20113536

Supervisor: Dr. Martin Flintham

Module Code: COMP3003

2022/04

Single Agent in 3D Imperfect Information Video Game with Deep Reinforcement Learning in Unity

Submitted April 2022, in partial fulfilment of
the conditions for the award of the degree **BSc (Hons) Computer Science with Artificial Intelligence**.

20113536
School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature: ***ESG***

Date: 25 / 04 / 2022

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive. *

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr. Martin Flintham for his extensive and comprehensive knowledge, helpful suggestions, uplifting advice, and patience in supervising me to complete the development of this project. I would also like to acknowledge the encouragement that I have received from my friends and family over the course of this project.

ABSTRACT

This paper presents the development of a Deep Reinforcement Learning(DRL) agent and a 3-dimensional imperfect information game environment designed by myself for the single agent to interact with. The DRL agent is trained with a Reinforcement Learning(RL) algorithm known as Proximal Policy Optimization(PPO) in a recurrent neural network known as Long Short-Term Memory(LSTM) where Intrinsic Curiosity Module(ICM) is implemented to help the agent to cope with the sparse rewards environment. The agent is trained with Curriculum Learning(CL) and Transfer Learning(TL), both of which are popular deep learning approaches along with environment parameter randomization to improve agent robustness. Robustness refers to the generalising capability or generality of the agent in a complex game environment. The model with best training results is used to control the agent for testing. Robustness is then quantitatively evaluated based on testing results, the agent behaviour is qualitatively studied based on human observations and discussed at length. This paper then concludes future work approaches for possible better testing results as well as a reflection on project aim, project workflow management, biggest challenge encountered and the lessons learned throughout this project.

TABLE OF CONTENTS

List of Abbreviations

3D	three-dimensional
2D	two-dimensional
AI	Artificial Intelligence
RL	Reinforcement Learning
DL	Deep Learning
DRL	Deep Reinforcement Learning
ML	Machine Learning
ANN	Artificial Neural Network
PPO	Proximal Policy Optimisation
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
ICM	Intrinsic Curiosity Module
TRPO	Trust Region Policy Optimization
NaN	not a number
CEC	Constant Error Carrousel
POMDP	Partially Observable Markov Decision Process
CL	Curriculum Learning
TL	Transfer Learning
DQN	Deep Q-Network
SL	Supervised Learning
APV-MCTS	Asynchronous Policy and Value Monte Carlo Tree Search
SARSA	State-Action-Reward-State-Action
FPS	First-person Shooter

List of Figures

List of Tables

1. Introduction

For the past few decades, there have been many successful attempts at creating game Artificial Intelligence(AI) that can play games without human intervention at all. Successful massive projects done by OpenAI and DeepMind in defeating top human players with trained AI agents are done through Reinforcement Learning(RL) and Deep Learning(DL). With these, an almost superhuman level of non-human players can be created. This project seeked to make use of Unity Machine Learning Agents [1], a powerful framework that allows users to train and embed RL agents using state-of-the- art deep learning technologies in virtual environments made with Unity. A Deep Reinforce -ment Learning(DRL) agent was created to play a game in which the agent was required to win the game by completing a complex task.

1.1 Deep Reinforcement Learning(DRL)

DRL is a mixture of DL and RL, DL is a subset of Machine Learning(ML) where ML algorithms are used to process data in a deep Artificial Neural Network(ANN). A deep ANN is composed of multiple processing layers that can process huge amounts of raw digital input data. Processing layers, usually known as hidden layers in a deep ANN, consist of interconnected nodes with each layer deriving from the previous layer to further optimise its decision making process. Generally, a simple ANN is known to only have a single hidden layer whereas a deep ANN has two or more hidden layers. The process of computing raw input data from the input layer through the hidden layers and eventually reaching the output layer is known as forward propagation. The input layer is where the network loads batches of input whereas the output layer is where the final predictions are produced. In the case of this project, it is where the decided actions for an agent are made, and the decision making process is done through the hidden layers. After a forward propagation, a process known as backpropagation is done to compute errors made in the forward propagation and the errors are amended by tuning the weights and biases of the network from the last hidden layer from the back to the first hidden layer at the front. Forward propagation and backpropagation are alternated repetitively for the network to eventually learn to produce more accurate predictions [2].

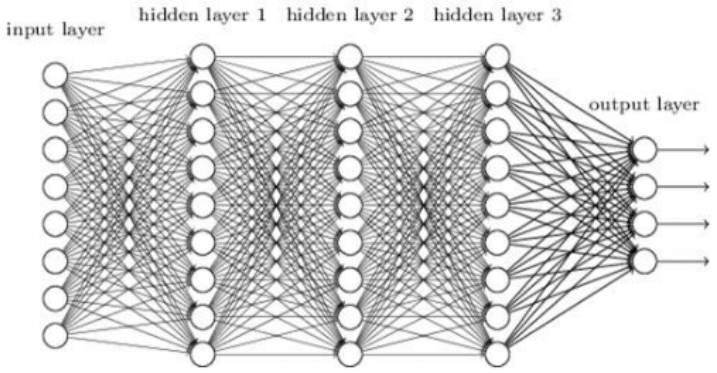

Figure 1: A typical network architecture of a Deep ANN for DL [3]

RL is one of the main 3 categories in ML. It is basically a set of methods of "teaching" a machine(agent) to learn sequences of actions in an environment for which the agent has to learn a task. In RL, the agent is able to learn from the consequences of its actio- ns in an environment similar to how humans learn from experience [4]. Fundame- ntally, an RL agent learns by utilising the basic working concept of trial-and-error to eventually figure out the optimal solutions to certain problems. An agent makes a decision and takes an action at each timestep in an environment then it will receive a state and reward or penalty(negative rewards).

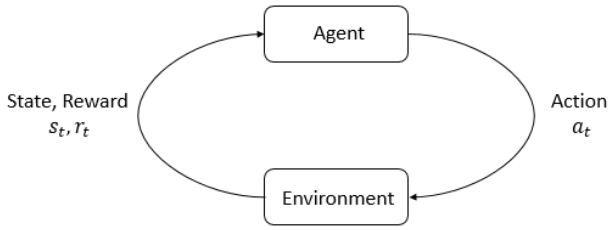

Figure 2: A basic diagram of a typical agent-environment interaction loop [5]

In RL, an ***agent*** is the AI machine that can take actions and an agent's ***environment*** is the world that it resides in and interacts with. At every instance of interaction, an agent is able to obtain the ***state*** of its ***environment*** and it can decide to make an ***action*** depending on what it observes. The ***environment*** could have a change in its state independently or it could be due to the agent's ***action*** [5]. After taking an ***action***, the agent will obtain a ***reward*** signal from its ***environment***, a vital element in RL that defines the "goodness" of the current ***state*** of the ***environment***.

It learns through the ***reward*** system in which the agent gets rewarded for ***actions*** that result in task completion or gets penalised for ***actions*** that are meaningless. The agent should find out on its own how to complete the task based on its ***observations*** and ***reward*** system as well as the ***actions*** it is allowed to make in the ***environment*** [6]. The goal here is for it to learn to maximise its rewards by optimising a ***policy***, which in RL, a ***policy*** can be thought of as a rule that an agent depends on when deciding to take any actions [5] [7]. Essentially, a ***policy*** is optimised based on the ***states*** and ***rewards*** received as the results of ***actions*** performed in the ***environment*** [8]. Thus it will eventually learn to piece together a few series of ***actions*** that can result in task completion and ***rewards*** acquisition.

2. Brief RL History & Motivation

RL came into the light as a main field of research and studies in the early 1980s when some of the preliminary works in artificial intelligence involved the idea of learning by trial-and-error that started in the psychology of animal learning, especially humans. Arguably, the first to explore and study the method of trial-and-error learning was Edward Thorndike during the 1910s who called it the "Law of Effect" which describes that when a response is followed by satisfaction in a situation, that response becomes more likely to happen again if that situation were to be repeated, and a response that is followed by discomfort becomes more unlikely to occur [9] [10].

Over the years, several researchers had explored trial-and-error learning, but research in this topic became uncommon in the 1960s and 1970s. However, the one that contributed the most in reintroducing trial-and-error in RL for AI was Harry Klopf during the early 1980s. Most learning researchers who at that time were focusing almost entirely on supervised learning overlooked an element that Klopf emphasised, which was the tendency to achieve some kind of positive results in an environment, to control the environment towards desired ends and away from undesired ends, in other words ,the gratifying aspects of behaviour [9].

Since then, there have been many breakthroughs and advances in the field of RL. For instance, OpenAI collaborated with DeepMind's safety team to train virtual robots to learn from human instructions to effectively solve complex RL problems without access to reward functions while providing feedback on less than one percent of an agent's interaction with its environment [11]. Besides, DeepMind successfully created the first AI agent that is able to learn to excel at a diverse array of challenging tasks. They successfully created an DRL agent that was able to surpass all previous algorithms by achieving a skill level that is comparable to that of a professional human game tester across a set of multiple games [12]. In the world of robotics, RL have helped engineers and researchers to successfully develop highly automated and intelligent robots to solve many real-world problems, reducing the need for manual human workers in exhaustive and dangerous tasks [13].

Edging into DRL applications that have a more direct inspiration for this project, the contributions and milestones that AI research laboratories such as OpenAI and DeepMind have achieved over the past decade in the gaming industry. For example, AlphaStar by DeepMind that became the first AI to reach top league in StarCraft II without any game limitations [14], OpenAI five that successfully defeated Dota 2 human world champions in 2019 [15], AlphaGo which according to DeepMind, is the world's first AI to defeat a professional human Go Player, the first to defeat a Go world champion and is arguably the strongest Go player to date [16]. There are also a number of brilliant individuals who have done similar work in training game-playing agents with RL. One such example would be a group of students who did a project in training an agent to solve a maze designed by themselves [17]. With the knowledge of the brief history in RL and its current advances, they served as the main motivation for me to explore the capacity of DRL to gain a firsthand experience in developing, training, experimenting and studying the performance and behaviours of an agent in a virtual game environment designed by myself.

3. Aim

The aim of this project was to develop a DRL agent to complete a task in a 3D video game with imperfect information and to study agent robustness in terms of generality as well as to study its behaviour.

4.Objectives

1. Created a game environment in Unity for which an agent can interact with.
    - An action-adventure game where the agent's task was to seek and collect treasures scattered around in a maze and also kill all the zombies.

2. Designed and developed a DRL agent that can perceive its environment and make a decision to take an action that can affect the state of its environment for which the agent can receive rewards or penalties.
    - DRL agent was developed using Unity ML-Agent where observation spaces, action spaces and reward or penalty signals were designed and implemented [1].

3. Designed, implemented and experimented with different training sequences for the agent to find the sequence that produced the best training performance.
    - The RL algorithm used to train the agent with deep ANN was Proximal Policy Optimisation(PPO), a state-of-the-art RL algorithm developed by OpenAI in 2017 [18]. Along with PPO, to enhance the memory of the agent, a recurrent neural network(RNN) known as Long Short-Term Memory(LSTM) [19] was implemented. Besides that, a module known as Intrinsic Curiosity Module(ICM) was also implemented for better agent exploration in a sparse rewards environment [20]. Curriculum Learning(CL) and Transfer Learning(TL) were designed, implemented and experimented for training in the Unity environment [21] [22]. Hyperparameters were manually tuned and environment parameters randomization was applied to all training sessions for the agent [23].

4. Quantitatively evaluated and analysed the robustness of the agent in its environment in terms of generality, this particular process will be known as testing for the rest of this paper.
    - Quantitative measures and visual analysis of each training session was recorded in Tensorboard [24].Descriptive analysis and explanation on the training performance was done based on data given in Tensorboard. Robustness evaluation was done using a trained model with the best training performance.

5. Qualitatively analysed the behaviours of the agent's interaction in its environment in terms of desired behaviours and emergent behaviours.
    - Behaviours of the agent and its interaction with its environment were observed during testing to study the desired and emergent behaviours as well as provide an extensive discussion on the studied behaviours.

