An AI Agent that plays FPS games

6CCS3PRJ Final Project Report

Author: Teoh Kai Shun

Student ID: 20016655

Supervisor: Prof. Dr. Agi Kurucz

Course : Computer Science with Intelligent Systems

April 2023

Acknowledgement

I thank myself for believing in me when nobody could.

I thank I for pulling myself through the toughest time.

I thank me for being me, staying true to what I want to do, and trying my very best.

I thank my supervisor for understanding me, and working with me, despite the fact that I have a atypical work style.

Table of Content

- Abstract

- Introduction
  - Relationship of AI and Games
  - Recent developments of AI in Games and its Significance
    - AlphaGO
    - The Director in Left 4 Dead series
    - Open AI 5
  - Aims and Objectives

- Background Knowledge
  - The Game of Counter Strike : Global Offensive (CSGO)
    - Skills demanded by game
  - Reinforcement Learning
  - Markov Decision Process (MDP)
  - Use of Neural Network in Reinforcement Learning
  - Deep Deterministic Policy Gradient (DDPG)
  - Hindsight Experience Replay (HER)
  - Goals and Universal Value Function Approximator
  - Actor Critic
  - Target Networks in Actor Critic
  - Asymmetric Actor Critic Algorithm (AAC)
  - Computer Vision
  - YOLOV5
  - Game State Integration
  - Game Interface
  - Open AI Custom Environment

- Related Studies
  - AI in First Person Shooting games
  - AI in CSGO

- Design of Project
  - Intuition of Approach
    - Hypothesis
  - Formularising CSGO as a Reinforcement Learning
    - Why Reinforcement Learning
    - Designing the solution system.
  - 4 key entities in our design
    - Agent
    - Virtual Environment
    - Player
    - Spectator

- Implementation
  - Implementation of Entities
    - Spectator
    - Player
    - Agent
      - DDPG
      - HER
      - AAC
    - The Virtual Environment
      - Physical Setup
      - Initialisation step
      - Restarting the Game
      - Interaction process
      - Evaluation process
      - States from Raw Information
      - Encoded Actions to Player action
  - Implementation of Relevant Methodologies

- Results and Critic
  - Results
  - Critic

- Limitations

- Future Endeavours

- Appendix
  - Code PDF

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service.

I confirm this report does not exceed 25,000 words.

Teoh Kai Shun, April 2023

Abstract

Artificial Intelligence has existed in games since as early as 1978, and games have proved themselves to be suitable environments for Artificial Intelligence to train in given its cost and ease of use, as compared to having the Artificial Intelligence train in the physical world. In this report, I tackle the game of Counter Strike : Global Offensive, which is a modern and popular First Person Shooting Game. Specifically, I attempt to train an Artificial Player that can play the game well, while at the same time exhibit human-like play style that is not expected from the current state of the art Bots that Counter Strike : Global Offensive uses. I will be using Reinforcement Learning in my approach, specifically using an Asymmetric Actor-Critic, Deep Deterministic Policy Gradient algorithm, with ϵ-greedy exploration strategy.

I : INTRODUCTION

Relationship of AI and Games

The earliest use of AI in commercial gaming dates to 1978, in the game of Space Invaders, an iconic fixed-shooter games. In these times, the Artificial Intelligence was very Naive, using stored patterns to direct movement. In 2000, when Artificial Intelligence was more refined, its application to games changed the nature of the games perceived by the gaming community, via the game The Sims, which is an iconic game allowing the player to assume the role of 'god' in the simulation world. The use of Artificial Intelligence allowed different objects to affect the NPCs' (Non Player Characters) behaviour and relationship. [1] This transition from mechanical arcade games, to a more interactive-styled game with social elements was brought possible by AI, and made games appealing to a wider public, and as such it can be said that the development of AI can bring about a development in games. Moving forward to recent times, likewise, games have been a popular choice of platform to not only train, but benchmark an Artificial Intelligent Player's performance. This is easily backed by the significance of AlphaGo[2] winning against a professional Go player in the field of Artificial Intelligence. Finally, Artificial Intelligence now aims to overcome Multi-Player Online Games, with the most notable example of Open AI 5 winning a professional team in Dota 2 [3]. As I can see, games and Artificial Intelligence compliments each other. In the near future, I can expect to see games that has intelligent empathetic characters who will interact with the player and optimise the player's experience while the player is playing.

Recent developments of AI in Games and its Significance

Up until recently, majority of academics work regarding AI in games have been focused on traditional card games and traditional board games, with the purpose of beating expert human players and becoming an unbeatable player. However, due to the recent increase in the quality, diversity and pervasiveness of games, the role of AI in games have now increased — for example it could be use in generating realistic game graphics [27]. With addition to the fact that the value of worldwide computer and video games is estimated to be $USD25 billion annually — and it is predicted to grow rapidly over the next decade, game developers can look into designing and adapting AI into their games, so as to provide a more entertaining and immersive game experience, thereafter make more profit. The profound profit-making ability of AI in games makes it a crucial player in the game industry for many game developers. To further elaborate, I will like to mention some of the use of AI in games that I felt was significant or interesting below, and its effect.

1)AlphaGO

AlphaGo is arguably one of the most significant milestone in both the study of AI in games, and the study on AI. Before AlphaGo's victory against a professional Go player on 9th March 2016, there was much doubt in the ability of AI to beat professional players in traditional board games, even though it has achieved remarkable success. For example, in 1997, Deep Blue won again the then-world chess champion Garry Kasparov with a 3½ - 2½ for Deep Blue [28] . The arguments before AlphaGo's victory has always revolved around the game of Go, with the public opinion being that the ability of AI to beat professional players in board games should only be acknowledge when the AI beat a professional player in one of the hardest board games, Go. This began decades of study on AI in Go.

The difficulty of Go lies, especially for AI, lies in its complexity. Despite decades of work on AI in Go, the best AI agent could only play at an amateur level. This is due to the fact that classical AI methods tackled games by holding onto a game tree and pruning it. The issue with this approach is that as the game gets more complex, the size of the game tree exponentially increases. Go has 10^170 possible board configurations, and using game tree can cause a bottleneck due to intractability. In addition to the configurations complexity, the game of Go requires multiple layers of strategic thinking to win. This meant that a new type of AI was required to tackle the problem [29]. OpenAI decided to to take the approach of Reinforcement Learning approach, combining an advanced search tree with an Actor-Critic algorithm with Neural Networks to deal with the complexity faced by traditional method. This approach eventually lead them to the victory on 9th March 2016. Following its victory, OpenAI dived deeper into the study of AI in games, moving from traditional board games to popular video games, developing more sophisticated AI like Open AI 5 for Dota 2 [30] and AlphaStar for StarCraft II [31], showing an increased confidence and interest in AI. I want to note that this transition from traditional board games to popular video games was made possible by the success of AlphaGo, which solved the problem of scalability for traditional approaches to AI in games. Additionally, a documentary [video link 3] was made in tribute the AlphaGo's victory, which has 33 million views on YouTube.

2)The Director in Left 4 Dead series

The Left 4 Dead is a single-player and multiplayer cooperative survival horror FPS game developed by Valve Corporation and Turtle Rock Studios [34]. The game consist of Survivors — who can be controlled by either Player of some AI — and mobs that I briefly refer to as the Infected — which is controlled by the game's AI. The goal of the Survivors is to help each other to safety while avoiding attacks from the Infected. If a Survivor is attacked by the Infected, the Survivor will become 'infected' and now his/her goal will be to stop the Survivor. There have been mentions of the use of game-AI above, but that is not to be mistaken for the Director. Unlike traditional AI, where the AI takes a more central part int the game like being opponents of human players, the Director takes a more peripheral role, which is to provide an immersive experience for the human player by dynamically mutating the gameplay.

The Director mutates the gameplay dynamically by studying the current gameplay and then from there decide what changes will make the game more interesting to the players. For example, instead of using preset spawn points for the Infected, the Director places enemies in varying positions and numbers based upon each player's current situation, status, skill, and location, creating a new experience for each play-through [33]. The most interesting capabilities of the Director is available in Left 4 Dead 2, which is the implementation of Dynamic Map Design. Dynamic Map Design allows the Director to measure how well the Survivors are doing, and change the map accordingly. These changes can include small path tweaks, object and item spawns, weather patterns, even re-pathing of entire events, creating an unique experience for every match, making the game captivating and challenging.

This is one of the new generation AI that I have aforementioned. The Director not only introduced a new role of AI in games, all while achieving its purpose of increasing the complexity of the game by introducing dynamic changes to live gameplay. This new approach to AI in games showed that AI in games is not necessary restricted to its stereotypical role of being the opponent; and that it has now a more versatile role, and this can potentially open up another source of revenue for many game developers. For example, Dota 2 implemented a in-game Personal Assistant known as Dota+ which aids the player in live games by recommending Heroes with respect to Heroes chosen by enemy team, recommending items to buy in-game, and even providing mechanical skills assistance like pulling creeps, which used to require memorising certain timing. It even has a live critic that tells the player how he is performing with respect

to the expected performance calculated from past datas. By charging a monthly subscription fee and offering packages for the use of Dota+, Dota 2 has found a new stream of revenue.

3) Open AI 5

Lastly, I want to briefly talk about Open AI 5, as it is undoubtedly the root motivation for this project. The first time I was exposed to AI in games was through Open AI 5 and it left a very significant impression on me. On April 15 2019, OpenAI 5 beat the world champions in an esport game, having won 2 back-to-back games against the then Dota 2 world champion — OG [32]. In addition, it is the first AI to beat esport pros on a livestream — previously, both OpenAI 5 and DeepMind's AlphaStar had beaten good professional players in private game session, but always lost during live pro matches. This made it extremely significant, as the match on April 15 was held live after the Finals of the Dota 2 competition — The International 9 — meaning that billions of esports fan watched the game. With such a huge exposure and exceptional results, many, including me, was convinced of the potential of AI in games.

Like my project, Open AI 5 tackled the game of Dota 2 as a Reinforcement Learning problem, and wanted to use Dota 2 to work on a problem that felt out of reach for existing deep reinforcement learning algorithms. After tackling the problem, they realised that it was not the lack of sophistication of current methods that is limiting the potential of Reinforcement Learning, but scalability issues with Reinforcement Learning. As such, majority of the research dealt with scalability. To deal with scalability, Open AI 5 created a system called Rapid which runs Proximal Policy Optimisation at a significantly larger scale. The results exceeded expectations, and OpenAI produced world class Dota bots without hitting any fundamental performance limits. However, OpenAI notes that the performance of the bots comes at the cost of massive amount of experience, which can be impractical in many cases where running simulations/training can be costly. To give a perspective, OpenAI 5 consumed 800 petaflops/s-days and experienced about 45,000 years of Dota 2 self-play over 10 realtime months, for an average of 250 years of simulated experience per day. One can imagine the amount of computation power required to run this algorithm in a day, and the amount of hardware required to support the training. This is incredibly costly when compared to training with other forms of Artificial Intelligence like Supervised Learning and Unsupervised Learning. However, its performance on this task remains unrivalled in comparison with the other forms of learning aforementioned.

Aims and Objectives

In this project, I aim to show that the current state of Art Artificial Intelligence Methodologies are able to exhibit better performance than the current bot implementation used in Counter Strike : Global Offensive, under a constraint situation, specifically in a 1 versus 1 after the bomb is planted, where the agent is playing as Terrorist, all while retaining human-likeness play style. By human-likeness, I mean that the agent's play style is not predictable as a bot's play style will be [4]. I intend to measure the agent's performance by using its performance against a bot, then I will personally play against it, for 20 rounds, measuring its win rate. To measure its human-likeness, I will perform a Turing Test by having people watch 5 games of 3 rounds played by 2 different entity {Human, Agent}, then have the spectator map the gameplay to respective entity player. A preferable outcome will be one where I get a high average error rate.

At the same time, I want the agent play as a human player would, as much as possible. This is so as I want our agent to be able to play at the same advantage as a human player, so that I can truly showcase that the agent is able to play on par or better than a human player. To play on the same advantage as a human player, the agent will need to take in inputs from 'human' sources like the monitor and speaker, and outputs like keyboard, mouse and noise. However, due to limitation in time and expertise, I have omitted audio I/O, and have opt to implement a Naive Visual System for the agent. By Naive I mean that it is only able to identify the location of the enemy on the screen, but it is unable to geo-locate the enemy's location just from the image. At the same time, it does not have the ability to geo-locate itself from image data alone.

II : BACKGROUND KNOWLEDGE

The Game of Counter Strike : Global Offensive

Counter-Strike: Global Offensive (CS:GO) is a 2012 multiplayer tactical first-person shooting game developed by Valve and Hidden Path Entertainment [23]. First-person shooter (FPS) is a sub-genre of shooter video games centred on gun and other weapon-based combat in a first-person perspective, with the player experiencing the action through the eyes of an antagonist or protagonist which is armed, and then controlling the player character in a three-dimensional space [24].

From here on out, CSGO will be used in abbreviation for Counter Strike:Global Offensive. CSGO have multiple game modes, but in our case, I consider only the classical game mode, which highlights both the mechanical aspects and the strategical aspects of the game. In the classical game mode there are two opposing teams, the Terrorists and the Counter-Terrorists, each made up of 5 players. These 2 teams compete for the best of 30 rounds, with team switch occurring after the 15th round. Terrorist win the round by either eliminating all members of the opposing team; or by planting the bomb and waiting for it to explode. Counter Terrorist win by either eliminating all members of the opposing team before they can plant the bomb; or by defusing the bomb if it is planted. At the end of each short round, players are rewarded based on individual and team performance with in-game currency to spend on other weapons or utility in subsequent rounds. Winning rounds generally rewards more money than losing does, and completing map-based objectives, including killing enemies, gives additional cash bonuses.

For our project, I utilise a reduced version of the classical game mode, and I codename it 'the CSGO problem' . Specifically, I ignore the economical aspect of the classical game mode, I only focus on a specific situation of the game and that I put restrictions on the utilities being used. I talk more about these below.

The classical game mode have an element of economical aspect to it, as teams and players need to control their economy throughout the game. As mentioned above, a lose leads to lesser reward money, which increases with consecutively losses, and they are other actions that lead to bonus money — eg getting a kill. A strong economy throughout the game is desired, but players at the same time need to invest money into rounds to secure the win. This then captures the economical aspect of the game, which is balancing the tradeoff of long term and short term goals. However, I do not concern our agent with this aspect as it will further complicate the task. Such an aspect could also be tackled with a simple heuristic or policy mapping, but I ignore it as I want to focus solely on the strategic gameplay of the agent. To circumvent this economical concern, I configure the game such that both agent and enemy has no buying limit, and that each round resets such that the result of the previous round does not affect the next round.

