TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT REPORT

ON

# AUGMENTING SELF-LEARNING AGENT IN FIRST-PERSON SHOOTER GAME USING REINFORCEMENT LEARNING

**SUBMITTED BY:**

SAMRAT SINGH ( PUL075BEI031 )

SKEIN NEUPANE ( PUL075BEI037 )

SUSHANT PANDEY ( PUL075BEI045 )

YACHU RAJA JOSHI ( PUL075BEI048 )

**SUBMITTED TO:**

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

April 30, 2023

# Page of Approval

TRIBHUVAN UNIVERSIY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certifies that they have read and recommended to the Institute of Engineering for acceptance of a project report entitled **"Augmenting Self-Learning Agent in First-Person Shooter game using Reinforcement Learning"** submitted by **Samrat Singh**, **Skein Neupane**, **Sushant Pandey**, **Yachu Raja Joshi** in partial fulfillment of the requirements for the Bachelor's degree in Electronics & Computer Engineering.

.............................
Supervisor
**Prakash Chandra Prasad**
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.............................
Internal examiner
**Person B**
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.............................
External examiner
**Person C**
Assistant Professor
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

Date of approval:

# Copyright

# Acknowledgments

We extend our deepest gratitude and appreciation to all those who have provided us with unwavering support and assistance in the successful completion of this collaborative project.
We would like to express our sincere appreciation to our project supervisor, **Assistant Professor Prakash Chandra Prasad**, for his exceptional guidance, unwavering support, and constant encouragement throughout our project. His profound expertise and unparalleled knowledge have played a pivotal role in shaping the direction and scope of our work, and his constructive feedback has been pivotal in refining and enhancing our project.
We express our gratitude to the faculty members of the **Department of Electronics and Computer Engineering** for equipping us with the essential resources and facilities that facilitated the successful completion of this project. Their technical and academic support played a significant role in ensuring the triumph of this endeavor.
We also extend our sincere appreciation to our colleagues and classmates whose invaluable assistance and support throughout the project were instrumental in shaping and enhancing our work.
Lastly, we acknowledge our families and friends for their unwavering support, encouragement, and understanding throughout our academic journey. Their continuous love and support have been a constant source of inspiration and strength.
We thank everyone for their encouragement, guidance, and support, and for making this group project an extremely enriching and gratifying experience.

# Abstract

This group project highlights the effectiveness of utilizing reinforcement learning (RL) along with the Proximal Policy Optimization (PPO) algorithm to train an agent to play a Wolfenstein3D-like game with multiple levels. The agent exhibited exceptional performance in relation to reward, time efficiency, and overall effectiveness. An in-depth analysis of its performance indicated marked enhancements in the reward curves, strategic navigation throughout the game levels, and expeditious completion of each level. The study highlights the potential of RL and PPO for training agents in complex video games with multiple levels, as well as in other applications such as agent-based modeling and machine learning.

Keywords: *Reinforcement Learning, Deep Reinforcement Learning, Proximal Policy Optimization, Curriculum Learning, Reward Shaping*

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **ANN** | Artificial Neural Network |
| **API** | Application Program Interface |
| **A2C** | Advantage Actor-Critic |
| **A3C** | Asynchronous Advantage Actor-Critic |
| **CNN** | Convolutional Neural Network |
| **DDA** | Digital Differential Analyzer |
| **DDPG** | Deep Deterministic Policy Gradient |
| **DP** | Dynamic Programming |
| **DQN** | Deep Q-Network |
| **FOV** | Field of View |
| **FPS** | First Person Shooter |
| **GAE** | Generalized Advantage Estimation |
| **IMPALA** | Importance Weighted Actor-Learner Architecture |
| **KL** | Kullback-Leibler |
| **MDP** | Markov Decision Process |
| **MC** | Monte Carlo |
| **ML** | Machine Learning |
| **MLP** | MultiLayer Perceptron |
| **PG** | Policy Gradient |
| **PPO** | Proximal Policy Optimization |
| **RL** | Reinforcement Learning |
| **SARSA** | State-Action-Reward-State-Action |
| **SGD** | Stochastic Gradient Descent |
| **TD** | Temporal Difference |
| **TRPO** | Trust Region Policy Optimization |
| **UI** | User Interface |
| **VGA** | Video Graphics Array |
| **VR/AR** | Virtual Reality / Augmented Reality |

# 1. Introduction

## 1.1 Background

Reinforcement Learning(RL) is a powerful machine learning technique in which an agent learns to solve a problem by interacting with its environment. Through a system of rewards and punishments, agents learn to make decisions that maximize rewards and minimize negative consequences. RL is rapidly becoming a prominent area of research, with a growing number of applications in a wide range of fields [3].

One area where RL has shown particular promise is in the development of first-person shooter(FPS) bot artificial intelligence. In FPS games, RL has the potential to create diverse behaviors without the need for explicit coding. Unlike traditional arcade games, FPS games offer a more realistic study environment for RL, with 3D graphics and partially viewable states.

Furthermore, FPS games directly simulate reality from a first-person perspective, presenting agents with new, competitive objectives like navigating and shooting. The ability to manage these objectives is highly desired for general intelligence. The availability of well-designed game environments also accelerates the development of RL algorithms [4].

Overall, RL holds great potential for creating intelligent agents that can operate in complex, dynamic environments. The use of FPS games as a testing ground for RL algorithms is particularly promising, and is likely to drive significant advances in the field of artificial intelligence in the coming years.

## 1.2 Problem statements

Training agents to operate successfully in 3D multiplayer games based on raw pixel inputs is a difficult task and a growing area of interest in AI research. Adding AI to games can improve Non-Player Character(NPC) intelligence and balance game complexity. The focus of this project is on training agents in a First-Person Shooting (FPS) game environment. The game consists 3 different levels each with a different objective and finish conditions. The primary objective is to replicate a human player and play against in-game NPCs and complete the levels.

## 1.3 Objectives

The objective of this project is to train a competent agent using RL in FPS games, with the aim of exhibiting human-like behaviors and outperforming both average human players and existing in-built game agents.

## 1.4 Scope

Following are the scope of our project:

- Implement RL algorithm for FPS games
- Train an agent to navigate game environment and compete against human players or in-game agents
- Explore and compare different RL algorithms for optimization
- Evaluate and analyze agent's performance and behavior in various game scenarios
- Investigate the impact of different reward functions on the learning process and performance of the agent.
- Investigate the use of curriculum learning to design a training curriculum that gradually increases the complexity of tasks and environments, enabling the agent to learn more effectively and efficiently.

## 1.5 Applications

Our project has a wide range of applications in various fields:

- **Gaming**-The development of more advanced and engaging video games with smarter in-game agents
- **Security**-The potential to use the intelligent agents for security and surveillance.
- **VR/AR**-The use of intelligent agents in virtual reality and augmented reality applications for training and simulation
- **Autonomous Robots**-The development of autonomous robots that can learn and adapt to new environments

- **Real-World Problems**-The potential to apply reinforcement learning to a wider range of real-world problems beyond gaming, such as traffic management or financial trading.

- **IEEE ViZDoom Competitions**- The agents trained in an FPS environment are able to compete in a multiplayer death match FPS game Doom.

# 2. Literature Review

## 2.1 Related work

Reinforcement Learning (RL) has been utilized extensively in the development of intelligent agents capable of playing games. Recent advancements in RL have led to significant progress in game-playing agents for various games such as Atari, VizDoom, and Wolfenstein. This literature review provides an overview of the advancements in these three domains, highlighting the key techniques, algorithms, and architectures used to build intelligent game-playing agents.

The Atari 2600 platform was introduced in the 1970s and included popular games such as Space Invaders, Pac-Man, and Breakout. In 2013, DeepMind introduced the Atari 2600 games as a benchmark for RL algorithms, which led to significant advancements in the field. [5] introduced the Deep Q-Network (DQN) algorithm, one of the earliest and most influential papers in this domain.

DQN is a value-based RL algorithm that learns to estimate the value of each action in each state. The algorithm utilizes a neural network to approximate the value function, and a replay memory to store past experiences. DQN achieved state-of-the-art results on several Atari games, even outperforming human players in some cases. However, DQN had several limitations, including slow learning and a tendency to overestimate the value function.

To address these limitations, several variants of DQN were introduced, including Double DQN [6], Dueling DQN [7], and Rainbow [8]. Double DQN addressed the overestimation issue by decoupling the selection and evaluation of actions. Dueling DQN introduced a new architecture that separates the estimation of the state value and the advantage of each action. Rainbow combined several techniques, including double Q-learning, prioritized experience replay, and distributional RL, to achieve state-of-the-art results on the Atari benchmark.

VizDoom ,a first-person shooter game that was introduced in 2016 as a benchmark for RL algorithms. The game includes several challenging scenarios, including navigating through a maze and killing enemies. The VizDoom benchmark presented several challenges that were not present in the Atari games, including partial observability and continuous action spaces.

Several RL algorithms have been applied to VizDoom, including DQN [9], Asynchronous

Advantage Actor-Critic (A3C) [10], and Proximal Policy Optimization (PPO) [11]. DQN achieved state-of-the-art results on some of the simpler scenarios, but struggled on more complex scenarios that required long-term planning. A3C and PPO both achieved state-of-the-art results on the VizDoom benchmark, with PPO outperforming A3C on some of the more challenging scenarios.

Wolfenstein 3D, a first-person shooter game that was introduced in 1992. In 2017, DeepMind introduced a version of Wolfenstein as a benchmark for RL algorithms. The Wolfenstein benchmark presented several challenges, including the need to navigate through a complex 3D environment and interact with enemies that could shoot back.

Several RL algorithms have been applied to Wolfenstein, including DQN [12], Asynchronous Actor-Critic (A2C) [13], and IMPALA [14]. DQN achieved state-of-the-art results on some of the simpler scenarios, but struggled on more complex scenarios.

## 2.2 Related Theory

### 2.2.1 Reinforcement Learning

Reinforcement learning (RL) is a sub-field of machine learning that focuses on how agents can learn to make optimal decisions in dynamic and uncertain environments. RL has shown great success in a wide range of applications, including robotics control, game playing, and recommendation systems. As stated by Sutton and Barto, "Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal" [15]. RL algorithms operate by trial and error, where the agent learns from feedback in the form of rewards or punishments, which encourages it to make better decisions. The success of RL lies in its ability to learn from experience, allowing it to improve its decision-making abilities over time.

### 2.2.2 Agent-Environment Interface

Figure 2.1 illustrates the agent-environment interaction in reinforcement learning, where the agent takes actions in the environment, receives rewards, and observes the resulting state. As Sutton and Barto state, RL consists of three key components: "an agent, an environment, and a reward signal" [15]. The agent is responsible for taking actions within the environment, with the goal of maximizing the cumulative reward it receives. The environment, on the other hand, is the context in which the agent operates, consisting of a set of states and the rules that dictate how the agent transitions between them. The reward signal is a numerical value that the agent receives from the environment after each action it takes, and it is used to guide the agent's learning process. By learning from experience through the feedback provided by the reward signal, the agent can improve its decision-making abilities over time. These three components are fundamental to the RL framework, and their interaction is essential to the learning process.

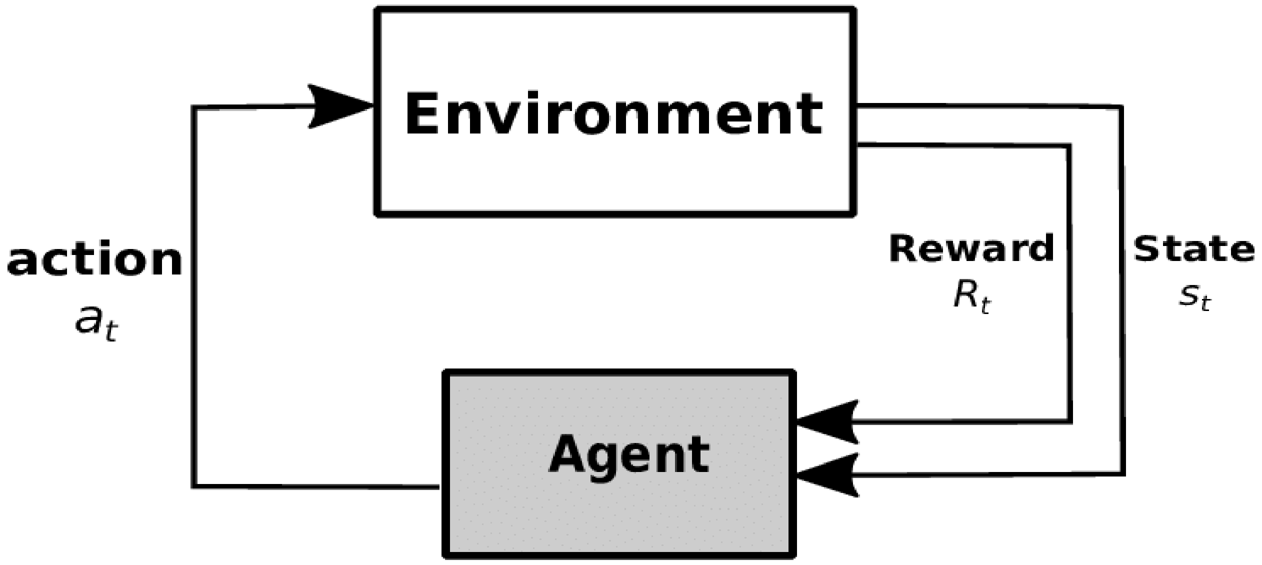

Figure 2.1: The agent-environment interaction in reinforcement learning.

### 2.2.3 Terminologies

Reinforcement learning (RL) has its own set of terminology that is used to describe the various concepts and algorithms within the field. Some of the key terms used in RL include:

**State:** A state is the condition of the environment that the agent is in.

**Action:** An action is the choice made by the agent in response to the state it is currently in.

**Reward:** A reward is a signal that the agent receives in response to its action in a particular state, that defines how good the action was.

**Policy:** A policy is the mapping from states to the actions the agent should take in those states.

**Value function:** The value function estimates the expected total reward an agent will receive in the future from a given state or action.

**Q-value:** The Q-value is the expected total reward an agent will receive in the future if it takes a particular action in a particular state.

**Discount factor:** A discount factor is a value between 0 and 1. It is multiplied to the reward signal to minimise the impact of future rewards.

**Model:** A model is an internal representation of the environment that the agent uses to predict how the environment will respond to its actions.

**Episode:** An episode is a single instance of the agent's interaction with the environment, starting from an initial state and ending when the agent reaches a terminal state.

**On-policy learning:** On-policy learning is a type of RL algorithm where the agent learns from experience that was generated by following the same policy that is being updated.

**Off-policy learning:** Off-policy learning is a type of RL algorithm where the agent learns from experience that was generated by following a different policy than the one being updated.

### 2.2.4 Key Characteristics of RL

1. **Trial and Error Learning:**
   Trial and error learning is a process by which the agent learns to take actions based on the feedback provided by the environment. The agent tries different actions and learns from the rewards received, gradually improving its behavior over time. Reinforcement learning algorithms use trial and error learning to discover the optimal policy that maximizes the cumulative reward over time [15].

2. **Goal-oriented Learning:**
   Reinforcement learning is a goal-oriented learning process, where the goal of the agent is to learn a policy that maximizes the cumulative reward over time. The agent learns to take actions that lead to desirable outcomes and avoids actions that lead to undesirable outcomes. The goal-oriented nature of reinforcement learning makes it well-suited for a wide range of applications, such as game playing, robotics, and autonomous driving [15].

