the agents that used the target's movement direction observation and those that did not, whith the ones that did, performing much better, on average being faster by ~ 58 seconds. The best configuration was the one with 128 units per layer, obtaining a time of 462.4213 seconds, with the other configurations lagging behind by over 38 seconds (at least 8.2% slower).

In summary, it seems that the extra observation improves the agent's performance only when multiple hidden layers are used, otherwise it hinders its performance. The surprise here is that the fastest configuration had only one network layer, compared to the results from Section 5.1, where the best results were obtained by agents that had a network with

Network Configuration	Observed target's direction	Time to complete (s)
1 layer, 128 units	No	482.4993
1 layer, 128 units	Yes	499.2063
1 layer, 256 units	No	495.7267
1 layer, 256 units	Yes	688.592
1 layer, 512 units	No	452.052
1 layer, 512 units	Yes	485.6294
3 layers, 128 units	No	484.7428
3 layers, 128 units	Yes	488.6792
3 layers, 256 units	No	491.0297
3 layers, 256 units	Yes	485.8719
3 layers, 512 units	No	602.9445
3 layers, 512 units	Yes	467.162
5 layers, 128 units	No	527.4669
5 layers, 128 units	Yes	462.4213
5 layers, 256 units	No	549.7428
5 layers, 256 units	Yes	500.7562
5 layers, 512 units	No	568.0677
5 layers, 512 units	Yes	506.722

Table 5.3: Test results for reaching a moving target using agent that was trained to reach moving targets

5 hidden layers. Here, the difference between the best achieved time, form the agent with 1 hidden layer, and the second best time, from the agent with 5 hidden layers, is of 10.3693 seconds, which means that the second best agent was slower by 2.29%.

5.2.2 Agent trained to reach static targets

Now, an agent that was trained to reach a static target will be tested to see how fast it will reach a moving target, and afterwards compare the results with the ones obtained by the agents that were trained to reach moving targets. The obtained results are in Table 5.3 and Figure 5.3.

Looking at the results for the architecture with a single hidden layer, it can be seen that the best result, out of all the agents tested, is obtained by the one with the configuration with 128 units per layer, and which finishes the test course in 427.4408 seconds. The configuration with 256 units did not finish the course since it was unable to be trained. The configuration with 512 units is slower by 11.3486 seconds (2.65% slower).

For the architecture with 3 hidden layers, the configurations with 128 units and 512 units obtain results that are very similar, the only difference being of 1.8478 seconds (0.41%), with the one with 512 units being the faster one. The configuration with 256 units is slightly slower, by 8.7071 seconds (1.94%).

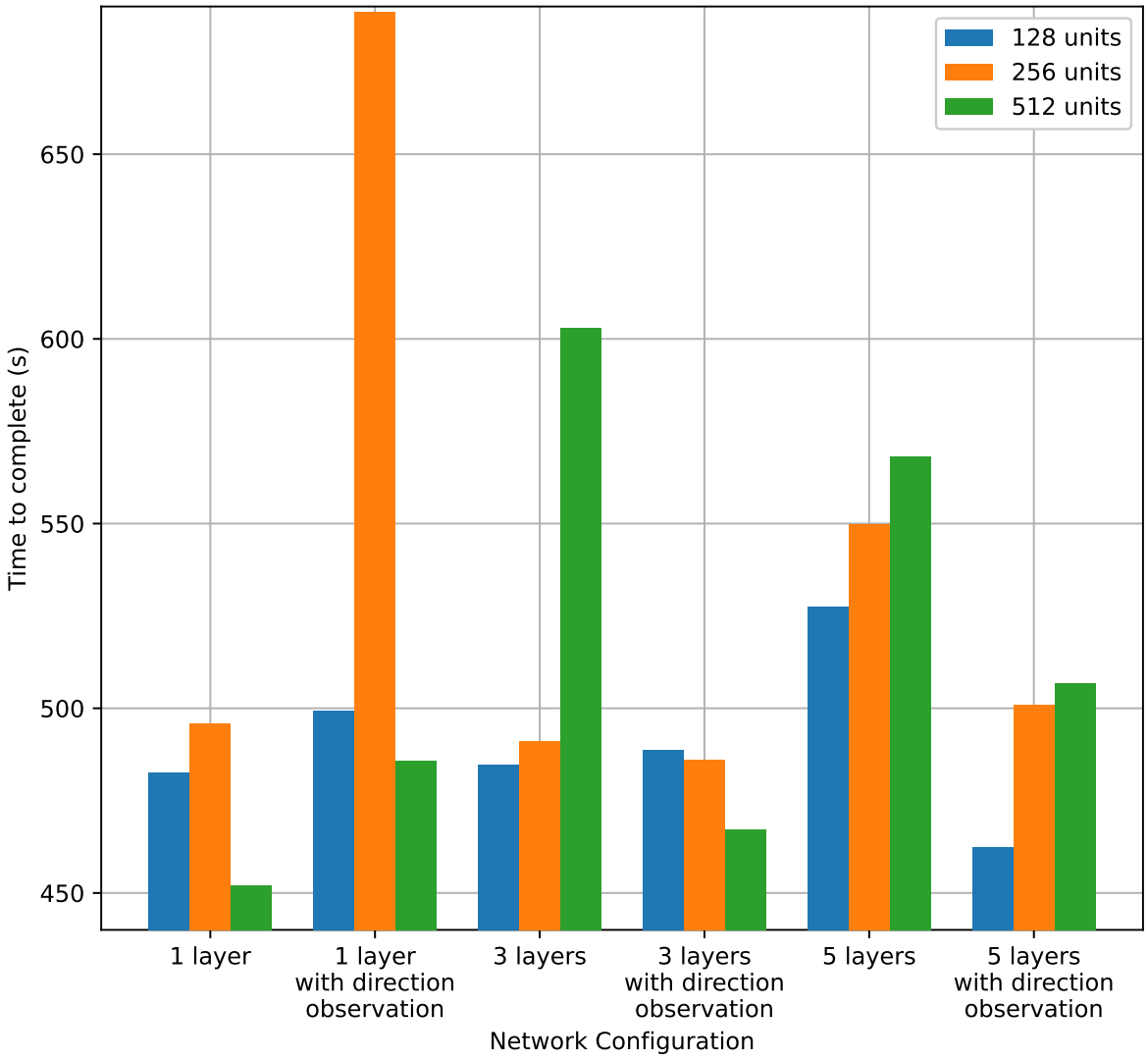

Figure 5.3: Test results for reaching a moving target using agent that was trained to reach moving targets

Network Configuration	Time to complete (s)
1 layer, 128 units	427.4408
1 layer, 256 units	DNF
1 layer, 512 units	438.7894
3 layers, 128 units	450.3436
3 layers, 256 units	457.2029
3 layers, 512 units	448.4958
5 layers, 128 units	429.3176
5 layers, 256 units	466.765
5 layers, 512 units	462.7256
7 layers, 128 units	465.1996
7 layers, 256 units	479.0871
7 layers, 512 units	546.0353

Table 5.4: Test results for reaching a moving target using agent that was trained to reach static targets

The architecture with 5 hidden layers has the second best performing agent, this being the one that has 128 units per layer and that finished the course in 429.3176 seconds. The other two agents managed to be slower, with the agent that had 256 units per layer being slower by 37.4474 seconds (8.72%), and the agent that had 512 units being slower by 33.408 seconds (7.78%).

The architecture with 7 layers, has the slowest results, with the best one from this architecture being the configuration with 128 units which managed to finish the test in 465.1996 seconds. The slowest configuration was the one that had 512 units per layer, finishing the course in 546.0353 seconds, 17.37% slower than the one with 128 units. Again, the agent tries to make very large turns to avoid other objects, which make him lose a lot of time.