I have mentioned above that I focus on a specific situation in the classical game mode, and that is the 1 Versus 1 Post plant situation. In this situation both teams only have 1 remaining agent left alive, meaning that there is only 2 player.The Terrorist player has just planted the bomb and now has to defend it. The Counter Terrorist Player is not in the bomb site and thus have to navigate to the bomb site and defuse the bomb. The bomb explodes in 40 seconds. This situation is perfect for my task as it captures sufficient strategical aspects of the game, while at the same time remains as a problem that is not too complex to be solved. Given that there is not much open-source research done on an AI bot agent, the simplicity of the situation makes it a great starting point to develop an AI bot agent. Furthermore, the 1v1 Post Plant situation demands quite a significant strategical approach as it involves outplaying your opponents. Disregarding better mechanical aim skills, which dictates how a duel between players goes, outplaying your opponent will require one to outsmart your opponent, which means to be able to predict how your opponent will move and account for that in your plan. The act of accounting for enemy movement captures the strategical aspects of the game. Additionally, at advanced level of CSGO, especially in the professional scene, the importance of strategy in 1v1 Post Plant situations are more pronounced, as can be seen in this [video link link 1]. This further prove to show that even in simple situations like 1v1 Post Plant Situations, there still exists strategies that can be played to favour chances of winning.

With regards to the utilities aforementioned, I mean the use of grenades. In CSGO, players can use utilities like smoke grenades, hand grenades or molotov in the game. However, the use of these utilities highlights a higher form of strategical aspect of the game as it introduces elements of change in the environment, and this elements of change in question can be utilised in strategy. For example, the smoke grenades creates a cloud of smoke that blocks player vision, and restricting visual information access, thus increasing uncertainty of the situation, which is the elements of change in question. I believe that this higher form of strategical aspect should not be pursued yet as it might be too complex to deal with.

Skills demanded by game

I have talked about the different aspects of the game -- economical, strategical, mechanical -- and now i want to talk about the skills demanded by these aspects. For ease of understanding, i will class the skills according to which aspects it is demanded by.

The first set of skills is called Mechanical skills, and it concerns the quality of control of the avatar (in game character) by the player. This particular set of skills are mainly dependent on reaction speed, spatial awareness, hand-eye coordination and precision of mouse and keyboard control. Mechanical skills are significant in game as the game is ultimately a first-person shooting game, and as such will inevitably involve a lot of duels. Outcome of duels can be dependent on Mechanical and Strategical skills, but in most cases Mechanical skills is the significant factor. This is so as duelling involves aiming and gunning down the enemy, and as such requires good moues control and good hand-eye coordination. At the same time, player can strafe [43] and counter strafe [44] -- which is a high level task that involves good keyboard and mouse control, and good hand-eye coordination -- to make it harder for opponent to aim at the avatar, while at the same time minimising recoil and holding the *crosshair on the enemy avatar. At the same time it involves spatial awareness, mainly sound localisation [46] to maximise usage of sound cues in locating enemy avatar locations and movements.
***crosshair is a logo that the game provides to show where the player is aiming at. It acts as a reference point for aiming. Here is a picture with crosshair [45].**

The second set of skills here is called Strategical Skills, and it concerns the planning , coordinating and execution of a strategic play. A strategic play in CSGO will involve planning a path [47] and a sequence of movements along this path that each ally player will make, and at higher levels pre-empting possible events when navigating through said path, like pre-empting a potential duel at a given location with the reasoning that there is a high chance of encountering an enemy player at said location. Strategical Skills largely rely upon Mechanical Skills for execution, and for deliberation is requires Strategical skills, which is dependent on higher-order thinking, spatial awareness, good decision making. For a plan to be successfully executed, it must account as much movements of the made by the enemy team, and such information come from higher-order thinking [48] which can be understood as the reasoning to predict the enemy's current strategy. Spatial awareness is also required as it gives information about enemy movements during execution of the plan. This allows the player to gauge the quality of the plan by cross-checking events pre-empted by the plan against the actual events happening. This then allow the player to determine whether a new plan should be made, or that they should follow through with it; and this is a crucial decision to make as a successful execution of plans can lead to winning games. Lastly, due to the large state and action space of the game and its multi-agent and continuous characteristics, many decisions needs to be made constantly about the next action or plan to take when playing the game. As aforementioned, these decisions are crucial to winning games. They are also chronically dependent upon one another due to the sequential nature of the game with respect to time. As such decisions made manifests decisions to be made, like a Markov chain, and I have to ensure that I make decisions that puts us in a position such that the options available for future decisions are preferred. This is achieved by good decision skills.

The last set of skills is called Economical skills, and since it is not my focus, i will briefly summarise it. It involves making smart economic decision after each round in the game to manage the team's economy, which is the money they have to buy guns and utilities in each round.

Reinforcement Learning

Reinforcement Learning is a type of Machine Learning, and it emphasises on learning what to do in a given situation so as to maximise a numerical reward [5]. Unlike supervised learning, the agent is not 'told' what to do, but instead must attempt a trial-and-error process to learn the right action to take in a given situation, or more formally state. In Reinforcement Learning, the learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. There are 6 main elements in Reinforcement Learning : Agent, Environment, Value Function, Policy, Reward Signal, and optionally, Model.

The Agent will be the entity that is being controlled by the Artificial Intelligence.

The Environment will be the space that the Artificial Intelligence will interact with via the Agent.

The Reward Signal communicates a numerical reward to the Agent every time it performs an action

The Value Function is responsible for mapping situations, known as states, to its respective worth, or value, to the Agent. The value being measured here can be thought of as the likelihood of the agent to achieve its goal(s) from this state.

The Policy maps states to actions that the agent should take. Policy can be deterministic or non-deterministic. Deterministic policies will always choose the same action given the same states, while a Non-Deterministic one can choose different actions given the same states. This is so as unlike Deterministic Policies that maps a state to a singular action, Non-Deterministic Policies maps states to a probability distributions over a subset of actions.

The Model can be thought of as an abstract simulation the agent has of the environment. The model allows the agent to 'visualise' the effects of its action. It is to be noted that the Model need not perfectly simulate the Environment, but it is crucial that the Model allows the agent to make some sort of inference about the Environment that it is in.

Majority of Reinforcement Learning methods can be generally classified into 2 types, Value based method — which focuses on improving the Value Function — and the Policy based method — which focuses on improving the Policy [38]. There is a separate class known as the model based method — which focuses on learning the model of the environment, and plan with the model to navigate the environment — but I will not elaborate any further due to their irrelevance.

Due to recent advances in technology, especially the adaptation of Neural Networks into algorithms, methods of Reinforcement Learning can also be can be classified under two classes depending on the algorithm's implementation. If the methods uses Neural Network in its algorithm, it is classified as a modern reinforcement learning agent. On the other hand, if it uses more traditional implementation for its algorithm, it is classified as a classical reinforcement learning method [40]. The implementation difference within the algorithm usually concerns the Value Function and the Policy. Traditional implementation of these methods uses some sort of data structure -- like a dictionary in python -- to store the mapping between each individual state and their respective value. This same line of implementation is done on the policy, but this time storing the mapping of each individual states to an action. It is easy to see how such implementation can face scalability issues as problems gets larger and more complex. The first scalability issue faced is the intractability of the Value Function and Policy. If state space is continuous, it is hard to accurate create a discrete mapping of discrete states to value or action, as quantising the state space can result in either information loss or exponential increase in state space after quantising. The same can be said for only the Policy, specifically, when faced with an continuous action space. Inaccuracy of the traditional implementation of Value Function and the Policy thus degrades as the problem becomes larger and more complex, and this effectively hurts the performance of the algorithm in progressively harder problems. The second scalability issue lies with training. Assuming that the traditional implementations are effective, it is still necessary for the agent to visit all distinct states sufficiently to achieve satisfactory performance. This can be difficult due to the probability distribution of states seen by the learning agent during interaction. As problems gets more complex, state space increases, causing the probability distribution to be sparse, meaning that the probability associated to each state is significantly small, and even worse for corner cases. This can make training difficult as it can be hard to ensure that the agent has visited every state sufficiently many times, and this is usually overcame with long training periods, which brings about increased cost and the risk of instability throughout long training periods. With the adaptation of Neural Networks in the implementation, the issue of intractability was better tackled, and the class of Reinforcement Learning Algorithm which uses Neural Network are known as modern Reinforcement Learning Algorithm. The use of Neural Network in Reinforcement Learning will be further elaborated below. An example of a classical Value based methods are Value Iteration [5], and an example of a modern Value based method will be Deep Q Networks [39]. An example of a classical Policy based method will be Policy Iteration, and an example of a modern Policy based method will be the one that is used by me, Deep Deterministic Policy Gradient.

Reinforcement Learning usually involves 2 main processes, interaction and evaluation. During interaction, the agent follows its current policy to interact with the environment, generating a sequence of events. These sequence of events are then used in the evaluation step to update the policy to improve performance. The order of these steps really depends on the implementation. In our implementation, I use

a Off-policy Reinforcement Learning, meaning that I do not learn while interacting, and instead I collect the sequences of events during interaction for some time, then use these collected sequences to evaluate our policy. To briefly elaborate, in On-Policy Reinforcement Learning the agent learns and evaluates in a single interaction, where in Off-Policy Reinforcement Learning, the interaction and evaluation step is separated.

Markov Decision Process

Markov Decision Process are usually used to formalise a Reinforcement Learning Problem. To define a Markov Decision Process, I need to define 5 items: State S , Action A , Reward Function R , Transition Function p and a Policy π .

S **: The set of states that the agent will encounter in the environment**
A **: The set of actions the agent can take in the environment**
R **: The reward receive for taking a given action, in a given state.**
ρ **: The function that determines the next state, given an action and a state**
π **: The function that determines that action the agent will take, given a state.**

The dynamics of Markov Decision Process can be summed up with both the reward function and the transition function. I define the Transition Function and The Reward Function :

$$\rho(s'|s,a) := Pr(S_t = s'|S_{t-1} = s, a_{t-1} = a)$$

$$R(r|s,a) := Pr(R_t = r|S_{t-1} = s, a_{t-1} = a)$$

<u>It is important to note that both the Transition Function and the Reward Function defines a probability distribution as the environment can potentially be Non-Deterministic or that there might be other agents affecting the state of the environment.</u>

After formularising Markov Decision Process, lets show how I formulate it as a Reinforcement Learning problem. Though the Markov Decision Process strictly outlines that the next state always encode all information about its preceding states, if I relax this constraint, I can use it to formularise Reinforcement Learning Problem. Reinforcement Learning is done via interaction between the agent and the environment, and from their interaction, I can generate a sequence of (state, action, reward) tuple over varying times. Furthermore, since the next preceding state is to some extent dependent on the current state and action, and by induction its initial state, I can maintain the relaxed-constraint that the preceding state retains some amount of information from its preceding states. To fully formulate it as a Reinforcement Learning Problem, I implement a suitable Reward Function that will allow the agent to learn a decent policy.

Use of Neural Network in Reinforcement Learning

Naive Reinforcement Learning solutions like Finite Markov Decision Process solvers usually approach Reinforcement Learning problem by keeping track of a mapping of states to their respective 'learnt' value function from interaction, and using the value function, a policy which maps states to action is created. These solutions are simple and intuitive, but they usually face traceability and scalability issue. This is because the growth of both state space and action space can exponentially increase the computation time and memory usage of these solutions, making it impossible to keep track of the value function and policy. As such, I want the agent to be able to generalise value functions from its experience and it just so happens that Neural Networks are great function approximator [9]. This then makes it possible for us to train a Neural Network that can generalise the values of state is has not seen, using the state it has experienced before. From these values, the Agent can then create a policy, and thus allowing the agent to tackle Reinforcement Learning Problem more adequately despite of the large state or/and action space. In CSGO, the state space for a 2 player game is calculated to be around 10^43, from the calculations I have made based on the environment I have created to represent CSGO. As such, for this project, I will need to use Neural Networks to approximate the Value Function.

The use of Neural Network is definitely a need in my case. By running some set space calculation, there is approximately 10^43 distinct states in a 1V1 game in CSGO, in the map Dust 2. Intractability and Computational Complexity becomes apparent bottlenecks if I were to take the Classical approach of keeping a table that maps each state to an action. Furthermore, it is impossible for the agent to

sufficiently explore all distinct states. This then necessitate the use of Neural Network. The use of Neural Network in this case is also suitable due to the similarity that exist between subsets of states in the state space. This then allow the Neural Network to approximate a unseen state's value based on similar states that it has seen before.

There are 2 approaches to creating a Neural Network that is capable of approximating the Value Function and the Policy Function in a game. The first approach is called the Gradient-based method, which include backward propagation algorithm. The second approach is called Non-Gradient based method, which includes stochastic processes in which changes to a neural network are proposed and accepted with certain probabilities [49]. These two approaches differ in the way they evaluate gradients of the loss function, Gradient-based method explicitly evaluating it with backward propagation, while the non-gradient method implicitly evaluates it through some stochastic process.

Deep Deterministic Policy Gradient

**Deep Deterministic Policy Gradient is an actor-critic Reinforcement Learning algorithm that learns a deterministic continuous action policy [8]. It takes on a off-policy and Gradient based approach. The algorithm maintains two neural
of networks: the policy (also called the actor)**

$$\pi_\theta : S \rightarrow A \text{ (with neural network parameters } \theta)$$

and a Q-function approximator (also called the critic)

$$Q^{\pi}_{\varphi} : S \times A \rightarrow R \text{ (with neural network parameters } \varphi).$$

During training, episodes are generated using a noisy version of the policy (called behavioural policy),

e.g.$\pi_b(s) = \pi(s) + N(0,1), where\ N\ is\ Normal\ noise.$

The transition tuples encountered during training are stored in a replay buffer [1].

