MCTS-PPO Based Dynamic Difficulty Adjustment in Games

A thesis submitted for the degree
Bachelor of Information Technology (Honours)

24 pt Honours project, S2/S1 2022–2023

By:
Zhicheng Zou

Supervisor:
Dr. Penny Kyburz

Australian
National
University

School of Computing
College of Engineering and Computer Science (CECS)
The Australian National University

May 2023

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

May, Zhicheng Zou

Acknowledgements

This was my first research project in my four years at university. The year I worked on this project was one of the most rewarding years of my college life. Looking back on all of these processes, from the initial selection of the topic, to defining the research direction, proposing and implementing the methodology, conducting the experiments, and completing the thesis report, I am so proud of what I have achieved. It has been a really tough and torturing year for me, but in the mean time, it is the most valuable time I have ever had.

First, I would like to express my deepest gratitude to my advisor, Dr. Penny Kyburz, for her constant and timely feedback throughout the course of my research project and for guiding me in writing my thesis.

I would also like to express my sincere gratitude to all those who have supported me and provided me with valuable advice throughout my journey.

Finally, I would like to express my sincere gratitude to my family for their financial and emotional support. Without their support, I would not have been able to study abroad and conduct this research project. Their love, encouragement, and trust in me have been a source of strength for me.

Abstract

Dynamic difficulty adjustment (DDA) in games is a highly active research field, and it aims to dynamically adjust the game difficulty to suit the individual player's skill level during the gameplay. The successful integration of the DDA mechanism into games can make the game stands out from other similar games as it provides a superior and more immersed gaming experience. Various algorithms have been applied to achieve the DDA mechanism in games. In this thesis, by combining the Monte Carlo Tree Search algorithm and the Proximal Policy Optimization (PPO), we proposed a novel DDA mechanism that uses a neural network to guide the Monte Carlo Tree Search algorithm to find an appropriate game difficulty level. While the MCTS algorithm has demonstrated impressive success in games such as Chess and Go, most existing DDA mechanisms based on the MCTS algorithm exhibit search deficiency in real-time game settings, leading to degraded performance in adjusting the game difficulty. Our proposed DDA mechanism was tested in a two-player real-time fighting game platform, and the results indicate that our approach can alleviate the search deficiency problem while satisfying the strict time constraint of real-time games. Furthermore, many existing DDA mechanisms may produce abnormal game behaviours to adjust the difficulty of the game, which may negatively impact the gaming experience. Our proposed DDA mechanism overcomes this issue and can effectively reduce the probability of abnormal actions taken by our agent while still effectively adjusting the game difficulty.

Table of Contents

Chapter 1

Introduction

The landscape of video games has witnessed remarkable growth, evolving from simplistic single-player games of the early 1970s (e.g., Speed Race (1974) and Space Invaders (1978)) to the current era dominated by complex multiplayer real-time combat games (e.g., StarCraft and Dota). Today, a significant portion of the population dedicates substantial time to playing games daily. According to a game survey conducted in 2022 in the United States, 25% of the participants reported spending at least 12 hours playing video games. Additionally, a study conducted in Australia during the Covid-19 pandemic revealed that, on average, Australians spent 83 minutes playing video games. The need for games to stand out amidst the plethora of similar games available has become of paramount importance (Zohaib, 2018). However, achieving this objective extends far beyond superficial considerations, such as the game's theme or visual aesthetics, with one crucial aspect being game difficulty.

The appropriate calibration of game difficulty is crucial in providing an engaging gaming experience. Games that are either excessively easy or overly challenging can result in negative player experiences (Lomas et al., 2017). In the field of psychology, the concept of FLOW state refers to a mental state where individuals can fully immerse themselves in a single activity. Drawing upon this idea in the context of video games, Chen (2007) proposed a FLOW state model specifically tailored for video games, as depicted in Figure 1.1. This figure portrays how the player's experience is influenced by the difficulty level of the game. The X-axis represents the player's gaming ability, which corresponds to their level of expertise in the game. The Y-axis, on the other hand, represents the difficulty level of the game. The model highlights that when the player's expertise level is high, but the game difficulty is low, they are prone to easily becoming bored. Conversely, when the player is not proficient in the game, but the difficulty level is high, they are likely to feel anxious and perceive the game as unsuitable for their abilities. In both scenarios, such situations lead to unfavourable gaming experiences, causing players to quickly abandon the game. From a game design perspective, this outcome is clearly

undesirable.

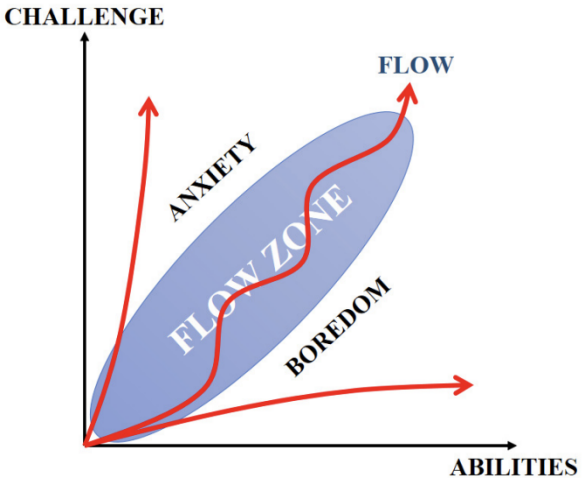


Figure 1.1: The FLOW state in games (Chen, 2007).

Moreover, considering that players possess varying levels of gaming abilities and learning capacities, designing a fixed difficulty level that caters to the expectations of all players is inherently impractical. Furthermore, the perception of game difficulty may differ significantly between game designers and players. Consequently, the concept of dynamic difficulty adjustment (DDA) in games emerged in the late 1980s. DDA aims to dynamically and automatically adapt the game's difficulty level during gameplay based on the information collected from the game state.

Extensive research has been conducted on various methods for developing or improving DDA in games, with innovative applications observed in diverse fields. Notably, the number of research papers focused on DDA has significantly increased over the years, with the count nearly tripled between 2009 and 2017 (Zohaib, 2018). One illustrative example of DDA application is in the popular racing game 'Mario Kart', where leading racers are more susceptible to the effects of other players' special attacks, which can impede their progress and create opportunities for other players to catch up (Vang, 2022). Furthermore, DDA has found adoption in official test systems, such as the Graduate Record Examinations (GRE), where the accuracy of a candidate's answers to previous questions can directly influence the difficulty level of subsequent questions.