3. **Exploration and Exploitation:**
   Exploration refers to the process of trying out new actions that the agent has not yet tried in order to learn about the environment and discover new strategies that can lead to higher rewards. In reinforcement learning, exploration is necessary to avoid the agent getting stuck in a suboptimal policy and to learn about the environment's unknown aspects.

   Exploitation refers to the process of selecting actions that the agent has previously found to be optimal in order to maximize the cumulative reward over time. In reinforcement learning, exploitation is necessary to ensure that the agent continues to take actions that have previously yielded high rewards. The exploration-exploitation tradeoff is a fundamental problem in reinforcement learning. The agent must balance its exploration of new actions with its exploitation of actions that have previously yielded high rewards. If the agent explores too much, it may waste time and resources trying out actions that lead to lower rewards, but if it exploits too much, it may miss out on potentially higher rewards that could be obtained by trying out new actions [15].

4. **Reward Signal:**
   The reward signal is the feedback provided by the environment to the agent, indicating the quality of the actions taken by the agent. The reward signal is used to guide the learning process of the agent, encouraging it to take actions that lead to higher rewards. The reward signal can be either positive or negative, depending on whether the action taken by the agent led to a desirable or undesirable outcome [15].

5. **Sequential Decision-Making:**
   Reinforcement learning is concerned with sequential decision-making, in which the agent takes a series of actions over time, with each action affecting the state of the environment and the rewards received. The agent learns to take actions that maximize the cumulative reward over time, taking into account the long-term consequences of its actions. The sequential decision-making nature of reinforcement learning makes it well-suited for applications such as decision making in finance, healthcare, and logistics.

### 2.2.5 Distinction from other forms of Machine Learning (ML):

Reinforcement learning (RL) differs from other machine learning algorithms such as supervised learning and unsupervised learning in several ways. In supervised learning, the algorithm is trained on labeled examples, whereas in RL, the agent learns through trial and error. In unsupervised learning, the algorithm learns to identify patterns in unlabeled data, whereas in RL, the agent learns to select actions that maximize the cumulative reward signal. Additionally, RL can handle problems with delayed rewards and continuous and high-dimensional state and action spaces, which makes it suitable for a wide range of applications [15].

### 2.2.6 Most Popular Breakthroughs in RL

RL has had several breakthroughs in recent years. Some of the most popular breakthroughs in RL are as follows:

- **Deep Q-Networks (DQN):** DQN is a deep RL algorithm that was introduced by [5]. DQN combines Q-learning with deep neural networks to enable agents to learn from high-dimensional sensory inputs such as images.
- **AlphaGo:** AlphaGo is a deep RL algorithm that was introduced by [16]. AlphaGo combines RL with deep neural networks to play the board game Go at a superhuman level, surpassing the performance of human experts.
- **AlphaZero:** AlphaZero is a deep RL algorithm that was introduced by [17]. AlphaZero combines RL with deep neural networks to play the board games Go, chess, and shogi at a superhuman level, without any prior knowledge of the game rules.
- **OpenAI Five:** OpenAI Five is a deep RL algorithm that was introduced by OpenAI in 2019. OpenAI Five combines RL with deep neural networks to play the multiplayer online battle arena game Dota 2 at a professional level, surpassing the performance of human teams [18].

### 2.2.7 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in situations where outcomes are uncertain and depend on previous states and actions. According to a research paper by Kaelbling, Littman, and Moore, an MDP consists of "a set of states, a set of actions, a transition function, a reward function, and a discount factor" [19]. In an MDP, the agent interacts with the environment by taking actions in a current state, transitioning to a new state, and receiving a reward based on the action taken and the new state reached.

The key components of an MDP are:

- **States**: These are the different possible situations or configurations that the decision-maker can find themselves in. In a given state, the decision-maker has some information about their environment, but may not know everything. The set of states is denoted by $S$.

- **Actions**: These are the different possible choices or decisions that the decision-maker can make. Actions are chosen based on the current state and the decision-maker's goals. The set of actions is denoted by $A$.
- **Transition probabilities**: These describe the probability of moving from one state to another when a particular action is taken. The transition probabilities are denoted by $P(s'|s, a)$, where $s$ and $a$ are the current state and action, respectively, and $s'$ is the next state.
- **Rewards**: These are the immediate benefits or costs associated with each action. The rewards are denoted by $R(s, a)$.
- **Policy**: This is the decision-making rule used by the decision-maker to choose actions based on the current state. The policy is denoted by $\pi(a|s)$.

An MDP is said to satisfy the Markov property if the probability of transitioning to a future state only depends on the current state and the action taken, and not on the history of past states and actions. This property simplifies the analysis and computation of optimal policies.

The goal of an MDP is to find a policy that maximizes the expected cumulative reward over time. The cumulative reward is given by the sum of the rewards received at each time step, weighted by a discount factor that reflects the importance of immediate versus future rewards.

MDP is explained by the mathematical equations as:

```
**Inputs:**
    State space $S$
    Action space $A$
    Reward function $R(s, a)$
    Transition function $P(s'|s, a)$
**Outputs:**
    Policy $\pi(s)$
    Value function $V(s)$
**Formulation:**
    For each state $s \in S$, initialize $V(s)$ arbitrarily.
    **Repeat until convergence:**
        **For each state** $s \in S$ :
            $V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s'|s, a)\,[R(s, a, s') + \gamma V(s')]$   (2.1)
    **For each state** $s \in S$ :
        $\pi(s) \leftarrow \operatorname{argmax}_{a \in A} \sum_{s' \in S} P(s'|s, a)\,[R(s, a, s') + \gamma V(s')]$   (2.2)
```

In the above formulation, an MDP is defined by a state space S, an action space A, a reward function R(s,a), and a transition function P(s'—s,a). The goal is to find a policy $\pi$(s) that maximizes the expected total reward over time. The value function V(s) represents the expected total reward starting from state s and following policy $\pi$(s) [15].

## 2.2.8 Policy and Value Functions

In reinforcement learning, a value function represents the expected return starting from a particular state and following a specific policy. The value function can be defined recursively using the Bellman equation:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V^{\pi}(s')]. \tag{2.3}$$

where $V^{\pi}(s)$ is the value function for state $s$ under policy $\pi$, $a$ represents an action, $s'$ is the resulting state, $P(s'|s, a)$ is the probability of transitioning to state $s'$ from state $s$ under action $a$, $R(s, a, s')$ is the immediate reward received after taking action $a$ in state $s$ and transitioning to state $s'$, $\pi(a|s)$ is the probability of selecting action $a$ in state $s$, and $\gamma$ is a discount factor.

On the other hand, a policy function specifies the action to be taken in a given state. A policy can also be evaluated using a similar recursive definition:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s')Q^{\pi}(s',a')]. \tag{2.4}$$

where $Q^{\pi}(s,a)$ is the action-value function for state-action pair $(s,a)$ under policy $\pi$ [15].

### 2.2.9 Policy Evaluation

Policy evaluation is the process of calculating the state-value function for a given policy. The state-value function is the expected return starting from a given state and following the policy thereafter. The formula for state-value function is given by:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V^{\pi}(s')\right] \tag{2.5}$$

Here, $V^{\pi}(s)$ is the state-value function for policy $\pi$, $\pi(a|s)$ is the probability of taking action $a$ in state $s$, $p(s',r|s,a)$ is the probability of transitioning to state $s'$ and receiving reward $r$ when taking action $a$ in state $s$, and $\gamma$ is the discount factor.

The optimal state-value function, denoted as $V(s)$, is the maximum state-value function over all policies, i.e., $V(s) = \max_{\pi} V^{\pi}(s)$. The optimal state-value function satisfies the Bellman optimality equation:

$$V(s) = \max_{a} \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right] \tag{2.6}$$

The optimal policy $\pi$ can be obtained by acting greedily with respect to the optimal state-value function, i.e., $\pi(a|s) = 1(a = \arg\max_{a'} Q(s,a'))$, where $Q(s,a)$ is the optimal action-value function and $1(\cdot)$ is the indicator function [16].

### 2.2.10 Policy Iteration

Policy iteration is an iterative method for solving a Markov Decision Process (MDP) that involves alternating between policy evaluation and policy improvement. The goal is to find an optimal policy that maximizes the expected cumulative reward.

The steps involved in policy iteration are as follows:

1. Initialize the policy $\pi$ arbitrarily.
2. Evaluate the policy $\pi$ by solving the Bellman equation for $V_\pi$:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')] \tag{2.7}$$

3. Improve the policy by selecting the action that maximizes the expected cumulative reward for each state:

$$\pi'(s) = \operatorname{argmax} a \sum s', rp(s',r|s,a)[r + \gamma V_\pi(s')] \tag{2.8}$$

4. If the policy has not changed significantly, stop and return the optimal policy $\pi$ and the corresponding value function $V_\pi$, otherwise set $\pi = \pi'$ and go to step 2.

In the above equations, $s$ represents the current state, $a$ represents the action taken, $s'$ represents the next state, $r$ represents the reward received, $p(s',r|s,a)$ represents the probability of transitioning to state $s'$ and receiving reward $r$ given the current state $s$ and action $a$, $\gamma$ represents the discount factor, and $\pi'(s)$ represents the new policy [16].

## 2.2.11 Value Iteration

Value iteration is another iterative algorithm in reinforcement learning used for finding the optimal value function for a given MDP. The algorithm starts with an initial estimate of the value function and updates it using the Bellman optimality equation until convergence.

The Bellman optimality equation for the value function can be written as:

$$V_{k+1}(s) = \max_{a\in A} \sum_{s'\in S} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right] \tag{2.9}$$

where $V_k(s)$ is the estimated value of state $s$ after $k$ iterations, $a$ is an action in the action space $A$, $P(s'|s,a)$ is the transition probability from state $s$ to state $s'$ under action $a$, $R(s,a,s')$ is the immediate reward obtained from taking action $a$ in state $s$ and ending up in state $s'$, and $\gamma$ is the discount factor.

The value iteration algorithm can be summarized as follows:

Initialize $V_0(s)$ for all $s \in S$ arbitrarily. For each iteration $k \geq 0$, update the value function $V_{k+1}(s)$ for all $s \in S$ using the Bellman optimality equation:

$$V_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right] \tag{2.10}$$

Repeat step 2 until the change in $V_k(s)$ between two consecutive iterations is less than a predefined threshold [15]. The optimal policy can then be obtained by selecting the action that maximizes the right-hand side of the Bellman optimality equation:

$$\pi(s) = \text{argmax}\, a \in A \sum s' \in SP(s'|s,a) \left[R(s,a,s') + \gamma V^*(s')\right] \tag{2.11}$$

where $V^*(s)$ is the optimal value function, obtained as the limit of $V_k(s)$ as $k \to \infty$.

### 2.2.12 Necessary Mathematical Quantities

A) **Expected Return**

The expected return is the expected cumulative reward that an agent can achieve by following a policy $\pi$ starting from a state $s$:

$$V(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s, \pi], \tag{2.12}$$

where $V(s)$ is the value function of state $s$, $R(s_t, a_t)$ is the immediate reward received when taking action $a_t$ in state $s_t$, $\gamma$ is the discount factor, and $E$ is the expectation operator [16].

B) **Q-Value**

The Q-value is the expected cumulative reward that an agent can achieve by taking a particular action $a$ in a given state $s$ and then following a specific policy $\pi$:

$$Q(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s, a_0 = a, \pi], \tag{2.13}$$

where $Q(s,a)$ is the Q-value function of state-action pair $(s,a)$ [16].

C) **Advantage Function**

The advantage function measures how much better an action is than the average action taken in a given state:

$$A(s,a) = Q(s,a) - V(s), \tag{2.14}$$

where $A(s,a)$ is the advantage function of state-action pair $(s,a)$, $Q(s,a)$ is the Q-value function of state-action pair $(s,a)$, and $V(s)$ is the value function of state $s$ [15].

D) **Bellman Equation**

The Bellman equation is a fundamental equation in reinforcement learning that describes the relationship between the value function and the expected reward:

$$V(s) = E[R + \gamma V(s')|s,a], \tag{2.15}$$

where $V(s)$ is the value function of state $s$, $R$ is the immediate reward, $\gamma$ is the discount factor, $V(s')$ is the value function of the next state $s'$, and $a$ is the action taken in state $s$ [15].

### 2.2.13 Reinforcement Learning Algorithms

Reinforcement Learning (RL) can be broadly categorized into two categories: Value-Based RL and Policy-Based RL. Value-Based RL algorithms aim to learn the value function of a state, which represents the expected cumulative reward that an agent would receive by following a specific action policy from that state. The agent learns to estimate the value function, which can be used to select actions that maximize the expected cumulative reward. On the other hand, Policy-Based RL algorithms aim to learn a policy directly, which is a mapping from states to actions. The agent learns to optimize the policy by adjusting the parameters of the policy directly. Both categories of RL algorithms have their advantages and disadvantages, and the choice of algorithm depends on the specific problem and the characteristics of the environment.

#### 2.2.13.1 Monte Carlo Estimate:

Monte Carlo (MC) is a specific type of reinforcement learning (RL) algorithm that works by simulating complete episodes of the environment and using the observed returns to estimate the value function. MC does not require a model of the environment, which makes it suitable for situations where the environment's dynamics are either unknown or hard to model. MC methods rely on the principle of sampling the returns of complete episodes of the environment, and they estimate the value function of a state by averaging the returns observed in all episodes that visit that state.The update rule for a first-visit MC algorithm is given by:

$$V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t)) \tag{2.16}$$

Where $V(s_t)$ is the estimated value function for state $s_t$, $\alpha$ is the learning rate, and $G_t$ is the observed return starting from state $s_t$ at time $t$. The return is defined as the sum of rewards received from time $t$ to the end of the episode [15].

MC methods offer several advantages compared to other RL algorithms. One major advantage is their unbiasedness, meaning that their estimates approach the true value function with enough samples. Additionally, MC methods can handle environments with stochastic rewards or transitions, making them applicable to a wide range of problems. However, MC methods also come with some limitations. One primary limitation is the high variance of estimates, particularly in scenarios where episodes are long or infrequent. This high variance can hinder the accuracy of value function estimates and slow down the learning process. Another limitation of MC methods is the computational expense of simulating complete episodes of the environment, which can be a challenge in large-scale problems.

#### 2.2.13.2 Temporal Difference:

Temporal Difference (TD) is a kind of reinforcement learning (RL) algorithm that updates its value function by estimating the value of the state-action pairs from the observed experience. TD methods use a blend of Monte Carlo (MC) and dynamic programming (DP) methods, enabling them to update their estimates based on partial experience.

TD algorithms estimate the value function by calculating the temporal difference between the predicted value of a state or action and the observed value of the next state. This difference is used to update the value function of a state-action pair, by adding the difference between the predicted value and the actual reward received. The update rule for a TD(0) algorithm is given by:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.17}$$

where $V(s_t)$ is the estimated value function for state $s_t$, $\alpha$ is the learning rate, $r_{t+1}$ is the reward received at time $t+1$, $\gamma$ is the discount factor, and $V(s_{t+1})$ is the estimated value function for the next state $s_{t+1}$ [16].

TD methods offer several advantages over other RL algorithms. One advantage is that they can learn online, meaning that the value function can be updated after each environment step. This allows the agent to quickly adjust to changes in the environment. Another advantage is that they do not need a complete model of the environment, making them suitable for situations where the environment's dynamics are unknown or difficult to model.

TD methods, despite their benefits, come with some drawbacks as well. One of the main limitations of TD methods is the tradeoff between bias and variance. If the value function estimate is incorrect, TD methods can become biased, and this bias can accumulate over time, leading to inaccurate estimates. However, increasing the number of updates can reduce bias, but it can also increase the variance of the estimates. Another limitation is the curse of dimensionality, where the number of possible states and actions can increase exponentially with the size of the problem, making it challenging to estimate the value function with accuracy.