Transition tuple :(s_t, a_t, r_t, s_{t+1})

Training examples sampled from the replay buffer are used to optimise the critic. By minimising the Bellman error loss the critic is optimised to approximate the Q-function.

$$L_c = (Q(s_t, a_t) - y_t)2, where\, y_t = rt + \gamma Q(s_{t+1}, \pi(s_{t+1})),$$

The actor is optimised by minimising the loss :

$$L_a = -E_s[Q(s, \pi(s)] .$$

**The gradient of L_a with respect to the actor parameters is called the deterministic policy gradient and can be computed by back-propagating through the combined critic and actor networks.
To stabilise the training, the targets for the actor and the critic y_t are computed on separate versions of the actor and critic networks, which change at a slower rate than the main networks.**

I have decided to use Deep Deterministic Policy Gradient (DDPG)t Algorithm due to both the large action space available to the agent, and the success it had in [8]. Value-based method are unsuitable in large action space [20], and as such is not chosen to be used, while DDPG is proven to be successful in large action space like those of a 6-axis Robt Arm. This is so as Value-based methods requires a good degree of exploration in the action space before it is able to find the optimal action. However, when the action space is large, such exploration is sometimes impossible, and possibly unnecessary there might exist redundant actions that should not be explored at all. However, with Policy-based method like DDPG, I need not explore the entire action space, as I are focused on finding the best action to take, making our search in the action space to be guided by some heuristic, which in our case will be the action that maximises reward. Additionally, Policy Gradient Algorithm, a predecessor of DDPG, is not chosen as it is not sample efficient, although it is able to deal with large action space. An algorithm is said to be sample efficient if it

learns as much as possible from each experience [21]. Policy Gradient is a on-policy method, meaning that it discards the previously learnt policies, meaning that previously learnt information is not retained, making it sample inefficient. However DDPG differs from the fact that it uses a Memory Buffer to store experienced transitions, and learn by sampling this Memory Buffer {more details about this in HER} [22], allowing it to learn from experience from different timepoints. Now more is learnt from the agent's experiences, allowing the agent to be sample efficient. Sample Efficiency is important as it concerns with the speed of convergence for our learning algorithm. I want our algorithm to be able to converge as fast as possible, as the task is highly complex, meaning that it will already take a long time for the algorithm to converge. Furthermore, it is expensive to train the agent as it requires the agent to actually play the game, and playing a game for a long time can be costly due to electricity usage. It also requires a person to supervise the learning as the game might disconnect or bug during training.

Hindsight Experience Replay (HER)

HER is a that can be used in combination to the Off-Policy Reinforcement Learning in the evaluation step aforementioned. HER combined with Off-Policy Reinforcement Learning allows sample efficient learning in an environment with sparse reward [7]. However, HER is applicable whenever there are multiple goals to be achieved. Goals in this context can either mean a predicate that determines if a given state has achieved the goal, or goals can specify some properties of a state. The basic idea behind HER is that it allowing Reinforcement Learning agent to undertake reason related to 'what went wrong' when faced with negative interaction, instead of deeming it as a bad sequence of plays. This is done by replaying the similar sequence of state transitions with the goal that the agent has during the generation of the sequence of state transitions, and an additional goal, which is unintentionally achieved during the sequence of state transitions. As such, the agent can then learn more efficiently as it now can learn with respect to 2 goals instead of 1. To implement HER, I first collect a number of state transitions via interaction with the environment, then sample randomly from this collected pool of data to use for evaluation. I also keep a limited memory, allowing the memory population is constantly replaced with new state transitions, creating a feedback loop that allows the agent to learn and improve. The last step is to decide on the choice of the additional goal. In the paper [7] they have chosen the additional goal to be the goal that is achieved by the terminal state for example, but other approaches like :
Future — replay with k random states which come from the same episode as the transition being replayed and were observed *after* it

Episode — replay with k random states coming from the same episode as the transition being replayed,

Random — replay with k random states encountered so far in the whole training procedure.

All of these strategies have a hyper-parameter k which controls the ratio of HER data to data coming from normal experience replay in the replay buffer.

HER aims to overcome the issue of sparse reward and sample efficiency in Reinforcement Learning. Sparse reward is the situation where there is little to no reward in the majority of the interaction between the agent and the environment, and only major rewards are given occasionally. Sample Efficiency is the problem of learning with limited samples. Both problems are major issues to applied Reinforcement Learning. For example, in robotic task where Reinforcement Learning is usually applied, reward is usually given at the completion of the task, and that it can be costly to operate the robot for training, resulting in limited samples to learn from [AAC]. This is true in the context of CSGO, where reward is only given at the termination of a round, and that full access to CSGO data, including keyboard and mouse control data, is limited. Furthermore, by overcoming the difficulty of sparse reward, HER circumvent the task of Reward Function Design, which is a tedious manual task that requires a lot of domain knowledge and hyper-parameter tuning. Even then, the Reward Function might hinder exploration and not translate well to other similar task, thus limiting the Agent's versatility.

Goals and Universal Value Function Approximator

The use of goal is lightly touched on in HER, but here I want to formally introduce its purpose. In HER, goals takes the form of a predicate or a condition which takes a state as a argument, and outputs a boolean value corresponding to the achievement our failure of the goal in the state given as arguments. Thus generally, the goal space contains just as much structure as the state space [9]. This is an important aspect of goal as I have already mentioned that I are able to approximate the Value function with Neural Networks. Approximating the Value function with neural network is effectively the idea of representing the utility of any states in achieving the agent's specified goal. What I want to do with goal is to extent the

idea of value function approximation to both states and goals so as to learn an Universal Value Function. Universal in this case means that the value function can generalise over all possible goals the agent can have in the Environment. With the value function being universal, the agent is now more versatile as it is able to achieve different goals, which effectively means that it can achieve different task, since tasks can be encoded as goals.

Actor Critic

Actor-Critic is a popular reinforcement learning algorithm that combines aspects of both value-based and policy-based methods to learn an optimal policy in an environment. In Actor-Critic, the actor is responsible for selecting actions based on the current policy, while the critic evaluates the quality of the selected actions by estimating the value function. The value function estimates the expected total reward that can be obtained from a given state under the current policy. The actor receives feedback from the critic in the form of a value function, which is used to update the policy by selecting actions that are more likely to result in higher rewards [55]. This feedback loop allows the agent to learn from its experiences in the environment and improve its policy over time. One of the advantages of the Actor-Critic algorithm is that it can handle continuous action spaces, which are common in many real-world applications. It also has good sample efficiency, meaning that it can learn an optimal policy with relatively few samples compared to other reinforcement learning algorithms. Overall, Actor-Critic is a powerful and versatile algorithm that has been successfully applied to a wide range of problems, including robotics, game playing, and natural language processing.

Target Network in Actor Critic

**Target networks are an extension of the Actor-Critic algorithm that is used to improve stability during the learning process. In standard Actor-Critic algorithms, the critic uses the same network to estimate the value function at each time step, which can lead to instability due to the correlations between consecutive estimates. To overcome this issue, target networks are used to separate the networks used for learning the policy and estimating the value function [54]. The idea is to use two sets of networks: one set of "online" networks that are used to compute the current policy and value estimates during learning, and another set of "target" networks that are used to generate the targets for the value function during training.
The target networks are updated less frequently than the online networks, typically after a fixed number of steps, and their weights are slowly updated by averaging them with the online network weights. This means that the targets for the value function are less correlated with the estimates produced by the online network, which improves the stability of the learning process. In summary, target networks are used in Actor-Critic algorithms to improve the stability and convergence of the learning process by reducing the correlation between consecutive value estimates. By using two sets of networks, one for computing the policy and value estimates and another for generating the targets for the value function, the Actor-Critic algorithm can more reliably learn an optimal policy in complex environments.**

Asymmetric Actor Critic Algorithm

Asymmetric Actor Critic Algorithm is a variant implementation of Deep Deterministic Policy Gradient (DDPG) [8]. Unlike conventional DDPG where both the Actor and Critic are trained on similar nature of states, in Asymmetric Actor Critic Algorithm, the Actor is trained on Partial States, while the Critic is trained on Complete States [8]. The idea of Partial States and Complete States emerge from the use of simulations in Robotic Reinforcement Learning task. In simulation, the environment is directly accessible to the agent, meaning that the agent has complete knowledge of the environment, be it the robot arm's orientation or velocity. However, such is not the case in the real world, where partial information like visual information is easily available to the agent. This discrepancy in Observability of the environment thus limits a simulation-trained agent to perform well in real life setting, and is also what give rise to the idea of Partial States and Complete States.

Asymmetric Actor Critic Algorithm aims to exploit the availability of Complete States in the simulation, while retaining the agent's ability to perform in the real world. This is the reason why the Actor is trained on Partial States, so that the policy is learnt with respect to partial states and actions can be chosen with just partial states, which then allows the agent to maintain its performance in the real world. At the same time, the Critic is trained on Complete States so that it learns better than the Actor, and as such is able to 'guide' the Actor, allowing for faster convergence and better policy learnt.

I have decided on using Asymmetric Actor Critic Algorithm to speed up convergence by exploiting the difference in Observability. In our case, I have access to 'player' data, which is data held by a player of the game, being the partially-observable states; and the 'spectator', which is data held by a spectator of

the game, being the completely observable states. It is true that I could just fully train the agent on the completely-observable states, but that will mean that when I deploy the agent and it plays against a live player, it will require the same completely-observable states in order to make decisions, as its policy was trained on completely-observable states. However, the only data available to it now only suffice to make up the partially-observable states, and due to the difference in observability, the agent will not perform well. Therefore, there is a need for Asymmetric Actor Critic Algorithm to ensure that the agent is able to play with partial observability, even though it is trained with complete observability.

Computer Vision

In order for the Agent to play the game of CSGO like a human would do, it will require the ability to take in inputs and produce outputs like a human. As I have aforementioned, I have decided on a Naive Implementation of the Visual System, and I will be outlining the methodologies I have used for the Visual System here.

The Visual System is made up of 2 parts, the Enemy Radar Detector and the Enemy Screen Detector. The Enemy Radar Detector utilises an simple RGB-value thresholding [10] for image segmentation, then followed by a Component Analysis [11] with statistics on the thresholded image.

The Enemy Screen Detector on the other hand utilises a more sophisticated method of computer vision. It uses Yolov5 [12], which is the fastest and most accurate as of Nov 2022, and had beaten all State Of The Art (SOTA) benchmarks, as of Nov 2022. More details of Yolov5 will be given right after, but for now it suffice to understand it as a vision AI that is able to detect the location of an enemy agent on the screen. Specifically, when it is given an image with an playing avatar, it is able to create a bounding box around the body and the head of the agent. This gives information about the exact pixel coordinates of these body parts on the screen, allowing the agent to aim at the agent.

I have outlined that the Visual System is Naive as I want to reduce the computation time required to run the agent as much as possible. In our case, a naive visual system like this will suffice as I want to put our focus and computation into helping the agent learn the strategic aspects of the game. Computer Vision is usually computationally intensive, and if I were to implement a sophisticated visual system that is able to locate itself with a given image, and locate objects in an image, the amount of computation required will be exponential to that of our current visual system. Furthermore, it will require a lot of data with location labelling from all over the map, and these data is not easily collected in bulk as they usually have to be manually labelled. Due to time limitations, I have decided to move forward with a visual system that is naive.

YOLOV5

YOLO (You Only Look Once) is a popular object detection model known for its speed and accuracy [25]. Object Detection is a high-level computer vision task that involves identifying and locating objects in images or videos. This usually involves lower-level computer vision task like edge enhancement, feature extraction and image segmentation, which helps to process the image information so that the object detection task is made easier. YOLOv5 is classified as a One-Stage/Proposal Free object detection algorithm. Single-shot object detection uses a single pass of the input image to make predictions about the presence and location of objects in the image. It processes an entire image in a single pass, making them computationally efficient. However, single-shot object detection is generally less accurate than other methods, and it’s less effective in detecting small objects. I have decided to trade-off accuracy for speed, as in our case, accurately detecting small objects is not really important, but keeping computation fast is as our resources are limited and I want the agent to receive information without much delay to maintain the information’s credibility — due to the non-deterministic nature of the environment, delay in receiving information might discredit the validity of the information as the environment might have changed after the information is received, making the information inaccurate, which effectively cause the learning agent to learn poorly. The accuracy for small objects is not important as I are detecting enemies in this case, and they will only appear small if they are far away — disregarding the case when they are being occluded by another object since our computer vision does not deal with this situation. Enemies far away do not pose much threat as usually inaccuracies in gun shot increases with distance. Thus ignoring far away agent seems to cause less problem for the learning agent as compared to introducing delay to the learning agent.

To put in perspective, if I were to use a more accurate algorithm like R-CNN, computer vision model will detect images at a frame rate of 16fps. On the other hand, using Yolov5 helps us achieve 60fps. The difference of 16fps and 60 fps can make a huge performance in the gunshot accuracy of the agent. The way our agent aim is to detect an image, then move its mouse position to the location of the detected coordinate on the screen. The issue with this is that the agent or enemy can potentially move during the interaction. With 60fps, there will be a smoother and better tracking of crosshair — keeping crosshair on enemy — as the agent receives information with little delay from the previous, meaning that the location to aim at should not have changed much and the mouse movement will be kept minimal, allowing for greater accuracy. However, if a 16fps algorithm was used, information will come at the agent at approximate ¼ of the speed, meaning that the enemy have more time to change position on the screen. This can cause a discrepancy in information, which may cause the agent to aim at the position the enemy was previously at, reducing accuracy. This is why I chose YOLOV5.

Game State Integration

The Game State Integration is a way for developers to pull information from live CS:GO games. You can access any information about each player in the game, which can be their health, ammo count and location [13]. You can also get information about the bomb, which includes its state ('planted' or 'defused') — and if a player interacts with it, there will be information about the player interacting with the bomb. This stream of information is used to inform the agent about the state of the environment while it is interacting with the environment to do Reinforcement Learning [14].

Game Interface

The Implementation of the Game Interface requires WinGUI32 as part of its implementation, and since this library only supports Windows machine, a Window machine is required to run the project. To control mouse and keyboard, pynput is used, which is also a open library.

The notion of Game Interface is quite important for our game AI as I do not want to be mistaken by the game as hackers. By using Raw mouse and keyboard input, the agent seems to be playing like a normal human player, using the expected game interface, and will not be flagged by the game as a hacker. This will allow us to be able to freely use CSGO as a training environment for our learning agent.

III : Related Research

Use of AI in First Person Shooting Game

A popular and recent use of AI in a First Person shooter video game is implemented in a game known as Quake III. Quite similar to CSGO, quake is a multi-player first person shooter game, with play modes that are quite similar, meaning they demand a similar set of skills from players. Thus I decide to consider their approach, and from then gain inspiration.

Deep Mind chose to tackle Quake III for its multi-agent characteristics. There has been progress made in artificial intelligence through the use of reinforcement learning on increasingly complex single-agent environments and two-player turn-based games, but these environments do not reflect the real world environment accurately.
In the real world, there exists multiple agents, each learning and acting independently to cooperate and compete with other agents, and such environment still remains an open challenge despite recent successes like OpenAI 5. End-to-end reinforcement learning have struggled in training agents in multi-agents systems [19] due to the high complexity of the learning problem that arises from the concurrent adaptation of other learning agents in the environment [40].