In summary, the best agent was the one that had an architecture with 1 hidden layer and 128 units per layer, finishing the test in 427.4408 seconds, and being closely followed by the agent with the architecture with 5 hidden layers and 128 units, which was slower by 1.8768 seconds. This shows, that unlike the test in Section 5.1, having a smaller network leads to better results.

5.2.3 Results comparison

Comparing the obtained results, it can be seen that the agent who was trained to reach static targets, performs better than the agent trained to reach moving targets. The fastest agent trained to reach static targets finished the test in 427.4408 seconds, while the fastest agent trained to reach moving targets finished the test in 452.052 seconds, being slower by 24.6112 seconds (5.75%). This means that training an agent for a particular

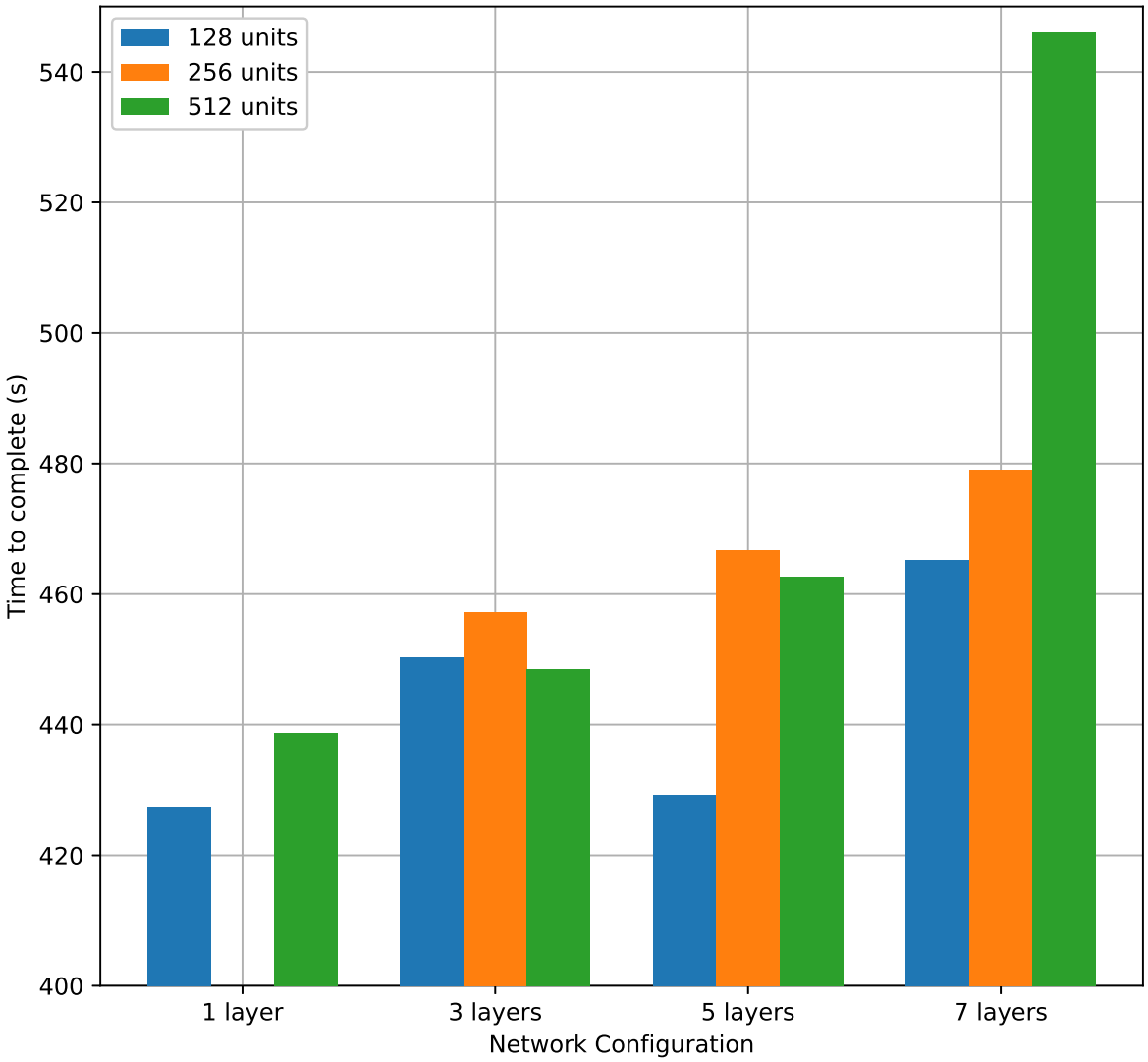

Figure 5.4: Test results for reaching a moving target using agent that was trained to reach static targets

task could make it perform a more advanced version of that task better than an agent trained specifically for that advanced task. It is also worth noting that both agents had a network with a single hidden layer.

5.3 Shooting a moving target

For shooting a moving target, the test is built just like it is described in Section 5.2, the only difference being that the agent is not supposed to reach the target, but to shoot it. The two types of trained agents will be tested, however, comparisons between them regarding how fast can they finish the test are not as important as before, since one of the approaches is supposed to be more aggressive (the *naive* one), and the other is supposed to be more defensive (the tactical one), with the result being that the more aggressive apporach should finish the test faster.

5.3.1 Agent trained in a *naive* manner

For the *naive* approach, the results can be seen in Table 5.5 and Figure 5.5. The architecture with a single hidden layer obtained the best results with the configuration that has 128 units per layer obtaining the best result, which is 499.8927 seconds. The configuration with 256 units was slower by 64.8033 seconds (12.96%), while the one with 512 units per layer was slower by 36.7426 seconds (7.35%).

For the architecture with 3 hidden layers, the configuration 128 units per layer was agian the fastest one, managing to finish the test in 590.7621 seconds, however it was slower than any of the configurations with a single hidden layer. The configuration with 256 units was slower by 46.3229 seconds (7.84%), while the one with 512 units was slower by 145.5966 seconds (24.64%) which is a significant increase.

Looking at the results for the architecture with 5 hidden layers, it can be observed that, again, the best configuration is the one with 128 units per layer, with a time of 606.5751 seconds. The configuration with 256 units was slower by 68.4348 seconds (11.28%), while the configuration with 512 units was slower by 211.0569 seconds (34.79%), which is a big increase in the time to finish the test.

The architecture with 7 layers, has the slowest results out of all the architectures, with the configuration with 128 units per layer obtaining a time of 837.6456 seconds, which is slower than any of the other architectures. The configuration with 256 units per layer is slower by 34.5383 seconds (4.12%), while the one with 512 units is slower by 76.8397 seconds (9.17%).

In summary, the fastest architectures were the ones that had only 128 units per layer, with the fastest one being the one with a single hidden layer and which finished the test in

Network Configuration	Time to complete (s)
1 layer, 128 units	499.8927
1 layer, 256 units	564.696
1 layer, 512 units	536.6353
3 layers, 128 units	590.7621
3 layers, 256 units	637.085
3 layers, 512 units	736.3587
5 layers, 128 units	606.5751
5 layers, 256 units	675.0099
5 layers, 512 units	817.632
7 layers, 128 units	837.6456
7 layers, 256 units	872.1839
7 layers, 512 units	914.4853

Table 5.5: Test results for shooting a moving target using agent that was trained using *naive* method

499.8927 seconds. The slowest architectures were the ones with 512 units per layer, with the slowest one being the one with 7 hidden layer and which finished the test in 914.4853 seconds, taking almost twice as much time as the fastest agent. It is worth noting that increasing the number of hidden layers and also the number of units per layer, is alost guaranteed to decrease the performance of the agent.