#### 2.2.13.3 Generalized Advantage Estimation:

The goal of RL is to learn a policy $\pi$ that maximizes the expected cumulative reward over time. To do this, we need to estimate the value function $V(s)$ and the advantage function $A(s, a)$ for each state $s$ and action $a$, which represent the expected cumulative reward starting from state $s$ or taking action $a$ in state $s$, respectively.

The advantage function is defined as:

$$A(s, a) = Q(s, a) - V(s) \tag{2.18}$$

where $Q(s, a)$ is the state-action value function, representing the expected cumulative reward starting from state $s$, taking action $a$, and following policy $\pi$ thereafter. The value function $V(s)$ can be estimated using various methods, such as TD learning or Monte Carlo sampling, but the advantage function is typically harder to estimate accurately.

The GAE technique is a method of calculating the advantage function, which combines TD and Monte Carlo estimates. GAE works by taking an exponentially-weighted average of the TD error signals at different time horizons. This results in a weighted average of the advantages over both short and long-term periods. The degree of weighting is controlled by the lambda parameter ($\lambda$), which helps balance the tradeoff between bias and variance in the estimation process.

The GAE estimate is defined as follows:

$$GAE(t) = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \tag{2.19}$$

where $\delta_{t+k}$ is the TD error signal at time step $t + k$, defined as $\delta_{t+k} = r_{t+k} + \gamma V(s_{t+k+1}) - V(s_{t+k})$, and $\gamma$ is the discount factor. The lambda parameter controls the weighting of the TD error signals at different time horizons, where $\lambda = 0$ corresponds to a purely TD estimate, $\lambda = 1$ corresponds to a purely Monte Carlo estimate, and $0 < \lambda < 1$ corresponds to a combination of both TD and Monte Carlo estimates [1].

**The trade-off between bias and variance:**

Monte Carlo methods estimate the value function by averaging the actual returns obtained from running a complete episode of the RL algorithm. This method is unbiased but has high variance, as it requires a large number of samples to obtain accurate estimates. In contrast, TD methods estimate the value function by bootstrapping from the estimates of the next state, using a temporal difference error signal. This method is biased but has lower variance than Monte Carlo methods, as it can update the value function after each time step.

The GAE method combines both Monte Carlo and TD estimates to obtain a more accurate and stable estimate of the advantage function. Specifically, GAE computes an exponentially-weighted average of the TD error signals with different time horizons, where the weighting factor is controlled by a hyperparameter called the $\lambda$ parameter. This allows for a bias-variance tradeoff in the estimate of the advantage function, resulting in a more accurate and stable estimate.

To see how GAE takes the better aspects of both Monte Carlo and TD estimates, let's start with the TD error signal:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.20}$$

This error signal represents the difference between the observed reward at time $t$ and the expected reward, based on the current estimate of the value function. In TD learning, we use this error signal to update the value function estimate:

$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t \tag{2.21}$$

where $\alpha$ is the learning rate.

However, the TD error signal has high variance, especially when the policy changes frequently or the reward signal is noisy. To address this, we can use Monte Carlo estimates of the return to provide a more accurate estimate of the value function. Specifically, we can define the return $G_t$ as:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k \tag{2.22}$$

where $T$ is the time horizon, and $\gamma$ is the discount factor.

The Monte Carlo estimate of the value function is then:

$$V_{MC}(s_t) = E[G_t|s_t] \tag{2.23}$$

This estimate is unbiased, but has high variance, as it requires a large number of samples to obtain accurate estimates.

The GAE method combines both TD and Monte Carlo estimates to obtain a more accurate and stable estimate of the advantage function. Specifically, the GAE estimate at time step $t$ is given by:

$$GAE(t) = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \tag{2.24}$$

where $\lambda$ is the lambda parameter, which controls the balance between bias and variance in the estimate. The lambda parameter can take values between 0 and 1, where $\lambda = 0$ corresponds to a purely TD estimate, $\lambda = 1$ corresponds to a purely Monte Carlo estimate, and $0 < \lambda < 1$ corresponds to a combination of both TD and Monte Carlo estimates.

The GAE estimate is a weighted average of the TD error signals at different time horizons, where the weight of each TD error signal is determined by the lambda parameter. When $\lambda$ is close to 0, the GAE estimate is dominated by the short-term TD estimates, which have lower variance but higher bias. When $\lambda$ is close to 1, the GAE estimate is dominated by the long-term Monte Carlo estimates, which have higher variance but lower bias. When $\lambda$ is between 0 and 1, the GAE estimate combines both short-term and long-term estimates, resulting in a more accurate and stable estimate of the advantage function.

The GAE estimate can be used to update the policy parameters using a gradient ascent algorithm. The policy gradient is defined as:

$$\nabla_\theta J(\theta) = \sum_t \nabla_\theta \log \pi(a_t|s_t) A(s_t, a_t), \tag{2.25}$$

where $J(\theta)$ is the objective function to be maximized, and $\theta$ represents the policy parameters. The policy gradient is estimated using the GAE estimate of the advantage function, resulting in a more accurate and stable estimate of the gradient [1].

#### 2.2.13.4 Value-Based Methods

Value-based methods are a class of reinforcement learning algorithms that learn to estimate the value function of state-action pairs in order to find an optimal policy that maximizes the expected cumulative reward. The value function represents the expected cumulative reward that an agent can achieve by taking a particular action in a given state and then following a specific policy [4].

### A) Q-Learning

Q-learning is an off-policy method that learns the optimal Q-value function by selecting actions that maximize the expected cumulative reward, regardless of the current policy. It updates the Q-value function by using the Bellman equation and the maximum Q-value over all possible actions in the next state:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{2.26}$$

where $Q(s, a)$ is the Q-value function of state-action pair $(s, a)$, $\alpha$ is the learning rate, $r$ is the immediate reward, $s'$ is the next state, and $a'$ is the action taken in the next state [15].

Popular Q-learning methods include:

- **Double Q-Learning:** Double Q-learning is an extension of Q-learning that addresses the problem of overestimation of Q-values. In Q-learning, the maximum Q-value in the next state is used to update the Q-value of the current state. This can lead to overestimation of Q-values, especially in environments with high variance in the rewards. Double Q-learning addresses this issue by using two Q-functions instead of one and alternating between them during updates [6].
- **Deep Q-Networks (DQN):** DQN is a variant of Q-learning that uses deep neural networks to estimate the Q-function. This allows DQN to handle high-dimensional state spaces, such as images, and can learn directly from raw sensory input. DQN uses experience replay to store and sample transitions from a replay buffer, allowing it to learn from past experiences and improve sample efficiency [6].

**Limitations of Value-Based Methods:**

- **Unstable:** One of the major limitations of value-based methods, especially deep reinforcement learning algorithms, is their instability. The problem arises due to the inherent nature of these algorithms, which involves updating the estimates of state or action values based on noisy samples. This noise can lead to oscillations in the learned policy and cause the algorithm to diverge. Additionally, the use of large neural networks in deep reinforcement learning can lead to issues with overfitting, making the learned policy too specific to the training environment.

- **Off-policy method:** Another limitation of value-based methods is that they are off-policy methods, meaning that they require the use of a separate behavior policy to generate the training data. This can be a disadvantage because it may not always be feasible to use the same behavior policy as the target policy, leading to poor convergence of the learned policy.

- **Value bootstrapping:** Value-based methods use value bootstrapping, which means that the value estimate for a state or action is updated based on the estimates of the next state or action. While this can lead to faster convergence and more efficient learning, it can also result in overestimation or underestimation of values if the bootstrapped estimates are inaccurate.

- **Limited exploration:** Value-based methods can suffer from limited exploration of the state-action space, especially in large state-action spaces. This can lead to sub-optimal policies that do not explore all possible actions and states, resulting in poor performance in complex environments.

- **Reward engineering:** Another limitation of value-based methods is that they rely heavily on the reward function, which can be challenging to design for complex environments. Reward engineering involves designing a reward function that provides a meaningful signal to the agent, but it can be difficult to strike a balance between encouraging exploration and rewarding the agent for achieving the desired task.

- **No natural handling of continuous action spaces:** Many value-based methods, such as Q-learning and SARSA, are designed for discrete action spaces and cannot be easily extended to continuous action spaces. While there are extensions, such as deep deterministic policy gradients (DDPG), they can be complex to implement and may require significant computational resources.

- **Difficulty in handling multi-agent environments:** Value-based methods can also struggle to handle multi-agent environments, where the behavior of one agent affects the rewards of other agents. This can result in non-stationary environments, where the optimal policy changes over time, leading to challenges in convergence [6].

#### 2.2.13.2 Policy-Based Methods

Policy-based methods are a class of machine learning algorithms that directly learn an optimal policy to maximize the cumulative reward. In contrast, value-based methods learn an optimal value function to estimate the expected reward from each state. Policy-based methods have several advantages over value-based methods, but they also have some limitations that need to be considered.

The key distinctions for policy-based methods over value-based methods include:

- Policy-based methods can handle continuous and high-dimensional action spaces, making them well-suited for real-world problems where the action space is continuous and high-dimensional.
- Policy-based methods can learn stochastic policies, which output a probability distribution over actions instead of a single action. Stochastic policies are more robust to changes in the environment and can handle exploration-exploitation trade-offs more effectively.
- Policy-based methods can learn optimal policies that are optimal in expectation, meaning that they can perform well even in non-stationary environments.
- Policy-based methods are less prone to the overestimation of Q-values that can occur with value-based methods, which can lead to suboptimal policies.

**Limitations of Policy-Based Methods:**

- Policy-based methods can be more sample-inefficient than value-based methods, meaning that they require more data to learn an optimal policy.
- Policy-based methods can be sensitive to the choice of hyperparameters and can be more challenging to tune than value-based methods.
- Policy-based methods can suffer from local optima, meaning that they can converge to a suboptimal policy that is not the globally optimal one.
- Policy-based methods can be more computationally expensive than value-based methods, especially for large and complex environments [1].

### A) Policy Gradient

A approach known as "policy gradient" allows agents to directly learn their policies, i.e., $\pi(a, s) = \Pr(a_t = a|s_t = s)$, from their experiences, i.e., $(s_t, a_t, s_{t+1}, r_t)$, which are acquired from the environment.The method attempts to estimate the true gradient of the anticipated return $\nabla_\theta E$ by gathering a sample of trajectories and comparing the mean gradient among these sampled trajectories to the actual gradient. A greater performance results by updating the parameters in the projected direction of the anticipated return gradient, which causes the expected return to grow roughly after every step. The following equation is used by the default policy gradient to update its parameters:

$$\widehat{g}t = \widehat{E}t \left[ \nabla\theta \log \pi\theta(a_t|s_t)\widehat{A}_t \right]. \tag{2.27}$$

In this algorithm, a stochastic policy $\pi_\theta$ and an estimator $\widehat{A}_t$ for the advantage function at timestep $t$ are utilized. The empirical average across a finite batch of samples is denoted by $\widehat{E}_t[\dots]$ in a sampling-and-optimization-alternating approach. Objective functions that utilize automated differentiation software generate a policy gradient estimator as the gradient; this estimator $\widehat{g}$ is produced by differentiating the objective as follow:

$$L_{PG}(\theta) = \hat{E}t \left[ \log \pi\theta(a_t|s_t)\hat{A}_t \right] \tag{2.28}$$

Multiple optimization steps for the same trajectory on this loss frequently result in incredibly large policy updates. Consequently, it is not advised to do so.

We may infer from the equation above that we select a few sample trajectories and compute their returns. Then, for a trajectory, the gradients of log probabilities at each time step are added and multiplied by the trajectory's return. The real gradient of the expected return, which is utilized to update the policy, is approximately approximated by averaging this number for various samples of our strategy.

The formula unambiguously demonstrates that a positive return for a trajectory increases the likelihood that these activities will be taken, but a negative return for a trajectory moves in the other direction, reducing the probability of taking the trajectory and so lowering its likelihood.

### B) Actor Critic Methods

Actor-Critic methods are a class of reinforcement learning (RL) algorithms that combine the benefits of both policy-based and value-based methods. The Actor-Critic approach can learn both the optimal policy and the value function of an environment simultaneously.

In Actor-Critic methods, the agent maintains two neural networks: an Actor and a Critic. The Actor network is responsible for determining the policy (i.e., the action to take) in a given state, whereas the Critic network evaluates the value of the state. The Actor network is trained using policy gradients, while the Critic network is trained using TD (Temporal Difference) learning.

The Actor-Critic approach can be further classified into two subcategories: on-policy and off-policy methods.

**On-policy Actor-Critic Methods/A2C:**

In on-policy Actor-Critic methods, the policy is updated after every action, which means that the policy is learned based on the current policy. This results in a slow learning process, but it has the advantage of being stable and converging to a near-optimal solution.

The most used variation of the on-policy actor-critic methods is the advantage actor-critic method (A2C).

The Advantage Actor-Critic (A2C) is an on-policy Actor-Critic algorithm that combines the advantages of policy gradient methods and value-based methods. A2C learns both the policy and the value function simultaneously by using two neural networks: an Actor network and a Critic network.

The Actor network takes the state as input and outputs the probability distribution of actions to be taken in that state. The Critic network takes the state as input and outputs the value of the state. The value function estimates the expected reward that an agent can get by being in a particular state and following the current policy.

The A2C algorithm updates the policy and value function using the advantage function, which measures the difference between the actual value of a state and the expected value based on the current policy. The advantage function is defined as:

$$Advantage = Q(s, a) - V(s) \tag{2.29}$$

where Q(s, a) is the expected reward of taking action a in state s and following the current policy, and V(s) is the expected reward of being in state s and following the current policy.

The A2C algorithm performs the following steps:

1. **Collect trajectories:** The agent interacts with the environment by taking actions based on the current policy and collects a set of trajectories, which are sequences of state, action, and reward tuples.

2. **Compute advantages:** The Critic network is used to estimate the value function for each state in the trajectory. Then, the advantage function is computed for each state-action pair in the trajectory.

3. **Compute policy gradients:** The policy gradients are computed using the advantage function. The gradients are then used to update the Actor network.

4. **Update value function:** The value function is updated using TD learning. The Critic network is trained to minimize the mean squared error between the estimated value and the actual value of the state.

The given steps are repeated for as long as the iterations are required and the desired goal is reached. A2C stands out as computationally efficient because it updates the policy and value function using the same set of trajectories. It is also stable because the value function provides a baseline for estimating the advantages[1].

## C) Trust Region Methods:

In TRPO, an objective function (the "surrogate" objective) is maximized subject to a constraint on the size of the policy update. Specifically,

$$\text{maximize}_{\ \theta}\ \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] \tag{2.30}$$

$$\text{subject to}\ \ \hat{E}_t\left[\text{KL}\left[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right]\right] \leq \delta. \tag{2.31}$$

Here, $\theta_{\text{old}}$ is the vector of policy parameters before the update. After approximating the objective and constraint using linear and quadratic approximations, this issue can be roughly solved effectively using the conjugate gradient algorithm.