Developing an effective mechanism for DDA to set appropriate challenges for players is a complex task. Various approaches exist for implementing DDA, which can be roughly categorized into two categories: one that adjusts game difficulty purely based on game state matrices obtained during gameplay, and the other that incorporates mental or brain information using external monitors. For example, some studies have utilized functional near-infrared spectroscopy (fNIRS) to obtain passive brain sensing data and detect prolonged periods of boredom or cognitive overload, while others have used self-

reported emotions to indicate when difficulty adjustments are needed (Afergan et al., 2014; Frommel et al., 2018). However, relying on explicit measurement of difficulty level through external monitoring may be impractical, as not all players have access to measuring tools at home, and the use of such tools may cause interference or discomfort due to being monitored. Moreover, privacy concern is another major problem related to external sensors. Therefore, in this thesis, we will primarily focus on the first type of DDA mechanism, as our primary interest is in exploring how different algorithms can generate an effective approach for game difficulty adjustment.

In this thesis, we proposed an approach that utilizes the proximal policy optimization algorithm (PPO) and the Monte Carlo tree search algorithm (MCTS) for the DDA mechanism. The MCTS algorithm, renowned for its capability to make informed decisions in complex decision spaces, will serve as the foundation for adjusting game difficulties. Meanwhile, the PPO algorithm, a state-of-the-art deep learning algorithm, has demonstrated its effectiveness in training AI agents in games (Engstrom et al., 2020). And we will employ it as a guiding mechanism for the MCTS algorithm, facilitating successful adjustments of game difficulties. Our proposed DDA approach will be evaluated in a two-player real-time fighting game named FightingICE (RITSUMEIKAN, 2023).

The reasons why we choose to propose an approach for a real-time fighting game are the following: First, it involves complex decision-making processes, requiring players to effectively evaluate the opponent's behaviours and strategies, which provides a good platform for evaluating the effectiveness of the proposed DDA approach. Also, the real time constraint is a prevalent characteristic in modern games. Consequently, it is crucial to propose an effective DDA approach that can perform optimally within these time constraints. Meanwhile, the nature of a fighting game necessitates a successful DDA approach to provide players with thrilling and immersive in-game experiences. The dynamic adjustment of game difficulty in a fighting game significantly impacts player engagement and enjoyment.

Implementing an effective DDA mechanism in real-time fighting games presents several challenges that need to be addressed. One challenge is to ensure that the player controlled by the DDA agent engages in normal game behaviour, even when the game character controlled by the real player is in a disadvantaged state (e.g., having very low HP compared to the DDA-controlled character). The DDA-controlled character should not remain still or perform repetitive defensive actions, such as continuous jumping, while these actions may reduce the challenge level, these obvious abnormal behaviours will negatively impact the player's experience and undermine the goal of difficulty adjustment. The real-time constraint is another crucial problem in implementing DDA in real-time video games, and it makes it challenging to select the most appropriate actions within strict time constraints. Real-time games require quick action execution, and the DDA mechanism must be able to adjust the game difficulty on the fly effectively without causing delays or disruptions in gameplay. Moreover, the performance of the MCTS algorithm, which forms the basis of our proposed DDA approach, is closely tied to the allocated search time for exploring actions within the action space. Insufficient execution

time can lead to insufficient exploration of the action space, resulting in the diminished performance of the MCTS algorithm. As a result, due to this inherent limitation, the adoption of the MCTS algorithm in real-time games is limited.

Our proposed approach in this thesis has two primary objectives. The first objective is to address the limitations of the MCTS algorithm by integrating it with the PPO algorithm. This integration aims to develop a robust and efficient DDA approach specifically tailored for real-time video games and alleviate the performance degradation of the MCTS algorithm in real-time settings by leveraging the capabilities of the PPO algorithm. The second objective is to harness the strengths of the PPO algorithm to reduce instances of abnormal behaviors exhibited by the agent controlled by our proposed DDA approach. By doing so, we aim to provide players with a more immersive and enjoyable in-game experience. By achieving these objectives, our proposed approach seeks to pave the way for future advancements and applications of the MCTS algorithm in video game designs and DDA mechanisms.

1.1 Thesis outline

This thesis will be presented in the following structure:

- In the Related Work chapter (Chapter 3), we will discuss different approaches to DDA mechanisms in different genres of games together with their pros and cons, as well as why these proposed approaches do not suit the real-time fighting game setting.
- In the Methodology chapter (Chapter 4), we will present our proposed DDA approach in detail. Our proposed DDA approach aims to reduce the search space of the MCTS algorithm and minimise the occurrence of abnormal behaviours during the gameplay with the opponents.
- In the experiment and evaluation chapter (Chapter 5), we will discuss different experiments conducted to evaluate the performance of the DDA agent we proposed.
- In the discussions chapter (Chapter 6), we will compare our proposed approach with other benchmark approaches and discuss the effectiveness and the limitations of our proposed DDA agent.
- In the conclusions chapter (Chapter 7), we will provide the takeaway messages and present future works.

Chapter 2

Related Work

Over the years, numerous approaches to DDA have been proposed, reflecting the evolving landscape of game design. This evolution has seen a transition from games that task players with self-assessing their competence and selecting an appropriate difficulty level, to games that employ automated mechanisms to determine and adapt the game difficulty on behalf of the players. From rule-based and probabilistic-based DDA approaches to the deep neural network based DDA approaches, we have seen a huge advancement in the implementation of DDA mechanisms.

2.1 Definition of DDA

DDA can be defined as the process of automatically modifying in-game features in real-time based on players' performance during gameplay. These in-game features encompass various aspects, including the performance of enemy AI, resource availability to the player, time limits, and so on. The DDA developer has the discretion to choose which feature or features to adapt during the game, with the ultimate goal of optimizing the player's experience by preventing them from feeling overwhelmed or underwhelmed so as to keep players within the "FLOW ZONE" mentioned previously, ensuring they stay fully engaged and immersed in the game. By achieving this, DDA has the potential to prolong the time players spend in games, enhancing their overall experience and satisfaction.

A well-crafted DDA mechanism possesses the capacity to offer a tailored gaming experience to players with different skill levels. With regard to novice players, it is imperative for the DDA mechanism to effectively reduce the game's level of difficulty, to prevent newcomers from becoming overwhelmed by the intricacies of the game environment, while granting them opportunities to progressively enhance their proficiencies. On the other hand, for proficient players, the DDA mechanism should adeptly escalate the game's

level of difficulty, thereby averting any potential boredom that may arise from a lack of challenge.

There are different approaches and algorithms for implementing the DDA mechanisms. We will first provide how difficulty is updated in early game designs, followed by a comprehensive examination of more sophisticated and contemporary approaches to DDA.