Just like in Section 4.3.3, there will be a comparison between the results obtained when changing the bullet observations. Each agent will have an architecture with a certain number of hidden layers and 128 units per layer. The observation combinations are the following:

- Bullet's trajectory and speed, and if the bullet was fired
- Bullet's speed and if the bullet was fired
- Only if the bullet was fired

The test results can be seen in Table 5.6 and Figure 5.6. For the architecture with a single hidden layer, the fastest result is obtained by the agent that observers only if the bullet was fired, finishing the test in 499.8927 seconds. In the second place is the agent that makes all three observations about the bullet, being slower by 28.301 seconds (5.66%), and in the third place is the agent that makes two observations about the bullet, and it's slower by 47.3781 seconds (9.47%).

For the architecture with 3 hidden layers, the fastest agent is the one that makes all thee observations regarding the bullet, and finishes the test in 529.0487 seconds, almost identical to the performance achieved by the same type of agent, but with a network with a single hidden layer. It is followed by the agent that make two observations regarding

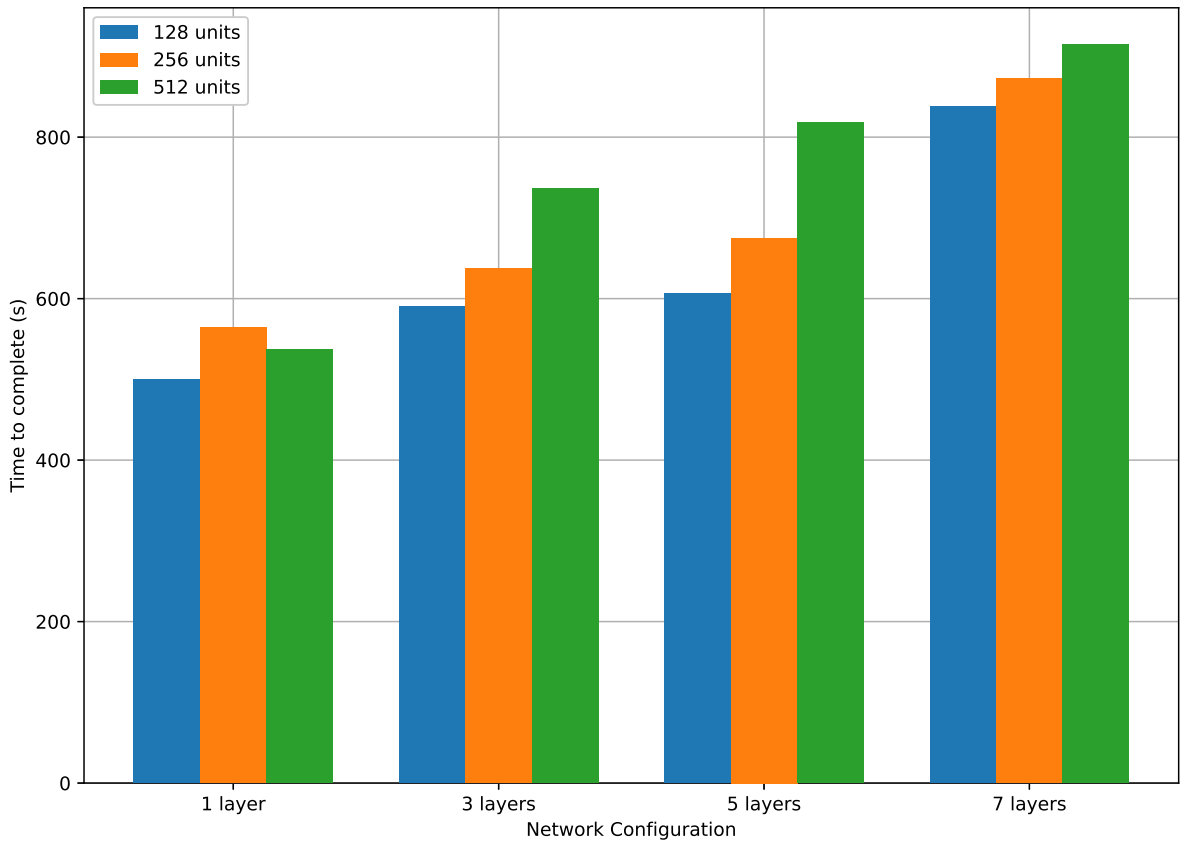

Figure 5.5: Test results for shooting a moving target using agent that was trained using *naive* method

Network Configuration	Bullet Observations	Time to complete (s)
1 layer, 128 units	Bullet's trajectory and speed	528.1937
1 layer, 128 units	Bullet's speed	547.2708
1 layer, 128 units	Only if bullet was fired	499.8927
3 layers, 128 units	Bullet's trajectory and speed	529.0487
3 layers, 128 units	Bullet's speed	557.6132
3 layers, 128 units	Only if bullet was fired	590.7621
5 layers, 128 units	Bullet's trajectory and speed	628.6615
5 layers, 128 units	Bullet's speed	688.095
5 layers, 128 units	Only if bullet was fired	606.5751
7 layers, 128 units	Bullet's trajectory and speed	687.3325
7 layers, 128 units	Bullet's speed	684.9317
7 layers, 128 units	Only if bullet was fired	837.6456

Table 5.6: Test results for shooting a moving target using agent that was trained using *naive* method and different bullet observation combinations

the bullet, being slower by 28.5645 seconds (5.39%), and finally, by the agent that only observes if the bullet is fired, being slower by 61.7134 seconds (11.66%).

The architecture with 5 hidden layers, again, has the the fastest agent being the one that observes only if the bullet was fired, and the agent finishes the test in 606.5751 seconds. It is closely followed by the agent that used all three bullet observations, being slower by 22.0864 seconds (3.64%), and then by the agent with two bullet observations, which was slower by 81.5199 seconds (13.43%).

Looking at the results for the architecture with 7 hidden layers, the agents that had 2 or 3 observations regarding the bullet performed virtually identically, having a difference of only 2.4008 seconds, while the agent that observed only if the bullet was fired was slower by 152.7139 seconds (22.29%).

In summary, it can be seen that increasing the number of layers also makes the agent perfrom more poorly, with the agents that had a network architecture witha single hidden layer performing the best. It can aslo be observed that when the number of layers is lower, it is best not to include additional observations regarding the bullet and only keep the one that tells if the bullet was fired.

5.3.2 Agent trained in a tactical manner

Test results for the agent that was trained to shoot a target in a more tactical manner can be seen in Table 5.7 and Figure 5.7. For the architecture with a single hidden layer, the best results are obtained by the agents with a configuration of 256 and 512 units per layer, with the oen with 256 units being the fastest, obtaining a time of 483.9203