The theory supporting TRPO actually recommends solving the unconstrained optimization problem by employing a penalty rather than a constraint.

$$\underset{\theta}{\text{maximize}}\hat{E}_t\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)}\hat{A}_t - \beta\text{KL}\left[\pi_{\theta_{\text{old}}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right]\right] \tag{2.32}$$

for some coefficient $\beta$. This follows from the fact that a certain surrogate objective (which computes the max KL over states instead of the mean) forms a lower bound (i.e., a pessimistic bound) on the performance of the policy $\pi$. TRPO employs a hard constraint as opposed to a penalty since it is challenging to select a single value of *beta* that performs well over several problems, or even within a single issue when the features vary throughout the course of learning. Experiments demonstrate that selecting a fixed penalty coefficient *beta* and optimizing the penalized objective equation using SGD alone are not sufficient to meet our aim of a first-order method that mimics the linear improvement of TRPO; more adjustments are needed. [1].

### Limitations with TRPO

1. **Computationally expensive:**

   TRPO requires computing the Fisher information matrix, which has a high computational cost, especially for high-dimensional policies. The Fisher information matrix is used to estimate the curvature of the policy objective function around the current policy, which is then used to constrain the magnitude of the policy update. This can make TRPO infeasible for large-scale problems where the number of parameters is large. The Fisher information matrix is defined as the negative expected value of the Hessian of the log-likelihood function of the policy. Computing the Hessian requires evaluating the second derivatives of the log-likelihood function with respect to each parameter of the policy. This can be computationally expensive, especially for high-dimensional policies, where the number of parameters can be in the millions or billions.

2. **Hyperparameter sensitivity:**

   TRPO has several hyperparameters that require careful tuning to achieve good results. These include the step size, the trust region size, and the penalty coefficient. The optimal values for these hyperparameters can be difficult to find and are highly dependent on the specific problem at hand. The step size determines the magnitude of the policy update, and a too large step size can result in policy divergence, while a too small step size can result in slow convergence. The trust region size determines the maximum allowable policy update, and a too small trust region can result in slow convergence, while a too large trust region can result in policy divergence. The penalty coefficient determines the trade-off between the policy improvement and the trust region constraint, and a too small penalty coefficient can result in policy divergence, while a too large penalty coefficient can result in overly conservative updates.

   Finding the optimal values for these hyperparameters can be difficult and time-consuming. Moreover, the optimal values may change during the training process as the policy changes.

3. **Limited Exploration:**

TRPO may suffer from limited exploration, especially in environments with sparse rewards or complex dynamics. This is due to the conservative nature of the trust region policy update rule, which may not explore new actions or states as effectively as other algorithms. The trust region policy update rule constrains the policy update to be within a certain region around the current policy. This constraint can limit the ability of the policy to explore new actions or states that may be far away from the current policy. Moreover, the trust region constraint is based on the current policy, which may not be optimal, and therefore may not encourage exploration towards the optimal policy.

### D) Proximal Policy Optimization:

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that is commonly used to train agents to interact with environments in order to maximize rewards. It was first introduced by OpenAI in 2017 following the limitations regarding TRPO and has since become a popular choice for many researchers and practitioners in the field of deep reinforcement learning[15].

PPO is a variant of policy gradient methods, which involve optimizing a parameterized policy to maximize the expected cumulative reward of the agent over multiple episodes. Unlike other policy gradient methods, PPO constrains the changes to the policy during each update step to prevent it from diverging too far from the previous policy, thereby ensuring stability and preventing catastrophic forgetting.

#### Clipped Surrogate Objective Function

By limiting the policy update to prevent significant changes in the policy parameters, the clipped surrogate objective function strategy of PPO works to increase the learning process' stability and prevent the policy from diverging.

Let $r_t(\theta)$ denote the probability ratio,

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{2.33}$$

so $r(\theta_{\text{old}}) = 1$.

TRPO maximizes a "surrogate" objective.

$$L^{\text{CPI}}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{2.34}$$

The primary objective we recommend is as follows:

$$L^{CLIP}(\theta) = \hat{E}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{2.35}$$

Where $\epsilon$ is a hyperparameter, say, $\epsilon = 0.2$. The motivation for this objective is as follows. The first term inside the min is $L_{CPI}$. The second term, $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving $r_t$ outside of the interval $[1-\epsilon, 1+\epsilon]$.The ultimate goal is a lower bound for the unclipped objective since we pick the lowest possible value of the clipped and unclipped objectives. With this method, we only take into account the probability ratio change when it would improve the objective and ignore it when it would worsen it.

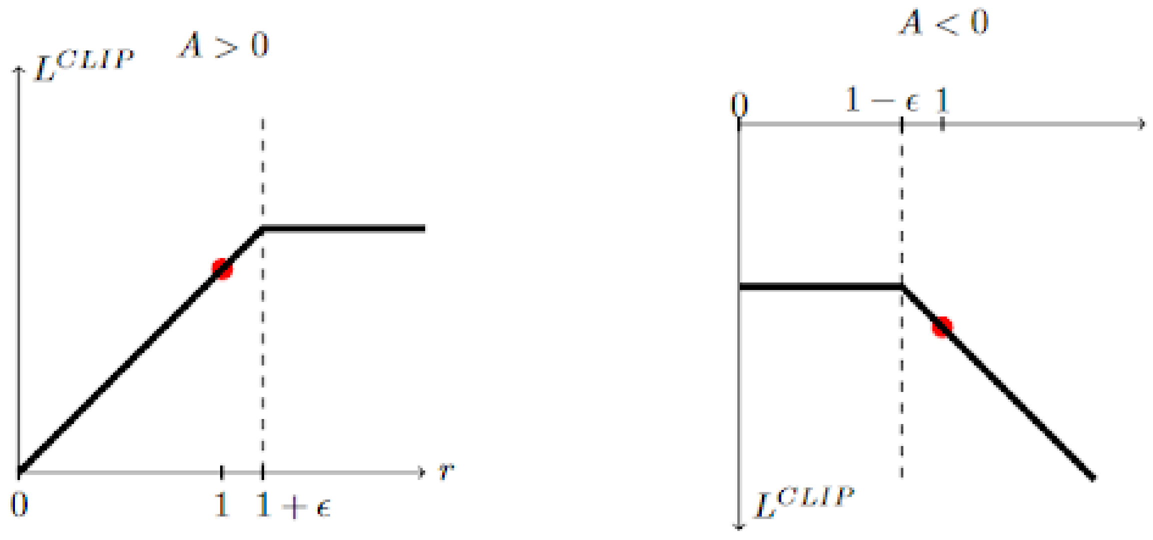

Figure 2.2: Single time-step of surrogate function [1]

### Overcoming Limitations of Value-Based Methods with PPO

1. **Exploration:** One of the major limitations of value-based methods such as DQN is that they tend to overestimate the value function, leading to a suboptimal policy. This occurs because the agent may not explore enough to find the optimal policy. PPO solves this issue by directly learning the policy, which encourages exploration. Additionally, PPO uses an objective function that constrains the policy update to a certain range, preventing the policy from changing too much at once and reducing the risk of catastrophic forgetting[1].

2. **Stability:**

Another limitation of value-based methods is the instability caused by the non-stationary target function in the Bellman equation. This can lead to divergence or oscillations in the learning process. PPO solves this issue by using a surrogate objective function that constrains the policy update to be close to the old policy, thereby ensuring a smooth learning process.

3. **Sample Efficiency:**

   Value-based methods such as DQN require large amounts of data to learn the value function accurately. This is because they learn from complete episodes of experience, which can be time-consuming and computationally expensive. PPO solves this issue by using a mini-batch of experiences to update the policy, which reduces the amount of data needed and makes the learning process more efficient. Additionally, PPO uses a value function that is trained on the same data as the policy, which reduces the variance of the value estimates and increases sample efficiency.

4. **Continuous Action Spaces:** Value-based methods such as DQN are designed for discrete action spaces and are not well-suited for continuous action spaces. PPO solves this issue by using an actor-critic architecture, where the actor directly outputs the mean and variance of a Gaussian distribution, and the critic estimates the value function of the state-action pairs. This allows the agent to learn a stochastic policy that can handle continuous action spaces.

### Limitations of PPO

While PPO solves some of the limitations of value-based methods, it also has its own limitations. One of the main limitations of PPO is that it is computationally expensive, especially in large-scale problems. Additionally, PPO is still prone to local optima, and it may not always converge to the optimal policy.

### Adaptive KL Penalty Coefficient

The application of a penalty on KL divergence and adaptation of the penalty coefficient such that we attain some goal value of the KL divergence $d_{\text{targ}}$ with each policy update is another strategy that may be employed as an alternative to the clipped surrogate objective or in addition to it. The KL penalty fared worse than the clipped surrogate objective in testing, but we've put it here because it's a crucial baseline.[1].

In the simplest instantiation of this algorithm, we perform the following steps in each policy update:

1. Using several epochs of minibatch SGD, KL-penalized objective is optimized as

$$L_{\text{KLPEN}}(\theta) = \hat{E}t\left[\frac{\pi\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}t - \beta\text{KL}[\pi\theta_{\text{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \tag{2.36}$$

2. Compute :

$$d = \hat{E}t[\text{KL}[\pi\theta_{\text{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \tag{2.37}$$

   - If $d < d_{\text{targ}}/1.5$, $\beta \leftarrow \beta/2$
   - If $d > d_{\text{targ}} \times 1.5$, $\beta \leftarrow \beta \times 2$

For the upcoming policy update, the revised $\beta$ is utilized. Another hyperparameter, $\beta$'s starting value is unimportant in reality because the algorithm swiftly modifies it. [1].

### 2.2.14 Game Rendering Technique

#### Raycasting

Raycasting is a technique used in computer graphics to render a 3D scene into a 2D image. Raycasting is a fundamental technique used in many classic video games such as Wolfenstein 3D and Doom.

The process of raycasting involves casting a ray from the player's viewpoint into the 3D environment. The ray is then traced through the environment until it intersects with an object or surface. The point of intersection is then used to calculate the distance between the player and the object, as well as the angle at which the object is seen. This information is used to render the scene on the player's screen.

In the context of first-person shooter games, raycasting is used to determine what the player can see and what obstacles are in the way. This information is then used by the game engine to determine which objects should be rendered and which should be hidden, based on their distance and angle relative to the player's viewpoint. Additionally, raycasting can be used to detect collisions between the player and other objects, such as walls or enemies.

Raycasting can be implemented using different algorithms, such as the Digital Differential Analyzer (DDA) algorithm or the Bresenham's line algorithm. In general, raycasting is a computationally efficient technique, making it a popular choice for real-time rendering in video games.

## Limitations of Raycasting

Although ray-casting is a fast and efficient technique for creating 3D environments, it does have some limitations. One major limitation is that it relies on certain geometric constraints, such as walls being at a 90-degree angle with the floor. As a result, the viewpoint in a ray-casting game cannot be rotated along the Z-axis. This means that walls cannot be slanted, and the benefit of drawing in vertical slices is lost. Because of this limitation, a ray-casting environment is not considered a true 3D environment. Players can only move forward, backward, and turn left or right, but cannot rotate or swing around the Z-axis

# 3. Methodology

## 3.1 Game Environment

### 3.1.1 Game Development

To develop our FPS game, the Pygame library was utilized. Pygame is a popular game development framework for Python that provides a range of features, including graphics and sound capabilities, event handling, and game physics. The library allowed to create a 2D game environment and implement various game mechanics, such as player movement, enemy AI, weapon mechanics, and health systems. We also utilized the library's sprite class to create and animate game objects and the Pygame mixer module to handle sound effects and background music. Overall, Pygame provided a robust and flexible toolset to create a functional and immersive game environment for our reinforcement learning agent.

### 3.1.2 Game Rendering

To render the game environment, the raycasting technique was utilized, which involves casting rays from the player's perspective and using their intersections with the game world to create a 2D projection of the 3D environment.

#### Implementation Steps

The following outlines the steps involved in raycasting for an FPS game:

1. **Setting up the Environment:** The first step is to set up the virtual 3D environment. This can be done by defining a 2D grid of cells, where each cell can either be a wall or empty space. In addition, we define the player's position and orientation [20].

   Following constraints are considered for the environment creation:

   - Walls are always at a 90° angle with the floor.
   - Walls are made of cubes that have the same size. (64*64*64 in our case)
   - The floor is always flat.

2. **Defining attributes for projections:** We define the projection attributes required to render the world. These attributes include the player's height, field of view (FOV), and position, as well as the projection plane's dimension and the relationship between the player and the projection plane as shown in the figure B.1. In the figure B.1, player is represented by yellow dot with a field of view of 60 degrees represented by green region. We set the FOV to 60 degrees for screen optimization, and the player's height to 32 units. The player's point of view is defined by their X and Y coordinates and the angle they are facing. We also define a projection plane with a resolution of 320x200 units, which corresponds to the VGA video card resolution [20]. When projected, the world look like the scene in Figure B.2.

   We now have obtained following attributes that will be used throughout our project:

   Table 3.1: Atrributes for projections

   | **Attributes** | **Values** |
   |---|---|
   | Dimension of the Projection Plane | 320 x 200 units |
   | Distance to the Projection Plane | 277 units |
   | Angle between subsequent rays | 60/320 degrees |
   | Center of the Projection Plane | (160,100) |

   From these values, the angle between subsequent rays and the distance between the player and projection plane can be calculated using trigonometry and the Pythagorean theorem. These values are important for later steps in the ray-casting process.

3. **Finding the walls:** In the third step of ray-casting, we need to find the walls of the world by tracing rays. The walls can be viewed as a collection of 320 vertical lines, which simplifies the process of ray-casting.

   The steps involved are as follow [21]:

   (a) Divide the projection plane into 320 vertical columns to represent the walls in the scene.

   (b) Starting from the left-most column (column 0), subtract half of the field of view (FOV) angle (30 degrees) from the player's viewing angle to get the direction of the first ray.

   (c) Cast the first ray from the player's position in the direction obtained in step 2.

(d) Trace the ray until it intersects with a wall, by checking the grid intersection points of the ray with the grid boundary (Points A, B, C, D ,E and F in Figure B.3.

(e) Record the distance to the wall, which is equal to the length of the ray.

(f) Add the angle increment of 60/320 degrees to the viewing angle to get the direction of the next ray.

(g) Repeat steps 3-6 for each subsequent column until all 320 rays have been cast.

The process to find the vertical and horizontal intersections are described in the next section.

**Finding the horizontal intersections**

Steps to find the horizontal intersections are as follow:

(a) Find the coordinate of the first intersection point A as shown in Figure B.3.

(b) For a ray facing up, $A.y = \lfloor Py/64 \rfloor * (64) - 1$.

(c) For a ray facing down, $A.y = \lfloor Py/64 \rfloor * (64) + 64$.

(d) Find $Ya$.

(e) For a ray facing up, $Ya = -64$.

(f) For a ray facing down, $Ya = 64$.

(g) Find $Xa = 64/\tan(\alpha)$.

(h) Extend the ray to the next intersection point, C.

(i) $C.x = A.x + Xa$ and $C.y = A.y + Ya$.

(j) Repeat steps 1-4 until a wall is encountered.

(k) Compare the distances to both intersection points and choose the closer one.

**Finding the vertical intersections**

Steps to find the vertical intersections are as follow:

(a) Find the coordinate of the first intersection point, B as shown in Figure B.3.

(b) For a ray facing right, $B.x = \lfloor Px/64 \rfloor * (64) + 64$.

(c) For a ray facing left, $B.x = \lfloor Px/64 \rfloor * (64) - 1$.

(d) Find $A.y = Py + (Px - A.x) * \tan(\alpha)$.

(e) Find $Xa = 64$.

(f) Find $Ya$ using the same equation as in the horizontal case.

(g) Extend the ray to the next intersection point.

(h) Repeat steps 1-5 until a wall is encountered.

(i) Compare the distances to both intersection points and choose the closer one.