To further study on multi agent systems, Deep Mind decided to tackle Quake III, specifically its Capture The Flag play mode [41] -- which involves securing the opponent team's flag from their base to your own base; and preventing them from doing so to your team's flag. This requires each agent to be able to strategically navigate through the map, evading and tagging opposing players in the process.Tagging is done by aiming and shooting at the opposing player with a laser gadget -- own by each player, and these shots can be evaded. If an player is tagged, it respawns in its base room after a sort delay.

The team devised an algorithm and training procedure that enables agents to acquire policies that are robust to variations in maps, number of players and choice of teammates and opponents [50]. They proposed a purely end-to-end learning and generalisation method, and this method stabilises learning process in *partially observable multi-agent environment by concurrently training a diverse population of agents who lean via self-play. The maintenance of the agent population also serves as a meta-optimisation.
*unlike our approach -- which is atypical, the partially-observable state refers to raw RGB values of the gameplay. This is usually the assumed meaning of partial-observability, I did not follow this standard definition of partial state due to insufficient computation capabilities.

To tackle the problem, they adopted a novel two-tier optimisation process in which a population of independent Reinforcement Learning agents are trained concurrently from thousands of parallel matches with agents playing in teams together and against each other on randomly generated maps. Each agent in the population learns its own internal reward signal to complement the *sparse delayed reward from winning , and selects action using a novel temporally hierarchical representation that enables the agent to reason at multiple timescales.
*same problem faced by CSGO, usually reward earned when match ends, as such making reward sparse and delayed.

In their formulation, the agent's policy uses raw RGB values from the agent's first person perspective at a given time t as input, and outputs a probability distribution over all actions, conditioned on the stochastic latent variables whose distribution is modulated by a more slowly evolving prior process. The agent samples this probability distribution to select an action. Like all Reinforcement Learning Problems, they approached the game as an optimisation task, with the goal of finding a policy that maximises the expected cumulative reward over a game with T time steps.

The agent uses a multistep actor-critic policy gradient algorithm with off-policy correction and auxiliary task. The auxiliary task in question here is different from Goals as aforementioned in Universal Policy Function. While the purpose of Goals is for the agent to achieve generalisation for task, auxiliary task here aims to help agent do more exploration. In complex environments, there exists a wide variety of training signals and these auxiliary task directs the agent to maximise other pseudo-reward function apart from the cumulative reward. This then allow the agent to do more exploration as it no longer exploits policy with respect to only maximising cumulative reward, and as such agent can adopt a better policy [51].

The agent's hierarchical architecture is conceptually closer to work (outside of Reinforcement Learning) on building hierarchical temporal representations and recurrent latent variables. The resulting

architecture constructs a temporally hierarchical representation space in a way that promotes the use of memory and temporally coherent action sequences. This differ from previous hierarchical Reinforcement Learning agents that constructs explicit hierarchical goals or skills.

I did not choose to take this approach as it is computationally intensive, so much so that I am confident that I do not have the proper hardware to run them. Furthermore, they have utilised some very advanced technology that i am not familiar with, and as such I did not feel confident in adopting their approach. Additionally, there are intrinsic difference in the game by design, and thus differentiating the emphasis put upon each individual skill by each game. To generally summarise in, in Quake III Capture The Flag play mode requires players to navigate a large space occupied by multiple agents, thereafter increasing uncertainty in environment. The increase in uncertainty can reduce the significance of any strategic play as these strategies are formed under assumptions made about uncertainties, and as such has a good chance of failing. This can cause the learning agent to disregard strategic plays and opt for more aggressive and mechanical-skill demanding plays, which is potentially more significant than strategic plays to win game, and potentially simpler to implement. In our case however, the amount of uncertainty in the environment is significantly lesser, making strategic plays more consistent and thus more significant. The learning agent may then adopt a policy that emphasises on strategic play. It is to note that this is just an interesting speculation by me. As such i did not take up this approach.

Use of AI in CSGO

There have been many AI-related works on the game of CSGO, and their objective varies. On one side, there are AI with an analytical objective, be it predicting win rate of a given team while watching the game, or analysing the skill set of a player given his or her game history and performance in them. On the other side, there are AI with the purpose of giving human players a challenge. Since our project concerns the latter, I will focus on the latter and ignore the former.

There has been 1 popular and general approach to a CSGO problem that is much simpler than this project. Specifically, in [52], the paper dealt with the DeathMatch game mode of CSGO. This game mode concerns solely mechanical skills, with little to no strategy involved, as the game mode is just a 10 minute round of players killing each other, and the one with the most kill is the victor. The approach taken by [52] uses Object detection and Imitation Training. The object detection model is used to detect enemy players, so as do mine in this project, and the Imitation Model is used to control the agent movements and orientation. Like our project, the idea was to have this 2 models collaborate with each other to play the game.

The paper decided to take on imitation training and supervised learning approach, due to the lack of available in-game API that gives access to real time game status. Well our project clearly proves the contradictory. However, the second reason as to why the author has taken the supervised learning approach remains valid. CSGO is unable to simulate games at scale for training, which makes Reinforcement Learning challenging due to scalability issues. This will be further elaborated in the Challenges section

IV : Design of Project

In this section, the design of the project is outlined in detailed. I start out first by introducing my intuition behind my approach. Then I formularise CSGO as a Reinforcement Learning problem, and show at an abstract level how this is implemented to give readers a concept of our implementation.

Intuition of Approach

The reason as to why i have chosen to discuss my intuition behind my approach as i believe this knowledge will put readers at a better headspace to understand the design of the project.

My hypothesis is as follows:

- **I start out with the believe that CSGO can be tackled as a Reinforcement Learning problem, reason being its game-like nature. Since the paper has already discussed why games fit into the framework of Reinforcement Learning, this believe will be taken as granted.**
- **I also believe that methodologies used in robotics task can be applicable in tackling the CSGO task, if it was to be formularised as a Reinforcement Learning Problem**
- **Getting inspiration from robotic task, I can then attempt to tackle the CSGO problem**

To use methods in robotic task to tackle CSGO, I need to reduce the CSGO problem to robotics tasks so that methodologies will be applicable. Below I outline my approach of 'abstractly' reducing the CSGO problem to robotic task by first breaking the CSGO problem down into sub-problems, then drawing similarities of sub-problems with robotic tasks. First i breakdown CSGO into 2 concurrent-subproblems:

- **I believe that a 1 v 1 Post Plant situation can be tackled by breaking it down into 2 concurrent sub-problems : positioning and balancing risk to prioritise conflicting tasks**
  - **This believe stems from the fact that there is many approaches in solving the CSGO problem due to its large problem space and complexity. These approaches lie on a spectrum, ranging from a passive strategy to a aggressive strategy.**
  - **With proper positioning, which can be understood as a path-planning task, the agent could run down time by avoiding the enemy, thereafter securing a win without taking the risk of a duel.**
  - **The issue with the above strategy is that the agent run the risk of giving the enemy the safety to defuse the bomb. This is where the trade-off balancing happens. The agent needs to weight risk of the bomb getting defused against the risk of duelling the enemy. The agent has to choose to prioritise 2 conflicting task : avoid the enemy or duel the enemy**
  - **In a way, this can be understood as the constraints of the problem, whereby the agent is constraint to prioritise either avoiding the enemy or duelling the enemy; and switch priorities when necessary throughout the game while positioning.**
  - **As such this 2 problems must be tackled concurrently.**

By formularising the CSGO problem as a positioning task, I can then draw inspiration from works of robotic task. Robotics task usually involve controlling a robot arm in a 3D space to achieve some sort of task, like pick-and-place for example. This is conceptually similar to the problem I want to tackle in CSGO. I want our agent to be able to control the character in a 3D space and our objective is to win. Reinforcement Learning have been largely successful in tackling robotic task, I believe that methodologies used in robotic task will be applicable to the task of CSGO.

The tackling of the trade-off problem is not that straightforward however. To tackle the trade-off problem, I first need the agent to be able to complete each individual conflicting task, without having to prioritise either of them. Studies in robotic task have attempted to achieve Universal Functionality for robots as aforementioned in II. These methodologies can be adapted for CSGO, thus allowing us to train our agent to learn how to perform both task individually. After being able to complete both task, I hope that the agent can gain some sort of intrinsic knowledge about each individual task, specifically the situations where the task should and should not be pursued. This knowledge can help the agent choose which task to pursue at any given situation, based on its previous experiences in similar situations, thus tackling the trade-off problem.

Another great merit in using robotic task implementation is that most of them runs with simulation, which have a very similar interface to one of a game, making integration and refactoring of implementation easy.

Formularising CSGO as a Reinforcement Learning Problem

Why Reinforcement Learning

Among the 3 types of Machine Learning available, I have chosen to use Reinforcement Learning, as I think that it is the most suitable learning method for the agent out of the three. The reasons for so are below:

Firstly, I want the agent to learn and explore the strategic aspect of the game of CSGO. There have been Supervised Learning approaches that attempt to tackle a similar problem as ours [17], but what they inherently teach their agent is how to imitate a human player, which is a very superficial form of understanding in the strategic aspect of the game. I consider this superficial as ultimately, the agent is purely reacting to the scene it is shown, reacting based on the data it was trained on, and there is no deliberation or planning process involved. This also implies that there is little amount of exploration involved, which is not desirable for me. Additionally, it is hard to objectively label the strategic value of each gameplay data as it is qualitative and subjective. Supervised Learning relies on labelled data to learn, and with the lack of data labelled for strategic importance, it is unable to learn the strategic aspect of the game. This further show that Supervised Learning is not well-suited for this Machine Learning Task. With Reinforcement Learning however, the agent is allowed to interact with the environment until it finds either an optimal policy or a policy sufficient for itself. This policy captures a deeper understanding of the strategic aspect of the game as the policy accounts for the value of the action in the long term and short term when selecting the agent — in this case, long term value measures how valuable a certain action now is to help the agent win in the future. Being able to take both short term and long term consideration captures the agent ability to balance trade off between long term goals (winning the game) and short term goals (going to strategic location). The long term consideration by the agent also captures its planning process. As such, I can see that the agent is taking on a more deliberative process. Additionally, the constant learning from the interactions that uses ϵ-greedy exploration means that the agent is able to explore the strategic aspect of the game while maintaining performance.

Secondly, usually in games, there are more than 1 way to approach a situation — of course each having different strategic value— as the environment is a Multi-Agent system, meaning that it is dynamic and non-accessible. This is so as a Multi-Agent system has multiple agents in it, and each agent is autonomous and is able to interact with and affect the environment [19] . As such, no single agent is able to predict the next preceding state accurately based on its own information, making the environment dynamic. Furthermore, each individual agent do not have direct access to other agent's information, which makes the environment non-accessible. To better understand in our context, the playing agent has no way of knowing the enemy location or action until it sees the enemy. This shows the lack of access, and the inability of the agent to fully 'know' the next state, which then show that the environment is both dynamic and non-accessible. As such, there is a need to explore. As mentioned above, using Reinforcement Learning, I are able to ensure a certain degree of continuous exploration of the policy space, which effectively allow us to search pass Local Optimum Policies and potentially reach a Global Optimum Policy. If such a task was done via Unsupervised Learning, I cannot ensure continuous exploration of the policy space. Unsupervised Learning usually involves finding some hidden structure in the given data, and in our context, this hidden structure will take the form of the mapping of action to all possible situations. Initially, I can say that there is a certain level of exploration as the agent tries to learn this mapping, but eventually, when the learning stabilises, which could happen in Global Optimum or Local Optimum, there is no

longer any sort of exploration. This potential lack of exploration could result in pre-mature convergence or inability to adapt to other play style, affecting performance of the agent.

Lastly, Reinforcement Learning is really well-suited for solving this task. CSGO has APIs that support live match data streaming, and can be played with any python code that is able to emulate the keyboard and mouse to interface with the CSGO action. This means that implementing an environment for an agent to continuously interact with is not impossible. Additionally, I have mentioned the importance of exploration above, and Reinforcement Learning not only allow us to tune the degree of exploration, but it also allow us to continuously explore, even after the policy 'stabilises' {quoted as the agent is still learning and so the policy is not permanently stable, just temporary}. With the ϵ-greedy exploration for example, ϵ will be the hyper-parameter that controls the probability that the agent 'explore', which means deviate from its policy. Implementation of exploration is flexible, and in this case I do a uniform sampling of the entire action space. It is easy to see how this approach ensures continuous exploration as it is expected that ϵ% of the time, I are exploring. In addition to continuous exploration, there is a hyper-parameter λ called learning rate which controls the degree of 'learning' the agent goes through per evaluation iteration. Learning rate can be decayed as learning progresses so that policy is allowed to stabilise while allowing for continuous learning. Continuous Learning is important for CSGO as there exist a variety of play style, and even more new ones can be made. Play style usually dictates the player's strategy, meaning that there exist many strategies to play in CSGO.
As such, it is very likely that the agent encounters a strategy that it has not encountered all strategies during training, making this particular strategy peculiar to the agent, causing it to not perform well. However, with continuous learning, as the agent learns more from playing with this certain strategy, it can eventually adapt its policy to this strategy, allowing it to perform well against strategies it has never seen before.

Designing the solution system.

Requirements:
For CSGO to be tackled as a Reinforcement Learning Problem, I first frame it as a Markov Decision Process problem. As aforementioned in II, Markov Decision Process is a great framework for Reinforcement Learning problem. For us to use the Markov Decision Process framework, I first need a few key components. I outline these components below:

i. **An interface for the game to communicate its current state to the learning agent.**
ii. **An interface for learning agent to act on the environment.**
iii. **A reward signal from the environment that serves as feedback for the learning agent after every interaction instance.**

In order to implement these interface, I first need to define what our states, action and rewards are. As such, I first start out by defining what is a state. The idea of Complete and Partial State is irrelevant here since both basically share the same structure, their difference is in dependent of their observability. State will be a dictionary with the follow keys:

Definition of State:

- **Enemy Location — 3d vector of floats**
- **Enemy Forward Velocity — 3d vector of floats**
- **Enemy Health — int**
- **Enemy last seen time —int**
- **Enemy Screen Coordinates — 2d vector of ints, if not available it will be set to (0,0)**
- **Agent Location — 3d vector of floats**
- **Agent Forward Velocity — 3d vector of floats**
- **Agent Gun — A constant integer**
- **Agent Bullet count — A constant integer**
- **Agent Health — int**
- **Bomb location — 3d vector of floats**
- **Bomb State — dictionary that stores the bomb state, and the corresponding time to that state**
- **Current Time — int**
- **Winner — int**

As I have mentioned above, the difference between Complete and Partial State is their Observability. In our context, Complete State will have access to updated enemy-related information at all times. To simulate partial observability with Naive Visual System, I will allow the Partial State to access the enemy-related information when it is able to detect the enemy with the 'Player'. This is our attempt to simulate actual gameplay by a human as best as possible given out Naive Visual System, as the Visual System will cannot geo-locate itself or any object it sees in a given image. I also do not want to increase computation time for the Visual System by implementing a Sophisticated one as it will severely affect the training and thus the performance of the Reinforcement Learning Agent by effectively slowing down the learning agent's inputs stream from the game, which causes it to have a delayed reaction.
I then move on to defining actions. Since actions are communicated from the 'Agent' to the 'Player', it takes on an encoded form, and I have decided to encode actions with binary due to its simplicity. As such, action is an array of binary, A, each representing a Boolean that decides what action to take. After taking in an array of action, the 'Player' applies the action to the game as specified by the action Array. Below outlines the definition of Action.