2.2 Traditional Approach to Difficulty Adjustment

Figure 2.1: A simple game diffculty adjustment strategy (Smith, 2021).

To underscore the significance of DDA, let us first introduce a widely used but non-dynamic difficulty adjustment mechanism commonly found in games. This mechanism involves players choosing a difficulty level from a series of pre-set levels at the outset of the game, as depicted in Figure 2.1. This classic approach is relatively straightforward to implement, with game developers setting different parameters during the development phase to achieve varying levels of difficulty. However, this design has severe limitations and falls short of maintaining player immersion in the game (Tremblay, 2011). Firstly, game developers may possess a divergent perception of a game's difficulty compared to the players themselves. Developers possess a better understanding of the game mechanics and have invested substantial time in testing and playing the game, making it challenging for them to accurately assess the difficulty level for players (Demediuk et al., 2019). Additionally, the traditional difficulty level selection mechanism is often only effective at the beginning of the game, since with the progression of the game, the players' gaming skills will also improve over time, and this approach requires players to subjectively assess their abilities to achieve the goal of adjusting the challenge level accordingly, however, it is usually hard for players to assess their gaming ability accurately. Also, player preferences may differ from each other, with some players desiring more challenge while others prefer a moderate challenge level, and a set of predefined difficulty levels lack of diversities and personalized elements. This traditional difficulty adjustment approach has remained prominent in early game designs and serves as a precursor to the modern DDA approaches.

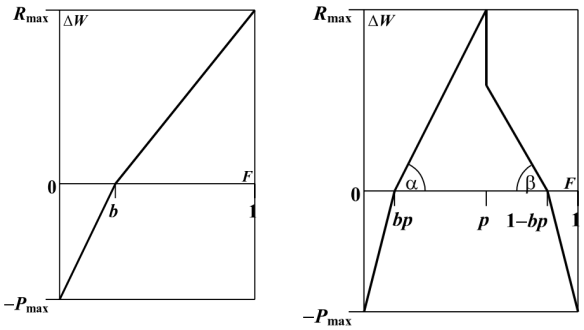
The adoption of DDA approaches presents a viable solution to overcome the limitations of the traditional non-dynamic approach, leading to a more personalized and captivating gaming experience. Through the utilization of real-time gaming data, advanced algorithms, and player modeling techniques, DDA can dynamically adapt numerous in-game features, ensuring an optimal equilibrium between challenge and enjoyment tailored to individual players. This paradigm shift entails a departure from players being required to conform to pre-established difficulty levels within the game, instead allowing the game itself to adapt to the distinct abilities and preferences of each player, which will ultimately resulting in a more gratifying gaming experience.

2.3 Rule based DDA Approach: Dynamic Scripting

"Scripting" is a frequently employed technique for implementing AI in commercial video games, as it offers numerous advantages (Spronck et al., 2003). Scripting entails the development of a predefined set of rules that govern the behavior of AI entities. However, scripting exhibits several drawbacks. For instance, it lacks creativity due to the limited number of rules, leading to suboptimal or foolish AI behaviors. Additionally, scripted AI is susceptible to counter strategies and struggles to adapt to new and evolving situations (Majchrzak et al., 2015). To address these challenges, the technique of "dynamic scripting" has been proposed Spronck et al. (2004). It is an online reinforcement learning method wherein the algorithm promptly responds to the evolving performance of players in real-time, allowing for greater adaptability and improved AI decision-making.

Dynamic scripting, as a rule-based DDA approach, involves the utilization of a rule database created by game developers. Similar to the scripting approach, dynamic scripting selects rules from the database based on players' performance, and these selected rules determine the actions performed by the DDA-controlledz AI. However, dynamic scripting diverges from traditional scripting in that it updates the weights of the rules in the rule database after each gameplay to optimize the AI's strategy. This approach was initially introduced by Spronck et al. (2004). with the aim of enabling adaptive performances by non-player characters (NPCs) in fighting games. To facilitate DDA, Spronck et al. introduced additional techniques such as high-fitness penalizing, weight clipping, and top culling for effective difficulty management. All of these techniques are simple to implement. As depicted in Figure 2.2, the left subfigure illustrates the conventional approach where higher fitness values correspond to higher rewards. In contrast, the modified approach depicted in the right subfigure demonstrates that the highest reward is not attained at the maximum fitness value, but rather at an intermediate point. This is to encourage the selection of actions that result in middle fitness values. The underlying principles of the other two approaches are similar to the concept of the penalizing approach.

According to empirical experimentation conducted on two different fighting games, Spronck et al. (2004) observed that dynamic scripting exhibited the ability to adapt to an opponent's strategy after only a few battles. However, the proposed dynamic script-



(a) without penalty (b) with penalty

Figure 2.2: Comparison of including high-fitness penalising (Spronck et al., 2004)

ing approach still possesses certain limitations and drawbacks. One primary limitation is its reliance on domain knowledge for constructing the rule database. The authors acknowledged that implementing dynamic scripting becomes challenging without comprehensive knowledge of the game's domain. Additionally, the process of hardcoding the rule database requires specialized game-specific knowledge, making the coding process intricate and time-consuming. In more complex games, constructing the rule database becomes even more complex, as multiple game factors need to be considered. To address this challenge, the integration of evolutionary algorithms into dynamic scripting has been proposed Szita et al. (2009). This approach reduces the manual effort required for rule encoding while achieving comparable performance to manual coding, although it still necessitates strong domain-specific knowledge. Secondly, after the rules are selected from the rule database, it still requires some mechanism to prioritise these rules, as some rules cannot be applied unless their prerequisites are satisfied. In the original dynamic scripting approach, this issue requires manual handling by the developers. An enhanced version is to use the relation-weights table with the selection bonus approach (RWO+B) approach combined with the dynamic scripting to enable the automatic generation of the rule priorities (Timuri et al., 2007). Another alternative dynamic scripting approach was proposed by Szita et al. (2009) that employs an automatic macro generation method to significantly reduce the search space for rules and accelerate the rate of adaptation in dynamic scripting, enhancing its effectiveness.

Despite the various enhancements made to the traditional dynamic scripting approach, there remains a crucial limitation: the process of updating rule weights only occurs at the end of each gameplay, which lacks full dynamism. Consequently, real-time adjustment of difficulty throughout the game is not fully realized. And as the gameplay strategy heavily relies on the rules selected at the beginning of the game, this results in a latency in the approach's ability to adapt to the opponent's performance. As pointed out by Majchrzak et al. (2015), dynamic scripting does not adapt to agents' behaviours in a real-time manner. Furthermore, while dynamic scripting may be capable of adapting to

opponent strategies in just a few turns, its performance may be significantly hampered when players frequently change their strategies. This will result in the dynamic scripting approach can not stably learn the opponent's strategy. An additional challenge arises when dealing with increasingly complex games. As games become more intricate, the rule database expands and becomes challenging to maintain. This limits the adaptability of dynamic scripting and its ability to be effectively applied to other games.