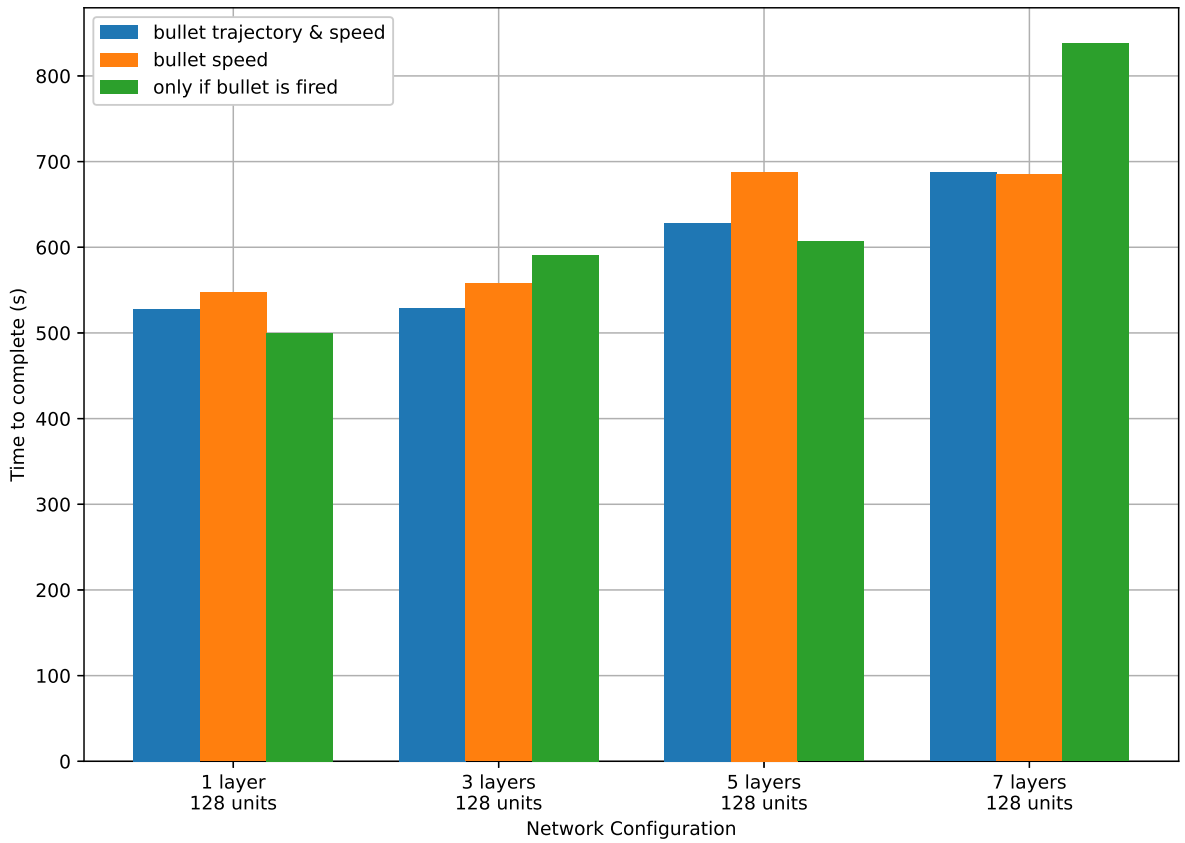

Figure 5.6: Test results for shooting a moving target using agent that was trained using *naive* method and different bullet observation combinations

seconds. The configuration with 512 units was slower by 10.6483 seconds (2.2%), while the configuration wih 128 units was slower by 121.107 seconds (25.02%).

The architecture with 3 hidden layers has the fastest configuration being the one with 128 units per layer, achieveing a time of 590.4073 seconds. It is followed by the configuration with 256 units being slower by 89.024 seconds (15.07%), and then by the configuration with 512 units being slower by 171.6053 seconds (29.06%).

For the architecture with 5 hidden layers, the fastest configuration is again the one with 128 units per layer which manages to finish the course in 650.9454 seconds. It is followed by the configuration with 256 units which is slower by 50.8067 seconds (7.8%). The configuration with 512 units falls massively behind these two other configurations, being slower than the one with 128 units by 877.8456 seconds (134.85%), taking more than twice as much time to finish the test. This is due to the fact that the agent was not able to learn how to accurately shoot at the target, missing most of its shots.

The architecture with 7 hidden layers, has the best configuration being the one with 128 units per layer, which manages to achieve a time of 687.3971 seconds. The configuration with 256 units takes the second place being slower by 250.0306 seconds (36.37%). Just like the architecture with 5 hidden layers, here also, the configuration with 512 units is the slowest and greatly falls behind the other two, being slower by 1513.4279 (220.16%) seconds than the configuration with 128 units, being more than 3 times slower than it. This also is due to the fact that the agent did not learn how to shoot properly, missing most of its shots.

In conclusion, in this test where it is measured how fast an agent can shoot a certain number of moving targets, even with the added tactical behaviour, the agents that had the network with the elast amount of hidden layers performed the best. It is worth noting that adding more units per layer for the architecture with a single hidden layer improved the agent's performance, while adding more units per layer with a network with nore hidden layers, severly impacted the agent's performance in a negative way. This could be due to the fact that the agents with a larger network needed more training steps due to the more complicated behaviour needed to be learned. Another thing worth noting, is that the agents did not seem to fully learn the added tactical behaviour of keeping a distance from the enemy and flanking it, instead opting to act more like the agents that were trained to shoot a target in a more *naive* way. This could be due to the added punishment and reward values that were added, being not optimal.

5.4 Fighting against an AI controlled enemy

The final test is for the agent to fight against an AI controlled enemy and see if it can defeat it, and how well it can do that. The test consists of a 1v1 fight against an

Network Configuration	Time to complete (s)
1 layer, 128 units	605.0273
1 layer, 256 units	483.9203
1 layer, 512 units	494.5686
3 layers, 128 units	590.4073
3 layers, 256 units	679.4313
3 layers, 512 units	762.0126
5 layers, 128 units	650.9454
5 layers, 256 units	701.7521
5 layers, 512 units	1528.791
7 layers, 128 units	687.3971
7 layers, 256 units	937.4277
7 layers, 512 units	2200.825

Table 5.7: Test results for shooting a moving target using agent that was trained using tactical method

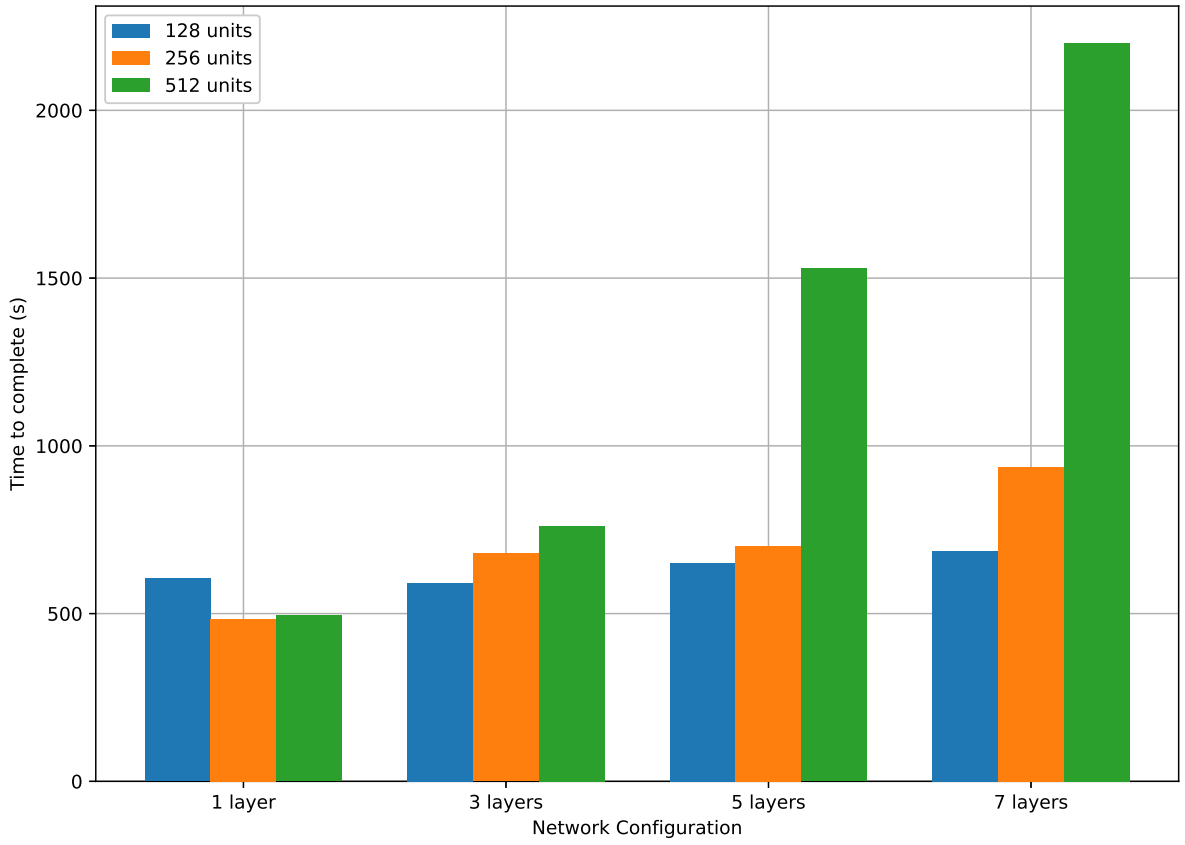

Figure 5.7: Test results for shooting a moving target using agent that was trained using tactical method

enemy tank on the same map where the agent was trained. The agent battles the enemy tank for 20 rounds. To win a round, a tank must hit it's enemy roughly 3 times to defeat it. Each round, the tanks start in randomized positions. The enemy AI is implemented using behavior trees and genetic algorithms, as described in [12]. To measure the agent's performance, several metrics will be used, and these are:

- how fast the agent finishes the test
- how many of the 20 rounds the agent won
- how many times the agent scored a hit on the enemy
- how many times the agent was hit by the enemy

Just like in the previous section (Section 5.3), both types of agents that can shoot a target will be tested: the agents trained in a *naive* way, and the ones trained in a more tactical way.