Definition of Action:

- **A[0] — 1 stands for idling agent, I can ignore 0**
- **A[1] — 1 stands for shift pressed, 0 stands for shift release {holding shift causes agent to walk, making no noise}**
- **A[2] — 1 stands for ctrl pressed, 0 stands for ctrl release {holding ctrl causes agent to crouch, lowering height and making no
noise}**
- **A[3] — 1 stands for press space, 0 stands for release space {pressing space causes agent to jump}**
- **A[4] & A[5]**
  - **(0,0) move forward**
  - **(1,0) move left**
  - **(0,1) move right**
  - **(1,1) move backward**
- **A[6] & A[7]**
  - **(0,0) no movement on mouse cursor**
  - **(1,0) means to move cursor to right by 100 pixels**
  - **(0,1) means to move cursor to left by 100 pixels**
  - **(1,1) no movement on mouse cursor**
- **A[8] & A[9]**
  - **(0,0) no movement on mouse cursor**
  - **(1,0) means to move cursor to down by 50 pixels**
  - **(0,1) means to move cursor to up by 50 pixels**
  - **(1,1) no movement on mouse cursor**
- **A[10] — 1 stands for left-click, 0 stands for no left-click**
- **A[11] and A[12] — coordinates of enemy on screen if any, else (0, 0)**

Initially, there existed a shaped-reward function that was complicated and extremely tailored for a subjective play style, meaning that it potentially could have hindered exploration, which is a really bad thing given the large state space of the environment it functions in. As such, I have used Hindsight Experience Replay (HER), which allows for sparse, binary reward. With HER, I was able to treat CSGO as a Zero-Sum Game and design a simple reward function that basically rewarded the agent for winning the game, penalise agent for losing the game. A draw is quite impossible, but if there is ever one, no reward will be given.

Reward Function : $R(x|x = win) = 1$, $R(x|x = lose) = -1$

- **There is 2 winning options, 1 is killing the agent and the other is defusing the bomb. This will be used as our distinct goals**
- **To lose is basically to have the bomb defused. The agent's death correlates to losing, but in the most ultimate sense is not the
100% causing factor that can result from a lose, meaning that the agent can still win from dying. As such I do not want to penalise agent death as a lost so as to not hinder exploration to strategy that involves higher-risk plays for the agent.**

Now I move on to define goals. Goals in this context of CSGO will be the 2 definite and distinct winning conditions : (1) Bomb explodes or (2) Enemy killed. Note that (2) is a special condition in this case since it is the last enemy standing that can defuse the bomb. Killing him will equate to having the bomb explode was there is no more players that can defuse the bomb. As said before, the abstract implementation of goals will be some sort of predicate or some description of the properties of the state. To simplify the implementation of goals, I use the formal implementation. I implement this abstract predicate as such:

Class Goal:

- **Index — int**
- **Predicate — function**

With the index, I can represent the goal as tensor in the Deep Deterministic Policy Gradient (DDPG) Algorithm. Index will be used to identify goal as well. With this implementation, I will be able to move the implementation of goals from the environment-side to the agent-side. This make sense as goals should be independent of the environment, and should be dependent of agent [9]. Note that the implementation uses functional programming. I will be using the Index of goals to retrieve the Function-predicate, and then pass states into it as argument to see if it returns True or False.

After having defined our states, actions and rewards, I can now move onto the interfaces that works with these elements. In order to implement this interface, I have introduced 4 entities, and have included a figure — figure (1) — below that outlines their interaction at an abstract sense.

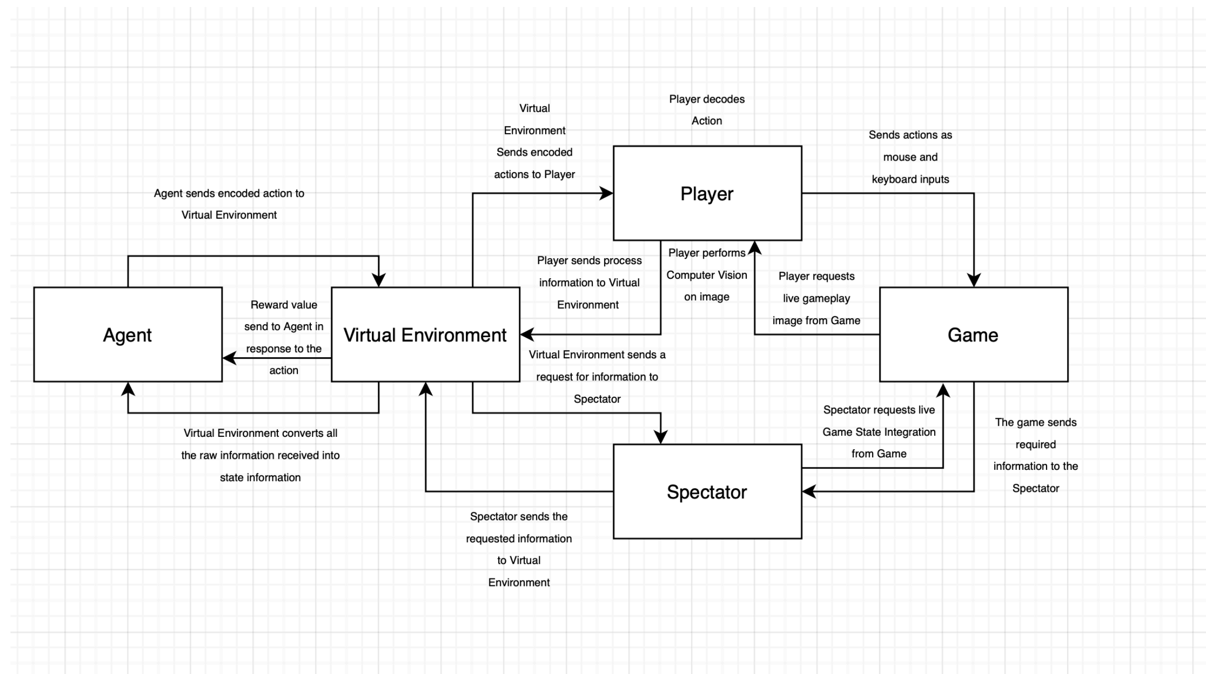


Figure (1) : the abstract relationship between each entity

4 key entities in our design :

- **Agent**
- **Virtual Environment**
- **Player**
- **Spectator**

The Agent

The purpose of the Agent entity is really to run the Reinforcement Learning Algorithm and learn the CSGO problem. The role of the Agent in this system will be to play the game via the Virtual Environment and learn the game. For the agent to learn the game, I will be using Deep Deterministic Policy Gradient Reinforcement learning method.

I have decided to use Deep Deterministic Policy Gradient (DDPG) Algorithm due to both the large action space available to the agent, and the success it had in [8]. Value-based method are unsuitable in large action space [20], and as such is not chosen to be used, while DDPG is proven to be successful in large action space like those of a 6-axis Robt Arm. This is so as Value-based methods requires a good degree of exploration in the action space before it is able to find the optimal action. However, when the action space is large, such exploration is sometimes impossible, and possibly unnecessary there might exist redundant actions that should not be explored at all. However, with Policy-based method like DDPG, I need not explore the entire action space, as I are focused on finding the best action to take, making our search in the action space to be guided by some heuristic, which in our case will be the action that maximises reward. Additionally, Policy Gradient Algorithm, a predecessor of DDPG, is not chosen as it is not sample efficient, although it is able to deal with large action space. An algorithm is said to be sample efficient if it learns as much as possible from each experience [21]. Policy Gradient is a on-policy method, meaning that it discards the previously learnt policies, meaning that previously learnt information is not retained, making it sample inefficient. However DDPG differs from the fact that it uses a Memory Buffer to store experienced transitions, and learn by sampling this Memory Buffer {more details about this in HER} [22], allowing it to learn from experience from different timepoints. Now more is learnt from the agent's experiences, allowing the agent to be sample efficient. Sample Efficiency is important as it concerns with the speed of convergence for our learning algorithm. I want our algorithm to be able to converge as fast as possible, as the task is highly complex, meaning that it will already take a long time for the algorithm to converge. Furthermore, it is expensive to train the agent as it requires the agent to actually play the game, and playing a game for a long time can be costly due to electricity usage. It also requires a person to supervise the learning as the game might disconnect or bug during training.

In addition, instead of the normal Actor-Critic model that baseline DDPG algorithm uses, I will be using an Asymmetric Actor-Critic model.

I have decided on using Asymmetric Actor Critic model to speed up convergence by exploiting the difference in Observability. In our case, I have access to 'player' data, which is data held by a player of the game, being the partially-observable states; and the 'spectator', which is data held by a spectator of the game, being the completely observable states. It is true that I could just fully train the agent on the completely- observable states, but that will mean that when I deploy the agent and it plays against a live player, it will require the same completely-observable states in order to make decisions, as its policy was trained on completely-observable states. However, the only data available to it now only suffice to make up the partially-observable states, and due to the difference in observability, the agent will not perform well. Therefore, there is a need for Asymmetric Actor Critic Algorithm to ensure that the agent is able to play with partial observability, even though it is trained with complete observability.

Within the Asymmetric Actor Critic Model, I have decided to use Recurrent Neural Network as the choice of Neural Network. I have 6 fully-connected layers — 1 input layer, 1 output layer and 4 hidden layers. The 4 hidden layers takes the form of 512-1024-1024-512 nodes per layer. The choosing of the number of nodes in the hidden layer and number of hidden layer are extremely important, as there is a need to balance the tradeoff between complexity and capability. With increased number of nodes per layer or number of layers, I can expect to see a stronger neural network that is more capable at tackling the problem. However, doing so increases training time exponentially. On another hand, using a simple Neural Network might result in faster convergence, but the neural network may potentially converge pre-maturely, causing it to be incapable of accurately approximating functions, and as such making it incapable of tackling the task. I have tried 2 other alternatives, both of which caused the aforementioned problem. A 512-512-512-512 architecture converged pre-maturely, and could not handle the task; while a

512-1024-2048-1024-512 architecture took around an hour per evaluation step to update its parameters, and as such was computationally impractical for me to utilise.

Lastly, for learning, I do an off-policy, Hindsight Experience Replay (HER) to update the Neural Networks of AAC.

As aforementioned, HER aims to deal with This is true in the context of CSGO, where reward is only given at the termination of a round, and that full access to CSGO data, including keyboard and mouse control data, is limited. Furthermore, by overcoming the difficulty of sparse reward, HER circumvents the task of Reward Function Design, which is a tedious manual task that requires a lot of domain knowledge and hyper-parameter tuning. Even then, the Reward Function might hinder exploration and not translate well to other similar task, thus limiting the Agent's versatility.

The Virtual Environment

The Virtual Environment is meant to be an interface for the agent to interact and learn from the environment. Specifically, it acts as a middle man for the agent to communicate with both the Player and Spectator. Its role in the system is to facilitate information transfer and communication between entities. It serves as a common ground for the other 3 entities to communicate with each other.
There are multiple reason as to why a virtual Environment is used. Firstly, especially with Reinforcement Learning, most algorithms are coded in a way that integrates smoothly into the OpenAI gym Environment interface. As such it is both a convention, and easier, to implement the algorithm on an OpenAI gym custom environment. This is what the Virtual Environment is. Secondly, it is to help keep the code clean. As mention, they are many entities in play in this entire system, as such I want to keep all the code that facilitates their interaction in one place. Doing so not only makes it easy to debug their interactions, but it also allows me to easily see in what order are the interactions taking place, since all the code of their implementation is kept in one place.

The Player

The Player will be the interface where the Virtual Environment communicates the agent's action to, so that the Player can decode the action into mouse and keyboard inputs; and send these inputs to the Game so that the action can be applied in the game.

The Spectator

The Spectator will be the interface where the Virtual Environment will be able to get live complete information about the game. This entity plays a crucial role in our Asymmetric Actor Critic model as it gives us complete Observability of the environment, allowing us to train the Critic with complete information instead of the available partial information. For convenience, this entity will provide both Complete and Partial information to the Virtual Environment.

V : Implementation

Implementation of Entities

I begin by introducing the implementation of each entity in isolation, outlining what are expected from the. Moving forward, I outlined the key interactions of the 4 entities in the Virtual Environment during training.

Spectator

The implementation of the Spectator is relatively straightforward as compared to the other 3 entities. The spectator needs to retrieve information and sends it to the Virtual Environment as previously stated. As such there is 2 subtask that is required of it – retrieving information and communicating with the Virtual Environment.
For it to retrieve information, specifically Game State Integration information, I have use code written by

https://github.com/Erlendeikeland/csgo-gsi-python and adapted it for my purpose. The need for spectator lies in the fact that I will not be able to retrieve the information about the match with full observability if I was to retrieve it as a player of the match. As such, there is a need to run a separate instance of the game, and have the player join the match as the spectator. By using the spectator, I have also allowed myself some leniency to grab all the information I require from the Spectator. This is also for code cleanliness and ease of debugging. Under strict conditions, I would have needed to grab information from the Spectator to make my fully observable states; and grab information from the Player to make my partially-observable states. However, I have decided to separate observability in the Virtual Environment. I have mentioned some new terms here, but I will elaborate them further below.

For the second task, I have done it with the use of the 'socket' class in the Python Standard library. Since there is only 1 instance of relationship between the Virtual Environment and the Spectator, the implementation is also really straightforward. The Virtual Environment sends to the Spectator what it wants, via a string, and the Spectator returns the required data, as a python dictionary. The key expectation of the Spectator here is to ensure that the correct data is sent in the correct form. Since the data is generated by the game, all I need to ensure is that the data requested corresponds with the data sent. I have ensured this by working with python dictionary and keys; and have manually tested it. It is really hard to write a fixed test from it, since the data is live generated, and as such it is difficult to write static assertions.