2.4 Probabilistic-based DDA Approach

Another genre of implementing DDA approaches the problem from a probabilistic perspective, considering it as an optimization problem. In this approach, the primary objective is to maximize player engagement throughout the game by calculating probabilities to dynamically adjust the game difficulty to maximize some objective functions.

2.4.1 Probabilistic Method via Parameter Manipulating

The first instance of a probabilistic-based DDA technique was introduced by Segundo et al. (2016), wherein probability calculations were integrated into the heuristic function to facilitate the implementation of DDA. The authors conducted a series of experiments employing a space shooter game as a testbed for their DDA mechanism. Within this framework, the DDA mechanism evaluates the player's remaining Health Points (HP) percentile and the associated probability of encountering potentially lethal damage, utilizing a Gaussian distribution. To illustrate, if a player has 40% of their HP remaining and there is a 70% probability of sustaining damage that could lead to their demise, the DDA mechanism would dynamically reduce the game difficulty. This adjustment may involve providing supplementary resources or diminishing the enemy's offensive capabilities, thus reducing the probability of death.

While the authors demonstrated the positive impact of their proposed probabilistic-based DDA on enhancing the user experience, several challenges are associated with this approach. One major concern is the necessity of a more sophisticated heuristic function to apply the technique to complex games. Acquiring and designing such a heuristic function is often a challenging task. Additionally, the reliance on a Gaussian distribution assumption may not always hold, leading to inaccurate predicted probabilities. Consequently, the performance of the DDA mechanism may be compromised due to these inaccuracies.

2.4.2 Probabilistic Method via Player's Progression Model

Xue et al. (2017) proposed a second probabilistic-based DDA approach, which focuses on modeling players' progression in a game as a probabilistic graph. The primary objective of this DDA approach is to optimize player engagement and gameplay duration by optimizing the player's progression through game levels. The authors evaluated their method using a level-based game (such as Plants vs. Zombies developed by Electronic

Arts, Inc.) where the player's progression can be modelled by using two essential features: the number of trials (indicating the frequency of gameplay attempts at a particular level) and the game level reached (advancing to the next level requires successful completion of the current level tasks). The progression model is illustrated as a graph structure, as depicted in Figure 2.3, where each circle represents a distinct player state. Within each layer of the graph, players may need to undergo multiple trials before progressing to the next level. And there exists a probability associated with advancing from one level to another. Notably, a churning state is introduced to represent players who leave the game permanently, without the intention of returning to the game.

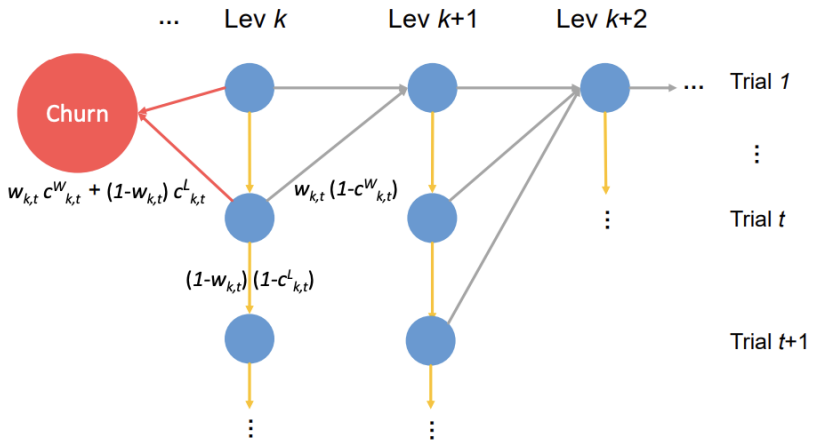


Figure 2.3: Example of the player progression model(Xue et al., 2017).

The utilization of a probabilistic-based approach for modelling player progression is indeed an innovative concept. The objective of the DDA mechanism becomes to maximize the likelihood of players remaining within the progression model before reaching a churning state. However, a significant challenge arises when attempting to implement this mechanism—the lack of prior knowledge regarding the churn rate, which is typically an unknown parameter in games. While Xue et al. (2017) suggested the use of advanced predictive models to estimate the churn rate based on individual features, this introduces additional computational complexity to the dynamic programming optimization process, potentially leading to performance degradation. Furthermore, this type of DDA mechanism exhibits limited adaptability. Its applicability is confined to specific game genres, particularly level-based games, where constructing a clear and straightforward progression model is feasible. Conversely, for games with more intricate settings such as role-playing games, this approach may not be as suitable. These games often involve multiple progressions occurring simultaneously, with non-linear progression paths, making it challenging to construct a comprehensive progression model (Zohaib, 2018).

2.4.3 Probabilistic Method via Bayesian Optimization

Khajah et al. (2016) introduced another probabilistic-based DDA approach by using Bayesian optimization to maximize players' engagement. Bayesian optimization is a valuable method for identifying the optimal solution by leveraging prior knowledge and data observations, making robust inferences from noisy data. The authors asserted that employing Bayesian optimization effectively prevents users from experiencing suboptimal designs, thereby enhancing their engagement during gameplay. However, a limitation of this proposed approach is that the optimization was performed on a group of players rather than targeting individual players. To address this, the authors proposed conducting optimization for specific users by conditioning it on their previous gaming history, thereby providing a personalized in-game experience. Nevertheless, the authors acknowledged that achieving this objective necessitates addressing the significant concern of data efficiency.

2.5 Deep Learning based DDA Approach

With the advancements in deep learning algorithms, there is a growing interest in utilizing these algorithms to implement DDA approaches. Or et al. (2021) introduced a neural network-based approach for implementing the DDA mechanism. The authors highlighted a limitation of existing DDA approaches, which often neglect the influence of other players, despite their significance in game difficulty adjustment. To address this, they modified the loss function used for training the neural network. The proposed loss function consists of two components. The first component is the variance of the expected difficulty levels across all players, emphasizing the importance of considering the gaming abilities of other players. The second component is the discrepancy between the current game difficulty level and the expected game difficulty level for each player in the game. By utilizing stochastic gradient ascent to minimize the loss function, the neural network parameters are iteratively updated to find optimal difficulty levels for individual players in the game. Additionally, the authors incorporated the complete rate, representing the percentage of players who successfully completed a specific task, as an additional constraint during neural network training. They argued that the complete rate can further enhance the accuracy of difficulty adjustment. Leveraging the expressive power and capability of neural networks, the experiments conducted by the authors demonstrated the superiority of the proposed DDA approach over heuristic methods devised by game designers in various multiplayer game environments.