5.4.1 Agent trained in a *naive* manner

The first type of agent that is tested is the one trained in a naive way, which means its only goal is to shoot the target no matter what. This should lead to a more aggressive and reckless approach. Unlike the previous section, there will be no comparisons between different observations for the bullet, since it was seen that the best observation combination regarding the bullet was the one where it is observed only if the bullet is fired.

The test results can be seen in Table 5.8 and in Figures 5.8, 5.9, 5.10 and 5.11.

Looking at the results, it can be seen that the best network configuration is the one with 1 hidden layer and with 128 units per layer, since it manages to finish the test in a short time, similar to the other agents with 1 and 3 hidden layers, manages to have the maximum number of rounds won, 18, an manages to get hit the least out of all agents. Where it does not manage to win is the number of times it manages to hit the enemy, falling behind a few agents, but still managing to be ahead of most other agents. Despite not being trained specifically to have an advanced behaviour, it sometimes tries to run from the enemy so that it can flank it. The other agents that have a single hidden layer perform similar to the one that has 128 units per layer, however they managed to be hit by at least 7 more times by the enemy.

For the agents with 3 hidden layers, the best performing one is also the one that has 128 units per layer, achieving results similar to the agent that has a network with a single hidden layer and 128 units per layer, being slightly worse than it due to the fact that it got hit more times by the enemy. The other 2 agents managed to finish the test with

Network Configuration	**Time to complete** (s)	**Rounds won**	**Successful hits**	**Hits received**
1 layer, 128 units	342.9979	18	58	26
1 layer, 256 units	342.3145	15	56	39
1 layer, 512 units	336.0863	18	60	33
3 layers, 128 units	327.6781	18	59	29
3 layers, 256 units	315.368	14	56	38
3 layers, 512 units	340.9718	14	53	32
5 layers, 128 units	363.3622	14	51	37
5 layers, 256 units	516.4233	13	52	40
5 layers, 512 units	428.2677	14	57	35
7 layers, 128 units	458.6925	16	62	34
7 layers, 256 units	425.0882	15	56	38
7 layers, 512 units	375.2268	15	54	32

Table 5.8: Test results for fighting an AI controlled enemy using agent that was trained using *naive* method

similar times, but won less rounds, did not hit the enemy as many times, and got hit by the enemy more.

The agents with 5 hidden layers manage to be the ones who finish the test the slowest, with the worst offender being the agent with 256 units per layer due to the fact that it keeps running away from the enemy. These agents are also the ones that won the least amount of rounds, possibly due to the fact that they were the ones to be hit by the enemy the most out of all agents.

Finally, the agents with 7 layers also have big completion times compared to the agents with 1 or 3 hidden layers, because they tend to run around the map. These agents with 7 hidden layers are also receiving more hits than the ones with 1 or 3 layers, however they manage to win more rounds than the agents with 5 layers. The agent with 128 units per layer manages to score the most hits on the enemy out of all the other agents.

In conclusion, it can be seen that using agents that have smaller network architectures, especially regarding the number of units per layer, perform better than the other ones. The approach taken by the agents in this test was one where they would just charge at the enemy and stay in front of it while shooting at every chance. A downside of this behaviour is that it's very predictable nad might not be as fun for a human player to go up against since it will, almost always, act the same way. Another problem would be that the agents did not have any *trigger discipline*, meaning that they would shoot bullets even if they were not aimed at the enemy.

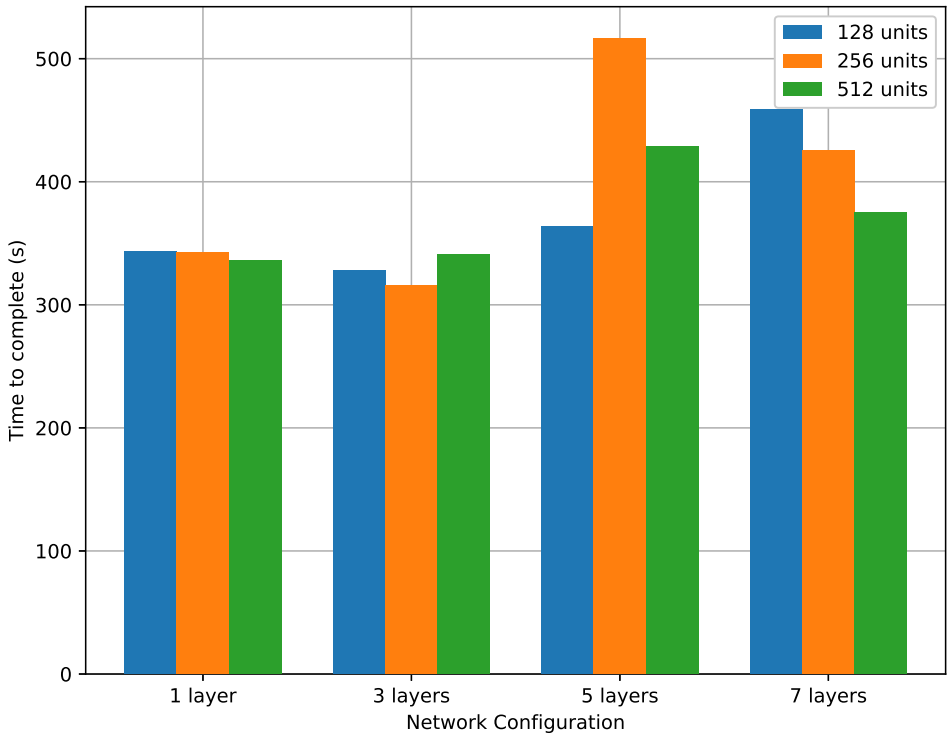

Figure 5.8: Test results for fighting an AI controlled enemy representing the time to finish the test