Player

The implementation of the Player is slightly more complicated than the Spectator. The Player has 2 main task – Process Game Play image and send it to the Virtual Environment, and apply actions sent to it by the Virtual Environment. For ease of understanding and coding, it consists of 2 elements, the game interface server and the enemy detection server.
The first complication that I faced firstly is managing the 2 different relationship such that they do not overlap and result in data inconsistency. It will severely affect training if the enemy detection server took the encoded data from the game interface server, which causes the action to not be applied, and as such causes the game interface server to idle and wait for a never coming response from the Virtual Environment that the action has been applied. I have tried many different methods, but I overlooked a really simple solution to this, one that does not involve complication like threading. I basically only had to connect the two servers to different ports, which is what I have done. Connection implementation wise, it uses the 'socket' library like the Spectator. The game interface server uses conceptually similar implementation as the Spectator -- where the Virtual Environment sends an action over to it, via a string, and the game interface server performs the action and reports back to the Virtual Environment. However, for the enemy detection server, data is constantly ported to the Virtual Environment as it is a one-sided relationship – only 1 kind of information is required by the Virtual Environment, so the Virtual Environment need not specify anything to the enemy detection server, and as such this makes it a one-sided relationship.
I now move onto the implementation of the game interface server. The game interface servers mainly perform 2 kind of actions – in-game actions and developer console commands. The latter actions are used to automate the configuration and restarting of the match. To implement the requested action, the server first needs decode the action, then perform it via keyboard and mouse inputs. Decoding of the statements are done via a list of if-else statements. I have a list of pre-defined strings that I will match with the requested actions, and these strings corresponds to function that will run hard-coded developer console commands if the requested action matches with the corresponding string during the if statements check. If there is no match, it proceeds to treat requested action as an in-game action. In-game actions are encoded in binary, and are defined above.
For the Enemy Detection Server, there is 2 subparts to it, getting the gameplay image, and processing the image. The first part is done via WIN32GUI library, which allows me to grab a specified region or the entire image of the current display. For the second part, it breaks down into 2 subparts, which is only made possible by the game. The game has a radar that shows enemy presence on map as a red dot. I have chosen to exploit this by first having the Enemy Detection Server detect for enemy presence in the radar. This is done by isolating the radar part of the image, and performing thresholding on the red dots, then statistical analysis (by cv2) to detect for any red dot presence in the radar. If there is enemy presence, I move onto the second subpart. The second subpart concerns itself with the entire image, and it uses a YOLOV5 model to scan the entire image for enemy presence, and outputs the screen coordinate of the enemy if it is detected on screen. I have used YOLOV5, and have taken the model weights and code provided by https://github.com/Lucid1ty/Yolov5ForCSGO. This saved time needed to train my own computer vision to detect enemy. Since much of the code remained similar after I have adapted them from my purpose, there is not much for me to implement.

Agent

The Agent implementation consist of 3 parts – the Deep Deterministic Policy Gradient algorithm, the Asymmetric Actor Critic and the Hindsight Experience Replay Buffer. Majority of the code was taken from baselines requiring minimum changes as the algorithm implementations are standard, and there is not much need to deviate from the base implementation. The major concern here will be the choice of Neural Network architecture. Thus, I will be briefly mentioning the implementation of each part, then discuss about the neural network architecture being used. This section will be divided into 3 subparts, each part talking about the key implementation of the aforementioned parts.
{Please go onto the next page!}

Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm is a Reinforcement Learning method. It being an off-policy method, it has 2 roles. Firstly it needs to allow the agent to interact with the environment and record their transitions, and these transitions will be used to update the neural network during evaluation. The DDPG algorithm is more involved in the evaluation step, and as such I will elaborate on how the neural networks are updated in each evaluation step. DDPG has 4 neural networks – Actor, Critic, Target Actor, Target Critic.

```
def update_policy(self):
    # Sample batch
    state_batch, p_state_batch ,action_batch,\
    reward_batch, next_state_batch, next_p_state_batch,\
    goal_batch, p_goal_batch , terminal_batch = self.memory.sample(self.batch_size)
    pl = 0
    vl = 0
    for i in range(self.batch_size):
    # Prepare for the target q batch
        print('state_batch[i]', state_batch[i])
        print('next_state_batch[i]', next_state_batch[i])
        print('p_state_batch[i]', p_state_batch[i])
        print('next_p_state_batch[i]', next_p_state_batch[i])

        next_q_values = self.target_critic(
            to_tensor(next_state_batch[i]),
            self.target_actor(to_tensor(next_p_state_batch[i]), to_tensor(p_goal_batch[i])),
            to_tensor(goal_batch[i]),
        )
        next_q_values.volatile=False

        target_q_batch = torch.tensor(reward_batch[i]) + \
            self.discount*torch.tensor(int(terminal_batch[i]))*next_q_values

        # Critic update
        self.critic.zero_grad()

        q_batch = self.critic( to_tensor(state_batch[i]), torch.tensor(action_batch[i]), to_tensor(goal_batch[i]) )
        # q_batch = q_batch.to(torch.long)
        # target_q_batch = target_q_batch.to(torch.long)
        value_loss = criterion(q_batch, target_q_batch)
        value_loss.backward()
        self.critic_optimizer.step()

        # Actor update
        self.actor.zero_grad()

        policy_loss = -self.critic(
            to_tensor(state_batch[i]),
            self.actor(to_tensor(p_state_batch[i]), to_tensor(p_goal_batch[i])),
            to_tensor(goal_batch[i])
        )

        policy_loss = policy_loss.mean()
        policy_loss.backward()
        self.actor_optimizer.step()

        # Target update
        soft_update(self.target_actor, self.actor, self.tau)
        soft_update(self.target_critic, self.critic, self.tau)
        vl += value_loss.data
        pl += policy_loss.data
        print("value loss: ", vl, "policy loss: ", pl)
    return vl/self.batch_size, pl/self.batch_size
```

Figure (2) : network updates in DDPG

The function goes as follows :
1. **I sample transitions from our Hindsight Experience Replay Buffer**
2. **Using the samples, I generate *q-value and target q-value with the Critic and Target Critic network, I calculate the Value Loss — which in this case is Mean Square Error Loss — and I use this loss to update our Critic network**
3. **Using the Critic, I get our policy loss, and use the policy loss to update our Actor network**
4. **Then I perform a soft update on our target networks using the corresponding networks. In this case, our tau is 0.98. This means I update our target network's parameters by using a weighted sum of both the target network and the corresponding network — with weights being 0.02 and 0.98 respectively.**

This is an implementation of the Deep Deterministic Policy Gradient algorithm that I took from https://github.com/openai/baselines , which is a set of high quality, implemented algorithm. This saved me time needed to implement the algorithm from scratch, and it took away the concern that my agent was not learning as I have implemented the algorithm incorrectly. I only made minor changes to the code, and as such I did not implement much here.

Hindsight Experience Replay (HER)

```
class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = []
        self.max_size = max_size

    def push(self, state, p_state, action, reward, next_state, next_p_state, goal, p_goal, done):
        transition = tuple((flatten_obs(state), flatten_p_obs(p_state), action, reward, flatten_obs(next_state), \
            flatten_p_obs(next_p_state), flatten_goal(goal), flatten_goal(p_goal), done))
        self.buffer.append(transition)
        print(len(self.buffer))
        if len(self.buffer) > self.max_size:
            self.buffer.pop(0)

    def sample(self, batch_size):
        state_batch, p_state_batch, action_batch, reward_batch, next_state_batch, next_p_state_batch, \
            goal_batch, p_goal_batch, done_batch = np.transpose(random.sample(self.buffer, batch_size))
        return state_batch, p_state_batch, \
                action_batch, reward_batch, \
                next_state_batch, next_p_state_batch,\
                goal_batch, p_goal_batch, done_batch
```

Figure(3) : the Memory Buffer for Hindsight Experience Replay

HER requires a memory buffer that can allow the agent to store transitions during interaction and sample transitions during evaluation. I implemented the memory buffer using a python list, and set a limit on the number of transitions that can be stored in it so that I have an upper bound on our Space Complexity. Specifically, I have set the limit to 10000. This balances the tradeoff of maximising sample efficiency by retaining samples for as long as possible, while keeping Space Complexity bounded.

To store transitions, the agent only needs to call the wrapper-function for 'push' function. Since the transitions stored in the memory buffer will be used during evaluation, it is a merit to process these data for evaluation when I are storing them. This ensures that data used during evaluation is always in the expected format. The 'push' function also controls the size of the memory buffer; if memory buffer is at limit, the push function will remove the transition at the head of the list; so as to accommodate for new incoming transition .

Sampling is implemented in a very straightforward manner. I use the python library to sample our data, and then reformat our data using np.transpose. This is so as memory are stored as transitions, and not as state batches for example, and as such a transpose is required to switch the axis.

Asymmetric Actor Critic

```python
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, goal_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(flatdim(state_dim) + flatdim(action_dim) + flatdim(goal_dim), 512)
        self.fc2 = nn.Linear(512, 1024)
        self.fc3 = nn.Linear(1024, 1024)
        self.fc4 = nn.Linear(1024, 1024)
        self.fc5 = nn.Linear(1024, 512)
        self.fc6 = nn.Linear(512, 1)

        #init weight
        nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc2.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc3.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc4.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc5.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc6.weight, nonlinearity='relu')


        # self.relu = nn.ReLU()
        self.relu = torch.nn.functional.relu
        self.sigmoid = nn.Sigmoid()

    def forward(self, state, action, goal):
        x = torch.cat((state.flatten(), action.flatten(), goal.flatten()), dim=0)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.relu(self.fc4(x))
        x = self.relu(self.fc5(x))
        x = self.fc6(x)
        x = self.relu(x)
        return x
```

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, goal_dim):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(flatdim(state_dim) + flatdim(goal_dim), 512)
        self.fc2 = nn.Linear(512, 1024)
        self.fc3 = nn.Linear(1024, 1024)
        self.fc4 = nn.Linear(1024, 1024)
        self.fc5 = nn.Linear(1024, 512)
        self.fc6 = nn.Linear(512, flatdim(action_dim)) # action_dim = 10
        # self.relu = nn.ReLU()
        self.relu = torch.nn.functional.relu

        #init weight
        nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc2.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc3.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc4.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.fc5.weight, nonlinearity='relu')
        nn.init.xavier_uniform_(self.fc6.weight)


        # self.tanh = nn.Tanh()
        # self.sigmoid = nn.Sigmoid()
        self.action_layer = ActionLayer(flatdim(action_dim))


    def forward(self, state, goal):
        x = torch.cat((state.flatten(), goal.flatten()), dim=0)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.relu(self.fc4(x))
        x = self.relu(self.fc5(x))
        x = self.fc6(x)
        x = self.action_layer(x)
        return x
```

Figure (4.1&4.2) : the Asymmetric Actor Critic

Asymmetric Actor Critic's implementation significantly focuses on the architecture of the Neural Networks. The actual implementation of the networks is relatively straightforward, as seen from the code, and as such I will not elaborate any further with regards to the code. In our architecture I have chose to have 4 hidden layers in each architecture — taking on a 512-1024-1024-512 node structure for both networks. The Actor networks takes in partial states and goals as input while the Critic network takes in Complete States, goals and action as inputs — this explains the difference in their input layers. The Actor networks outputs an action while the Critic network outputs the a value function — this explains the difference in their output layers.

For selecting actions, I treat the output of the Actor layers as a list of confidence to take the action. Using a custom Action Layer, I sample from this list of confidence to generate our action. Do remember that actions in this context are in-game actions that are encoded in binary.