5. Project Key Components

5.1 Proximal Policy Optimization(PPO)

This project focused only on the use of one RL algorithm which was PPO. A PPO is a model-free RL method that is part of the policy gradient methods class for RL. It alternates between a “surrogate” objective function optimization with stochastic gradient ascent and sampling of data that an agent obtains through its interaction with the environment. While a standard policy gradient method would update policy gradient per data sample PPO instead utilises an objective function that allows policy gradient updates in multiple epochs of mini batches, an epoch is a complete cycle of both forward propagation and backpropagation in an ANN [18].

In a typical policy gradient method, it operates by first computing an estimator of the policy gradient then applying a stochastic gradient ascent algorithm to it to maximise the policy. A classic gradient estimator is shown below:

$$\hat{g} = \hat{\mathbb{E}}_t\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\hat{A}_t\right]$$

Figure 3 : Classic gradient estimator [18]

Π_θ - stochastic policy
$\hat{A}_t$ - estimator of the advantage function at timestep t
$\hat{E}_t$ - the expected empirical average over a finite batch of data samples in an algorithm that switches
between optimization and data sampling

In PPO, a modified version of the surrogate objective function is implemented into a typical policy gradient method. The base version of that function is used in another algorithm called Trust Region Policy Optimization(TRPO) in which its surrogate objective function is maximised while constrained to the size of the policy update as illustrated below:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t[\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

Figure 4: Surrogate objective function maximisation subject to policy update size constraint [18]

Θ_{old} - the vector of policy parameters before the update

However, the maximisation will lead to an extremely large policy update without the constraint so this is where the modification is done , eventually making it a clipped surrogate objective function:

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

Figure 5: Clipped surrogate objective [18]

This modification causes the surrogate objective to have clipped probability ratios that forms a lower bound of policy performance.Therefore, PPO is essentially an improved TRPO which is capable of achieving similar performance reliability and data efficiency to that of TRPO but it is a lot simpler to implement as it only need a few lines of codes modification in a vanilla policy gradient implementation and also it is applicable in more general settings. Other than those, it has better sample complexity, it is capable of outperforming most online policy gradient methods and it can achieve better balance between sample complexity, simplicity and wall time [18]. Thus, this state-of-the-art RL algorithm is the main choice of project.

5.2 Long Short-Term Memory(LSTM)

To understand what LSTM is, one must first know what a recurrent neural network(RNN) is. A RNN is a type of ANN that primarily works with sequential or time series data. The difference between a RNN and a traditional ANN is that RNN has a “memory ” system that can take in information from prior inputs to affect current inputs and outputs while traditional ANN does not. A traditional ANN treats inputs and outputs as independent elements but in a RNN, the output depends on the previous elements in the sequence.

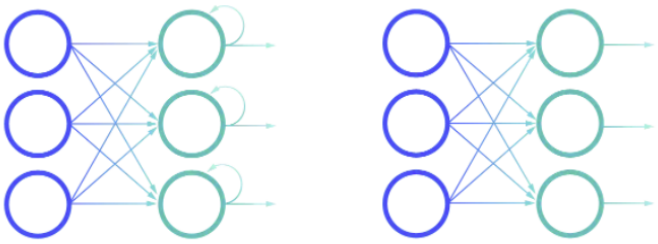

Figure 6: Difference between a RNN(left) and a traditional ANN(right) [25]

LSTM is a RNN with a specific type of architecture that is developed to solve problems of vanishing or exploding gradients in RNN when learning time series data with long-term dependencies [26]. Vanishing gradient is a condition in which the gradient continuously gets smaller to the point where it causes the weight updates to be so insignificant, that the algorithm is not able to learn anymore. On the contrary, exploding gradient is when the gradient gets too big that it causes the learning process to be unstable as the weight updates would be so significant that eventually they would be considered as NaN [25].

LSTM enforces ***constant error*** flow in a number of specialised units, called Constant Error Carrousels (CECs). Putting CEC to use along with input, output, and forget gate is called a memory cell. Forget gate is a part of a memory cell that learns to reset CEC activation function when the information stored in CEC is deemed useless.

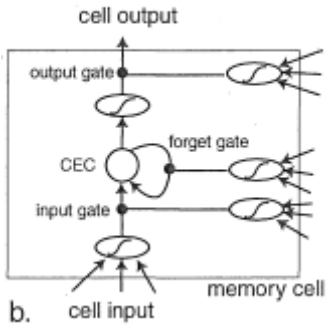

Figure 7: A visualisation of a memory cell [26]

In the context of applying LSTM with PPO, it is used to directly approximate the value function. With this, current observation can be used to approximate the state of the environment along with the agent's history. Therefore, LSTM is a good approach when training an RL agent that needs to deal with non-Markovian tasks with long-term dependencies. An agent that has to achieve a goal or solve a task in an environment where some parts of the state of its environment is hidden from the agent would be known as a non-Markovian task or Partially Observable Markov Decision Processes (POMDP). A good example of a non-Markovian task with long-term dependency would be a maze navigation RL task where different positions may coincidentally provide the same observations but the same action would lead to distinct subsequent states or rewards. The long-term dependency issue emerges when an agent needs to distinguish different observations that seem identical, so to solve this issue the agent needs to be provided with the capability to remember past observations or actions a long time before encountering the identical observations [26]. Similarly for this project, LSTM is a good addition because a big part of the agent's challenge is to navigate itself in a maze.

5.3 Intrinsic Curiosity Module(ICM) in DRL

In most real world RL problems, rewards are sparse, for which it is also the case for the game environment in this project especially as the game progresses. When extrinsic rewards are sparse, it is difficult for the agent to learn anything useful when most of what it does at the start of training should be highly random and will lead to situations where the agent does not receive any extrinsic rewards. A good approach to solve the sparse reward environment would be to implement Intrinsic Curiosity Module(ICM). It is an intrinsic reward signal that can help an agent to learn to explore its environment to familiarise itself with the environment and learn skills that might be needed later on[20].

Curiosity is an independent predictive model that can be trained together with a policy model, taking observations from current actions and the corresponding states to predict next states of the environment. Loss is presumably small when the predicted state is close to the actual, meaning that the observations that an agent has at this point are trajectories that are well explored. On the other hand, loss is presumably big when the predicted state is far from the actual, representing that the observations that the agent encountered are trajectories which are less explored. With that, the intrinsic reward signal would give intrinsic rewards to the agent corresponding to the loss of the predictive

model, essentially using prediction error as a curiosity reward. Therefore, when an agent explores new regions in its environment it will receive high intrinsic rewards, which usually should be the case at the initial stages of training and the intrinsic rewards will slowly drop as the agent get used to its environment more while learning to maximise the extrinsic rewards from its environment. ICM consists of two subsystems, a reward generator that provides intrinsic reward signals driven by Curiosity and a policy to maximise those intrinsic reward signals through a series of actions [20].

The use of ICM is a good approach as it can help the agent to perform well in a sparse reward environment by making use of new, unexplored interactions with the environme -nt as a form of internal motivation to encourage exploration. This could enable the agent to learn to explore its options, slowly but surely finding the best possible way to maximise its extrinsic rewards and achieve its goal. Since the environment for my project is considered to have sparse rewards as well, I decided to utilise ICM to train the agent.

5.4 Environment Parameter Randomization

Environment parameter randomization is a feature in Unity ML-Agents that can be used to configure the environment to automate the randomization of parameters in an agent's environment. For example, the positions in which the agent spawns in the environment on every new episode, the positions of collectibles in the environment and more. As long as it is an element in the environment that can be parameterised within the rules or constraints of the agent's task, it can be randomised. To Improve the robustness of an agent, randomising the environment parameters allows the agent to learn to adapt better to similar but different environment states throughout its training. The agent is able to improve its ability to generalise to unseen variation of its environ- ment instead of only learning to complete its task in one specific deterministic way in its environment and unable to complete the same task with a slightly different or more complex environment [27].

This approach is based on the idea of domain randomization, a popular approach in the RL research field proposed in the paper by Tobin et. al.. In their paper, a training method known as domain randomization is explored and applied to randomise the rende- ring sequence or process of a simulator when training models on simulated images. According to them, the real world could be treated by the model as simply yet another variation with sufficient variability during the simulation. The first successful transfer of a deep ANN to the real world for robotic control trained solely on simulated RGB images along with the absence of pre-training on real images is demonstrated in their paper [27]. According to speakers at the GDC Machine Learning Summit at 2020, domain randomization can be categorised into 3 different forms: variation of environment, random level sampling and procedural level generation [28].

Tobin et al. stated that the purpose of domain randomization is to train the model with sufficiently large variability so that the model is capable of generalising to real-world, often highly complex data during test time.They concluded that training with domain randomization enabled an object detection system to perform with sufficiently high accuracy in the real world with training only done in simulation. They went on to further conclude that domain randomization could be used as a vital tool to make complex

policies in DRL to be learned in simulation through large scale exploration and optimization before applying the learned policy models on real robots [27]. Environment parameter randomization in Unity ML-Agent works in a similar manner. For instance, position of a collectible is randomised to appear at different places in the environment so that the agent is able to learn to look for it not just at one specific spot in the environment.

5.5 Curriculum Learning(CL)

Curriculum learning(CL) is a ML training approach that draws inspiration from the typical learning order in human curricula.CL has proved to be a powerful and easy to implement method in boosting the generalisation capability and convergence rate of machine learning models in a diverse array of ML research fields such as computer vision, natural language processing and and also various RL problems. CL entails training a ML model with an initial learning process of smaller data subsets or easier subtasks and gradually increasing the complexity or difficulty level until the full dataset or main task is covered as shown in figure 8 [21].

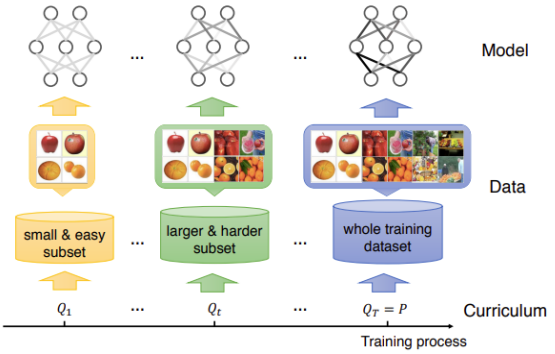

Figure 8: A general example of CL workflow

Without CL, a standard ML algorithm would be randomly presenting data samples to the learning model during training while being oblivious to the learning capability of the model and the different level of complexities within a certain dataset. CL application in ML would benefit the learning process but it is not guaranteed. The amount of advan- tage in employing a CL based training strategy depends on the specific ML problem, the type of datasets the ML problem requires and the design of the CL strategy itself [21].

Generally, there are two incentives as to why CL is a good and popular approach in modern ML training. First ,in terms of optimization problems, it is “to guide” the agent’s training process in a more structured and hierarchical manner, regularising towards better regions in parameter space. Second, in terms of data distribution, it is to “denoise” by reducing the interference of useless data for the learning model to focus more on the high-confidence easier data. The first incentive typically is applied to ML problems that are highly difficult to solve with direct training. Without CL, the model would have a slower convergence or an overall poorer performance[21].

The description of “to guide” incentive on applying CL fits well into this project because the task that the DRL agent needs to complete in its environment is a considerably complex one. Training the agent without CL implementation has proven to be imprac- tical

as the agent is not able to learn to cope with such complexity at the early stages of the training process. Thus, a customised CL strategy is designed to break the task into multiple easier subtasks during training.

5.6 Transfer Learning(TL)

Transfer Learning(TL) is a ML training approach with a similar concept to that of CL but it is implemented and applied in a slightly different way [29]. TL is a technique in which a trained model produced from one training process is reused as the starting point of another training process, ergo a "transfer" of learning. In other words, TL allows the use of a trained model to be further trained and improved from one separate training process to another instead of having to start training from the ground up everytime a model is to be trained. TL is capable of retaining the knowledge gained from solving one task and applying that knowledge to a related but different task. TL is usually applied in deep ANN in which a pre-trained deep ANN is applied on a new task to achieve a reduced training period [22]. For this project, TL is a provided feature in Unity's ML-Agent package and it is used side by side with CL.

5.7 Unity ML-Agents

This unity package is used for this project because it is high level yet flexible and highly customisable. The package allows users to use Unity editor as the tool and platform to develop the agent that can receive inputs from the game environment and output actions to affect the state of the environment. The python trainer, a python package known as ***mlagents*** [30] is the one responsible for the heavy work of receiving inputs from the Unity environment, processing the information and output a decision for actions back to the Unity environment. Inputs and outputs are channelled through a python API provided by a package called ***mlagents_envs*** that is automatically installed along with ***mlagents***. The package also made use of ***torch***, a python package for efficient deep learning processes using GPU or CPU [31]. Below is a simple illustration of its workflow [8].