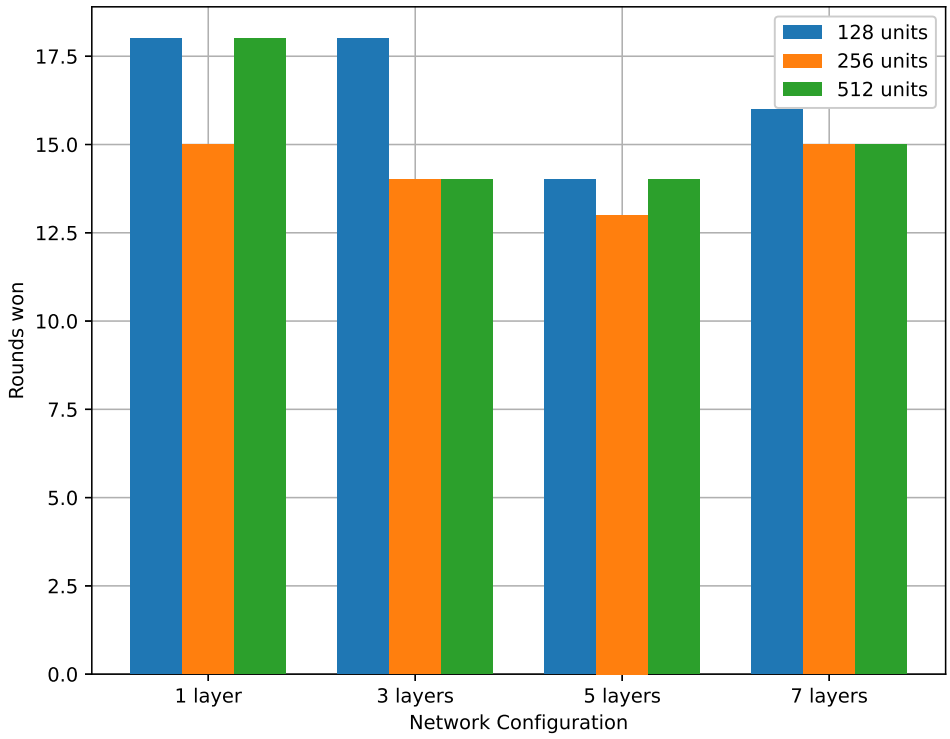

Figure 5.9: Test results for fighting an AI controlled enemy representing the number of rounds the agent won

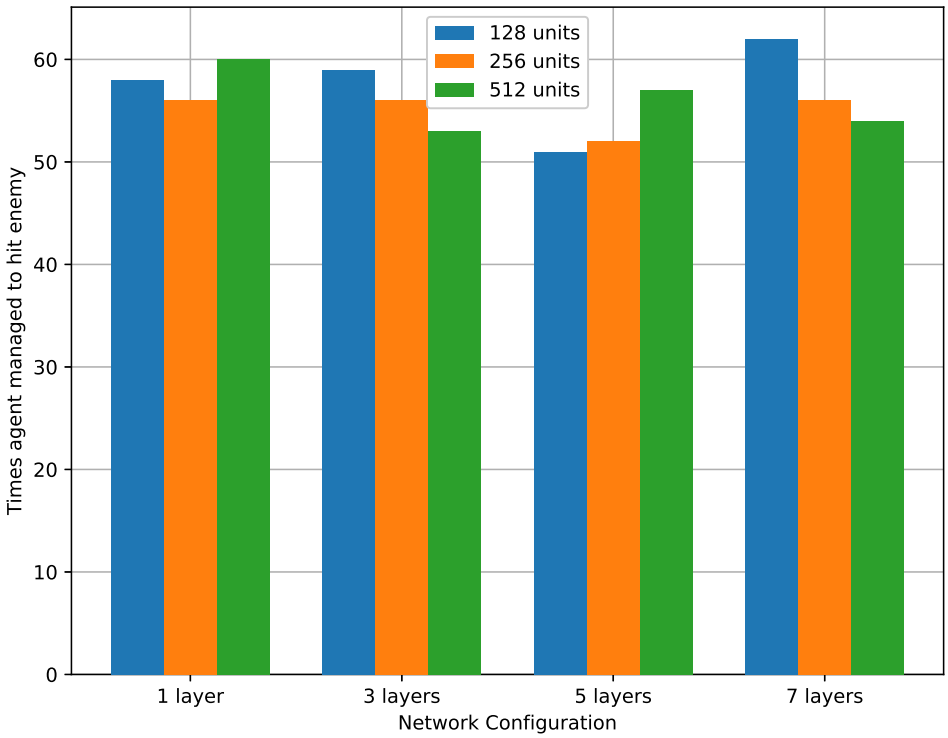

Figure 5.10: Test results for fighting an AI controlled enemy representing the number of times the agent hit the enemy

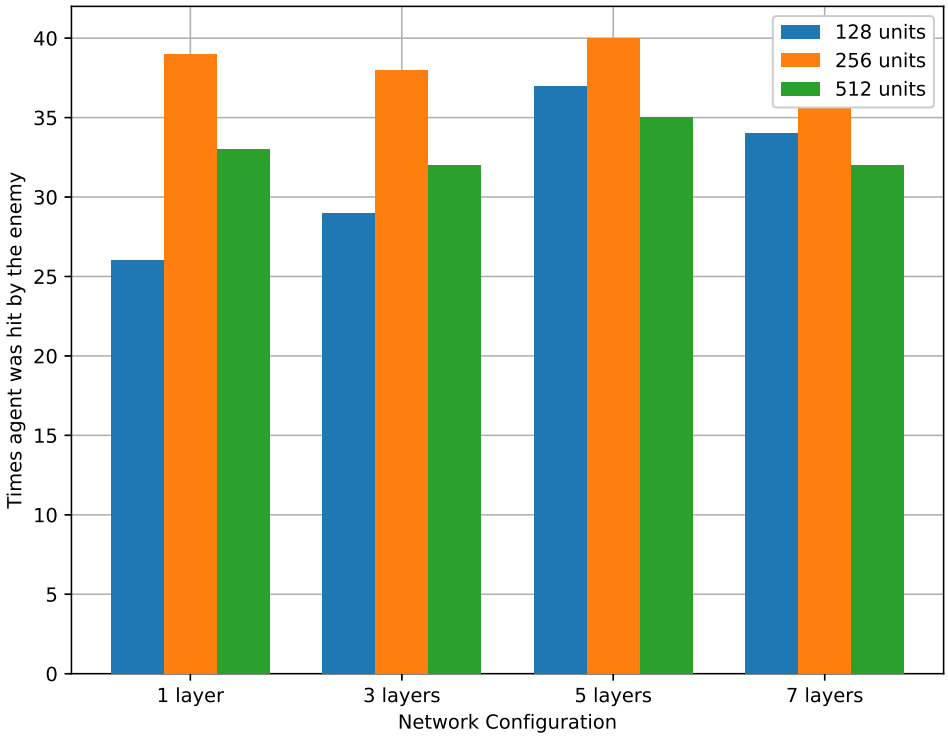

Figure 5.11: Test results for fighting an AI controlled enemy representing the number of times the agent got hit by the enemy

5.4.2 Agent trained in a tactical manner

The second type of agent that is tested is the one that was trained in a more tactical way, which should keep a distance from the enemy, get out of the enemy's crosshairs and try to flank it.

The test results can be seen in Table 5.9 and in Figures 5.12, 5.13, 5.14 and 5.15.

Looking at the results, the agent that has a network with a single hidden layer and 256 units per layers, is the best performing one out of all the other agents, being the one to finish the test in the least amount of time, winning the most rounds, being seconds at the number of times that it manages to hit the enemy, and being the agent that got hit the least amount of times. When fighting the enemy tank, it has the tendency to try and flank the enemy, just like it was trained to do, however, most of the time it tries to rush the enemy directly. The agent with 128 units per layer performs slightly worse than the one with 256 units, while not exhibiting any of the tactical behavior that it should. The agent with 512 units per layer performed visibly worse than the one with 256 units, losing half of the rounds and taking more time to finish the test. However, it did try to move away from the enemy to try and flank it.

For the agents with a network with 3 hidden layers, it seems that the best performing one is the agent that has 128 units per layer, managing to beat the other two in each category. Still, just like the one that has a single hidden layer, it did not exhibit any tactical behavior, instead opting to rush the enemy, just like the agents that were trained in a *naive* way. The other two agents did manage to exhibit the correct behaviour, but it was done rarely and quite poorly.

The agents with 5 hidden layer, have the best performing one be the one with 256 units per layer, managing to obtain the most victories, hitting the enemy the most times, and being hit the least out of the three. It did finish the test more slowly, however, this could be due to the fact that it managed to correctly apply what it was trained to do, and performed tactical retreats so that it could flank the enemy. Just like in Section 5.3.2, the agent with 512 units performed the worst out of the three, losing more than half the rounds, missing its shots, and getting hit by the enemy. The behaviour exhibited was one of confusion, with periods of time where it would move aimlessly around the level, giving the enemy the opportunity to shoot it.