```python
class ActionLayer(nn.Module):
    def __init__(self,action_size):
        super(ActionLayer, self).__init__()
        self.action_size = action_size
    def forward(self, x):
        action = torch.zeros(self.action_size)
        for i in range(self.action_size):
            rand = torch.rand(1)
            if rand < x[i]:
                action[i] = 1
            else:
                action[i] = 0
        return action
```

Figure 4.3 : Action Layer:

As seen from the implementation, for each confidence level, I generate a random number within the range of [0:1]. If generated number is below confidence, I set the current binary as 1, else 0.

The Virtual Environment

For this section, I have taken a a very different approach in explaining the implementation. The Virtual Environment links all entities together, and as such I felt that it will be better to go from a top-down approach for easier understanding of the implementations. I will first talk about how entities interact in the Virtual Environment, and then elaborate in details key internal functionalities of the Environment.

Since the Virtual Environment main function is to facilitate communications for the 3 Entities in 4 key processes that occur during training. However before I start talking about the 4 key processes, I want to elaborate on our physical setup, as it will help us understand how the virtual environment interacts with the entities.

Physical Setup

As I have mentioned, in an abstract sense, I have 4 entities. I will like to talked about how they are implemented in the physical system. Firstly I have used 3 laptops to implement the entire system, as I am trying to split up the computation task so that I can achieve better performance on the game. The original 4 entities abstract idea still applies in our physical system, but I have combined both the Agent and the Environment into 1 laptop. As such, from now on, unless specifically stated, I treat the agent entity and the environment entity as 1 complete entity, known as 'Agent'. I realise the abuse of notation, but I struggle to find another notation for the physical system.

The physical set up of is actually quite complicated. This is due to the fact that I am trying to split up the computation task so that I can achieve better performance on the game. The laptops have client-server Relationship. I will codename the computers 'Agent', 'Player' and 'Spectator'

- **'Agent' is responsible for running the Reinforcement Learning Algorithm.**
- **'Player' is the meta-interface that the 'Agent' will use to play the game**
- **'Spectator' will be spectating the game and streaming the Game State Integration information to the 'Agent'**

Relationship:
'Agent' <CLIENT> —— 'Player' <SERVER> {Gameplay Screen Information}
'Player' <CLIENT> —— 'Agent' <SERVER> {Game Interface}
'Agent' <CLIENT> —— 'Spectator' <SERVER> {Game State Integration}

Before I can train the environment, I must do some manual set up :
1. **First ensure that both the 'Player' and the 'Spectator' has CSGO installed, and both computers are configured to allow for Game State Integration —— Instructions can be found in [13]. Also ensure that both CSGO have developer consoles allowed —— Instructions can be found in [15]**
2. **Then have the 'Player' run 'python src1/main.py' from the root of the project Directory. This starts the server for the 'Player'. As the model takes time to load, ensure that the server is started before proceeding to next step**
3. **Then have the 'Player' create a lobby and add the 'Spectator'. With the 'Player', choose a difficulty, 'Hard' in my case, and have it start a casual bot match with the map DE-DUST 2.**
4. **After starting the game, ensure that the 'Player' joins the Terrorist team, and have the 'Spectator' join the Spectator team. Wait until both have entered into their respective teams.**
5. **Using the 'Player', open the console and type 'bot_kick' and 'bot_add_ct'. When 'bot_add_ct' is called, ensure that a bot is added to the game, as there are bugs with the implementation of adding bots like this.**
6. **After that, on the 'Spectator', run 'python src2/gsi_server.py'**
7. **Now you can start Training by running on the 'Agent' computer 'python src/main.py'**

This then starts the training process. The training process is basically several nested for loops, but the main process are the initialisation of the environment to start training, the interaction process, restarting a game, and finally the evaluation process.

The Initialisation steps goes like this :

1. **The 'Agent' sends a request to the 'Player' for it to run the necessary steps to configure the game for our specific training environment.**
2. **The 'Player' then uses the console to configure the environment, upon completion, the 'Player' updates the 'Agent'**
3. **After receiving the response, 'Agent' then request from both the 'Player' and the 'Spectator' for current game information to generate initial partial states and complete states. At the same time, initial complete and partial goal is generated. Initial reward is 0.**
4. **All these initial information are then used by the 'Agent' to initialise the DDPG algorithm.**
5. **Training then begins by a fresh restart of the game**

Restarting the game:

1. **The 'Agent' first sends a request to the 'Player' for it to run the necessary steps to restart the game**
2. **The 'Player' then uses the console to restart the game, which involves :**
    1. **uniformly sampling for a new bomb site**
    2. **From the bomb site, a bomb plant location is sampled uniformly. These bomb plant locations are hand-picked to ensure stability and preventing unexpected situation (being stuck in the wall)**
    3. **I then freeze the enemy, so the enemy can't move.**
    4. **Then I send the agent to said location to plant the bomb**
    5. **After planting the bomb, I unfreeze the enemy and the game begins.**
3. **After receiving the response, 'Agent' knows that the game has restarted, and from there starts training.**

A single iteration of the Interaction process goes like this :

1. **The 'Agent' first request from the 'Spectator' for the current Game State Integration Information of the game**
    - **Upon request, the 'Spectator' sends the current Game State Integration information to the 'Agent'**
2. **The 'Agent' then request information from the 'Player' about the current screen information of the game**
    - **Upon request, the 'Player' processes the current screen for enemy information and sends it back to the 'Agent'**
3. **The 'Agent' then processes both informations into Partial State and Complete States.**
4. **The 'Agent' then uses the Actor to selects an action, with the current partial state and current partial goal**
5. **The 'Agent' then encode this action, and sends it to the 'Player'**
6. **The 'Player' decodes the message, and then apply the action to the current game play. The 'Player communicates with the 'Agent' when it has finished applying the action.**
7. **After receiving a response from the 'Player', the 'Agent' sends another request to both the 'Spectator' and the 'Player' for the current game information. Like above, this triggers subprocesses.**
8. **The 'Agent' then processes both informations into new Partial State and Complete States. At the same time the agent rolls for a new complete goal and new partial goal.**
9. **Using the consecutive states, the agent calculates the reward for the action.**
10. **The current partial and complete state, the action taken, the new partial and complete states are then saved into the memory buffer as a Transition.**

A single iteration of the Evaluation process goes like this :

1. **From the memory buffer, then agent uniformly samples a Transition batch of size 128 that will be used for updating the Actor and the Critic. A batch consist of current partial and complete states batches, next partial and complete state batches, reward batches, action batches, partial and**

complete goal batches; and a terminal batch, which outlines if the current Transition is terminated or not.

2. **I then use the target Critic to optimise the Critic**
    1. **This is done by first using the current complete state batch and complete goal batch to generate q-values with the target Critic and the actor. Lets codename the respective q-values as q_target and q.**
    2. **Using q_target, reward batch, terminal batch and a discount factor, I calculate the target action values (Q_target) of each respective transition. Calculation goes like this :** $Q = r + \delta * P(terminal) * q_{target}$ **where** $P(x = True) = 1; P(x = False) = 0$
    3. **I do the same using q, calculating Q. {Sorry for the abuse of notation}**
    4. **I calculate the value loss using Mean Squared Error between Q and Q_target, then use it to backward propagate the Critic Neural Network, and finally optimising it with Adam Optimiser.**
3. **With the updated Critic, I use it to update the Actor.**
    1. **I first use complete state batch and complete goal batch to calculate policy loss with the critic.**
    2. **Then I find the mean of the policy loss, and use it to backward propagate the Actor Neural Network.**
    3. **Finally I optimise the Neural Network with Adam Optimiser**
4. **Lastly I perform a soft update on the target Actor and Critic using the updated Actor and Critic; and a tau value that specifies the extent of the update.**

States from Raw Information

Now I move on to detail the process of converting raw information to states. As aforementioned, I have 2 sources of raw information : (1) the 'Player' with screen information and (2) the 'Spectator' with the Game State Integration. I first talk about the 'Player' as it does some computer vision processing, and then sends the outputs to the 'Agent'. The Computer Vision process has 2 sequential sub-processes, which I will define in sequence below:

Sub-process 1: Enemy Radar Detector

1. **I first take grab the image of the current gameplay with the WinGUI32 python library**
2. **Then I crop out the radar part of the image. The dimensions of the radar can be easily process by the measuring its boundary pixels with some kind of image editing tool like Adobe**
3. **Then I pass the Radar image to the Enemy Radar Detector, which detects for enemy presence in the radar. The process goes :**
    1. **The Enemy Radar Detector first thresholds the image for within a range of RGB-values. This works as the enemy presence I want to detect is always a red dot, which has constant RGB-values. If I threshold a muffled range to account for noise, I will be able to isolate majority of the red dot form the entire radar, allowing is to better detect for enemy presence**
    2. **Then using statistics supported by CV2 python library, I connect the isolated components, in some sort of 'dilation-like' process, which attempts to 'grow' the isolated region so that the region looks complete. The 'growing' of these isolated component in this case depends on a 4-way connectivity component analysis. There was not much detail on the implementation of this, and I found the implementation on [11] and edited it for my own personal use**
    3. **Then I return a boolean value that determines if a enemy is detected on the radar.**
4. **The enemy radar detector output is then prepared to be send to the 'Agent'**
5. **If the enemy radar detector returns True, I proceed to detect for enemy presence in the entire screen.**
    - **There is 2 reason as to why I decide to only detect for enemy presence in the entire screen only if I detect Radar image. The first reason comes from domain knowledge. In every situation where the enemy is seen on the screen, the radar automatically records it. This means that if there is no enemy presence in the radar, there is no enemy presence in the screen. This ultimately means that doing enemy detection on the whole screen when there is no enemy presence ion the radar is redundant.**
    - **Secondly, the detection of enemy presence in the entire screen is the bottleneck for the learning agent when it is interacting with the environment. Unlike the radar image, which has dimensions less than (200,200), the entire screen image has a dimension of (1920,1080). I use multiple convolution in the detection of enemy presence in the entire screen and since convolutions runtime is** $O(n^3)$**, the increase in the image dimension really slows down the computation time per interaction if I was to always run it.**

Sub-process 2: Enemy Screen Detector

1. **As aforementioned, the Enemy Screen Detector uses convolution to detect the enemy. Specifically, it uses a openly-available Yolov5 Model to detect enemy presence [16].**
2. **It is really straightforward in abstract terms, and to save the details already covered in II, I will take on an abstract view of the model, and say that it outputs the coordinates of head and body if it finds any.**
    1. **Note that since there is only 1 enemy player playing with the agent, the visual system can be so naive that it is unable to pair corresponding head and body coordinates, and that it is unable to distinguish enemy or ally. This is an 'logical-jump' made possible by the restriction placed on the environment**
3. **The 'Player' then stores the coordinates, and send both outputs from enemy screen detector and enemy radar detector to the 'Agent' in a dictionary form with keys :**
    - **enemy_on_radar : [0/1]**
    - **enemy_screen_coords, head : (x,y)**
    - **enemy_screen_coords, body : (x,y)**

So the 'Agent' receives a dictionary containing the processed visual information of the current gameplay from the 'Player'. As mentioned above, the 'Agent' as receives Game State Integration information [14] from the 'Spectator'. I process Game State Integration as dictionaries. Details about the data I use is as below:

1. **All player data:**
    - **List of all player data in format of:**
        - **Name — name of the player**
        - **State — a dictionary with information about the state of player which includes :**
            - **Health — health of current player**
        - **Player position — a dictionary which includes the location and orientation of player:**
            - **Location — the location of the player in 3d coordinates**
            - **Forward — the velocity of the player's forward movement in 3d coordinates**
        - **Player weapon**
            - **Include a list of weapon datas about each weapon, with the information:**
                - **Ammo-clip — the number of bullets in current weapon**
                - **State — the state of the weapon, 'active' if player is using it**
2. **Bomb**
    - **state : the state of the bomb — eg 'planted', 'carried', 'defused' …**
    - **position — the location of the bomb in 3 axis coordinate**
    - **player {optional} — the player interacting with the bomb, if any**
3. **Round**
    - **phase : the phase of the current round eg — 'live', 'over'**
    - **win_team{optional} : the winning team of the current round, if any**
    - **bomb : the state of the bomb — eg 'planted', 'carried', 'defused' …**
4. **Phase Countdown**
    - **Phase : the phase of the current round eg — 'live', 'over'**
    - **Phase ends in : the time in which the current phase ends**

From here, it is easy to see how after a bit of simple information processing, I can make the current Complete and Partial State from these 2 sources of information.

Encoded Actions to Player action

Above I have mentioned how I have encoded the actions into an array of binaries, with a pair of coordinates. Before I explain how I decode this action and apply the action into the game interface, let me first explain the movement mechanic of CSGO.

There are multiple types of movements in CSGO, and their mechanical difficulty to the player can vary from beginner-level — straightforward, no complex combinations of buttons — to a very advanced-level — complex, complex combinations of button that requires timing.

I concern ourselves with beginner-level manuever since I do not consider learning these advanced-level movement part of our task — I are only concerned with playing the game, and simple movement will

suffice. Below I talk about the implementation of action from encoded form, to being applied to Game Interface.

The first 3 binaries basically configures the keyboard movement of the agent. Keyboard movement controls the physical movement that causes location changes in the game.

	1	0
A[0]	No Keyboard Movement allowed	Keyboard Movement Allowed
A[1]	Walk	Run
A[2]	Crounch	Uncrouch

Here I want to define the different configuration of movement. By default, the agent's movement configuration will be 'Run'. This is its fastest movement configuration, but it comes at the cost of making noise and increasing recoil. Making noise in the game can expose your location to the enemy agent if it hears you, and that increased recoil makes it harder to shoot with precision, even with perfect aim. When 'shift' button is held while moving, the agent enters into 'Walk' movement configuration. This is a slower movement configuration, but it makes no noise and does not increase recoil as much. Manoeuvring with no noise is strategically important in the game as it introduces uncertainty in the opposing agent, making the agent's location unpredictable, and effectively making the agent's strategy unclear to the opposing agent. The last configuration is 'Crouch'. This movement configuration is the slowest, and it puts the agent in a crouching position, but it makes no noise and does not increase recoil at all. 'Crouch' is usually required to navigate through low-ceiling places (not present in the map I have), but that's not the purpose as to why I chose to include it. I included it as crouching improves the agent's shooting precision, and I want to see how the agent will utilise this position to its advantage.

Then the next binary controls if the agent jumps, while the next 2 binaries determine the direction of keyboard movement

	1	0
A[3]	Jump	Don't jump

	(0,0)	(0,1)	(1,1)	(1,0)
(A[4],A[5])	Move Forward	Move Left	Move Backward	Move Right