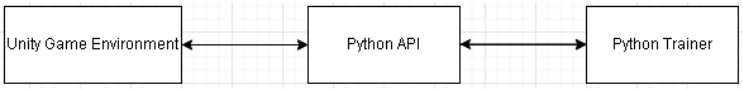

Figure 9: A high-level flowchart of how Unity ML-Agent works [8]

6. Background & Related Works

As previously mentioned in sections above, RL is a relatively big field of research in AI and any topics of AI in Robotics are no stranger to this ever growing research field. Plappert et al. from OpenAI demonstrated a set of robotic continuous control tasks simulation integrated from OpenAI Gym based upon up-to-date existing robotics hardware. The RL model is trained based on a multi-goal reinforcement learning framework for sparse binary rewards tasks where the agent is to follow commands through an additional input. The model is used to control a Fetch robotic arm to complete tasks like pushing ,sliding, picking and placing and also control a Shadow Dexterous Hand to manipulate in-hand objects [32].

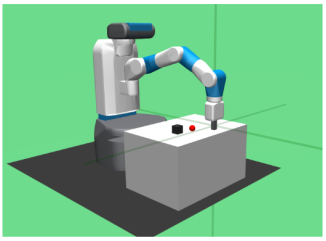

Figure 10: Fetch robotic arm [33]

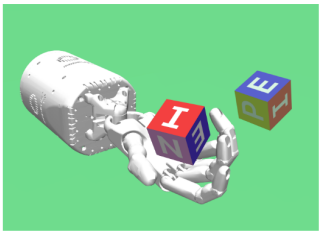

Figure 11: Shadow Dexterous Hand [33]

They stated in their paper that choosing a specific robotics continuous control problem to work on is probably the most difficult part of RL robotics research and they concluded the paper by presenting a set of research problems that they believe have the potential to induce even more widely applicable RL improvements [32].

In relation to my project, there are many other researches and projects involving the application of RL or DRL in games regardless of video or tabletop games. As clarified before, DRL is commonly used to handle large scale high dimension raw inputs to optimise deep ANN-based policies. Shao et al. reviewed successes that DRL had so far in various video games, achieving superhuman performances as well as some emphasis on a few important key takeaways on aspects of DRL application in video games such as exploration, exploitation, sample efficiency, generalisation and transfer, multiagent learning, imperfect information and delayed sparse rewards which most of them are highly relevant to my project [34]. Based on their paper, the typical workflow of DRL in video games is as such:

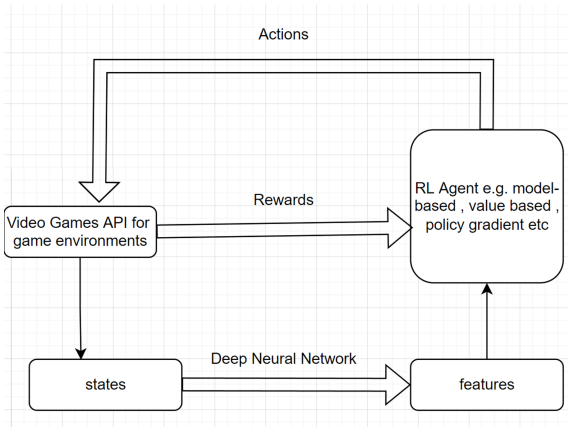

Figure 12: Typical workflow of DRL in video games [34]

The deep ANN is able to gain meaningful features automatically from raw inputs from the video games environment. Based on those meaningful features, agent actions are made causing its environment to respond in some ways, integrating the agent actions, producing the states of the environment [34].

In DRL, other than PPO which has been clearly stated as the only focus in my project, there are many other RL algorithms. Most of which can be classified into different categories, namely value based methods, policy gradient methods- which PPO belongs to and model based methods. Shao et al. concluded that game AI with DRL is a promising approach though there are still big problems in the field of DRL application in video games especially those of 3D imperfect information multi-agent video games [34].

In the realm of classic 2D games, the ones that are famously used for RL agents are the Atari 2600 games integrated from the Arcade Learning Environment, a platform for AI agents development and benchmarking in various 2D game environments [15]. Mnih et al. from DeepMind introduced a DRL model that can learn to control the agents by directly learning policies from raw high-dimensional pixel inputs. It was made possible by training the model in a convolutional neural network with a Q-learning variant to produce a value function that estimates future rewards. Deep ANN combined with Q-learning, a value-based RL algorithm, this approach is known to be called Deep Q-Network(DQN). DQN was proven to be capable of state-of-the-art performance in 6 out of the total 7 Atari 2600 games it was tested on without calibration in hyperparameters space or the network architecture [35].

OpenAI revealed that on 13th April of 2019, OpenAI Five or simply known as Five , one of OpenAI's project became the first ever AI system that successfully defeated world champions-Team OG in Dota 2, a competitive 5 versus 5 multiplayer online battle arena(MOBA) video game developed and published by Valve Corporation [36] [37]. This is considered as one of the big breakthroughs in AI research as Five managed to control 5 different characters simultaneously through having 5 distinct neural networks [38].

Figure 13: An example of OpenAI Five in action [39]

The idea behind Five in solving Dota 2 as a complex environment was to "scale existing reinforcement learning systems to unprecedented levels, utilising thousands of GPUs over multiple months". Five made use of large scale self-play DRL through existing techniques, a well designed distributed training system and tools for continual training and it was able to be trained for up to 10 months. Dota 2 was considered to have given reinforcement learning multiple big challenges due to the fact that the game itself had a decade of development and production history with thousands of complex data for an AI model to comprehend. Given that Five was trained over the course of 10 months, changes and improvements are made to the underlying architecture and design of the DRL system due to the constant change of game environment and mechanics in Dota 2. The team behind OpenAI Five built a distributed training system for which they termed "surgery" which was an approach to proceed with training across changes to the model and game environment."Surgery" was viewed as a collection of tools that perform offline operations on old models in order to obtain a new model that was compatible with the new game environment, in a way "surgery" was like a powerful and flexible version of a traditional TL. OpenAI Five Arena was made available to the public so that human players can play against Five competitively in which Five successfully won 99.4% of the total 7257 games it has played. This further proves that with proper scaling, modern DRL techniques can perform very well with massive complex data [37].

Prior to the brilliant achievement by OpenAI Five, Silver et al. from DeepMind outlined an approach that they applied to an AI known as AlphaGo in playing Go, a classic tabletop game famous for its extremely large search space that poses an extreme challenge for modern AI to process board positions and moves. That approach involved the use of policy based deep ANN - “policy network” for move selections and value based deep ANN - “value networks” for board positions evaluation. Other than RL in games of self-play, these deep ANN were also trained by a curated mixture of Supervised Learning(SL) from human expert games for which SL is another major subfield of ML that differs slightly from RL. They successfully presented a new search algorithm known as Asynchronous Policy and Value Monte Carlo Tree Search algorithm (APV-MCTS) that efficiently combined deep ANN with MCTS, a state-of-the-art search algorithm. With that, AlphaGo achieved almost a 100% winning rate against many other AI Go players, and defeated the European Go Champion by 5 games to 0 [40].

In an effort to further advance the capability of AlphaGo, Silver et al.later on established a more powerful AI Go player known as AlphaGo Zero that performed even better than AlphaGo. It was able to learn without any sort of human knowledge outside of fundamental game rules entirely with RL. This approach essentially made AlphaGo Zero a self-learner, teaching itself how to play the game. Predictions on move selections by a deep ANN for both AlphaGo Zero and the winner of AlphaGo Zero’s games during training allowed the deep ANN to improve the strength of the tree search leading towards better quality of move selections and self-play in subsequent training iterations. AlphaGo Zero was able to achieve even more powerful superhuman skills compared to AlphaGo, defeating it by 100 games to 0. This proves that even without SL from human expert data, training a model to play Go by learning from the ground up through RL, or in the words of the authors, starting *tabula rasa*, can achieve greater asymptotic performance albeit slightly longer training time compared to AlphaGo. To top that off, AlphaGo Zero was capable of learning a large amount of Go knowledge over a few days that is comparable to the amount humankind has accumulated over thousands of years with millions of games played [41].

StarCraft II, a highly complicated real-time strategy game that is often regarded as another major challenge for AI research as it is considerably the most difficult professional esports due to its large raw inputs, high complexity and multi-agents setup. Vinyals et al. demonstrated the use of DRL for multi-agent in which deep ANNs are used to process both adapting strategies and counter-strategies from human and agent games. The AI StarCraft II player in this paper known as AlphaStar was successfully rated at Grandmaster level, the highest rank in the game, performing better than 99.8% of officially ranked human players at that time [14].

Figure 14: An example of AlphaStar playing StarCraft II [42]

Before the success of AlphaStar, there were AI agents that were capable of playing the game with decent performance with simplification done on vital components of the game or with more privileges given to the AI compared to a typical human player. However, AlphaStar was playing the game as a whole, just like how a normal human player would. StarCraft II is a good representation of a real world AI task in a complex environment and a 3D imperfect information multi-agent video game with a wide range of action spaces. The achievement of AlphaStar is a proof that DRL is a good approach to building robust general purpose ML algorithms. Given the right deep ANN architecture and design, DRL has high potential in solving real-world complex problems [14].

McPartland and Gallagher demonstrated the capacity of tabular State-Action-Reward -State-Action(SARSA), a form of value-based RL algorithm in a simplified First-Person Shooter(FPS) game created by themselves. Separated bot controllers were trained using SARSA, each learning tasks such as navigation, item collections and combat individually. Their results showed that SARSA was able to learn a decent policy for navigation control though not up to the level of a standard A* algorithm(a heuristic search algorithm specifically built for robust pathfinding). Whereas the combat bot controller produced promising performance against a finite state machine opponent. In the second fold of their project, McPartland and Gallagher combined pre trained RL controllers through a number of differently curated methods to produce a more general AI bot. They concluded that RL was a successful approach and more complexities could be added by increasing both the state and action spaces for further benchmarking [44].

Building upon the paper mentioned above, Piergigli et al. made use of DRL to train multiple agents of a multiplayer survival FPS instead of just one agent. Piergigli et al. provided some preliminary results in their paper in their effort to train multiple agents to survive in unfamiliar environments, building up their combat skills , adapting to unexpected states , cooperating and coordinating with one another as well as competing for good ranking in the game. Exploiting PPO to train the agents without human expert data using their own “Learning System”, essentially a training design that they developed for the agents with the eventual aim of developing a team of AI players. They concluded that their agents’ performance seemed promising but still had plenty of room for improvements [45].

Juliani et al. from Unity Technologies highlighted that many of the existing environments for RL agents testing and benchmarking are not capable of providing flexible simulation

configurations. In their paper, they concluded that modern game engines, in this case, Unity Game Engine, is a suitable general platform for the development of a game environment alongside the creation of the agent using Unity ML-Agent Toolkit [1],all of which aligns with my project.

7. Design & Implementation

7.1 Game Environment

One of the major components of this project was the game environment for the agent to interact with. According to Schell,a game typically consists of four different elements in its design, namely story, technology, mechanics and aesthetics [48]. As this project aimed to focus on experimenting and studying the capabilities of DRL, there was no story and aesthetics were kept at the necessary minimum. Mechanics was the most vital element in terms of clearly defining what the agent can or cannot do in its environment. Core mechanics of the game are as such:

1. Player can move in any direction to explore the maze environment
2. Player can collect any treasure by collision
3. Player can eliminate any zombie by shooting at it with the only weapon given

The elements that directly affect the behaviour of the agent were the rules and constraints of the game:

1. Imperfect information environment.
    a. The environment was only partially observable by the agent, meaning that at any given point in the game, the agent was unable to “see” the layout of the environment as a whole.

2. Agent had a fixed limited amount of time to complete the game before the environment resets.
    a. There is a built in time limit system known as MaxSteps that is implemented along with the use of Unity Ml Agent package to develop the agent. It is a timestep system where all parameters in the environment and the agent’s progress will be reset once an instance of the environment ends, the instance is known as an episode.