One limitation associated with utilizing the game difficulties directly as the loss function is the reliance on the ability to compute these difficulties based on in-game features. However, determining an explicit representation of the difficulties and selecting the appropriate features for computing the difficulty level poses significant challenges. Consequently, directly computing the game difficulty level remains an unrealistic approach, emphasizing the need for alternative methods that can adjust the game difficulty without relying on explicit difficulty level computations.

2.6 MCTS-based DDA Approach

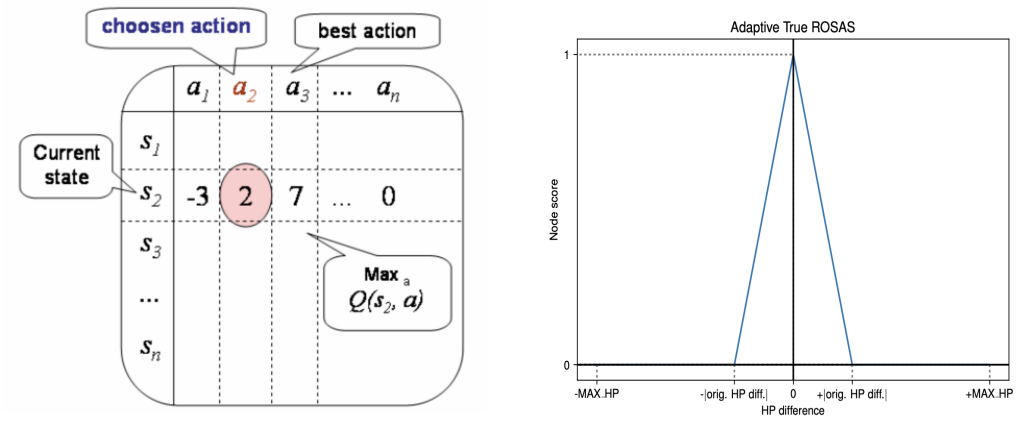
The successful application of the MCTS algorithm in traditional board games, such as Go and Chess, including its notable achievements with AlphaGo and AlphaGo Zero (Silver et al., 2017), has sparked extensive research on the potential applicability of MCTS in other similar games. However, real-time games present significant challenges for various AI techniques, including MCTS, primarily due to their real-time nature and the large number of possible actions and states, which makes it hard for the algorithms to find an optimal action within the limited time constraint. To overcome these challenges and further expand the utilization of MCTS in real-time games, researchers have proposed various enhanced algorithms that build upon the core MCTS framework. These enhancements encompass techniques such as combinatorial multiarmed bandit (CMAB)-based sampling (Ontanón, 2017), state/action abstractions, Bayesian models (Ontanón, 2016), pruning techniques (Ouessai et al., 2020), and machine learning algorithms (Świechowski et al., 2018)(Kim et al., 2020). These advancements have facilitated the adaptation of MCTS for eal-time games. However, despite these notable progressions, there is a limited number of research that has specifically investigated the integration of MCTS with DDA in the context of real-time games.

In this section, we will present several DDA mechanisms that are based on the MCTS algorithm. These agents have been evaluated using the FightingICE test platform. The MCTS algorithm consists of four main phases: selection, expansion, simulation, and backpropagation. These four phases together form an iterative process. In each iteration, an action from the action space is selected and being simulated to estimate the reward will receive. By performing a large number of iterations within the given time and computational constraints, a game tree is constructed. Each node in the tree represents an action that has been selected and simulated. The algorithm then selects the action with the best simulation result to perform. A detailed description of the MCTS algorithm will be provided in the methodology section.

2.6.1 MCTS with Modified Action Selection Criteria

Demediuk et al. (2019) proposed three distinct MCTS based approaches, each with different modified node selection rules. The first agent, named "Challenge Sensitive Action Selection" (CSAS), builds upon the traditional MCTS algorithm and utilizes the Upper Confidence Bound applied to Trees (UCT) node selection strategy. The UCT strategy is widely employed in MCTS to guide the selection phase, aiming to identify the optimal node for the simulation process. The UCT node selection strategy maintains a balance between exploration of unexplored nodes and exploitation of already explored nodes, preventing the algorithm from getting stuck in suboptimal situations. The traditional MCTS-UCT algorithm selects the node with the highest evaluated score as the final action to perform. In the CSAS agent, modifications are made to adapt the MCTS-UCT algorithm for DDA, where a Q-table is employed during the final node selection to rank each action based on values obtained from MCTS simulations. The

algorithm then selects an action with a rank that aligns with the current game difficulty level. For instance, if the player's HP value is higher than the opponent's HP value, the action's rank is decreased by one, encouraging the DDA agent to select a less competitive action. The overall process is illustrated in Figure 2.4(a).



(a) Challenge Sensitive ACTION Selection

(b) selection strategy of the Reactive Outcome Sensitive Action Selection

Figure 2.4: MCTS-DDA agents proposed by Demediuk et al. (2019)

Another agent proposed by Demediuk et al. (2019) is called "Reactive Outcome Sensitive Action Selection" (ROSA), which is another variant of the MCTS-UCT algorithm. In contrast to CSAS, ROSA evaluates the value of a node based on the difference in HP between the player and the opponent. To determine the simulation result of a node, the HP difference is normalized to a range of 0 to 1. A HP difference of 0 corresponds to a node value of 0.5. The formula for final node selection in ROSA is defined as follows:

$$action = \arg\min \begin{cases} 0.5 - r[a].score & \text{if } r[a].score \leq 0.5 \\ r[a].score - 0.5 & \text{otherwise} \end{cases}$$

where r is the root node, and $r[a]$represent the child node a of the root node r. This formulation implies that the agent prefers actions that have a node value close to 0.5, as it aims for the HP difference to be as close to 0 as possible.

The final MCTS-based DDA agent proposed by Demediuk et al. (2019) is called "Adaptive True Reactive Outcome Sensitive Action Selection" (ATROSAS). This agent builds upon the second agent but incorporates additional considerations. It takes into account both the HP difference before and after taking a specific action when calculating the node value in the MCTS simulation phase, unlike the previous two agents that only consider the HP difference after taking the action. This added adaptation enhances the evaluation of the node value, providing a more refined assessment. The assignment of

value to the node is illustrated in Figure 2.4(b). The interpretation is as follows: if the absolute value of the HP difference after taking the action is smaller than the absolute value of the HP difference before taking the action, the action can be regarded as a good action. The closer the current HP difference is to 0, the better the selected action is considered to be. This evaluation strategy encourages the actions that will reduce the HP difference.