Lastly, the agents with 7 hidden layers had the worst performing agent out of all the other agents tested, taking the most time to finish the test, winning the elast amount of rounds, constantly missing the target, and getting hit the most. Again, its behaviour was a confused one, choosing to move at random around the level. The other 2 agents performed better, with the agent with 256 units winning more rounds and scoring more hits than the one with 128 units, but also getting hit more.

Network Configuration	**Time to complete** (s)	**Rounds won**	**Successful hits**	**Hits received**
1 layer, 128 units	354.3039	16	54	36
1 layer, 256 units	286.458	17	59	31
1 layer, 512 units	393.5259	10	47	45
3 layers, 128 units	381.8488	17	60	33
3 layers, 256 units	368.9066	15	55	38
3 layers, 512 units	598.802	15	56	37
5 layers, 128 units	345.1217	14	55	38
5 layers, 256 units	448.2226	17	55	36
5 layers, 512 units	499.2554	9	42	43
7 layers, 128 units	359.0904	13	54	34
7 layers, 256 units	424.311	16	56	37
7 layers, 512 units	719.5594	7	30	50

Table 5.9: Test results for fighting an AI controlled enemy using agent that was trained using tactical method

In sumamry, just like the results obtained in Section 5.3.2, the agents did not manage to fully manifest the tactical behaviour that they were trained, instead opting to mostly charge the enemy tank at shoot at it from point blank range, with the occasional retreat to try and flank it. Again, the best performing agents were the ones that had a smaller network architecture, especially those that had 256 units per layer. The agents that had more hidden layer and 512 units per layer performed the worst, perhaps due to the fact that they were not able to fully learn their objective, requiring a longer period of training. However, the behavior exhibited by these agents could still be used against a player if they would have to

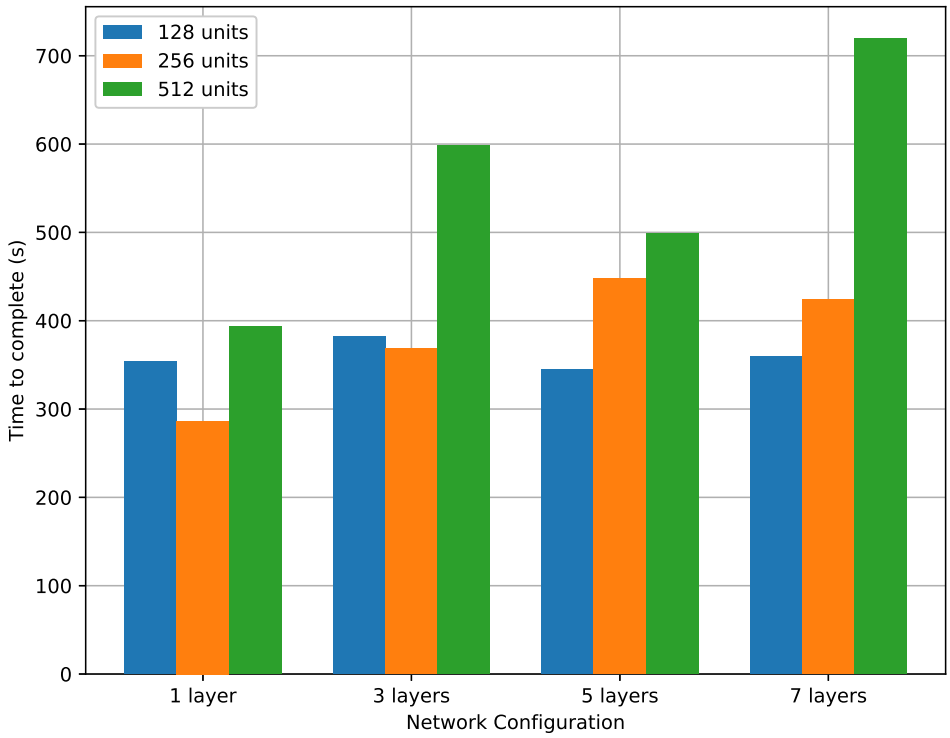

Figure 5.12: Test results for fighting an AI controlled enemy representing the time to finish the test

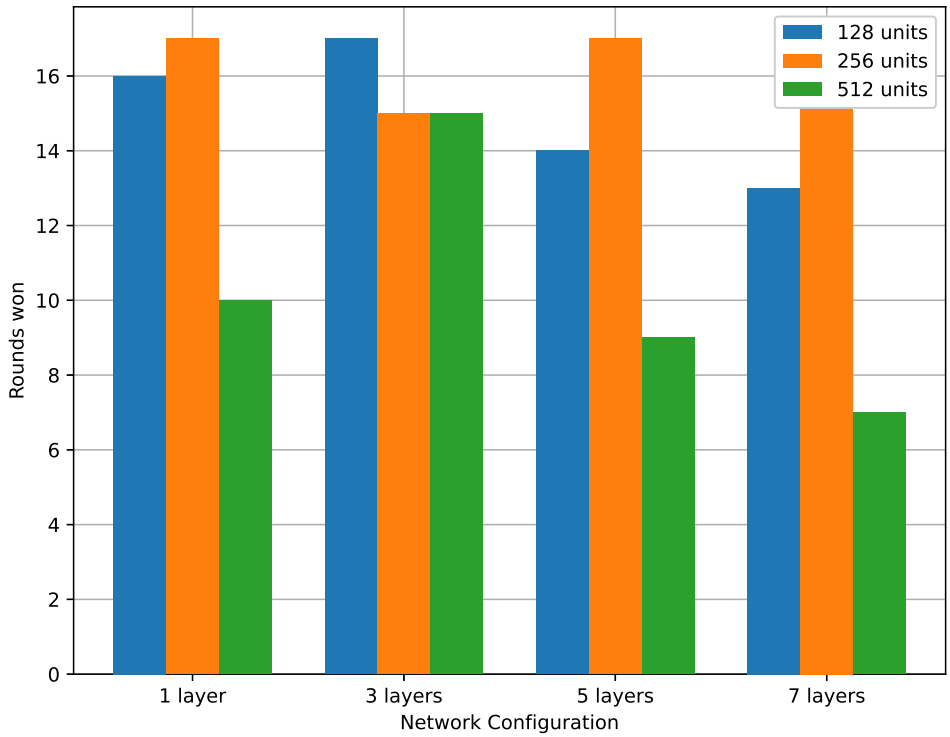

Figure 5.13: Test results for fighting an AI controlled enemy representing the number of rounds the agent won