4. **Finding distance to the walls:** The distance from the player's viewpoint to the wall slice is determined using trigonometric functions such as sine or cosine. These functions can be precomputed and put into tables for efficiency. However, a problem known as the "fishbowl effect" (as shown in Figure B.4) can occur when using these equations, causing viewing distortions. To correct for this, the resulting distance is multiplied by the cosine of the angle of the ray being cast relative to the viewing angle (BETA). BETA is calculated based on the player's field of view and the angle of the ray being cast.

**Fishbowl effect**

The "fishbowl effect" refers to a visual distortion that can occur in ray-casting implementations when trying to render 3D scenes. This effect arises from the fact that ray-casting combines polar and Cartesian coordinates, which can result in an inaccurate representation of distance when rendering walls that are not directly in front of the viewer.

Specifically, the distance calculated using trigonometric functions in polar coordinates can become distorted when mapped onto a 2D Cartesian plane. This is because the distance between two points on a 2D plane is not the same as the distance between those same two points in a polar coordinate system. As a result, the walls in the rendered scene may appear to bulge outwards or inwards, giving the impression of a "fishbowl" shape.

To correct for this distortion, the distance calculated using trigonometric functions in polar coordinates needs to be adjusted using a correction factor. This correction factor is based on the angle of the ray being cast relative to the viewer's line of sight (ALPHA), which is referred to as BETA [20]. The correction factor is simply the cosine of BETA, which ensures that the distance to the wall slice is accurately represented on the 2D Cartesian plane, and eliminates the "fishbowl effect" as shown in Figure B.5.

5. **Drawing the walls:** Once the distances to the walls are computed, the height of the projected wall slice needs to be found. This is done using a simple formula which takes into account the actual height of the wall slice, the distance to the projection plane and the distance to the wall slice as shown below [20]:

$$\text{Projected Slice Height} = \frac{\text{Actual Slice Height}}{\text{Distance to the Slice}} * \text{Distance to Projection Plane} \quad (3.1)$$

Our world consists of cubes, where the dimension of each cube is 64x64x64 units, so the wall height is 64 units. We also already know the distance of the player to the projection plane (which is 277). Thus, the equation can be simplified to:

$$\text{Projected Slice Height} = \frac{64}{\text{Distance to the Slice}} * 277 \quad (3.2)$$

Once this value is computed, a vertical line can be drawn on the corresponding column on the projection plane. This line will represent the projected wall slice.

6. **Texturing the walls:** To add texture to the walls in the ray-casting world, the technique of texture mapping is used. This technique involves adding a texture (bitmap) to the walls, making them more visually appealing. The bitmaps used in this technique have a size of 64 by 64 pixels, which is also the size of the cube facets used in the world.

   Texture mapping in the ray-casting world is simplified as it only requires scaling a slice (a column) of the bitmap to map it onto the walls. This means that for each column of the screen, the program can take the corresponding slice of the bitmap and scale it to match the height of the wall. This scaled slice is then used to texture the wall in that column, creating the illusion of a textured surface.

   By using texture mapping, the walls in the ray-casting world can be made to look more realistic and interesting as shown in Figure B.6, without requiring complex 3D models or rendering techniques. It is a simple yet effective way to enhance the visual appeal of the world and create a more immersive experience for the player.

7. **Motion of the player:**

   **Horizontal Motion**

   Horizontal motion in the context of a ray-casting game involves allowing the player to move forward, backward, and turn. To achieve this, the player's position is defined by a coordinate and a viewing angle, and two additional attributes are needed: the player's movement speed and turning speed.

#### A Moving Forward and Backward

Moving forward or backward involves finding the displacement based on the player's movement speed. This is done by calculating the X and Y displacement using trigonometry and adding or subtracting it from the player's current position. It is important to check for boundaries to ensure the player does not go outside the map or walk through walls.

#### B Turning

Turning is achieved by adding or subtracting an angle increment to the current viewing angle. The angle increment determines how fast the player turns, and larger increments can make the movement appear less smooth. The viewing angle should wrap around when it reaches a full circle.

## 3.2 Use of OpenAIGym:

The wolfenstein game was wrapped with OpenAIGym Environment to leverage the power of Gym Environments and its API. OpenAI is an organization that is dedicated to advancing artificial intelligence in a safe and beneficial way. One of their major achievements is the development of language models that can generate coherent and relevant text based on user inputs. In the context of gaming, OpenAI has also developed tools that allow game developers to wrap their existing games in environments that are compatible with reinforcement learning algorithms. This means that the game can be used to train AI agents to learn how to play the game autonomously, without the need for explicit programming.

The game mechanics and rules were defined in a way that could be understood by the AI algorithms, and the necessary data and resources were provided for the AI agents to interact with the game. By doing so, it was easier to train AI agents to play the FPS game, using reinforcement learning algorithms.

Overall, the use of OpenAI tools for wrapping our FPS game in a Wolfenstein environment represents a promising approach for advancing AI research in the gaming domain. This approach can help accelerate the development of intelligent agents that can learn to play games at a human-level or beyond, and could have potential applications in other areas such as robotics and automation.

## 3.3 Game Settings

### 3.3.1 Game Scenarios and Action Spaces

The project consists of three levels that progressively increase in difficulty. Each level presents unique challenges and objectives for the player to complete, allowing the reinforcement learning agent to learn and improve its decision-making skills. The following section provides a detailed description of the three levels in the game, including their objectives, challenges, and action space.

1. **Basic Level:** The Basic scenario is a rectangular map surrounded by walls on all sides. The player/agent spawns along one of the longer walls in the center and along the opposite wall, an enemy is spawned in a random position and moving either towards the left or right randomly. The enemy deals no damage to the player and a single hit to the enemy kills it. Episode ends when the enemy is killed.

   **Action Space:** Move left, Move right, Shoot

2. **Defend The Center:** Defend The Center is a square map enclosed by walls on all sides similar to basic. The player spawns at the center of the map. Four enemies are spawned, one at each corner of the room. The enemies cannot shoot the player, but move towards the player at constant speeds and deal damage to the player on collision. If an enemy is shot by or if it collides with the player, a new enemy replaces it in the same corner that it spawned from. The player has limited ammo to play with. Episode ends if the player's health decreases to zero.

   **Action Space:** Turn Left, Turn Right, Shoot

3. **Deadly Corridor:** Deadly Corridor is a long corridor like map with shooting enemies at both sides of the player in multiple locations. The enemies cannot move but can deal damage to the player by shooting. The player is spawned at one end of the corridor and has to reach a flag at the opposite end of the corridor. To reach the flag, it has to move through multiple shooting enemies and kill them. The player has limited ammo to play with. The episode ends on capturing the flag or if the player's health decreases to zero.

   **Action Space:** Move Forward, Move Backward, Turn Left, Turn Right, Shoot

### 3.3.2 Observation Spaces

The observation space on each timestep is the actual frame of the game window i.e raw pixel of the game and has a shape of (100,160,1). PPO is utilising CNNs instead of MLP policy. So, the input to the network is a grayscaled image of width 160 and height 100. The original window size was 320 x 200 which was rescaled to a size of 160 x 100.

### 3.3.3 Reward Structure

The following section outlines the reward structures for each level of our game, which are designed to incentivize the reinforcement learning agent to complete objectives and improve its performance over time.

1. **Basic Level:**

   For basic level, we used sparse rewards as follows:

   a) +100 for killing an enemy

   b) -5 for missing a shot, or wasting ammo

   c) -1 for each time-step

2. **Defend The Center:** For 'Defend The Center', we used only 2 rewards per time step, one for killing an enemy and another for dying. The agent was left to figure out the rest for itself.

   a) +1 for killing an enemy

   b) -1 for dying i.e, player's health $< 0$

3. **Deadly Corridor:**

   **Sparse Reward Problem:**

   The problem of sparse rewards was not visible in the first two levels of the game. In fact, sparse rewards worked pretty well for those two levels. We tried using similar sparse rewards for deadly corridor too, +1000 for getting to the flag and -1000 for dying and -500 for losing all ammo before killing all enemies. But this reward structure trained the agent to only move towards the flag and not shoot any of the enemies in its way. Thus, the agent got many hits and died pretty early if the damage taken was comparatively higher.

**Reward Shaping:**

The new reward structure introduced movement rewards for the agent,$+dx$ for every step taken towards the flag and $-dx$ for every step taken away from the flag. Along with the movement rewards, we introduced 3 more variables to the agent to help shape its reward structure and add some intermediate rewards for its actions: *damage-taken*, *hitcount* and *ammo-count*. *damage-taken* tells how much more damage the agent has taken in this step than the previous step, so a negative reward corresponding to *damage-taken* is given to the agent. Second is *hit-count*, which tells how much more enemies our agent hit in the current step as compared to the previous step. A positive reward to the agent corresponding to this value is given. The last one is the *ammo-count* which gives the difference between numbers of available ammo from the previous step to the current step. So, we give a negative reward to the agent corresponding to this value advising it not to waste bullets.

After shaping, the reward structure looked quite better than the previous sparse rewards and it showed in training as well. The agent started to consider killing its enemies, saving ammo and at the same time moving towards the goal flag.

**Curriculum Learning:**

Curriculum Learning (CL) is a training strategy that trains a machine learning model from easier data to harder data, which imitates the meaningful learning order in human curricula. There are a few generic types of curriculum learning available for reinforcement learning problems. Single-task curriculum, task specific curriculum and sequence curriculum are the prominent ones. An important question when designing curricula is determining the stopping criteria: that is, how to decide when to stop training on samples or tasks associated with a vertex, and move on to the next vertex. In practice, typically training is stopped when performance on the task or set of samples has converged. Training to convergence is not always necessary, so another option is to train on each vertex for a fixed number of episodes or epochs[22]

Our use of curriculum learning includes dividing the deadly corridor level into 3 sublevels: easy, medium and hard. Easy level is a small map with only 2 enemies, medium is a slightly larger map with 4 enemies and hard is the complete map with 6 enemies. The agent is first trained in the easy level to help it learn to kill enemies and move ahead in the map. Subsequent training in easy, medium and hard levels using curriculum learning makes it possible for the agent to play well in a hard environment like deadly corridor.

### 3.3.4 PPO Core Implementation Details

#### 3.3.4.1 PPO Core Algorithm:

A proximal policy optimization (PPO) algorithm that uses fixed-length trajectory segments is shown below. Each iteration, an actor collects rollout of data in N environment by stepping M steps. Then we construct the surrogate loss on these NM steps of data, and optimize it with minibatch SGD (or usually for better performance, Adam), for K epochs.

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

```
1: for $iteration = 1, 2, ...$ do
2:     for $environment = 1, 2, ..., N$ do
3:         Run policy $\pi_{\theta_{\text{old}}}$ for $M$ steps
4:         Compute advantage estimates $\hat{A}_1, ..., \hat{A}_M$
5:     end for
6:     Optimize surrogate $L$ with respect to $\theta$, with $K$ epochs and minibatch size $B \leq NM$
7:     $\theta_{\text{old}} \leftarrow \theta$
8: end for
```

where $L(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$ is the surrogate objective loss.

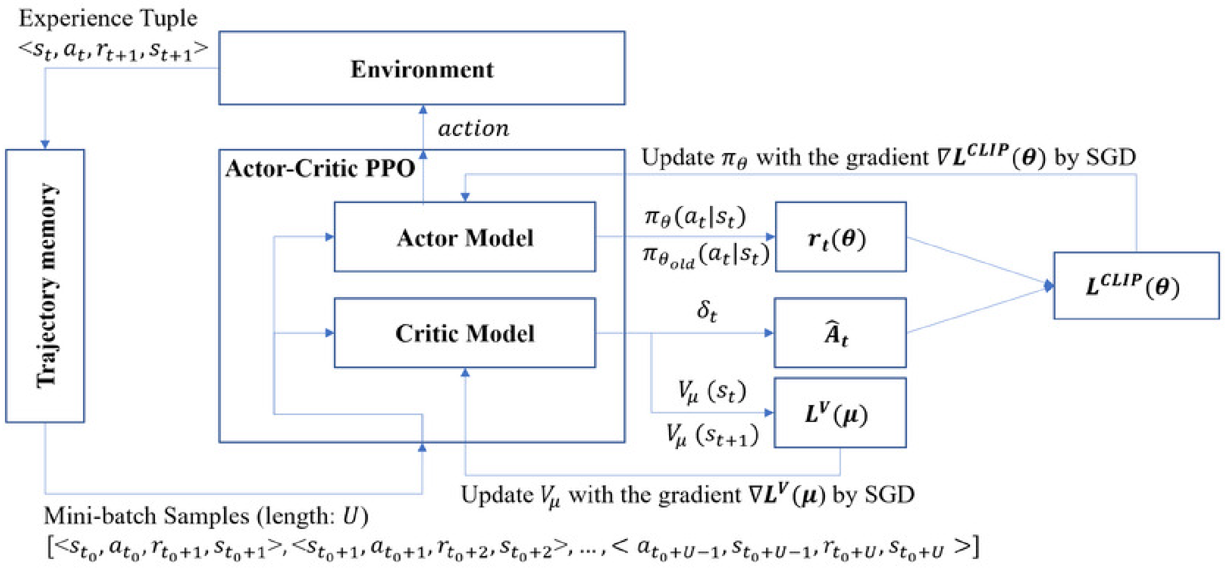
Experience Tuple
$<s_t, a_t, r_{t+1}, s_{t+1}>$
Environment
action
Trajectory memory
Actor-Critic PPO
Actor Model
Critic Model
Update $\pi_\theta$ with the gradient $\nabla L^{CLIP}(\theta)$ by SGD
$\pi_\theta(a_t|s_t)$
$\pi_{\theta_{old}}(a_t|s_t)$
$r_t(\theta)$
$L^{CLIP}(\theta)$
$\delta_t$
$\hat{A}_t$
$V_\mu(s_t)$
$V_\mu(s_{t+1})$
$L^V(\mu)$
Update $V_\mu$ with the gradient $\nabla L^V(\mu)$ by SGD
Mini-batch Samples (length: $U$)
$[<s_{t_0}, a_{t_0}, r_{t_0+1}, s_{t_0+1}>, <s_{t_0+1}, a_{t_0+1}, r_{t_0+2}, s_{t_0+2}>, \ldots, < a_{t_0+U-1}, s_{t_0+U-1}, r_{t_0+U}, s_{t_0+U} >]$

Figure 3.1: Actor-Critic PPO algorithm process, Source:[2]

#### 3.3.4.2 Vectorized Architecture:

PPO leverages an efficient paradigm known as the vectorized architecture that features a single learner that collects samples and learns from multiple environments.

Pseudocode of Vectorized Architecture

```
1:  $envs = VecEnv(num\_envs = N)$
2:  $agent = Agent()$
3:  $next\_obs = envs.reset()$
4:  $next\_done = [0, 0, ..., 0]$
5:  for $update$ in $range(1, total\_timesteps//(N * M))$ do
6:      $data = []$                                        ▷ ROLLOUT PHASE
7:      for $step$ in $range(0, M)$ do
8:          $obs = next\_obs$
9:          $done = next\_done$
10:         $action, other\_stuff = agent.get\_action(obs)$
11:         $next\_obs, reward, next\_done, info = envs.step(action)$
12:         $data.append([obs, action, reward, done, other\_stuff])$
13:     end for                                           ▷ LEARNING PHASE
14:     $agent.learn(data, next\_obs, next\_done)$
15: end for
```

In this architecture, PPO first initializes a vectorized environment `envs` that runs $N$ independent environments either sequentially or in parallel by leveraging multi-processes. `envs` presents a synchronous interface that always outputs a batch of $N$ observations from $N$ environments, and it takes a batch of $N$ actions to step the $N$ environments. When calling `next_obs = envs.reset()`, `next_obs` gets a batch of $N$ initial observations. PPO also initializes an environment done flag variable called `next_done` to an $N$-length array of zeros, where its $i$-th element `next_done[i]` has values of 0 or 1 corresponding to not done and done state of the $i$-th sub-environment respectively.