The three DDA agents proposed by Demediuk et al. (2019) represent a significant advancement in utilizing MCTS for implementing the DDA mechanism. However, these approaches do not fully address the two main objectives that we aim to achieve. Firstly, they primarily focus on designing different selection criteria to choose the best action for adapting to different levels of difficulty, without adequately addressing the challenge of ensuring sufficient exploration of all actions within the limited response time and large action space of real-time games. And as we mentioned, the primary constraint that hampers the adoption of the MCTS algorithm in real-time games is the compromised performance caused by inadequate exploration of the action space due to time limitations. Secondly, these agents only consider adjusting game difficulties based on the HP difference, neglecting the potential consequences of selected actions leading to abnormal behaviors.

2.6.2 MCTS-DDA with Believable Behaviours

The MCTS-based DDA approach proposed by Ishihara et al. (2018) represents an improvement aimed at addressing the problem of abnormal behavior exhibited by MCTS-DDA agents during gameplay, so that to provide enhanced gaming experience. The authors introduce the concept of "believable behaviors" and "abnormal behaviors." Believable behaviors are actions that align with the goal of defeating the opponent, while abnormal behaviors are actions that contradict this goal, such as defensive actions taken even when the agent is not in unfavorable conditions. To mitigate this issue, the authors propose an MCTS-DDA agent with an updated evaluation function that incorporates believability as a factor in the evaluation process. The evaluation function for each node is defined as follows:

$$eval_j = (1 - \alpha)B_j + \alpha E_j$$

where:

$$B_j = \tanh \frac{BeforeHP_j^{opp} - AfterHP_j^{opp}}{Scale}$$

$$E_j = 1 - \tanh \frac{|AfterHP_j^{my} - AfterHP_j^{opp}|}{Scale}$$

$$\alpha = \frac{\tanh \frac{BeforeHP_j^{my} - BeforeHP_j^{opp}}{Scale} + 1}{2}$$

In the above equations, $BeforeHP_j^{opp}$ represents the HP value of the opponent before

taking action j, and $BeforeHP_j^{my}$ denotes the HP value of the agent itself before taking action j. $AfterHP_j^{my}$ represents the HP value of the agent itself after taking action j. $Scale$ is a constant that is used to scale the HP difference. B_j represents the believability of action j, which is calculated by the difference in the opponent's HP before and after taking action j, divided by the scale factor. The higher the difference in HP, the higher the believability value, indicating a more aggressive action. E_j represents the desirability of action j to achieve an HP difference of 0. Here we can consider B_j and E_j are two variables on two sides of a lever, where B_j prefers actions that cause huge HP difference, and E_j prefers actions that cause HP difference to be 0. The coefficient α is dynamically adjusted based on the HP differences, so as to control the balance between B_j and E_j, and thus dynamically adapt the difficulty of the game.

While the proposed agent has shown success in restraining abnormal behaviors, its method of calculating believability solely based on actions that cause damage to the opponent may overlook scenarios where actions without immediate damage can still be considered believable. Furthermore, similar to the three MCTS-DDA agents mentioned earlier (in 2.6.1 section), this approach still operates on the entire action space and does not fundamentally address the issue of under-exploration of all actions within the constrained time limit.

2.6.3 Mental state based MCTS-DDA Approach

Another MCTS-based DDA approach that incorporates deep learning models was proposed by Moon et al. (2022). In their work, instead of evaluating nodes based on the HP difference, the authors introduced a novel approach that utilizes deep learning algorithms to train four distinct mental state models. These models are specifically designed to predict the mental state of players, enabling the evaluation of how stressful a particular action would be for the player, and the prediction results will server as evaluation matrix to evaluate the goodness of the selected actions.

The authors trained four distinct player state models, each serving as a binary classifier, to capture different user mental states. These mental states, namely "Competence (CO)", "Valence (VA)", "Challenge (CH)" and "FLOW (FL)", were derived from the Game Experience Questionnaire (GEQ) survey. To train these player state models, participants with varying levels of competence were recruited and asked to engage in gameplay while providing feedback through a questionnaire that pertained to their game-playing experience. The collected feedback was then utilized to manually label the dataset, creating a series of "[game log, player state]" data pairs that were employed to train the four mental models.

In contrast to previous DDA approaches based on MCTS, which primarily relied on in-game features such as HP differences to evaluate the goodness of actions, this proposed DDA approach takes a direct approach by utilizing trained mental state models. These models directly predict the mental states of players that would result from performing selected actions. For instance, if one of the mental state models generates a score of

0.8, it corresponds to a positive classification result with a score of 0.8. Conversely, if a model outputs a score of 0.2, it corresponds to a negative classification result with a score of -0.8. This approach allows the four mental state models to provide scores that directly reflect the anticipated game experience associated with a specific action for the opponent, without relying on indirect indicators such as HP differences.

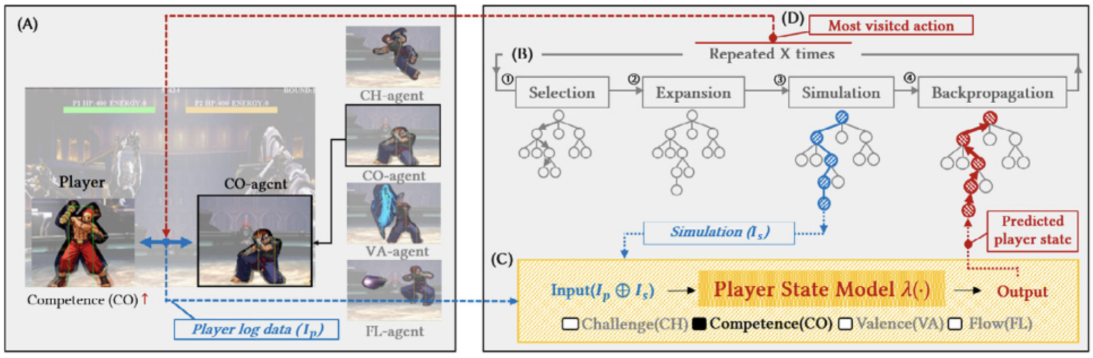


Figure 2.5: The mental state model incorporated MCTS-DDA(Moon et al., 2022).