3. Agent had restricted movements.
    a. The agent was not allowed to move left or right and only allowed to move forward, backward and turn to any direction. For a typical mouse and keyboard first-person setup for a human player, this would mean that pressing on ‘A’(move left) and ‘D’(move right) were not integrated into the game.

4. The agent only had a limited amount of weapon ammunition in each episode.

5. Agent had a fixed timer in between shots.
    a. The agent was only able to make one shot at any given point in time during the game and each subsequent shot was allowed to be made after the countdown of a fixed timer.

6. Agent had a fixed shooting distance
    a. The agent's shot was only registered and integrated into the game if the shot was made at a fixed distance from the target, meaning that if the agent shot at a zombie out of that fixed range, the zombie will not take damage because the "bullet" did not reach the zombie.

7. Agent spawn region was constant but it spawned at random positions in that region
    a. The agent has its own spawning region in the environment where no zombies or treasures or maze will be spawned. The spawning region was a fixed, designated area in the environment but the agent spawned at random positions within that region at the start of each episode.

8. Zombies had fixed pattern of behaviours
    a. Zombies were always alternating between wandering around and chasing the agent with a fixed countdown timer for each state.
    b. Each episode always started off with the zombies wandering around the place before it started moving towards the agent, constantly aware of where exactly the agent was throughout the entire chasing state.
    c. Zombies' movement speed was always 1 value higher during chasing state than the base speed during wandering state.
    d. A zombie can be killed as long as it was shot by the agent once.

9. Treasures were always scattered at random positions at the start of each episode.

10. Number of zombies and treasures were fixed at the start of each episode and will not increase at any point during the episode.

The game environment was a complex task environment with sparse rewards for the agent. In addition, the rewards get even more sparse as the game progresses within an episode, assuming that the agent was able to kill some zombies or collect some treasures as the game progresses. The conditions to win the game were as follow:

1. Seek and collect all treasures in the environment within the time given
2. Kill all the zombies in the environment with the only weapon given and within the time given
3. Avoid getting touched by any zombie, one touch leads to game over
4. Game is over when there is at least one zombie in the environment and no ammunition left to use

Fundamentally, the environment is a maze but it does not function like a typical maze game. Instead of looking for an exit for the maze where the agent was expected to solely learn navigation control, the agent was to complete its task, which was to meet all the

conditions aforementioned to secure a win while abiding to the implemented mechanics, rules and constraints of the game. Therefore, the maze was essentially there to just act as a hindrance for the agent to complete its task in the environment. The game environment was made based on the game objects provided by a package called StarterAssets [49], though none of the scripts provided by the package were used.

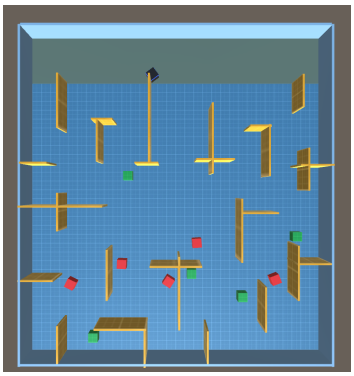

Figure 15: A top-down view of the game environment

An example of the game environment is shown in Figure 15, the black cube is the agent, greenish region at the top is the spawn region for the agent, the striped blue region is where the maze, zombies and treasures are put, red cubes are the zombies and the green cubes are the treasures whereas the yellowish cuboids are the walls that made up the maze. The entire environment is surrounded by 4 blue cuboids acting as borders. The layout of the maze is designed by myself ,drawing inspiration from classic Pac-Man game [50]. The idea was to always confuse the agent at every junction. Due to how the agent was set to partially perceive its environment, the walls were made to block the agent's field of view.

The zombies were implemented as navigation mesh agents [51] that allows the zombies to move in the environment based on Unity's AI system for pathfinding and navigation through spatial queries known as NavMesh [52]. The zombies were implemented as simple finite state machines as well, with only two states, wandering state and chasing state. The wandering state allowed the zombies to just move around in a random direc- tion set using a predefined function from Unity's Random class known as insideUnit- Sphere [53]. The chasing state allowed the zombies to move towards the agent by using SetDestination from NavMeshAgent class[54] through the Update function from Unity's MonoBehaviour Class [55]. The zombie died when the agent shot at it accurately, and that particular zombie game object will be deactivated in the environment which was done by setting false on the SetActive() function from Unity's GameObject class [56]. The zombie will respawn in the next new episode by setting true on SetActive().

The treasures were implemented as basic 3D cube game objects that were instantiated at random positions within the striped blue region at the beginning of each new episode by using Instantiate() function[57] and each will be destroyed from the environment through the use of Destroy() function [58] when the agent touched it.

7.2 Agent

The agent was implemented as a simple black cube by attaching a script that inherits from Unity ML-Agent class called *Agent* [59]. Besides, pre-made scripts called *Behavior Parameters* and *Decision Requester* are also attached to the agent. *Behavior Parameters* allows the agent's basic parameters to be set and adjusted whereas *Decision Requester* allows the agent to request for output actions from the trainer.

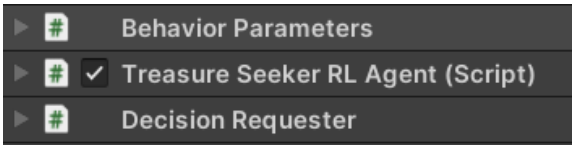

Figure 16: 3 main components for the agent

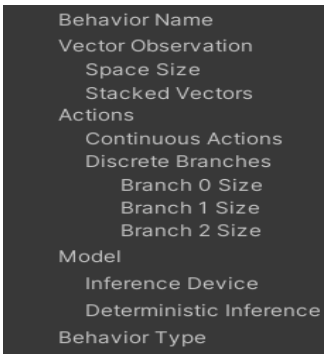

Figure 17: The adjustable parameters under *Behavior Parameters*

7.2.1 Observations

7.2.1.1 Raycast Observations

The agent was using two sets of Raycast observations and a set of vector observations. Raycast observations made use of rays[60] that were casted from the agent itself into the game environment and the tags[61] given to each game object that were hit by the casted rays will be observed by the agent. Other than the raycast observations, the vector observation had a space size of 4.

Raycast observations were implemented by attaching premade scripts provided by Unity ML-Agent package to the agent:

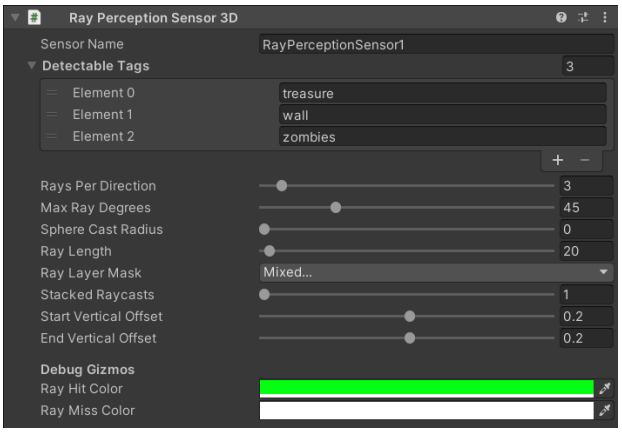

Figure 18: First set of Raycast Observations

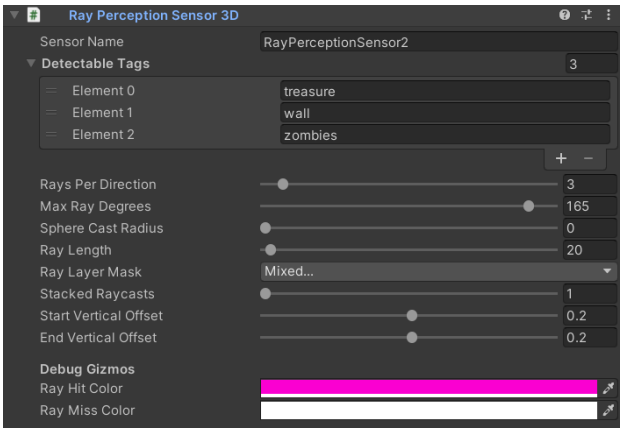

Figure 19: Second set of Raycast Observations

Detectable game object tags detectable by the rays were the treasures , the zombies and the walls which includes the walls that made up the borders and the maze. The first set was primarily for the agent to perceive objects in front of it hence the max ray degrees were set to 45 whereas the second set had max ray degrees of 165 , allowing the agent to have side views and back view. Providing the extra set of raycast observations was to

allow the agent to be able to observe its sides and back, similar to providing a mini map to a human player. Both sets had a default built-in ray each casted from the agent directly to its front making that rays to be at degree 0. In addition, both sets had 3 rays per direction - left and right to the default ray, each direction with a maximum of 180 degrees, leading to a total of 14 rays as shown below.

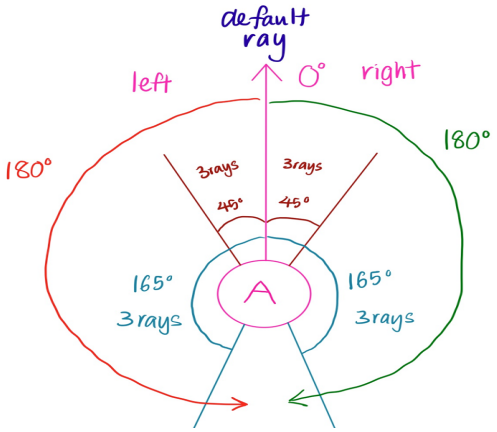

Figure 20: The layout of the observation rays

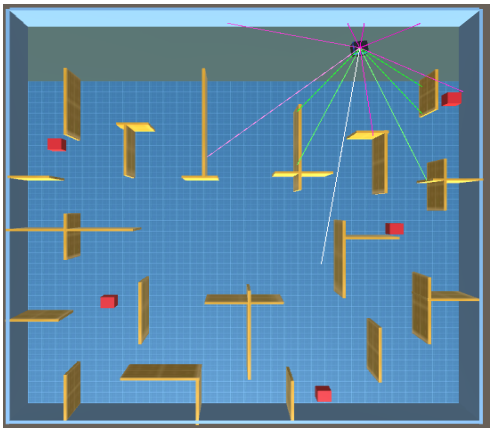

Figure 21: An example of the rays originating from the agent(black cube)

Based on the equation below:

(Observation Stacks) * (1 + 2 * Rays Per Direction) * (Num Detectable Tags + 2)

The total number of created raycast observations is 70. The length of all rays are fixed at 20, allowing the agent to only have imperfect information on its environment regardless of the agent's position in the environment, any objects that are out of the range for which the rays can reach, those objects' tags are undetectable. The number of rays per direction are kept as minimal as possible because redundant rays will only cause the agent's learning process to be less stable and more difficult for it to learn any meaningful behaviours [62].

7.2.1.2 Vector Observations

```
sensor.AddObservation(allTreasuresFound);
sensor.AddObservation(allZombiesDead);
sensor.AddObservation(canShoot);
sensor.AddObservation(tsarea.treasureboxList.Count);
```

Figure 22: Agent vector observation spaces

The vector observation set consists of 3 boolean observations and 1 integer observation:
allTreasuresFound - boolean flag to allow the agent to constantly observe whether or not all the treasures in the environment are found
allZombiesDead - boolean flag to determine whether all the zombies in the environment are dead
canShoot - boolean flag to determine whether or not the agent is allowed to take a shot
tsarea.treasureboxList.Count - number of remaining treasures to collect in the environment

The vector observations were designed as such because it was expected that the agent should learn to realise that when the number of treasures in the environment was reduced to 0, allTreasuresFound will turn from false to true. Besides that, it was also expected that the agent should learn to realise that when allZombiesDead turned from false to true along with true for allTreasuresFound, it has won the game. The canShoot flag was to allow the agent to learn to realise that it was only able to shoot once every fixed time so it should learn to time itself so that it can shoot when it encounters a zombie.