The next 4 binaries each controls the mouse movement. Mouse movement changes the orientation of the character being controlled. The first pair of binaries control horizontal mouse movement, while the second pair controls vertical mouse movements. I use XOR to determine the direction of mouse movement in a given axis. (Pixel is the unit used in the X,Y coordinate system of a screen)

	(0,0)	(0,1)	(1,1)	(1,0)
(A[6],A[7])	No Mouse Movement	Move Mouse to the left by 5 pixel	No Mouse Movement	Move Mouse to the right by 5 pixels

	(0,0)	(0,1)	(1,1)	(1,0)
(A[8],A[9])	No Mouse Movement	Move Mouse to the up by 5 pixel	No Mouse Movement	Move Mouse to the down by 5 pixels

I have used an extra bit (here bit means the datatype bit) of information here — instead of encoding it in 3 bits — as I want these actions to happen simultaneously. In the game, it is common to manuever the

mouse diagonally, and to simulate this diagonal movement, I utilise 4 bits so that horizontal and vertical movement can happen simultaneously. There might be more complex method out there that utilises 3 bits of information to achieve this, but I want to keep the implementation as simple as possible, even at the cost of increasing the action space by a factor of 2 due to the extra 1 bit of information.

Lastly the last binary determine whether the agent Aim and Fire. Recognise that this is a combined action as the agent performs both aiming and firing in a single action, and this is designed by choice so that I keep the action space small. Aiming here is assisted by the computer vision system through the parsing of the enemy screen location coordinate to the agent. This pair of coordinate will made up the last 2 element of the array.

	1	0
A[10]	Fire gun	Don't Fire gun

(A[11], A[12])	Coordinate of enemy on screen if enemy is detected, else (np.nan, np.nan)

I want to point out here that this is a very 'aggressive' and naive play style assumption made by us, and that it might not translate as well at more advanced level gameplay. However now it will suffice. To give more context to understand why it will suffice, there are times whereby it is worth to only aim without immediately shooting, and this is called 'trigger discipline'. The purpose of trigger discipline is to find a better time to shoot. One scenario when trigger discipline will be important is when an agent is leading its team, and is unaware that there is an enemy player aiming at it. If the enemy agent were to practice trigger discipline, it potentially can hide its presence from the enemy team for longer, allowing it to get more information about the enemy team by watching the enemy team's movement, and potentially even catching all of them off guard at the right timing and eliminating them all. Back to the context of our task, there are little time where you would not want to kill the enemy agent immediately after aiming at it, as it is the last enemy standing, meaning there is no longer a need to consider any future benefits of keeping the enemy agent alive as I can just eliminate the enemy agent and end the round with a win. As such, trigger discipline is not required of our agent.

VI : Results and Critic

Results

To test the performance of the agent, I had decided to have it play against a bot for 100 rounds, and get the win rate. Out of the 100 rounds, the agent managed to win 39 rounds, and as such I conclude that the agent did not manage to outperform the bots. Below shows the match history for the 100 rounds. The video of the gameplay is available here [video link 2]. Since the game play style of the agent was terrible, I have decided to omit the Turing test aforementioned, as the result of the Turing test is very clear.

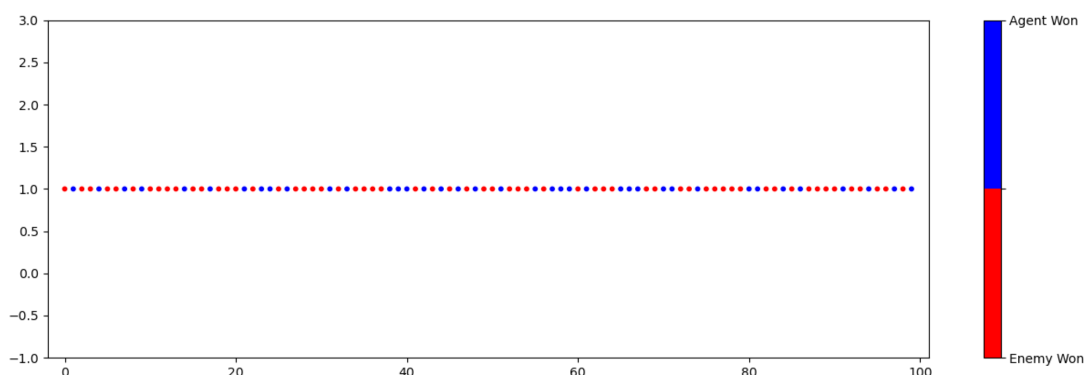


Figure (5): Match history of the 100 testing rounds. Red represents a round won by Enemy, while Blue represents a round won by Agent.

The results was really unexpected and I will elaborate on this further. The agent has learnt a policy solely meant to play against bots. It has exploited the fixed behaviours of the bots, and the bot's terrible aim to its advantage. Firstly, it abused the fact that bots will approach a situation more cautiously and slowly if gunshots are heard. This meant that meaningless shooting was an effective strategy to run time down, which contributes to winning. Secondly it has abused the bot's terrible aim, which was made terrible by choice. Bots takes around 1-2 seconds to adjust its crosshair before firing. This meant that constant movements by the agent will give the agent higher chances of not getting shot at. These 2 points are well reflected when you see the agent play. The agent employs a pretty random play style, it constantly moves around, and fire shots once in a while. There is also an issue with aim, but this is likely due to latency of information and action, and this will be explained in VII.

Critic

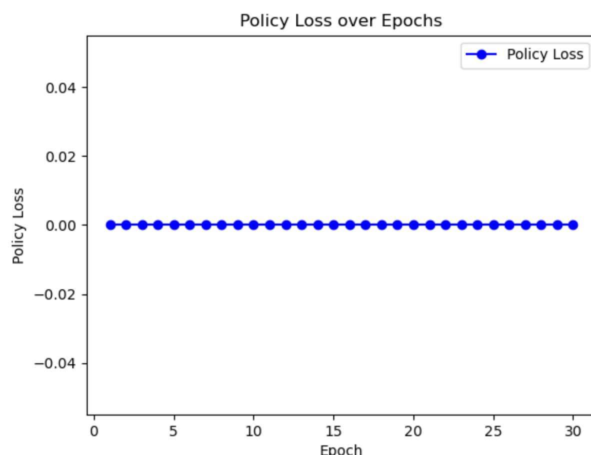
By looking at the result, I had a few ideas of what went wrong with my implementation. Firstly, I felt that the way I handled mouse control was insufficient. The way that I have encoded mouse actions has made it very difficult for the agent to control the mouse freely, and this severely hindered exploration for the agent. Since mouse movements were hard-coded, it was difficult for the agent to explore other potential orientations available to it, effectively hindering learning.

Secondly, my visual system was too naïve. Optimally, the agent should turn towards an unseen enemy player when it is being shot at, by using available in-game image cues. However, due to me choosing to omit this information, the agent could not use the cues to geo-locate the enemy or even orientate itself to face the enemy for a duel; and as such what usually happens is that the agent does not orient itself towards to enemy when it is being shot at. If a more sophisticated visual model is seen, I can potentially see the agent being able to shoot back at the enemy when it is being shot at.
Thirdly, the lack of sound cues also made geo-locating enemy difficult. By omitting sound cues, I have taken away the information that the agent could have used to pre- empt the enemy. This is so as sound cues gives a sense of where another player is going, and how close it is to the listening player. With such information, and especially with Asymmetric Actor Critic ,the agent might be able to learn to geo-locate the opponent with sound cues to a sufficient accuracy; as sound cues information from partial states could be cross-checked with location information from complete state. In a supervised learning sense, the network will learn how to predict enemy location via sound cue information. This system could even be separated out as an individual entity by itself, and trained like a Supervised Learning model, with inputs as in-game noise, and labels as enemy location.

Fourthly, I should have further broken down the task, so that the agent is able to learn more effectively. This is evident in the fact that it has not learnt to stop the enemy from defusing the bomb. I should have started out with simpler tasks, and progress the complexity of the task. For example, I could have isolated 'stopping bomb defuse' as a individual task, then have the agent learn this task by having it run through multiple bomb defuse scenarios. After the agent has learnt the bomb defuse scenario, I can transfer this agent into a more complex task, like ours, and have it learn this task. This potentially might lead to better performance, but it is also to note that this can lead to unexpected behaviours as well. For example, due to the many interactions with bomb defuse scenarios, the agent might be biased to play around the bomb, and this can limit exploration.

Lastly, the neural network I had used is too naive for the current task and this is evident in the graph showing policy loss throughout training. The policy loss throughout training has been 0, and I initially thought that it just needed time to explore, but up until the end, it remains as 0, and this is very concerning. Below shows the graph of the policy loss with respect to epoch

Figure(6) Policy loss over 30 training Epochs



VII : Limitations

Computation Power was the main limitation that I faced. I have tried to defer the computational demand by the project to reduce the load that I was putting on each computer, but it was still too much on the computers. This I very evident in the severe latency in the receiving of state information and application of action, and the latency had significant impact on the performance on the agent, especially its aim. Due to the time difference in receiving the coordinate of the enemy agent, and actual shooting done by the Player, the enemy has time to reposition itself, causing its position on screen to change, which results in really bad aim majority of the time. Additionally, due to the latency, the agent's interaction with the Environment was very limited given the time limit. If there was no latency, the agent could have interacted more with the environment, and as such learn better. Furthermore with higher computation power, I could have implemented a more complex Neural Network without worrying about training time.

Additionally, due to cost of training and speed of training, the agent was not given much training. It is very much possible that if given enough time, it might perform better. However unlikely this is, due to the simplicity of the Neural Network, it is still worth noting that training the agent on a single-instance for 5 days is insufficient for the agent to learn anything meaningful. For benchmark, OpenAI 5 experienced about 45,000 years of Dota 2 self-play in order to achieve its performance, and it shows the severe lack of training being done on the agent.

Furthermore, I also want to note that the way I have trained the agent might have put limits on its learning. There is two parts to this. Firstly, I have implemented our exploration of the action space by generating random actions. It is highly unlikely for the agent to encounter any meaningful sequence of actions, and made worse, the probability decays exponentially as the length of the sequence increase. It is thus very likely that the agent did not undergo any meaningful exploration in the action space. The second issue has to deal with the difficulty of the bots. The bots used in training were too passive, and as such had allowed the agent to picked up a strategy that will otherwise not work with a human player. To give an example, the bots' aim is terrible, and the agent abused this by constantly moving around so as to not get hit by shots and run down time.

IIX: Future Endeavours

I felt that I was too Naïve and should have done more research with the different kinds of Neural Network architecture and their different purpose. I have used a relatively Naïve implementation of Recurrent Neural Network, but on hindsight I should have used Long-Short Term Memory Neural Networks instead. The advantage that LSTM have over naïve RNNs is that it deals with long term dependencies better than RNNs [53]. I felt like what my system lacked was the lack of temporal awareness when it comes to choosing actions, making the agent's control look very incoherent and random. This could also be improved with some sort of AI planning system, which can allow the agent to plan its movements.

If given the computation power and hardware, I would also have liked to implement the self-play learning done by Deep Mind for Quake III. I would like to argue that part of the reason why the agent failed to learn the CSGO problem was due to the fact that it is trained against a predictable and low skill level player – the bot. With the bot being extremely low skilled, the agent does not get interactions that are representative of the usual high skill game that I want it to learn. Furthermore, the predictability and low skill level of the enemy bot can cause the agent to optimise on a strategically low-level policy as the agent exploits the weakness. This explains the constant movement and shooting done by the agent. The constant moving was meant to exploit the enemy bot's slow aiming skill, which buys the agent some time; while the constant shooting was an act to deter the enemy bot to get near, which works on the bot as the bot is hardwire to play more cautiously when gun shots are heard. This too buy the agent time. It is interesting to note however that this strategy was legitimately discovered by the agent, and that it did won the agent some games. However, the agent will mostly likely not perform when playing against humans, as this strategy is easily broken by using sound cues to locate agent, and having proper aim. If I were to train the agents by using self play, not only will I be able to train the agents to play both teams, but the exploration in the *strategic space will also be wider, since the agents are constantly learning, unlike bots which are

stagnant, and thus can only offer a small subsection of the strategic space for the agent to explore.
***by strategic space, I mean a set of sequence of meaningful actions that human CSGO players will find to have strategic value**

IX: Links to Video

[video link 1] https://www.youtube.com/watch?v=eC1XbG9ezdk
[video link 2] https://youtu.be/UjUoyWqHDyc
[video link 3] https://www.youtube.com/watch?v=WXuK6gekU1Y

REFERENCE

[1] https://bigcloud.global/the-evolution-of-ai-in-gaming/

[2] https://www.deepmind.com/research/highlighted-research/alphago

[3] https://openai.com/research/openai-five-defeats-dota-2-world-champions

[4] Patel, U.K., Patel, P., Hexmoor, H. *et al.* Improving behavior of computer game bots using fictitious play. *Int. J. Autom. Comput.* **9**, 122–134 (2012). https://doi.org/10.1007/s11633-012-0625-5

[5] Reinforcement Learning: An Introduction second edition, Francis Bach

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforce- ment learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[7] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welin- der, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, “Hindsight experience replay,” *arXiv preprint arXiv:1707.01495*, 2017.

[8] arXiv:1710.06542 **[cs.RO]**

[9] T. Schaul, D. Horgan, K. Gregor, and D. Silver, “Universal value function approximators,” in *ICML 2015*.

[10] https://datacarpentry.org/image-processing/07-thresholding/

[11] https://pyimagesearch.com/2021/02/22/opencv-connected-component-labeling-and-analysis/

[12] https://github.com/ultralytics/yolov5

[13] https://developer.valvesoftware.com/wiki/Counter-Strike:_Global_Offensive_Game_State_Integration

[14] https://www.reddit.com/r/GlobalOffensive/comments/cjhcpy/game_state_integration_a_very_large_and_indepth/ {Its reddit but its a very informative post}

[15] https://apexminecrafthosting.com/how-to-open-console-on-csgo/

[16] https://github.com/Lucid1ty/Yolov5ForCSGO

[17] arXiv:2104.04258 **[cs.AI]**

[18]Russell Stuart J and Peter Norvig. 2020. Artificial Intelligence : A Modern Approach. 4th ed. Boston: Pearson.

[19]Michael Wooldridge. 2009. An Introduction to MultiAgent Systems (2nd. ed.). Wiley Publishing.

[20] https://medium.com/analytics-vidhya/policy-gradients-in-deep-reinforcement-learning-83d99575cfca

[21] https://ai.stackexchange.com/questions/5246/what-is-sample-efficiency-and-how-can-importance-sampling-be-used-to-achieve-it

[22] https://medium.com/analytics-vidhya/deep-deterministic-policy-gradient-for-continuous-action-space-9b2b9bacd555

[23] https://en.wikipedia.org/wiki/Counter-Strike:_Global_Offensive

[24] https://en.wikipedia.org/wiki/First-person_shooter

[25] https://www.v7labs.com/blog/yolo-object-detection

[26] IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 1, NO. 1, MARCH 2009

[27] https://thenewstack.io/deep-learning-ai-generates-realistic-game-graphics-by-learning-from-videos/

[28] https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov

[29] https://www.deepmind.com/research/highlighted-research/alphago

[30] https://openai.com/research/openai-five

[31] https://www.theverge.com/2019/1/24/18196135/google-deepmind-ai-starcraft-2-victory

[32] https://openai.com/research/openai-five-defeats-dota-2-world-champions

[33] https://left4dead.fandom.com/wiki/The_Director

[34] https://left4dead.fandom.com/wiki/Left_4_Dead

[35] IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 9, NO. 1, MARCH 2017

[36] https://www.whitman.edu/documents/Academics/Mathematics/2019/Felstiner-Guichard.pdf

[37] https://arxiv.org/pdf/2106.08717.pdf

[38] https://towardsdatascience.com/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1

[39] S. Yoon and K. -J. Kim, "Deep Q networks for visual fighting game AI," *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, New York, NY, USA, 2017, pp. 306-308, doi: 10.1109/CIG.2017.8080451.

[40] arXiv:1807.01281v1 [cs.LG] 3 Jul 201

[41] https://quake.fandom.com/wiki/Capture_the_Flag_(Q3)

[42]https://dignitas.gg/articles/blogs/CSGO/12354/csgos-hierarchy-of-skill

[43] https://www.youtube.com/watch?v=O9Dg1lPPMmM

[44] https://www.youtube.com/watch?v=Sh_t0at5-iw

[45] picture with crosshair here

[46]https://doi.org/10.1016/B978-0-08-045089-6.X5001-1

[47]https://esportsinsider.com/2021/06/csgo-strategies-bayes-esports

[48] https://www.readingrockets.org/article/higher-order-thinking#:~:text=What%20is%20higher%20order%20thinking,I%20call%20that%20rote%20memory.

[49] Whitelam, S., Selin, V., Park, SW. et al. Correspondence between neuroevolution and gradient descent. Nat Commun 12, 6317 (2021). https://doi.org/10.1038/s41467-021-26568-2

[50] Max Jaderberg et al. ,Human-level performance in 3D multiplayer games with population-based reinforcement learning.Science364, 859-865(2019).DOI:10.1126/science.aau6249

[51] arXiv:1611.05397 [cs.LG]

[52] CS self-play AI agent with object detection and imitation training

[53] https://www.shiksha.com/online-courses/articles/rnn-vs-gru-vs-lstm/

[54] https://towardsdatascience.com/target-networks-slow-and-steady-wins-the-race-214ed14e97e7

[55] https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f