Figure 2.5 provides an overview of the proposed MCTS-based DDA approach that incorporates mental state models. This approach distinguishes itself from the traditional MCTS algorithm in the evaluation strategy, as depicted in Part (C) of the figure. In the traditional MCTS algorithm, the evaluation of simulation results is based on direct calculation of the HP difference. However, in the proposed approach, player-state models utilize real-time game log I_p and simulated game log I_s as inputs to generate scores for each simulation. Part (A) of the figure illustrates the gameplay of the player with one of the four AI behaviors, where the currently active behavior is represented by the "CO-agent". Part (B) represents a single iteration in the MCTS algorithm. Finally, Part (D) showcases the most frequently visited nodes is selected as the next action which will be performed by the agent.

This proposed DDA approach, despite its innovative integration of player mental state into the DDA mechanism without requiring additional sensors, faces several challenges. Firstly, the training of player state models necessitates a substantial dataset to achieve satisfactory prediction accuracy. This implies the need for a large number of players with varying levels of competence to engage in multiple gameplays, thereby collecting diverse training data to enhance prediction accuracy. Meanwhile, manually labeling the training data can be time-consuming and impractical, particularly for complex games that are more challenging and intricate. The authors themselves also acknowledged that the performance of the player state model is not yet optimal. Secondly, relying solely on game logs as a measure of players' emotional states may pose certain issues. Game logs might have limited information and may not provide a comprehensive indication of players' emotions. Additionally, considering that players have diverse perceptions of the game difficulty and their perceptions of the stress are also different, training a

high-quality player state model can be challenging due to the inherent randomness in player behavior. Furthermore, similar to the previously discussed MCTS-based DDA approaches, this approach also lacks consideration for reducing the search space to enhance search efficiency.

2.7 Summary

In reference to the aforementioned DDA mechanisms, we have explored their suitability across various game genres alongside their respective merits and limitations. When considering the MCTS-based DDA approaches, a recurring issue emerges: the failure to adequately consider the extensive search space and the temporal constraints inherent in real-time scenarios. Consequently, the proposed approaches may exhibit substantial performance degradation when confronted with more intricate game environments.

In the forthcoming chapters, our research is centered on the MCTS-based DDA mechanism. We aim to tackle the obstacle posed by the restricted search capabilities resulting from the vast search space and real-time limitations. To address this challenge, we introduce a novel approach that combines the MCTS algorithm with the PPO deep learning algorithm. Our primary objective is to effectively alleviate the performance degradation by significantly reducing the search space required for the MCTS algorithm to achieve optimal performance. Additionally, in our proposed approach, we place a strong emphasis on ensuring the reliability of the DDA agent's behavior and minimizing the occurrence of anomalous actions taken by the agent.

Chapter 3

Methodology

In this section, we will thoroughly discuss our approach to implementation approach for the MCTS-based DDA agent within the FightingICE framework. Our proposed approach combines the PPO and MCTS algorithms together. Specifically, we leverage PPO to train a robust policy network capable of making informed decisions, and we utilize the output of the policy network to guide the search process of the MCTS algorithm effectively.

3.1 FightingICE Framework

We have selected the FightingICE platform as our test environment to evaluate the performance of the proposed DDA approach. FightingICE is an open-source fighting game test platform developed and maintained by the Intelligent Computer Entertainment Lab at Ritsumeikan University, Japan. It offers a real-time combat setting where two players can engage in battles. The platform is primarily implemented in Java, although Python support is also available through additional packages.

In the FightingICE framework, players can choose from three distinct characters for each round: "ZEN", "GARNET", and "LUD", each possessing unique skills and abilities. For the purpose of this thesis, we will focus solely on the character "ZEN" to simplify our analysis and implementation. We believe that extending the DDA approach to other characters within the game should be relatively straightforward to accomplish.

3.1.1 Game Features of FightingICE

In the FightingICE game environment, players are characterized by two essential parameters: "Hit Points" (HP) and "Energy." The HP value represents the player's health, indicating the amount of damage they can withstand before being defeated. On the

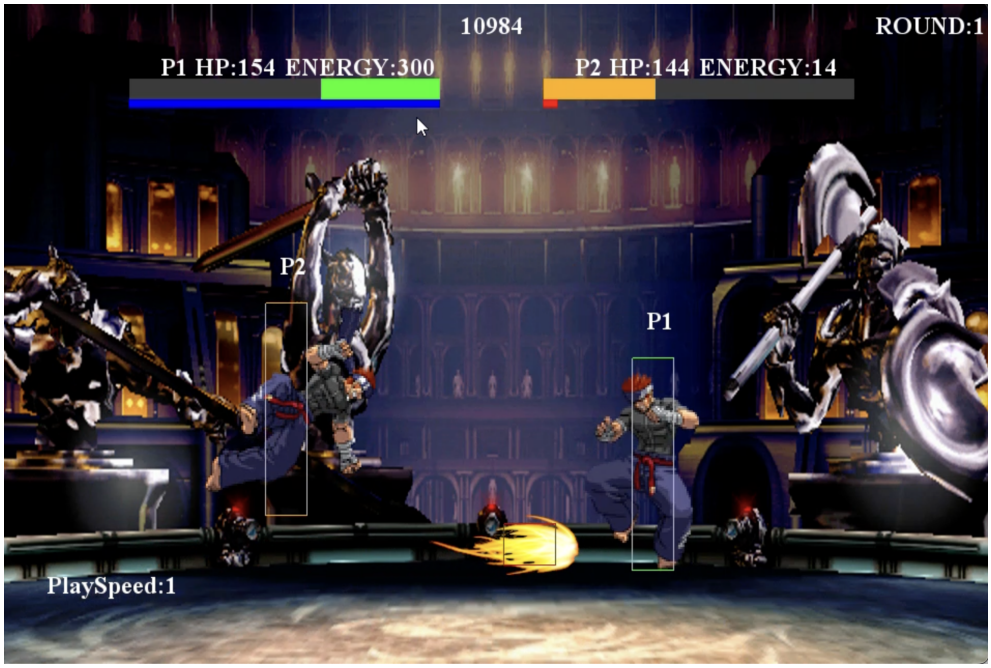


Figure 3.1: The screenshot of the fighting game.

other hand, the Energy value represents the player's resource for executing special actions or abilities. At the start of each game, both players have their HP values set to the initial default value, indicating their full health. Additionally, their Energy values are initialized to 0.

The winning condition in FightingICE is determined based on the following rules: If, at any given moment during the game, one player's remaining HP is greater than 0 while their opponent's HP is 0, the player with the remaining HP emerges as the winner. If both players have remaining HPs greater than 0 when the maximum game time is reached, the player with the higher remaining HP is declared the winner.