Then, the vectorized architecture loops two phases: the rollout phase and the learning phase:

**Rollout Phase:**

The agent samples actions for the $N$ environments and continues to step them for a fixed number of $M$ steps. During these $M$ steps, the agent appends relevant data in an empty list `data`. If the $i$-th sub-environment is done after stepping with the $i$-th action `action[i]`, `envs` would set the `next_done[i]` to 1, auto-reset the $i$-th environment, and set `next_obs[i]` to the initial observation of the new episode of the $i$-th environment.

**Learning Phase:**

The agent learns from the collected data in the rollout phase: `data` of length $N \cdot M$, `next_obs`, and `done`. Specifically, PPO can estimate value for the next observation `next_obs` and calculate the advantage `advantages` and the return `returns`, both of which also have length $N \cdot M$. PPO then learns from the prepared data `[data, advantages, returns]`, which is called "fixed-length trajectory segments" by [23].

#### 3.3.4.3 Orthogonal Initialization of Weights and Constant Initialization of Biases:

The weights of hidden layers use orthogonal initialization of weights with scaling `np.sqrt(2)`, and the biases are set to 0. However, the policy output layer weights are initialized with the scale of 0.01 and the value output layer weights are initialized with the scale of 1. [24] find orthogonal initialization to outperform the default Xavier initialization in terms of the highest episodic return achieved. Also, [25] finds centering the action distribution around 0 (i.e., initialize the policy output layer weights with 0.01") to be beneficial.

#### 3.3.4.4 Generalized Advantage Estimation:

Our PPO implementation use Generalized Advantage Estimation (GAE) for estimating advantages. Two important sub-details include:

a) **Value Bootstrap:** If a sub-environment is not terminated nor truncated, PPO estimates the value of the next state in this sub-environment as the value target.

b) **TD($\lambda$) Return Estimation:** PPO implements the return target as

```
returns = advantages + values
```

which corresponds to TD($\lambda$) for value estimation (where Monte Carlo estimation is a special case when $\lambda$=1).

#### 3.3.4.5 Mini-Batch Updates:

During the learning phase of the vectorized architecture, we shuffle the indices of the training data of size `N*M` and break it into mini-batches to compute the gradient and update the policy.

#### 3.3.4.6 Normalization of Advantages:

After calculating the advantages based on GAE, we normalize the advantages by subtracting their mean and dividing them by their standard deviation.

#### 3.3.4.7 Clipped Surrogate Objective:

$$L^{CPI}(\theta) = \hat{E}_t \left[ \left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \right)^{CPI} \hat{A}_t \right] = \hat{E}_t \left[ r_t(\theta)\hat{A}_t \right] \tag{3.3}$$

TRPO maximizes this "surrogate" objective.

$$L^{CLIP}(\theta) = \hat{E}_t \left[ \min \left( r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{3.4}$$

This is the main clipped surrogate objective proposed by [23].

#### 3.3.4.8 Shared Nature-CNN Network for Policy and Value Functions:

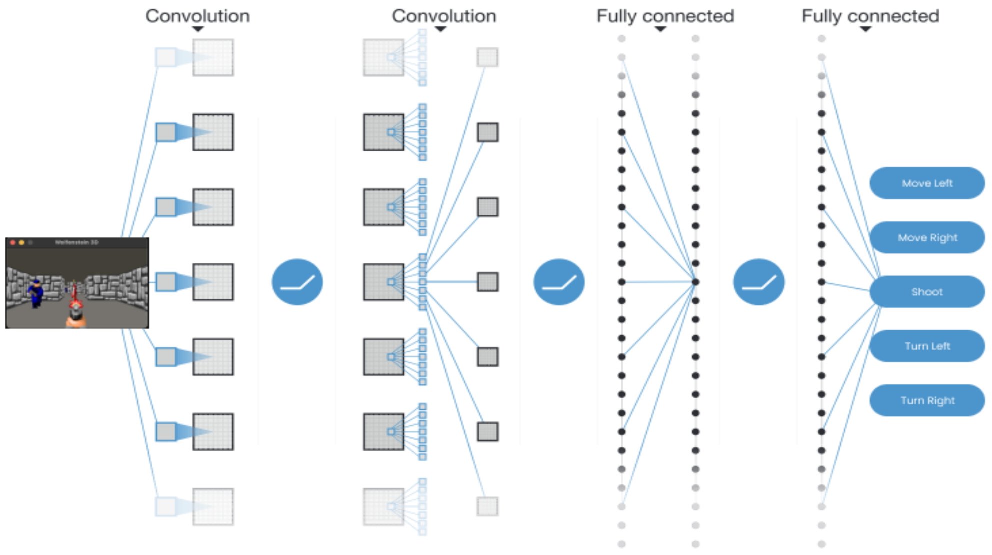

Figure 3.2: Shared Nature Actor Network

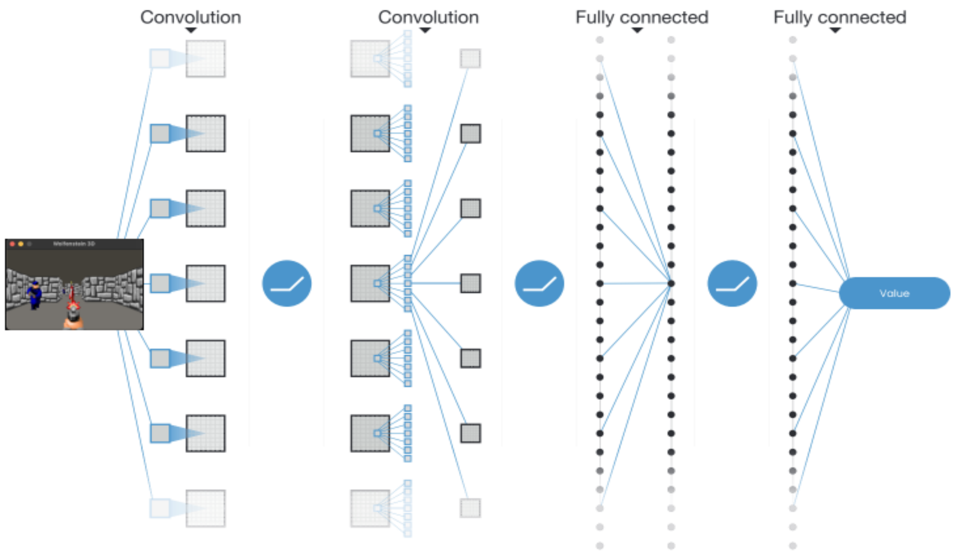

Figure 3.3: Shared Nature Critic Network

We implemented PPO so as to use the same Convolutional Neural Network (CNN) in [3] along with the layer initialization technique of weights and biases mentioned earlier to extract features, flatten the extracted features, and apply a linear layer to compute the hidden features. The policy and value functions share parameters by constructing a policy head and a value head using the hidden features.

Pseudocode of Shared Nature-CNN Network

```
hidden = Sequential(
    layer_init(Conv2d(n_input_channels, 32, kernel_size=8, stride=4, padding=0)),
    ReLU(),
    layer_init(Conv2d(32, 64, kernel_size=4, stride=2, padding=0)),
    ReLU(),
    layer_init(Conv2d(64, 64, kernel_size=3, stride=1, padding=0)),
    ReLU(),
    Flatten()
)

# Do a forward pass to get the shape
with torch.no_grad():
        n_flatten = hidden(
            torch.as_tensor(envs.single_observation_space.sample()[None]).float()
        ).shape[1]

# Policy output layer weights are initialized with the scale of 0.01
policy = Sequential(
    layer_init(Linear(n_flatten, 512), std=0.01),
    ReLU(),
    layer_init(Linear(512, envs.single_action_space.n), std=0.01)
)

# Value output layer weights are initialized with the scale of 1.0
value = Sequential(
    layer_init(Linear(n_flatten, 512), std=0.01),
    ReLU(),
    layer_init(Linear(512, 1), std=1.0)
)
```

#### 3.3.4.9 Overall Loss and Entropy Bonus:

The overall loss is calculated as:

$$loss = policyLoss + entropyCoeffient * entropy + valueCoeffecient * value$$

[3] has reported this entropy bonus to improve exploration by encouraging the action probability distribution to be slightly more random.

#### 3.3.4.10 Scaling the Images to Range [0, 1]

The input data has the range of [0, 255], but it is divided by 255 to be in the range of [0, 1].

#### 3.3.4.11 Global Gradient Clipping:

For each update iteration in an epoch, the gradients of the policy and value network was rescaled so that the "global l2 norm" (i.e., the norm of the concatenated gradients of all parameters) does not exceed 0.5. [25] find global gradient clipping to offer a small performance boost.

#### 3.3.4.12 Debug Variables:

The PPO implementation comes with several debug variables, which are:

1. **Policy Loss:** This is the mean policy loss across all data points. It correlates to how much our policy is changing. Ideally, policy loss should decrease with training time.

2. **Value Loss:** This is the mean value loss across all data points. It correlates to how well the model is able to predict the value in each state. Ideally, it should increase as the agent is learning and then decrease once the reward stabilizes.

3. **Entropy Loss:** This is the mean entropy value across all data points. A high entropy loss value indicates that the policy is taking a wide range of actions with relatively equal probability, while a low entropy loss value indicates that the policy is taking actions with high certainty..

4. **Clip Fraction:** This is the fraction of the training data that triggered the clipped objective.

5. **ApproxKL:** This is the approximate Kullback–Leibler divergence, measured by,

```
(-logratio).mean()
```

which corresponds to the KL estimator. An alternative estimator,

```
((ratio - 1) - logratio).mean()
```

which is unbiased and has less variance can also be used. Spikes in the Approx KL plot suggests that the policy is changing rapidly and one might want to increase the clip range.

#### 3.3.4.13 Evaluation Metrics:

**Mean Episode Length:**

Mean episode length is a basic but important metric to evaluate how our agent is performing. It tells whether the behavior of our agent is as expected or opposite. For example, in basic environment the mean episode length is expected to decrease with training time as it indicates the agent is taking less and less time to kill the enemy. Contrary to that, in defend the center environment, the mean episode length is supposed to increase with training. It tells that the agent is doing well to survive and killing the enemies before they reach the center. In deadly corridor, a very small episode length might mean that the agent is dying early. But a gradually decreasing episode length in the training curve indicates that the agent is taking less and less time to reach its goal, which is the flag.

**Mean Episode Reward:**

Mean episode reward is another important metric to evaluate the agent's performance. If the per episode reward is increasing with time, it means that the agent is doing well to meet its objectives and is doing what it is supposed to do. The ideal trajectory of the reward curve depends on the specific problem and the objectives of the agent. Generally, the reward curve should increase over time as the agent learns to maximize its cumulative reward in the environment. The rate at which the reward curve increases can vary depending on the complexity of the environment and the learning algorithm being used. Ideally, the reward curve should converge to a high level of cumulative reward, indicating that the agent has learned an optimal policy for the task.

# 4. System Design

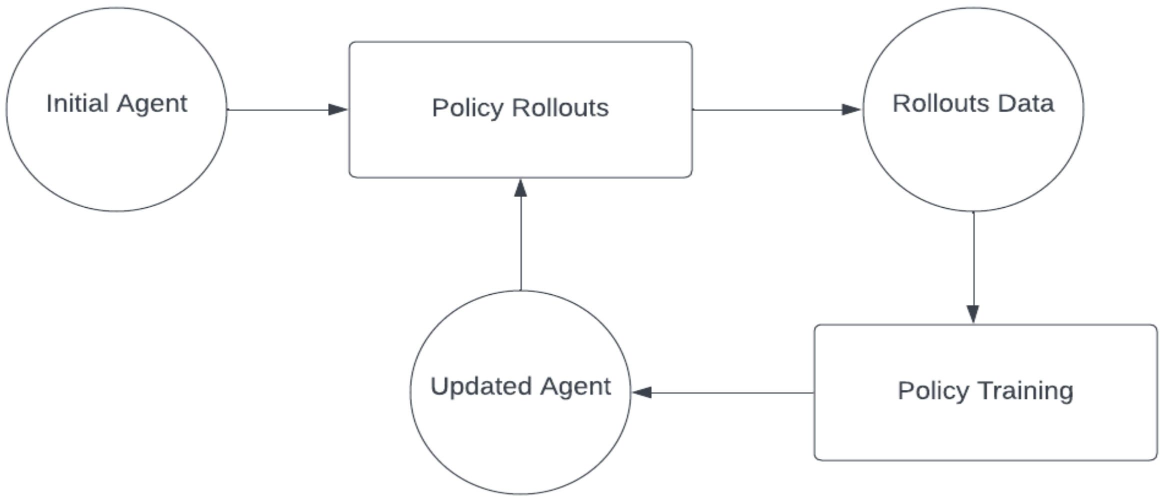

Figure 4.1: PPO Implementation Flowchart

On a high level, PPO takes the agent initalized to do policy rollouts i.e. to play the game. This generates rollouts data that can be used for policy training / optimizing the model. It then takes the updated agent to do policy rollouts and continue the training loop.

# 5. Results & Discussion

This section presents the findings and conclusions of our project for different levels of the game.

### 5.0.1 Basic Level

We trained the agent using PPO for about 110K timesteps which was enough to notice a satisfactory performance for the first level of the game. The episode length decreased continuously which is as intended for this level. On the other hand, the mean episode reward showed an increasing trend.