7.2.2 Actions

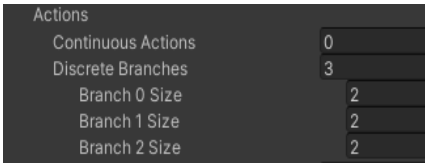

Figure 23: Agent action spaces

The actions for the agent were separated into 3 branches of discrete actions with 2 distinct discrete values in each branch representing 2 different actions. Continuous actions were not used at all due to the use of LSTM for training the model as LSTM works better with discrete action spaces. Side movements were not available for the agent because it was redundant as it can move considerably smoothly with only branch 0 and branch 1. The actions were implemented as follow:

- Discrete value of 1 for branch 0 → *forward*
- Discrete value of 2 for branch 0 → *backward*
- Discrete value of 1 for branch 1 → *left turn*
- Discrete value of 2 for branch 1 → *right turn*
- Discrete value of 1 for branch 2 → *shoot*
- Discrete value of any integer but 1 for branch 2 → *do not shoot*

The shooting mechanism for the agent was implemented based on Raycast[63] for which a ray was constantly casted from the agent towards the front for a fixed distance and when the ray hit a game object with layer mask of “zombies”, the zombie will be deactivated from the environment, indicating that the damage from the shot was registered and the zombie was dead.

7.2.3 Rewards

In this project, both positive and negative rewards are implemented in cumulative form. For instance, if the agent managed to collect a treasure, it will receive 1 reward point, and the second treasure collected will accumulate to 2 reward points. Positive rewards were given to the agent for actions and behaviour that was considered to be correct or desired, and the more those actions were carried out, the more reward it accumulated, which for PPO, the agent should aim to maximise its rewards. Whereas for negative rewards, though usually only applied to “punish” the agent to let it learn to complete its task quicker by cumulatively penalising the agent for simply doing any actions in the environment, negative rewards in this project were applied to agent actions that were meaningless [64]. For instance, the agent repetitively bumping into a wall instead of

moving around it to get to the other side. If the agent was not penalised for bumping into a wall, it might not learn that behaving that way did not assist it in any way towards completing its task.

There are a total of 22 training sessions forming one long sequence of training processes with each session transferring to the next to produce the final model used for testing. The reward system were as follow:

Sessions	Rewards
1 → 4	+1 for killing zombie, +1 for collecting treasure, -0.025 for missing shot, -0.0025 for touching walls, -1 for getting killed by zombie
5 → 7	+1 for killing zombie, +1 for collecting treasure, -0.025 for missing shot, -0.001 for touching walls, -1 for getting killed by zombie
8 → 10	+1 for killing zombie, +1 for collecting treasure, -0.015 for missing shot, -0.001 for touching walls, -1 for getting killed by zombie
11→ 22	+1 for killing zombie +1 for collecting treasure, -0.015 for missing shot, -0.001 for touching walls, -0.5 for getting killed by zombie

Table 1: Reward systems for each training session

Negative rewards were given in extremely small amounts because excessive negative rewards will lead to the agent not being able to learn any meaningful behaviour. This was especially applicable to my agent, since there were plenty of walls around, almost anywhere the agent moved it would encounter walls, so if simply colliding with the walls or missing a shot caused it to get penalised, it should only be penalised for a much smaller amount compared to positive rewards. On top of that, the negative rewards are also cumulative, so it could easily be accumulated during training to the point where it overwhelms the positive rewards, causing the agent to not be able to find the optimal policy that can maximise its positive rewards [65].

7.3 CL, TL & Environment Parameter Randomization

The DRL model was trained with CL for more effective learning of easier sub tasks to more difficult sub tasks and TL to transfer learned models from one training session to another , transferring learned knowledge and skills to form a long sequence of learning processes. Environment parameter randomization, inspired by the concept of domain randomization [27] was applied in an effort to build and improve agent robustness.

The training design involved these 3 components functioning side by side. TL was done manually from one individual training session to another, initialising a training session from the previously completed session, continuing the learning process from previously saved models, intuitively making the first model produced from the first training session as the base. CL was automatically applied to each training session for each environment parameters. Within each sub task in CL, environment parameters were randomised. Starting with the easiest setting with the bare minimum amount, more environment parameters were added to the environment to provide more features or complexity to make the game gradually more difficult towards the end of the TL sequence. Figure 24 is a

general visualisation of the design alongside the full list of environment parameters used in the entire sequence:

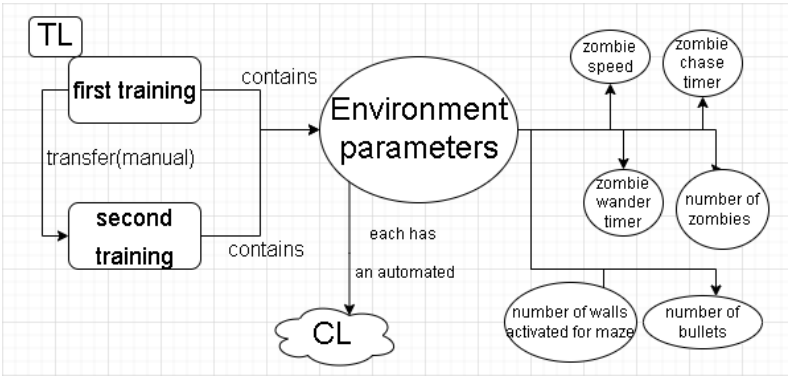

Figure 24: A simple visualisation of the agent training design

The details on the exact training sequence is much too elaborative so the general idea of the training design involving all the 3 components are as stated below:

1. Began training the agent in the environment without the maze alongside static zombies and treasures.
2. If the performance thus far was considerably good, gradually add in more features and environment parameters:
   Moving zombies → Partial maze → Full maze → limited weapon ammunition→ shooting timer

3. Any training performance that shows the learning model was unable to stabilise and converge to a presumed optima, its performance was considered to be poor. Then that particular training session will be repeated before any TL can be done.

Other than CL, TL and environment parameters randomization, PPO, LSTM and ICM were all implemented for training through simple configuration in a YAML file to manage the settings of the training process [66] as well as the use of Academy, a singleton class to manage and automate the randomization of environment parameters in the environment [67].

8. Training

8.1 Network Architecture

There were three different deep ANN used to train the DRL model:

1. A LSTM for PPO to process states observed from environment

```
network_settings:
  normalize: false
  hidden_units: 128
  num_layers: 2
  vis_encode_type: simple
  memory: #recurrent neural network activated
    memory_size: 128
    sequence_length: 64
```

Figure 25: Network settings of LSTM for PPO

Inputs were not normalised as it may be detrimental for discrete actions that the agent in this project primarily used. “Hidden units” refers to the number of processing nodes in each hidden layer which was kept at a constant value of 128 and the number of layers was kept to a bare minimum of 2 because any more layers was not recommendable as LSTM causes the complexity of the neural network to increase greatly. “memory” enables LSTM to be implemented to the deep ANN. “memory size” is the size of an array for the network to retain information as floating point values at any given point during the training and “sequence length” refers to the amount of sequential observations that the network should input at once [68].

2. A deep ANN to process extrinsic reward signals received in the environment

3. A deep ANN to process ICM reward signals

Both deep ANN for extrinsic and curiosity reward signals have the exact same architecture which did not contain LSTM:

```
network_settings:
  normalize: false
  hidden_units: 128
  num_layers: 2
  vis_encode_type: simple
  memory: null
```

Figure 26: Network settings for extrinsic and ICM reward signals

All three of these neural networks were simultaneously working together on policy optimisation to maximise the agent’s cumulative extrinsic rewards.

8.2 Hyperparameters Tuning

Available hyperparameters for each of the 3 deep ANN:

```
hyperparameters:
  batch_size: 512
  buffer_size: 5120
  learning_rate: 0.0001
  beta: 0.15
  epsilon: 0.2
  lambd: 0.9
  num_epoch: 3
  learning_rate_schedule: linear
```

Figure 27: Hyperparameters of LSTM for PPO

```
  extrinsic:
    gamma: 0.99
    strength: 1.0
```

Figure 28: Hyperparameters of extrinsic reward signals

```
  curiosity:
    gamma: 0.99
    strength: 0.25
    learning_rate: 0.0003
```

Figure 29: Hyperparameters of ICM reward signals

There was another hyperparameter known as Max Step which was a built-in timesteps limit for each episode regardless of training or testing. It is not the number of steps an agent can take in an episode but the amount of timesteps based on the elapsed time simulated by Unity. It was consistently fixed at 5000 during training.

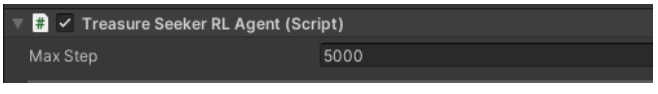

Figure 30: Adjustable maximum timestep given for the agent

All the listed hyperparameters were tuned manually for subjectively more control in monitoring and logging of the training performances. If a training session resulted in poor performance, the hyperparameter that was most likely to cause the poor performance will be tuned accordingly before re-training. Hyperparameters were also occasionally tuned for TL not due to poor performance but for refining purposes. The important hyperparameters were the learning rate, beta, epsilon and the strength for both extrinsic and curiosity [68].

Learning rate was tuned based on the stability and consistency of the reward maximisation, usually lower values means slower but more stable learning. Beta essentially determined the randomness of the RL policy, so it was tuned depending on the drop rate of the entropy graph. Epsilon affected the update rate of the policy, which for this project, was kept at 0.2 consistently. The strength of both extrinsic and curiosity-driven reward signals were the magnifier for the reward signals so it was important for the curiosity to not overwhelm the extrinsic reward signals [68]. Extrinsic strength was kept at 1.0 consistently whereas curiosity strength was tuned based on the results on the curiosity rewards graph.

8.3 Training Results

The training sequence consists of 22 consecutive sessions, the agent was trained for a total of 76.5 million timesteps in around 26 hours. Below are some of the important statistical evaluation of the training results from the final training session, the 22nd. The graphs below were updated every 5000 steps for a total of 7 million steps.

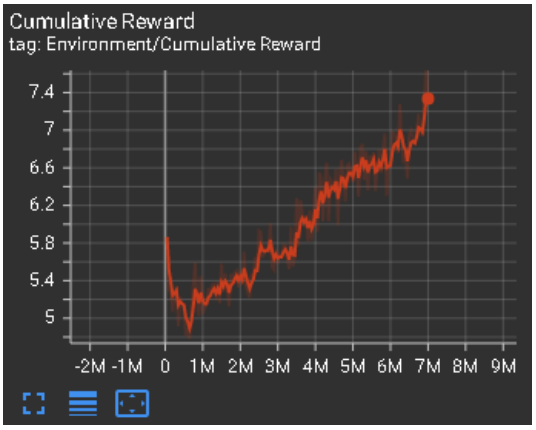

Figure 31: Cumulative reward against timesteps

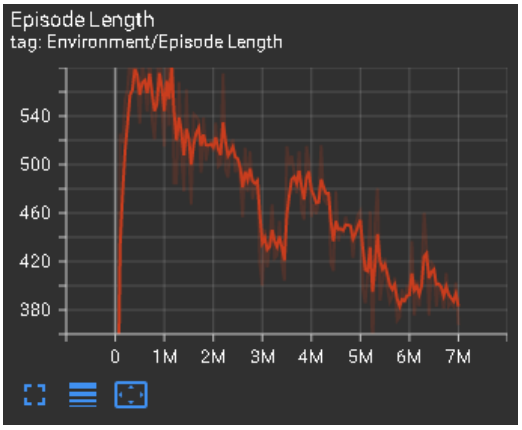

Figure 32: Episode length against timesteps

The initial cumulative reward at the first few thousand steps were considered high but it dropped before it started to rise consistently towards the end, indicating that the agent was learning to maximise the extrinsic rewards. The drop at the earlier phase was due to the existence of negative rewards, so at the start the agent should tend to explore more which will then lead to making more meaningless behaviours and causing more accumulated penalties. It was a good sign that it was able to learn from its mistakes and gradually learn to behave in a certain way that allowed it to complete its task. The episode length should generally have a consistent drop because the agent required lesser

timesteps to complete its task as it learned to optimise its policy. The sudden surge between 3M to 4M was due to the increase in difficulty and big change of environment implemented through CL. Therefore, the agent required slightly longer time to learn to adapt to the more difficult version of the environment before gradually learning to optimise its policy again.