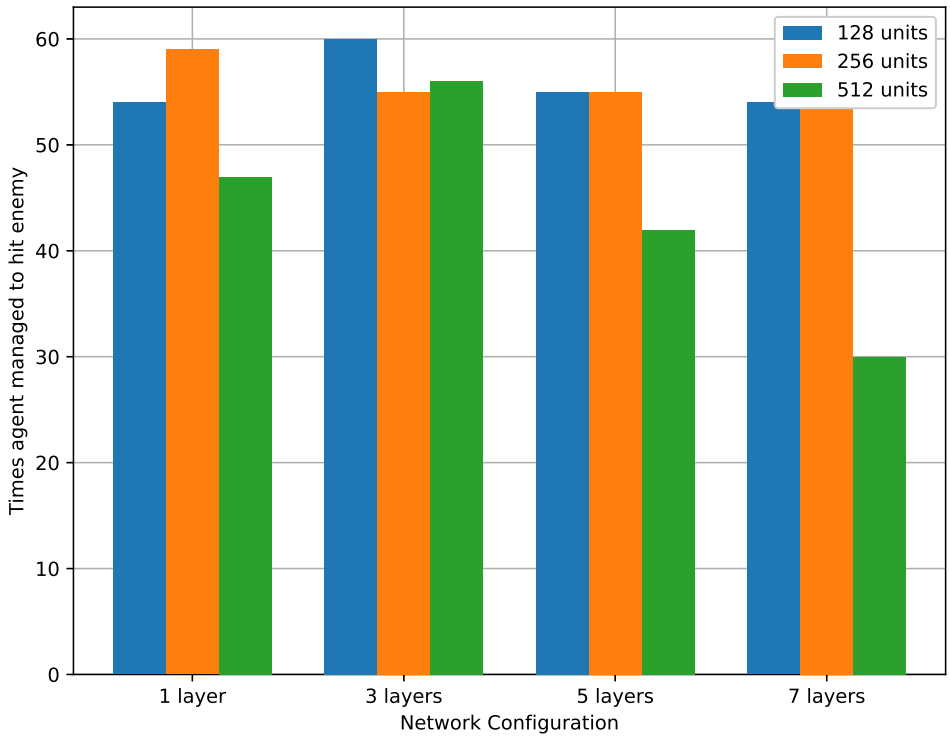

Figure 5.14: Test results for fighting an AI controlled enemy representing the number of times the agent hit the enemy

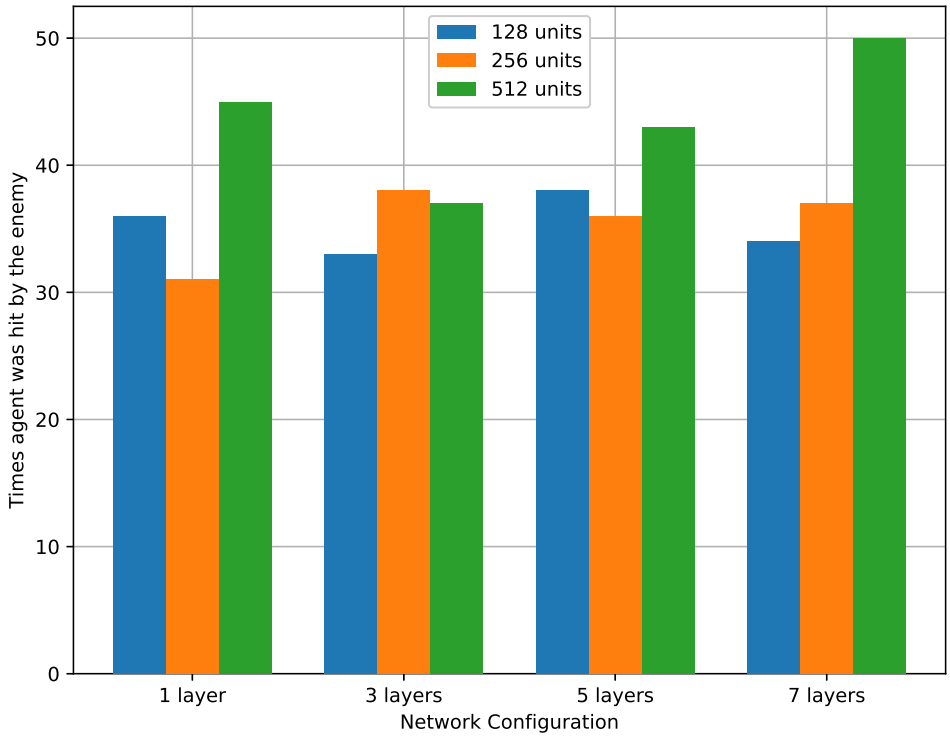

Figure 5.15: Test results for fighting an AI controlled enemy representing the number of times the agent got hit by the enemy

Chapter 6

Conclusions

In this paper, several behaviours of an AI that could be used in video games were implemented and tested using reinforcement learning, specifically PPO algorithms. The behaviours ranged form more simple ones, such as reaching given coordinates to more complex ones such as fighting an enemy and using a smart approach to the fight. To achive this, several rewards and observations were implemented so that the agent could learn the desired behaviour. The agent training was done using multiple network configurations, with different number of hidden layers and number of units per layer, in some cases testing different kinds of observations, by adding or removing specific ones. For the basic tasks, such as reaching a given point, or simply chasing a moving target, the experiments were successful, managing to correctly learn these behaviours and actually providing an alternative to more classical ways of moving a non-player character in the game's world. It has been shown that for these simple tasks, smaller networks perform much better than the larger ones and have several advantages over them, such as faster training times and less resources used during gameplay. The more complex behaviours of fighting an enemy did not manage to be successfully implemented like the previous ones so that they could be used in a real game, with the learned behaviour being suboptimal, not consisting of an interesting experience for a human player. This could be due to a number of factors such as reward and penalty values not being balanced correctly, agents trained for a shorter amount of time than what would be necessary to successfully learn a more complex behaviour, and not training the agent to fight a tank, but a target that is unable to fight back.

For further work on this topic, to improve the obtained results for the behaviour of fighting an enemy, new observations and rewards should be implemented or existing ones should have their values modified, the agents should be trained for longer periods of time and by fighting against AI controlled enemies, and different hyperparameter values should be tried.

Bibliography

[1] Eloi Alonso, Maxim Peter, David Goumard, and Joshua Romoff. *Deep Reinforcement Learning for Navigation in AAA Video Games*. 2020. arXiv: 2011.04764 [cs.LG].

[2] Richard Bellman. "A Markovian Decision Process." In: *Indiana Univ. Math. J.* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[3] Ronen Eldan and Ohad Shamir. *The Power of Depth for Feedforward Neural Networks*. 2016. arXiv: 1512.03965 [cs.LG].

[4] Vincent Franç ois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. "An Introduction to Deep Reinforcement Learning." In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. DOI: 10.1561/2200000071. URL: https://doi.org/10.1561%2F2200000071.

[5] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. *Unity: A General Platform for Intelligent Agents*. 2020. arXiv: 1809.02627 [cs.LG].

[6] Ruo-Ze Liu, Zhen-Jia Pang, Zhou-Yu Meng, Wenhai Wang, Yang Yu, and Tong Lu. *On Efficient Reinforcement Learning for Full-length Game of StarCraft II*. 2022. arXiv: 2209.11553 [cs.LG].

[7] *ML Agents Unity API documentation*. https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/. Accessed: 2023-08-29.

[8] *mlagents*. https://pypi.org/project/mlagents/. Accessed: 2023-08-29.

[9] Vasileios Moschopoulos, Pantelis Kyriakidis, Aristotelis Lazaridis, and Ioannis Vlahavas. *Lucy-SKG: Learning to Play Rocket League Efficiently Using Deep Reinforcement Learning*. 2023. arXiv: 2305.15801 [cs.LG].

[10] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. *Gotta Learn Fast: A New Benchmark for Generalization in RL*. 2018. arXiv: 1804.03720 [cs.LG].

[11] Inseok Oh, Seungeun Rho, Sangbin Moon, Seongho Son, Hyoil Lee, and Jinyun Chung. *Creating Pro-Level AI for a Real-Time Fighting Game Using Deep Reinforcement Learning*. 2020. arXiv: 1904.03821 [cs.AI].

[12] Ciprian Paduraru and Miruna Paduraru. *Automatic difficulty management and testing in games using a framework based on behavior trees and genetic algorithms*. 2019. arXiv: 1909.04368 [cs.AI].

[13] Tim Pearce and Jun Zhu. *Counter-Strike Deathmatch with Large-Scale Behavioural Cloning*. 2021. arXiv: 2104.04258 [cs.AI].

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[15] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. *A Survey of Deep Reinforcement Learning in Video Games*. 2019. arXiv: 1912.10944 [cs.MA].

[16] Yaodong Yang and Jun Wang. *An Overview of Multi-Agent Reinforcement Learning from Game Theoretical Perspective*. 2021. arXiv: 2011.00583 [cs.MA].