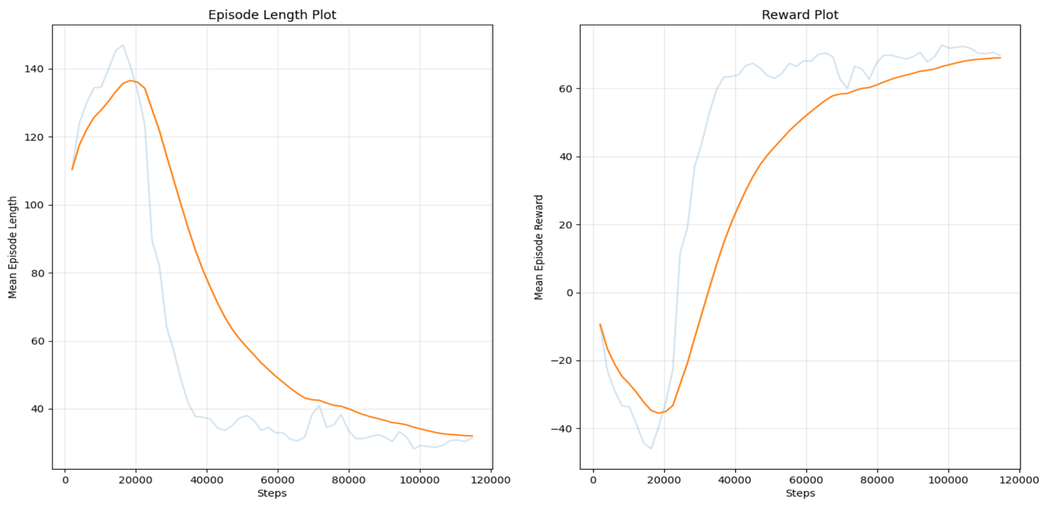

Figure 5.1: Mean Episode Length and Mean Episode Reward for Basic Level

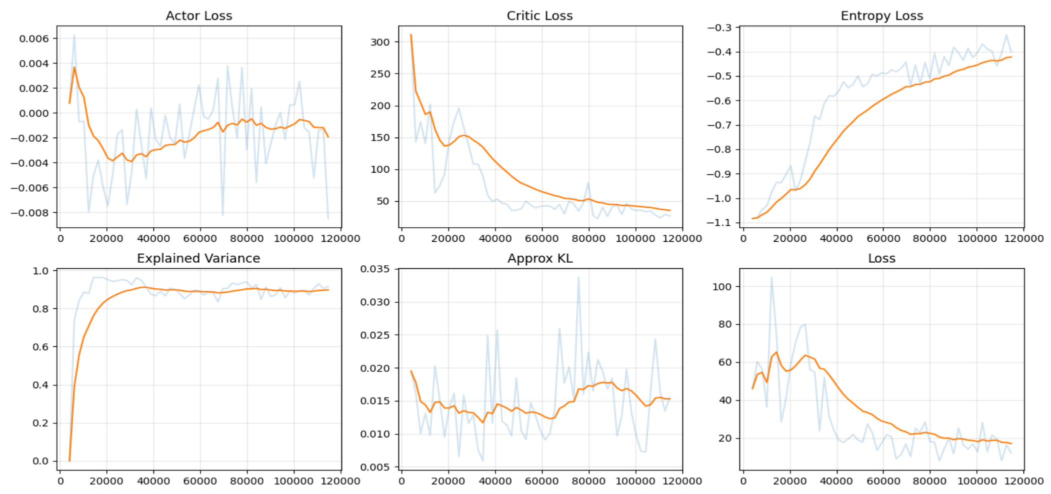

Figure 5.2: Losses in Basic Level

### 5.0.2 Defend The Center

The agent was trained for about 800K timesteps during which the agent showed good learning until about 700K timesteps and reached pretty near optimal results but started performing worse after that. Now might be a good time for early stopping. One reason for the agent to start performing worse might be that it started saving ammo causing it to shoot less often and hence, reduced its returns. Though trying out the model at 600K steps showed pretty good performance by the agent, the agent was wasting a minimum number of bullets and was hitting its targets with a good accuracy. During test with a total of 50 bullets given to the agent at start of the episode, it is consistently able to get 40 to 42 kills, wasting only 8 to 10 bullets per episode.

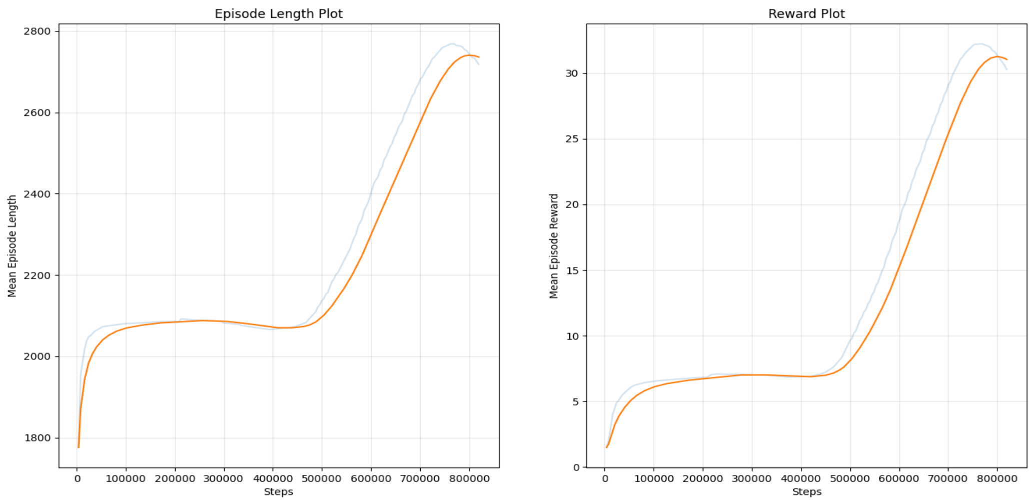

Figure 5.3: Mean Episode Length and Mean Episode Reward for Defend the Center

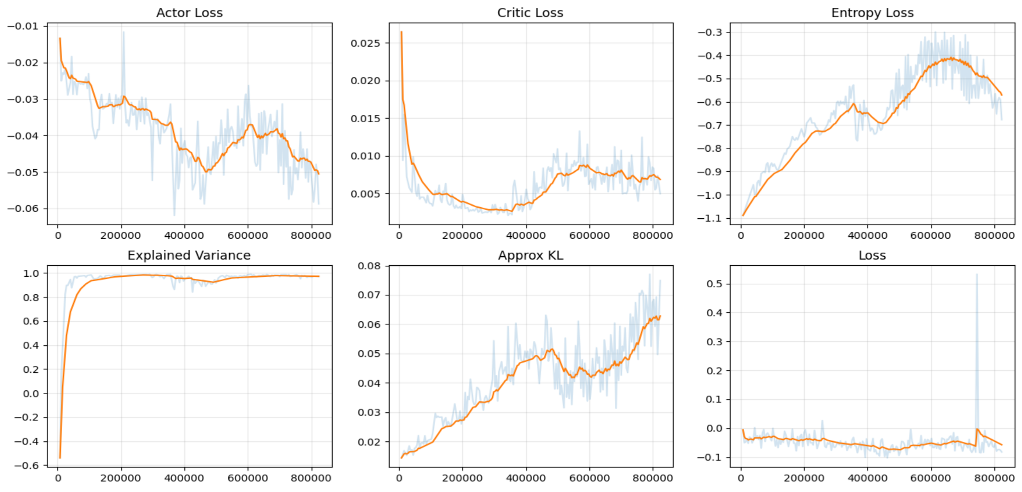

Figure 5.4: Losses in Defend the Center

### 5.0.3 Deadly Corridor

Training in the deadly corridor environment was particularly difficult due to the larger action space and a comparatively harder environment to navigate through. Firstly, training with sparse rewards showed no improvement in the agent's behavior whatsoever. The agent was not taking the intended actions and seemed confused. Though after a careful reward shaping and introduction of intermediate rewards, the agent's performance did not improve all that much. The agent was still not performing optimally. So to tackle this problem, we opted to make the agent learn via curriculum learning. The curriculum we tried primarily was simple and naive. By dividing the levels into 4 categories: EASY, MEDIUM, HARD and INSANE. The agent quickly figured out that it was able to complete the task inspite of taking damage. So, for those instances where it wasn't enough damage to kill it, it would simply run towards the goal. It had no plan and was lost as the levels got harder showing minimum to no progress. On introducing the second level in the curriculum, i.e, the medium level, with a larger damage from the enemies, the learning didn't go as expected as the agent was still not shooting as many enemies and died before reaching the flag. So, we had to modify the curriculum by taking a different approach. The sublevels were now easy, medium and hard. Easy level was a small map with only 2 enemies. We increased the enemy count by 2 and also the map size with each level. This allowed the agent to learn to kill 2 enemies that it encounters initially and then follows up by eliminating the rest of the enemies likewise thus progressively learning to move towards its goal, the flag, by eliminating the enemies.

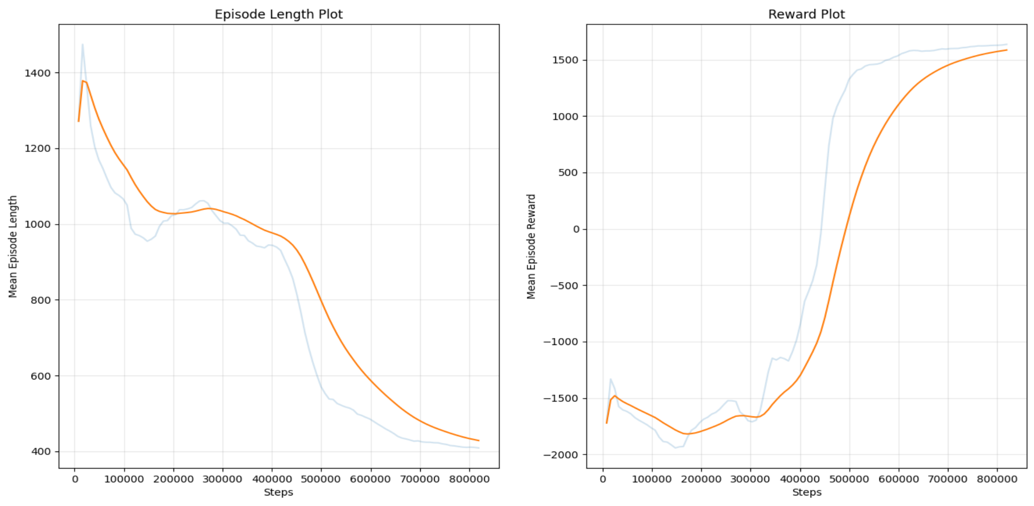

Figure 5.5: Mean Episode Length and Mean Episode Reward for Deadly Corridor

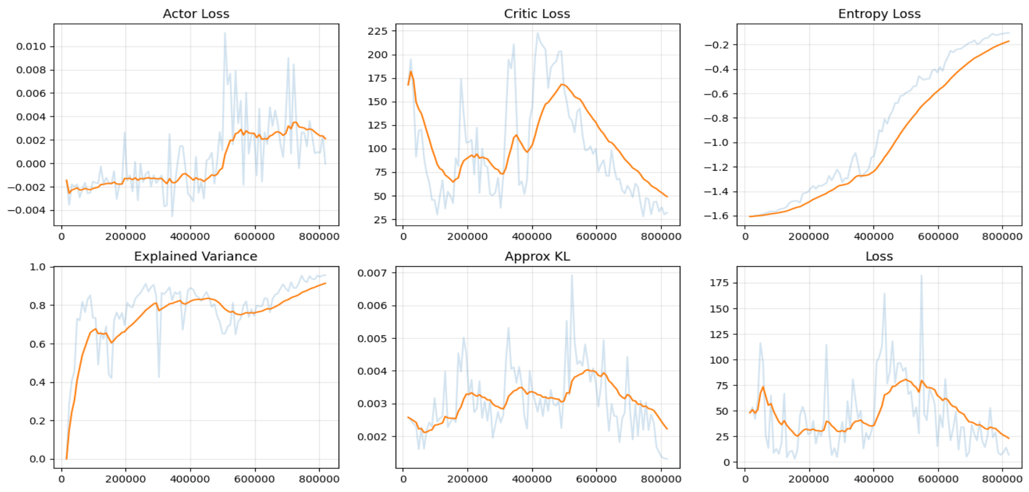

Figure 5.6: Losses in Deadly Corridor

After training the agent, we tested its performance in each level for 10 episodes. The testing showed pretty good results and the agent's performance was near optimal. It was successful in its intended tasks almost 100 % of the time.

# 6. Conclusions

In this paper, we presented a PPO-based reinforcement learning agent that was trained to perform in the Wolfenstein environment using raycasting for graphical rendering, and tested in three different game modes: basic, defend the center, and deadly corridor. Our agent achieved impressive results in terms of both its ability to navigate through the environment and its ability to accurately identify and target enemies.

Our agent was able to easily accomplish the objectives in the basic and defend the center game modes, demonstrating its versatility and ability to adapt to different tasks. However, the deadly corridor game mode proved to be more challenging, requiring our agent to navigate through narrow and highly dangerous environments.

To overcome these challenges, we utilized a convolutional neural network (CNN) alongside our PPO-based RL agent to improve its perception and decision-making capabilities. Our CNN-based approach enabled our agent to efficiently process the visual input from the environment and make more informed decisions, leading to improved performance in the deadly corridor game mode.

The performance of our agent was evaluated through various experiments, including the analysis of its learning curves, as well as its success rates in completing the game's objectives. Our results demonstrate that our approach is both effective and efficient, and it outperforms several existing methods that have been used to train RL agents for this environment.

One of the key advantages of our approach is that it utilizes raycasting for graphical rendering, which enables the agent to efficiently perceive its surroundings and accurately detect obstacles and enemies in real-time. Our agent was able to learn complex strategies and successfully navigate through the environment, demonstrating the effectiveness of our approach.

Overall, our approach combining PPO-based RL with CNN-based perception provides a promising framework for training agents to perform complex tasks in virtual environments. We believe that our results demonstrate the potential of RL-based approaches for solving real-world problems and provide a foundation for further research in this field.

# 7. Limitations and Future Enhancement

Training a game agent using reinforcement learning has an advantage of being able to learn without any supervision and flexibility in learning objectives. However RL is limited by several factors including its sensitivity to hyper-parameters and difficulty to interpret decision making technique of RL agents. Designing a good curriculum and shaping rewards properly is a difficult task itself. Self-playing AI agents can become too reliant on their learned strategies and may not be able to adapt to new situations or unexpected gameplay scenarios. This can make them predictable and less challenging to play against. There are also ethical concerns around the use of AI agents in games, particularly if they are designed to deceive or exploit human players.

Currently the agent in our game replicates a human player playing against one or many in-game NPCs(Non-Playable Characters). Future Enhancements might include using the trained model in the NPCs and letting a human play against the trained agent. For example, the trained model in basic level might be used in the enemy NPC and a human can control the playable character and try to stay alive against the trained enemy. These trained NPCs or AI bots if well trained might be a challenge for human players to play against.

# References

[1] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[2] Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo, and Youn-Hee Han. Federated reinforcement learning for training control policies on multiple iot devices. *Sensors*, 20:1359, 03 2020.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and Stig Petersen. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] Yusheng Li, Qing Zhu, Siqi Shao, Hui Zhao, and Jing Zhou. Reinforcement learning-based game ai: A survey. *IEEE Transactions on Games*, 13(2):144–175, 2021.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[8] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2018.

[9] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Michal Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learn-

ing. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[10] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[11] Wenpeng Yin, Plamen Angelov, and Xiaowei Jiang. Visual-based navigation and search in doom using deep reinforcement learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[12] Charles Beattie, Joel Z Leibo, Dmitry Teplyashin, Tom Ward, Daan Wierstra, and Demis Hassabis. Deep reinforcement learning with relational inductive biases. *arXiv preprint arXiv:1706.01399*, 2017.

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, and Shane Legg. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

[14] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Dmitriy Doronichev, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[18] Bowen Baker, Oleg Sushkov, Julian Michael, Michael Laskin, Kimin Lee, Richard Kurin, and Igor Mordatch. Openai five: A competitive multi-agent environment for reinforcement learning. *arXiv preprint arXiv:1905.06676*, 2019.

[19] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[20] Michael Abrash. *Zen of Graphics Programming.* Coriolis Group Books, Scottsdale, AZ, USA, 1995.

[21] Greg Anderson et al. *More Tricks of the Game Programming Gurus.* Sams Publishing, Indianapolis, IN, USA, 1995.

[22] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint arXiv:2103.01940*, 2021.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, Jul 2017.

[24] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Linnea Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020.

[25] Marcin Andrychowicz, Alexander Raichuk, Piotr Stańczyk, Marco Orsini, Semih Girgin, Rémi Marinier, Ludovic Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, and Sylvain Gelly. What matters in on-policy reinforcement learning? a large-scale empirical study. In *International Conference on Learning Representations*, 2021.