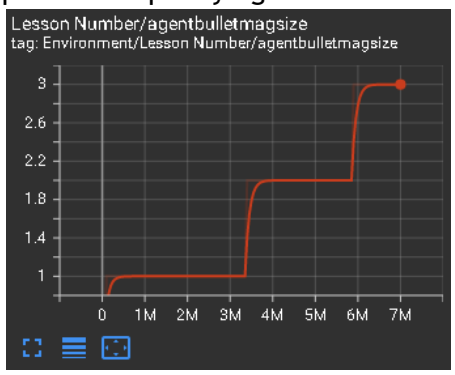

Figure 33: Lesson number(environment parameter) against timesteps

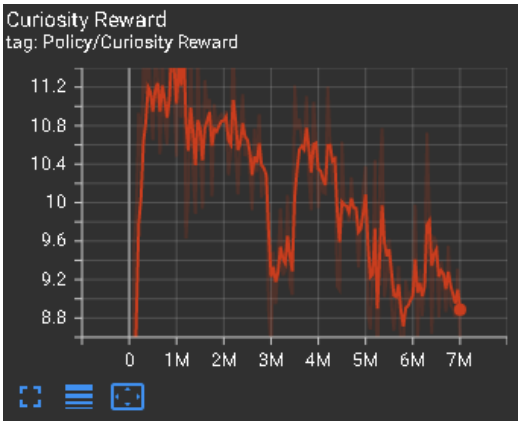

Figure 34: Curiosity reward against timesteps

This is an example of the CL for one of the environment parameters. There were a total of 4 lessons, 0 to 3, and a set of criteria to meet before a subtask can be considered as completed and allow the agent to move on to the next subtask with higher difficulty. Curiosity reward should have a consistent drop after rising during earlier training steps because the agent should explore its environment more earlier on, be curious and obtain new states from the environment so that it can learn to familiarise itself with its environment. The gradual drop indicates that it was starting to be less curious about its environment and focus more on exploiting specific regions to optimise its policy.

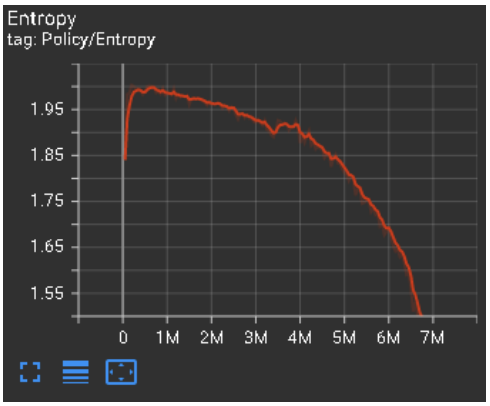

Figure 35: Entropy against timesteps

Aligning with the concept aforementioned, the “explore more at first then exploit more after” was applicable here as the entropy should rise up at the earlier training steps so that it can do more random actions to “test the waters” before it can slowly learn to be more deterministic about actions to carry out to complete its task.

9. Testing

The main objective was to test the robustness of the agent, which was in the case of project, robustness refers to the agent's capability to generalise to its environment.

9.1 Testing Setup

Testing was done by running 1000 episodes of 5000 Max Steps per episode() and 10000 Max Steps per episode. The model was tested on the same environment used during training with all environment parameters set at the most difficult stage : full maze with 5 moving zombies to kill, 5 treasures to collect and a total of 10 ammunition given for each episode. Zombies and treasures were spawned at random positions and the agent was spawned randomly within its designated spawn region at the start of each episode. The model was "plugged" into the agent, serving as the "brain" of the agent as shown below:

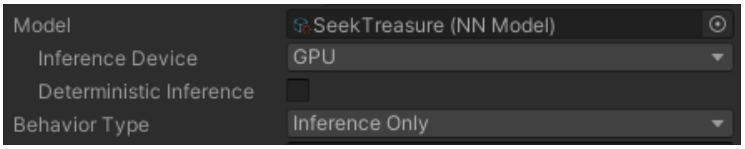

Figure 36: Agent model and inference device

During testing the agent only inferred from the trained model and it was no longer learning. The performance of every 100 episodes was recorded up to 1000 episodes. Below is a list of components that were recorded:

1. Number of successful episodes out of total
2. Number of missed shots(successes) → maximum, minimum, average
3. Amount of scores(failures) → maximum, minimum, average
4. Cumulative rewards → maximum, minimum, average

Testing the model on two distinct values of maximum timesteps was to find out whether increasing the maximum time steps by twice would produce better statistical results. The number of missed shots was measured only for the successful episodes to find out the severity of failure in success by measuring the amount of mistakes(missed shots) made in those episodes. On the contrary, to find the capacity of success in failure, scores were measured only for the failed episodes. Cumulative rewards were measured regardless of success or failures and it follows exactly the same way extrinsic rewards were measured during training which includes both positive and negative rewards. Unlike cumulative rewards, positive scores were only given when the agent killed a zombie or collected a treasure while there were no negative scores given for any actions.

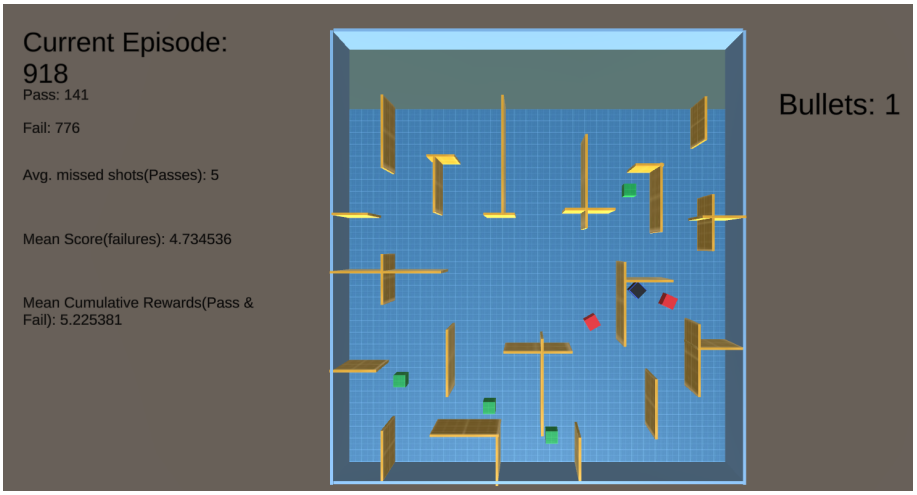

Figure 37: An instance example of testing in progress

9.2 Testing results

9.2.1 5000 Max Steps

Episode	Success	Rate	Scores(failure)			Cumulative reward		
			Max	Min	Average	Max	Minimum	Average
100	13	13%	9	0	4.885057	9.913000	-0.566000	5.235949
200	32	16%	9	0	4.601191	9.914001	-0.566000	5.146673
300	46	15.3%	9	0	4.69685	9.914001	-0.566000	5.192046
400	63	15.75%	9	0	4.673591	9.917000	-0.566000	5.195723
500	72	14.4%	9	0	4.628505	9.917000	-0.566000	5.076532
600	86	14.3%	9	0	4.675097	9.917000	-0.566000	5.120435
700	101	14.43%	9	0	4.661102	9.917000	-0.566000	5.117445
800	116	14.5%	9	0	4.646199	9.917000	-0.566000	5.112566
900	129	14.33%	9	0	4.640727	9.917000	-0.566000	5.098929
1000	144	14.4%	9	0	4.632010	9.917000	-0.566000	5.093273

Table 2: Testing results based on 5000 Max Steps

9.2.2 10000 Max Steps

Episode	Success	Rate	Scores(failure)			Cumulative reward		
			Max	Min	Average	Max	Minimum	Average
100	12	12%	8	0	4.693182	9.911001	0.3850001	4.988699
200	29	14.5%	8	0	4.555555	9.922001	-0.530000	5.010468
300	48	16%	8	0	4.623016	9.922001	-0.530000	5.158147
400	60	15%	8	0	4.729412	9.922001	-0.530000	5.200534
500	78	15.6%	8	0	4.703792	9.922001	-0.530000	5.203167
600	87	14.5%	8	0	4.692008	9.922001	-0.530000	5.137199
700	104	14.86%	8	0	4.68792	9.922001	-0.554000	5.152338
800	123	15.38%	8	0	4.726736	9.922001	-0.554000	5.218905
900	138	15.33%	8	0	4.730971	9.922001	-0.554000	5.220318
1000	153	15.3%	8	0	4.730814	9.922001	-0.580000	5.219267

Table 3: Testing results based on 10000 Max Steps

Based on both the tables above, the overall performance of the agent can be considered as consistent because most measurements are within a small, tight range of values. The number of missed shots were not presented in the table as it would be redundant because the average, maximum and minimum consistently have values of 5 throughout all the tested episodes regardless of timesteps. This shows that even when it succeeded, the number of mistakes made by the agent was always the highest possible. The success rate was essentially the primary measurement for robustness of the model. Considering both sets of measurements involving two distinct timesteps given in each episode, the success rate was within the range of 12 to 16%. Unfortunately, this can be considered as poor performance due to a few possible factors i.e. the network architecture, the training techniques as well as the hyperparameters. The hyperparameter tuning process could be automated to find best possible values instead of manual tuning, though setting up the automated selection process for optimal hyperparameter values would require much longer time. In terms of network architecture and training techniques, a deeper and better understanding of how and why they can affect the performance of a model would be beneficial as that would increase the chance of creating a better approach.

Maximum values for scores obtained for failed episodes were consistently high which was 9 and 8 respectively for 5000 and 10000 Max Steps tests. Those scores were considered to be high because obtaining 10 in an episode would mean it was no longer a failed episode. This was usually because the agent ran out of time to collect the last one or two treasures positioned at some corners in the maze or the agent ran out of ammunition before having a chance to kill the last one or two zombies or perhaps even because of the last one or two zombies that managed to kill the agent before it can retaliate. The minimum scores were always 0 simply because there were bound to be instances where the agent was killed by a zombie before it could even collect any treasure or kill any zombie. Average scores for failures were within the range of 4.5 to 5 indicating that even when the agent failed an episode,on average,it was at least halfway to success. This proves that the model

performance is not as poor as it seems if one is to evaluate the performance solely based on the measured success rate.

The average cumulative rewards were within the range of 4.9 to 5.3. With 10 being the highest possible amount, the maximum cumulative rewards were 9.9 and above. The minimum cumulative rewards were slightly below 0. This shows that the average overall performance of the agent is roughly half as good as the best performance. It was entirely impossible for the agent to gain cumulative rewards of 10 because of the existence of negative rewards, obtaining that value would require the agent to not make even one single mistake. The amount of failed episodes were larger than the amount of successful episodes by a large margin, causing the cumulative rewards to be relatively close to that of the scores of failed episodes.

Agent robustness was rated at the average value of 14.641% in 1000 episodes with 5000 Max Steps per episode and 14.847% for the ones with 10000 Max Steps. This shows that increasing the timesteps for testing only had an extremely small improvement in its robustness. Though it is worth noting that the small difference in success rate throughout all 1000 episodes between the two sets of tests was fairly consistent simply because the agent consistently had more time to complete its task.

10. Discussion

10.1 Desired behaviours

The way the agent was designed and trained was to eventually enable it to learn a few skills that can lead to behaviours that were desirable so that it was capable of interacting with its environment in a certain way that was expected. Below is a list of skills the agent was expected to learn:

1. Navigation
The agent should learn to control its own navigation, constantly finding the right balance between exploration and exploitation. Generally, it should learn to explore more at the beginning of a game match(which is an episode), familiarise itself with its environment. Once it was more used to the environment, it should lower its exploration capability and maintain it at a decent level while it starts to exploit more in the meantime. With this, theoretically, it should learn to check out every region and corners of the environment when it encounters a zombie or a treasure, it should exploit that region by moving close enough to that zombie to kill it or move towards the treasure to collect it.

2. Combat
The agent is given the freedom to choose to shoot in any direction it wants while abiding to a short timer in between shots.Therefore, the agent was expected to learn to be accurate enough, since there is a fixed range as to how far the shot can reach, it should also learn to be close enough when attempting to kill a zombie. On top of that, it should also learn to be responsive by learning to turn its body quick enough to shoot at any zombie coming its way, especially when facing up against multiple zombies rushing towards its position all at once.

3. Resource management
The agent was expected to learn to manage the limited amount of ammunition it had. Given only 10 bullets and 5 zombies to kill , there was not any room to make much of a mistake. The agent was expected to learn to only utilise the bullets given when a zombie was on sight and an attempt to shoot the zombie to death was necessary to be made, it should not just be simply shooting at anything all the time because doing that would mean that the agent was not actually learning but instead just depending on mere chances that out of those many shots made, some of it will hit the zombies.

4. Survivability
A big part of an action shooter game is not just about combat skills but also survivability. Given the freedom to move anywhere anytime it wants, the agent was expected to learn to survive the zombies' attack while they were all at once chasing the agent. The agent would be terminated once any zombie touched it, so it should learn to retreat and run away when necessary to reset its position so that it can have a better chance of killing the zombies one by one instead of getting killed due to it being unable to handle multiple dangers from different directions at the same time.

The agent is able to pick up on these skills but not up to the level that can be deemed as good. It was only able to occasionally behave the way it was expected to(behave well enough to complete its task on time) and in some instances it behaved poorly. Therefore,the agent being able to complete its task in the environment was surefire, so the more important question to ask oneself was: ***Was the agent able to consistently behave the way it was expected to,in other words, how robust was the agent in terms of its generality(generalising capability)?*** This was the question that this project ultimately seeked to answer, and as evidenced by the quantitative results provided in the previous section, the agent's robustness was considerably low.

10.2 Emergent behaviours

Other than behaviours that were expected, an observation study on any emergent behaviours from the agent was also conducted. The agent tends to exhaust all of its ammunition if there was any left even after all the zombies have been killed, this behaviour was even more apparent with the quantitative results on missed shots in successful episodes. For instance, if the agent manages to kill all 5 zombies within the first 7 bullets, it will still make use of the last 3 bullets, shooting at the walls, it was almost as if the agent was deliberately wasting the last 3 shots just because it can affor- d to do so. This fur- ther shows the weakness that the agent had in terms of resource management. The ag- ent occasionally stayed on the same spot to rotate for a few full 360 degrees before "locking" on a direction to move towards instead of constantly rotating to the desired direction while moving. A possible reason for that behaviour would be that the agent was "scanning" its environment to look for any sort of stimulus such as a zombie or a treasure to move towards. Besides that, a behaviour that was most likely to cause most of its failures would be that the agent was often oblivious to the zombies' existence until the zombies were a lot closer to the agent before it was willing to respond. A zombie can be detected via the agent's observation rays from a distance of 20 units away provi- ded that there was not any walls in between to block its sight, the agent was expected to move

towards the zombie to be close enough so that it can shoot at the zombie from a distance of 10 units away(which was the maximum distance the bullets can reach), the agent preferred to be up close to the zombie before shooting at it. It could be that the agent somehow developed a sense of risk-taking capability because it knew that getting very close to the zombies will secure the kill shots even more though it might be riskier to do so. Furthermore, even with the implementation of LSTM which essentially enhan- ces the memory capabilities of the agent, it can sometimes still be seen having a hard time remembering where it had been causing it to easily be stuck moving within the sa- me region in the maze, constantly revisiting the same place a few times before it was willing to move to another different region. One could argue that mayhaps the LSTM network should be deeper so that the agent was able to remember more as well as longer. However there is always a tradeoff between complexity and stability, making it more complex more often makes the learning process even worse and less stable as opposed to keeping it simple but giving it more time to learn.

11. Conclusion

Although the training results were considered to be good, the testing results did not reflect that. The final model performance can be considered as subpar due to various reasons. For possible better results, different forms of future work approach could be experimented with. For one, the task that the agent is trained to complete could be separated into multiple tasks with each task focusing on learning one specific skill using one deep learning model before combining the models into one through TL. Hyperpa- rameters tuning could be done with state–of-the-art hyperparameter optimization algo- rithms such as exhaustive search like grid search or random search, Bayesian Optimiza- tion, Particle Swarm Optimization etc [69] [70]. Besides, every component involved in the training design has plenty of potential to be further experimented with. For insta- nce, I could experiment with CL and TL approaches that had proven to be effe- ctive by researchers instead of attempting to stick to my own approach. Other than that, a deeper mathematical understanding of the importance of observation design would be beneficial for future work approaches because choosing the correct inputs are extremely important not just for RL but for ML in general as well. Perhaps even a search optim- ization technique to find the optimal observation spaces like the one proposed by Kim and Ha in their paper where they demonstrated the effectiveness of their search algor- ithm on improving model learning speed while comparing it to manually designed observ- ation spaces [70]. Furthermore, more profound research on deep neural networks would also be advantageous, understanding the effects of every parameter in the architecture and experimenting with different architecture optimization techniques could result in better deep learning models. For example, Luo et al. who developed a technique known as Neural Architecture Optimization(NAO) to efficiently automate neural architecture design based on continuous optimization [71].

12. Reflection

12.1 Project Aim Justification

The project aim is considered to be achieved. The aim is to create a single agent to play a 3D game that I designed and developed from scratch and the agent was able to comp- lete the game though it was not able to do it as well as I thought it would. However, the point of this project was to study how well the agent can perform in the game based on the exact way it was trained so it was never about creating an agent to have “superh- uman” capabilities in the game like those of OpenAI Five or AlphaStar.

12.2 Project Workflow

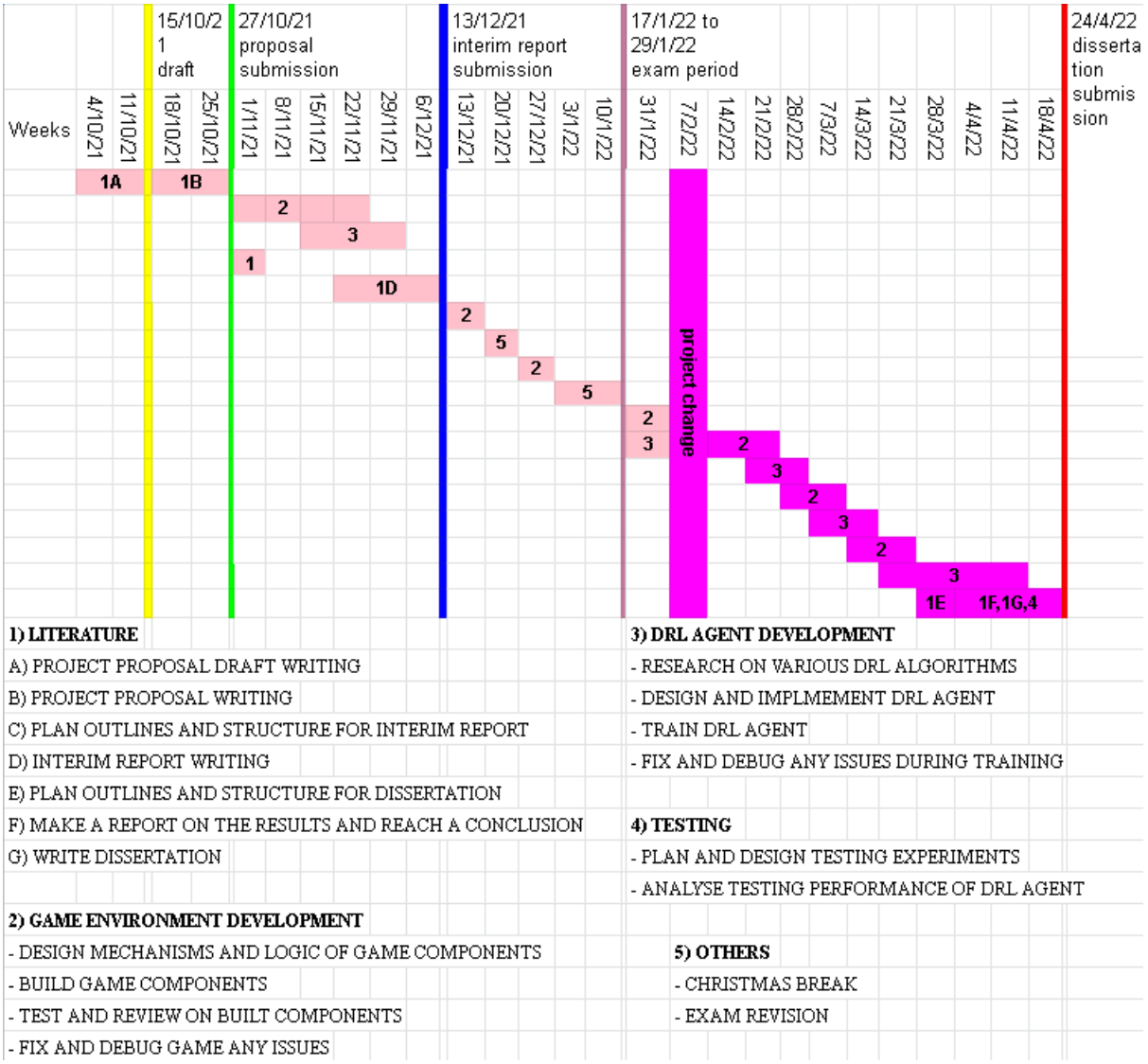

Figure 38: Gantt Chart for project workflow

According to the gantt chart above, most time spent on the Autumn semester was on learning to use Unity and developing the game environment with basic implementation of the DRL agent. A few weeks into the Spring semester, I decided to change the entire

trajectory of my project due to time constraints and the level of knowledge that I had was not sufficient for the original project idea to come to fruition. Most of what was developed in Autumn semester were irrelevant to the work that had to be done to achieve the new project aim. Therefore, I was in a way starting from scratch when I decided to change my project trajectory. The game and agent development were done alternatively with most of the training done around late March to early April 2022 and testing was done mid April alongside dissertation writing.

12.3 Biggest Challenge & Lessons

Throughout the entire 6 to 7 months duration of this project, the biggest challenge I encountered was learning to use Unity when I started to work on this project while simultaneously learning everything else regarding DRL. The lessons that I have learned through this project are three fold. First, time management is very important, high priority work should never be procrastinated and most importantly, one should never overestimate oneself. Overestimating myself is the main reason why I had to switch the trajectory of my project in the Spring semester. Second, "healthy" training performance based on statistical analysis does not guarantee a highly robust learning model, there are always some other factors to consider depending on the problem. Lastly, depth over breadth. For this project, I decided to stick with the goal of training an agent that is able to learn a considerably wide range of skills instead of focusing on mastering one primary skill. I should have experimented with training the agent to build one specific skill or a range of skills that are highly related to one another and dive deeper by making the task more challenging but only requires the agent to master the skill up to a higher capacity.

13. References

[1] A. Juliani *et al.*, 'Unity: A General Platform for Intelligent Agents', *ArXiv180902627 Cs Stat*, May 2020, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1809.02627
[2] 'What is Deep Learning?', Aug. 16, 2021. https://www.ibm.com/cloud/learn/deep-learning (accessed Apr. 20, 2022).
[3] 'Deep Neural Networks', *KDnuggets*. https://www.kdnuggets.com/deep-neural-networks.html/ (accessed Apr. 20, 2022).
[4] 'What is deep reinforcement learning?', *Bernard Marr*, Jul. 02, 2021. https://bernardmarr.com/what-is-deep-reinforcement-learning/ (accessed Apr. 20, 2022).
[5] 'Introduction — Spinning Up documentation'. https://spinningup.openai.com/en/latest/user/introduction.html (accessed Apr. 20, 2022).
[6] 'What is reinforcement learning? The complete guide - deepsense.ai'. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ (accessed Apr. 20, 2022).
[7] W. van H. PhD, 'The Four Policy Classes of Reinforcement Learning', *Medium*, Jul. 06, 2021. https://towardsdatascience.com/the-four-policy-classes-of-reinforcement-learning-38185daa6c8a (accessed Apr. 20, 2022).
[8] S. Rahman, 'Intelligent NPCs with Unity's ML Agents Toolkit - Graphics, Gaming, and VR blog - Arm Community blogs - Arm Community', Nov. 05, 2021. https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/intelligent-npcs-with-machine-learning (accessed Apr. 20, 2022).
[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. The MIT Press, 2018. [Online]. Available: https://www.pdfdrive.com/reinforcement-learning-an-introduction-2nd-edition-d185852969.html