# Appendix A. Hyperparameters

Table A.1: Hyperparameters for different environments

| **Hyperparameters** | **Basic Level** | **Defend The Center Level** | **Deadly Corridor Level** |
|---|---|---|---|
| Adam Learning Rate | 1e-4 | 1e-4 | 1e-5 |
| N Steps (M) | 2048 | 4096 | 8192 |
| Batch Size | 64 | 64 | 64 |
| N Epochs | 10 | 10 | 10 |
| Discount Factor | 0.99 | 0.99 | 0.95 |
| GAE Lambda | 0.95 | 0.95 | 0.9 |
| Clip Range | 0.2 | 0.2 | 0.1 |
| Entropy Coefficient | 0.05 | 0.05 | 0.05 |
| Value Coefficient | 0.5 | 0.5 | 0.5 |

# Appendix B. Game Environments

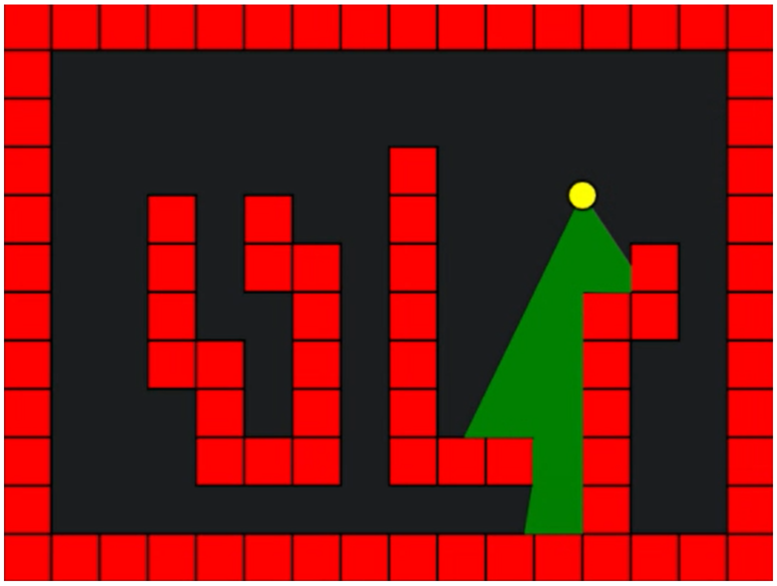

Figure B.1: Initial 2D representation of the world

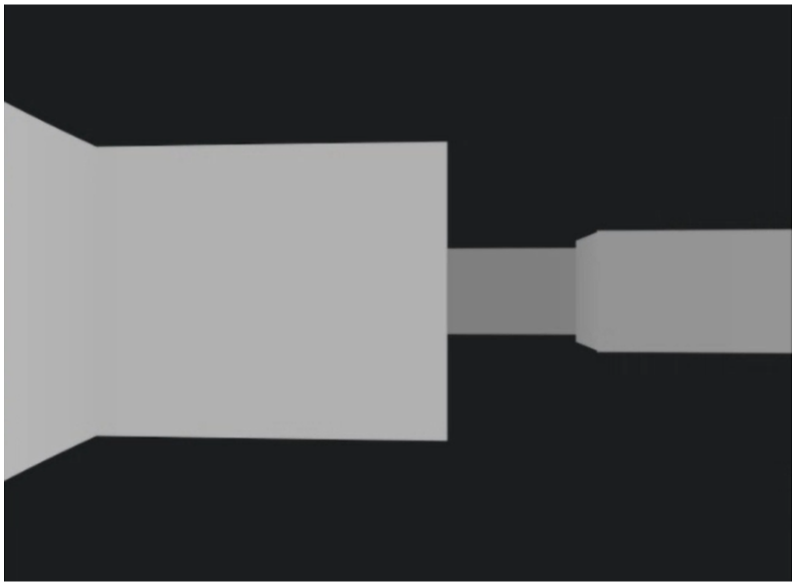

Figure B.2: Pseudo 3D world after raycasting

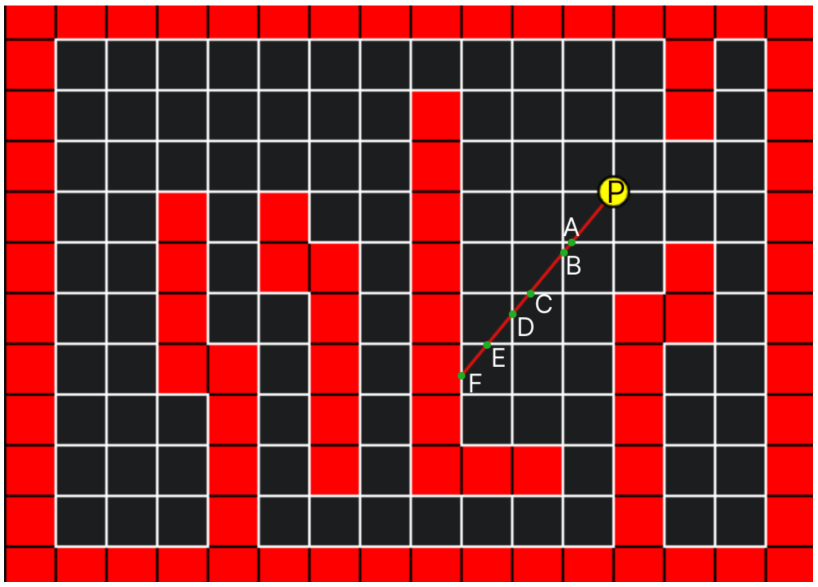

Figure B.3: Game world with horizontal and vertical intersection with the grid

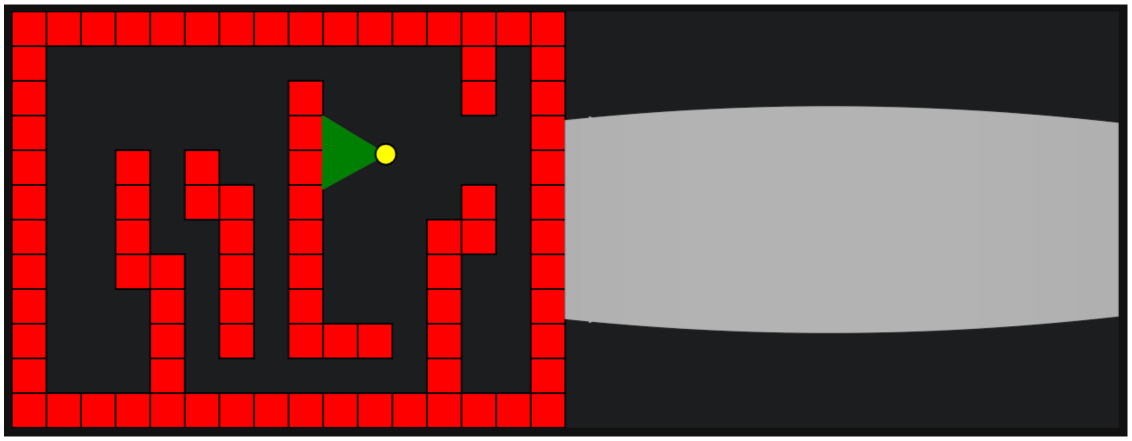

Figure B.4: Fishbowl effect

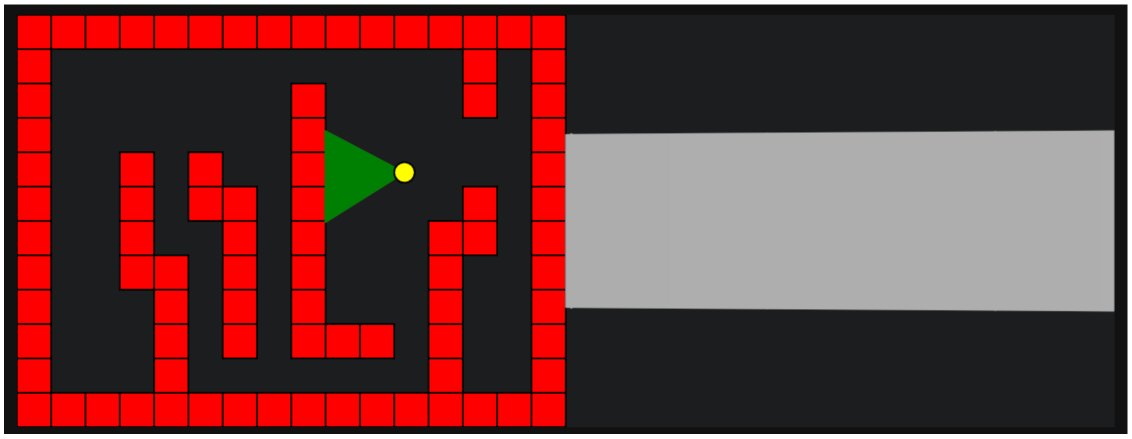

Figure B.5: After correcting the fishbowl effect

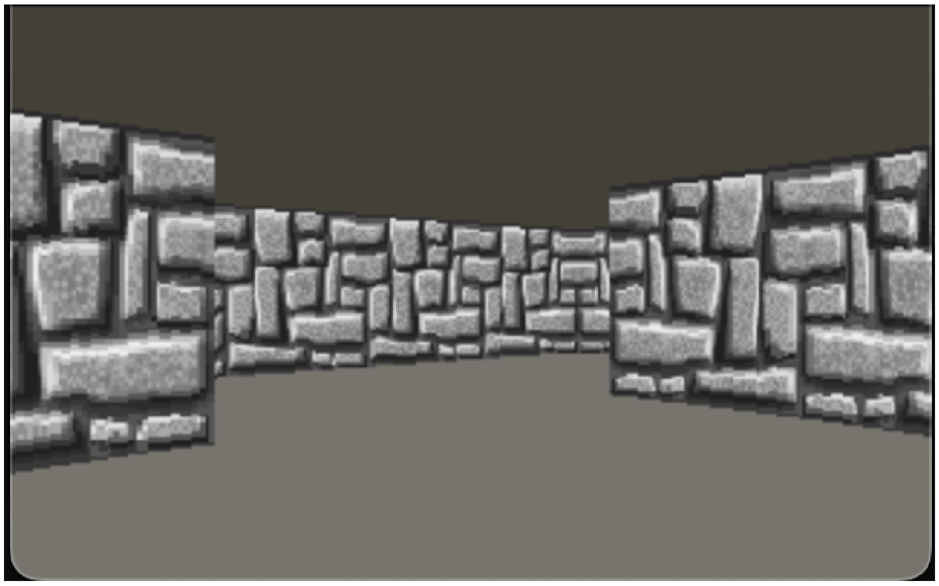

Figure B.6: Textured Walls

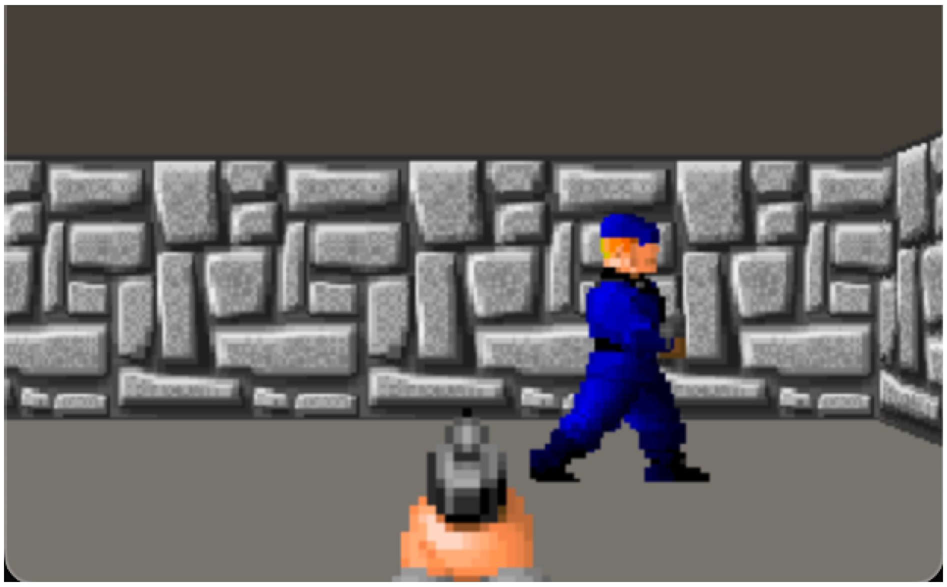

Figure B.7: Basic Level Game Environment

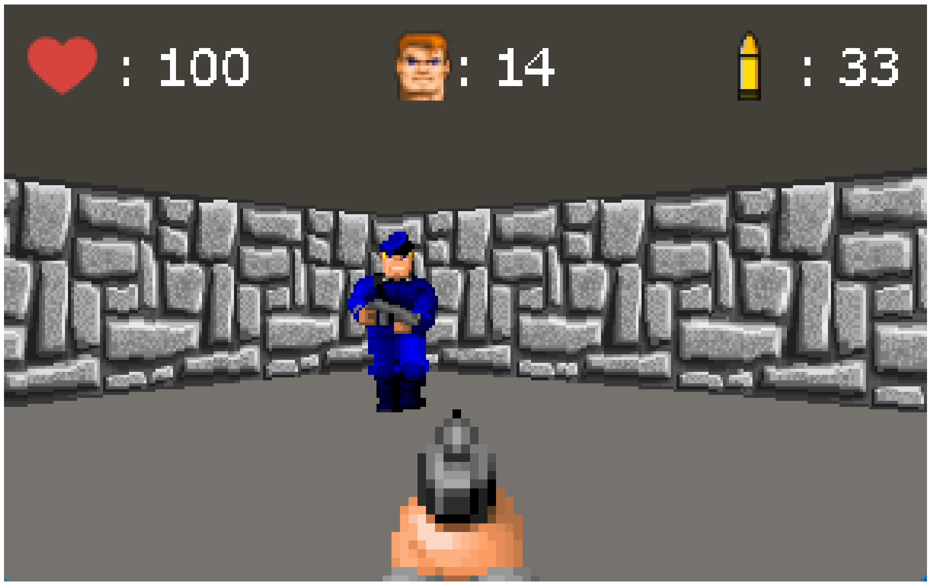

Figure B.8: Defend the Center Level Game Environment

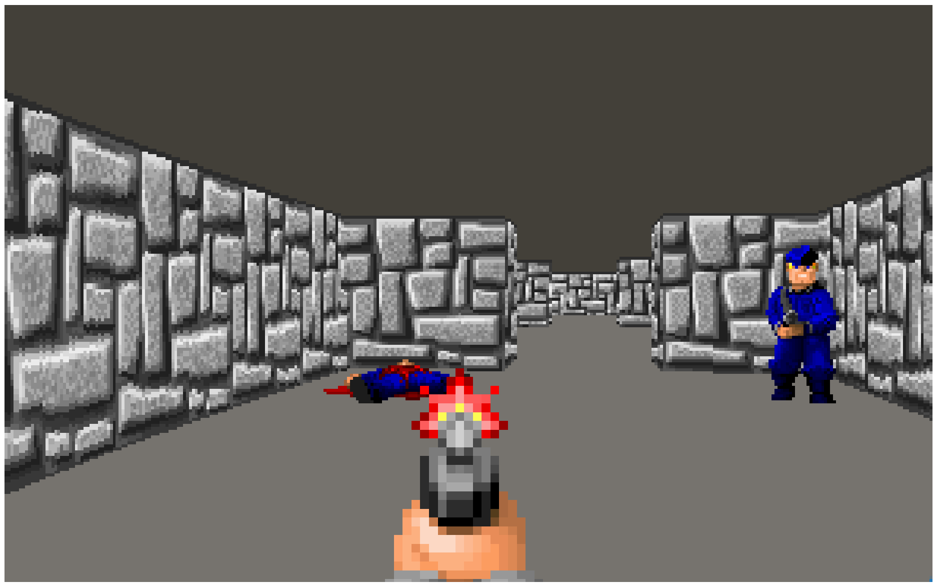

Figure B.9: Deadly Corridor Game Environment