[10] 'Law of effect', *Wikipedia*. Oct. 14, 2021. Accessed: Apr. 20, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Law_of_effect&oldid=1049833206
[11] P. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, 'Deep reinforcement learning from human preferences', *ArXiv170603741 Cs Stat*, Jul. 2017, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1706.03741
[12] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, Art. no. 7540, Feb. 2015, doi: 10.1038/nature14236.
[13] T. Zhang and H. Mo, 'Reinforcement learning for robot research: A comprehensive review and open issues', *Int. J. Adv. Robot. Syst.*, vol. 18, no. 3, p. 17298814211007304, May 2021, doi: 10.1177/17298814211007305.
[14] O. Vinyals *et al.*, 'Grandmaster level in StarCraft II using multi-agent reinforcement learning', *Nature*, vol. 575, no. 7782, Art. no. 7782, Nov. 2019, doi: 10.1038/s41586-019-1724-z.
[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, 'The Arcade Learning Environment: An Evaluation Platform for General Agents', *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, Jun. 2013, doi: 10.1613/jair.3912.
[16] 'AlphaGo'. https://www.deepmind.com/research/highlighted-research/alphago (accessed Apr. 20, 2022).
[17] Q. Chen, H. Gao, D. Wang, G. Su, and H. Zhao, *Unity Maze Game AI*. 2020. Accessed: Apr. 20, 2022. [Online]. Available: https://github.com/tavik000/MazeGameAI
[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, 'Proximal Policy Optimization Algorithms', *ArXiv170706347 Cs*, Aug. 2017, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1707.06347
[19] Michaelwolf95, *Memory-enhanced agents using Recurrent Neural Networks*. 2018. Accessed: Apr. 20, 2022. [Online]. Available: https://github.com/Michaelwolf95/Hierarchical-ML-agents/blob/5a2a61faaa8d6bcdca5f027a158cd693dd99c80e/docs/Feature-Memory.md
[20] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, 'Curiosity-driven Exploration by Self-supervised Prediction', *ArXiv170505363 Cs Stat*, May 2017, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1705.05363
[21] X. Wang, Y. Chen, and W. Zhu, 'A Survey on Curriculum Learning', *ArXiv201013166 Cs*, Mar. 2021, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/2010.13166
[22] S. Bozinovski, 'Reminder of the First Paper on Transfer Learning in Neural Networks, 1976', *Informatica*, vol. 44, no. 3, Art. no. 3, Sep. 2020, doi: 10.31449/inf.v44i3.2828.
[23] *Training Robust Agents using Environment Parameter Randomization*. Unity Technologies, 2022. Accessed: Apr. 20, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/71b121ca7ae3da493110aa7b652230cd2c3cb219/docs/ML-Agents-Overview.md
[24] 'TensorBoard', *TensorFlow*. https://www.tensorflow.org/tensorboard (accessed Apr. 20, 2022).
[25] 'What are Recurrent Neural Networks?', Apr. 07, 2021. https://www.ibm.com/cloud/learn/recurrent-neural-networks (accessed Apr. 20, 2022).
[26] B. Bakker, 'Reinforcement Learning with Long Short-Term Memory', in *Advances in Neural Information Processing Systems*, 2001, vol. 14. Accessed: Apr. 20, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2001/hash/a38b16173474ba8b1a95bcbc30d3b8a5-Abstract.html
[27] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, 'Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World', *ArXiv170306907 Cs*, Mar. 2017, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1703.06907
[28] R. L. Nilsson, J. Shih, and E. Teng, 'Machine Learning Summit: Successfully Use Deep Reinforcement Learning in Testing and NPC Development'. https://gdcvault.com/play/1026688/Machine-Learning-Summit-Successfully-Use (accessed Apr. 20, 2022).
[29] S. Narvekar, J. Sinapov, and P. Stone, 'Autonomous Task Sequencing for Customized Curriculum Design in Reinforcement Learning', in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, Melbourne, Australia, Aug. 2017, pp. 2536–2542. doi: 10.24963/ijcai.2017/353.
[30] Unity Technologies, *mlagents: Unity Machine Learning Agents*. Accessed: Apr. 20, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents
[31] 'PyTorch documentation — PyTorch 1.11.0 documentation'. https://pytorch.org/docs/stable/index.html (accessed Apr. 20, 2022).
[32] M. Plappert *et al.*, 'Multi-Goal Reinforcement Learning: Challenging Robotics Environments and

Request for Research', *ArXiv180209464 Cs*, Mar. 2018, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1802.09464
[33] 'Ingredients for Robotics Research', *OpenAI*, Feb. 26, 2018. https://openai.com/blog/ingredients-for-robotics-research/ (accessed Apr. 20, 2022).
[34] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, 'A Survey of Deep Reinforcement Learning in Video Games', *ArXiv191210944 Cs*, Dec. 2019, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1912.10944
[35] V. Mnih *et al.*, 'Playing Atari with Deep Reinforcement Learning', *ArXiv13125602 Cs*, Dec. 2013, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1312.5602
[36] 'Dota 2', *Wikipedia*. Feb. 25, 2022. Accessed: Apr. 20, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Dota_2&oldid=1074020661
[37] OpenAI *et al.*, 'Dota 2 with Large Scale Deep Reinforcement Learning', *ArXiv191206680 Cs Stat*, Dec. 2019, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1912.06680
[38] D. Cattelan, 'Evolution of AI in Video-Games from 1980 to Today', *Medium*, Dec. 06, 2019. https://medium.com/@DylanCa_/evolution-of-ai-in-video-games-from-1980-to-today-e3344acaed4c (accessed Apr. 20, 2022).
[39] 'OpenAI Five - A team of 5 Algorithms is Beating Human Opponents in a Popular Game', *Analytics Vidhya*, Jun. 26, 2018. https://www.analyticsvidhya.com/blog/2018/06/openai-five-a-team-of-5-algorithms-is-beating-human-opponents-in-a-popular-game/ (accessed Apr. 20, 2022).
[40] D. Silver *et al.*, 'Mastering the game of Go with deep neural networks and tree search', *Nature*, vol. 529, no. 7587, Art. no. 7587, Jan. 2016, doi: 10.1038/nature16961.
[41] D. Silver *et al.*, 'Mastering the game of Go without human knowledge', *Nature*, vol. 550, no. 7676, Art. no. 7676, Oct. 2017, doi: 10.1038/nature24270.
[42] 'AlphaStar: Mastering the real-time strategy game StarCraft II'. https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii (accessed Apr. 20, 2022).
[43] J. Suarez, Y. Du, P. Isola, and I. Mordatch, 'Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents', *ArXiv190300784 Cs Stat*, Mar. 2019, Accessed: Apr. 20, 2022. [Online]. Available: http://arxiv.org/abs/1903.00784
[44] M. McPartland and M. Gallagher, 'Reinforcement Learning in First Person Shooter Games', *Comput. Intell. AI Games IEEE Trans. On*, vol. 3, pp. 43–56, Apr. 2011, doi: 10.1109/TCIAIG.2010.2100395.
[45] D. Pierigli, L. A. Ripamonti, D. Maggiorini, and D. Gadia, 'Deep Reinforcement Learning to train agents in a multiplayer First Person Shooter: some preliminary results', in *2019 IEEE Conference on Games (CoG)*, Aug. 2019, pp. 1–8. doi: 10.1109/CIG.2019.8848061.
[46] O.-O. D. Science, 'Watch: Imitation Learning: Reinforcement Learning For The Real World', *Medium*, Sep. 12, 2019. https://odsc.medium.com/watch-imitation-learning-reinforcement-learning-for-the-real-world-65200d7288e6 (accessed Apr. 20, 2022).
[47] C. Renman, 'Creating Human-like AI Movement in Games Using Imitation Learning', KTH ROYAL INSTITUTE OF TECHNOLOGY, Stockholm, Sweden, 2017. [Online]. Available: https://kth.diva-portal.org/smash/get/diva2:1120710/FULLTEXT01.pdf
[48] J. Schell, *The Art of Game Design: A Book of Lenses*, Second. 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742. 20140929: CRC Press, Taylor & Francis Group, 2015.
[49] 'Starter Assets - Third Person Character Controller | Essentials | Unity Asset Store', *Unity Asset Store*, Jun. 09, 2021. https://assetstore.unity.com/packages/essentials/starter-assets-third-person-character-controller-196526 (accessed Apr. 21, 2022).
[50] 'Pac-Man', *Wikipedia*. Apr. 18, 2022. Accessed: Apr. 21, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=1083321408
[51] Unity Technologies, 'Unity - Scripting API: NavMeshAgent'. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/AI.NavMeshAgent.html (accessed Apr. 21, 2022).
[52] Unity Technologies, 'Unity - Scripting API: NavMesh'. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/AI.NavMesh.html (accessed Apr. 21, 2022).
[53] Unity Technologies, 'Unity - Scripting API: Random.insideUnitSphere'. https://docs.unity3d.com/ScriptReference/Random-insideUnitSphere.html (accessed Apr. 21, 2022).

[54] Unity Technologies, 'Unity - Scripting API: AI.NavMeshAgent.SetDestination'. https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.SetDestination.html (accessed Apr. 21, 2022).
[55] Unity Technologies, 'Unity - Scripting API: MonoBehaviour.Update()'. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/MonoBehaviour.Update.html (accessed Apr. 21, 2022).
[56] Unity Technologies, 'Unity - Scripting API: GameObject.SetActive'. https://docs.unity3d.com/ScriptReference/GameObject.SetActive.html (accessed Apr. 21, 2022).
[57] Unity Technologies, 'Unity - Scripting API: Object.Instantiate'. https://docs.unity3d.com/ScriptReference/Object.Instantiate.html (accessed Apr. 21, 2022).
[58] Unity Technologies, 'Unity - Scripting API: Object.Destroy'. https://docs.unity3d.com/ScriptReference/Object.Destroy.html (accessed Apr. 21, 2022).
[59] 'Class Agent | ML Agents | 2.3.0-exp.2'. https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/Unity.MLAgents.Agent.html (accessed Apr. 21, 2022).
[60] Unity Technologies, 'Unity - Scripting API: Ray'. https://docs.unity3d.com/ScriptReference/Ray.html (accessed Apr. 21, 2022).
[61] Unity Technologies, 'Unity - Manual: Tags'. https://docs.unity3d.com/Manual/Tags.html (accessed Apr. 21, 2022).
[62] *Raycast Observations*. Unity Technologies, 2022. Accessed: Apr. 21, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/71b121ca7ae3da493110aa7b652230cd2c3cb219/docs/Learning-Environment-Design-Agents.md
[63] Unity Technologies, 'Unity - Scripting API: Physics.Raycast'. https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (accessed Apr. 21, 2022).
[64] Bonsai, 'Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions', *Medium*, Nov. 16, 2017. https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0 (accessed Apr. 21, 2022).
[65] *Rewards Summary & Best Practices*. Unity Technologies, 2022. Accessed: Apr. 21, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/71b121ca7ae3da493110aa7b652230cd2c3cb219/docs/Learning-Environment-Design-Agents.md
[66] *Training ML-Agents*. Unity Technologies, 2022. Accessed: Apr. 21, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/71b121ca7ae3da493110aa7b652230cd2c3cb219/docs/Training-ML-Agents.md
[67] 'Class Academy | ML Agents | 2.3.0-exp.2'. https://docs.unity3d.com/Packages/com.unity.ml-agents@2.3/api/Unity.MLAgents.Academy.html (accessed Apr. 21, 2022).
[68] *Training Configuration File*. Unity Technologies, 2022. Accessed: Apr. 21, 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/71b121ca7ae3da493110aa7b652230cd2c3cb219/docs/Training-Configuration-File.md
[69] R. G. Mantovani, T. Horváth, R. Cerri, S. B. Junior, J. Vanschoren, and A. C. P. de L. F. de Carvalho, 'An empirical study on hyperparameter tuning of decision trees', *ArXiv181202207 Cs Stat*, Feb. 2019, Accessed: Apr. 21, 2022. [Online]. Available: http://arxiv.org/abs/1812.02207
[70] J. T. Kim and S. Ha, 'Observation Space Matters: Benchmark and Optimization Algorithm', *ArXiv201100756 Cs*, Nov. 2020, Accessed: Apr. 21, 2022. [Online]. Available: http://arxiv.org/abs/2011.00756
[71] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, 'Neural Architecture Optimization', *ArXiv180807233 Cs Stat*, Sep. 2019, Accessed: Apr. 21, 2022. [Online]. Available: http://arxiv.org/abs/1808.07233