3.1.2 Gaming Process of FightingICE

The overall gaming process in FightingICE can be depicted as shown in Figure 3.2, illustrating the sequence of events during gameplay. The process begins with the initialization stage, where the player has the option to choose between playing against another human player or selecting an AI player as the opponent. Once the game is initialized with the desired settings, it progresses to the main game loop. Within each iteration of the game loop, the current frame data is collected, encompassing crucial information such as the HP and energy values of both players, as well as their positions and other relevant in-game features. Based on this data, both players make decisions regarding their actions for the current frame. These selected actions are then executed, resulting in the generation of new frame data. This iterative loop continues indefinitely as long

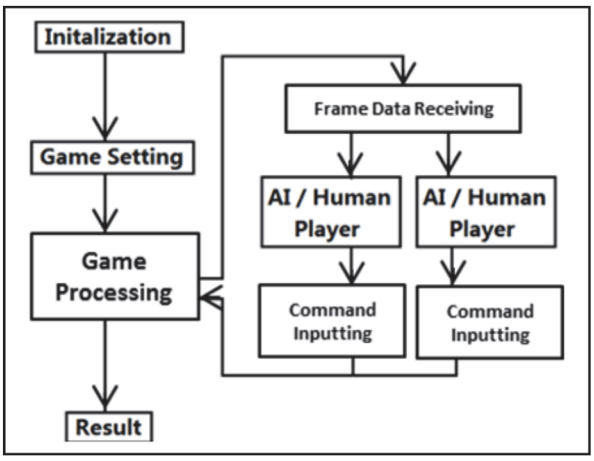


Figure 3.2: The overall game flow of game FightingICE (Lu et al., 2013).

as the HP values of both players remain greater than zero and the time limitation has not been reached.

In the FightingICE game, the minimum time unit is 1 frame, and the game runs at a rate of 60 frames per second. Consequently, AI players are required to provide a response within approximately $1/60 \approx 16.67$ milliseconds to ensure timely gameplay. Failure to meet this time constraint results in the previous action being reused. Additionally, there is a 15-frame information delay is introduced specifically for AI players. This delay corresponds to an approximate duration of 0.25 seconds, providing a simulation of the time it takes for human players to react and make decisions during gameplay.

For the "ZEN" character, there are 56 actions in total that the character can perform with. There are five different types of actions that the agent can perform in the FightingICE game, as summarized by Majchrzak et al. Majchrzak et al. (2015). These include *Basic actions*: These actions describe the neutral actions of the agent, such as standing, crouching, and being in the air. *Movement actions*: These actions are used to control the movement of the agent. *Guard actions*: These actions are performed in order to reduce the damage received when the agent is hit by the opponent's skills. They are used to reduce the damage received after being attacked. *Recovery actions*: These actions are performed automatically when the agent is hit by the opponent's actions. They are used to recover from being hit by the opponent. *Skill actions*: These actions are used to perform actions that can pose damage to the opponent.

Each skill action in FightingICE consists of three phases, as shown in Figure 3.3. The total number of frames for the action is divided into three phases: startup, active, and recovery. In this example, the total number of frames for this action is 18 frames, and the startup phase consists of 6 frames, the active phase consists of 2 frames and the recovery phase contains 10 frames. During the startup phase and the first few frames of the recovery phase, the agent cannot be controlled and is more vulnerable to being attacked.

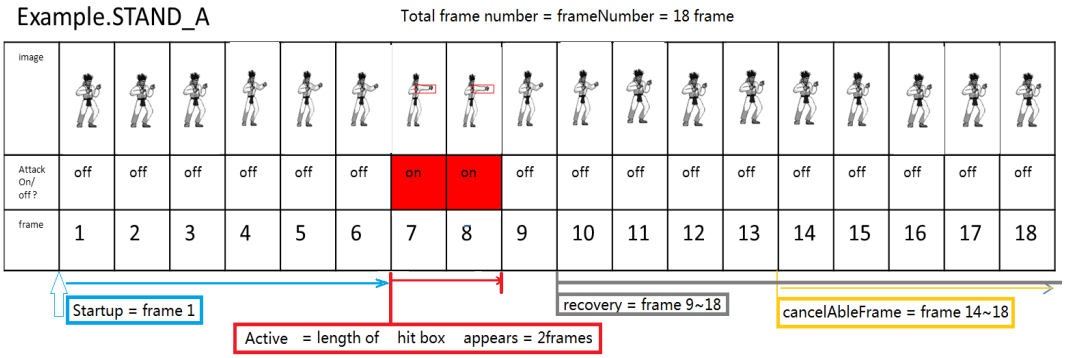
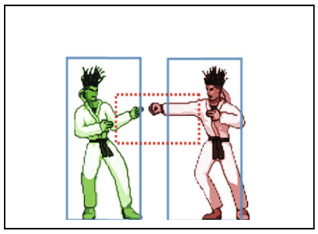
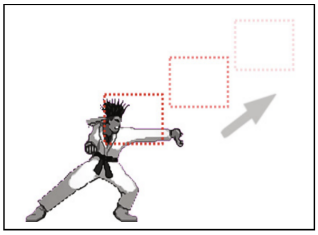


Figure 3.3: Three phases of an example action (RITSUMEIKAN, 2023).

And at the last few frames of the recovery phase (frames 14-18 in the example), these frames are called cancellable frames where the agent can start to perform new actions. The active phase is when the hitbox appears and will cause damage to the opponent if the opponent is touched by the hitbox as shown in figure 3.4.



(a) The hitbox touches the left agent and will cause damage to left agent.



(b) A projectile action will result in a hit box move away from the agent slowly.

Figure 3.4: An example of the hitbox (RITSUMEIKAN, 2023)

The distinctive features mentioned above have led to the widespread adoption of FightingICE by researchers, particularly in the realm of artificial intelligence (AI) development and research, as evidenced by previous studies (Cherukuri and Glavin, 2022). And the FightingICE framework provides comprehensive support for AI and machine learning development. The framework offers a lot of APIs that facilitate the creation and testing of custom AI models, making it accessible for developers to implement and evaluate their own approaches. Furthermore, the availability of numerous AI characters developed by fellow researchers provides a convenient platform to test our implemented agent.

3.1.3 Game Parameter Settings

To evaluate the effectiveness of our proposed DDA approach, we set the initial HP values for both players as 400, accompanied by setting their initial energy values to 0. The

duration of each game round has been defined as 80 seconds, corresponding to 4800 frames. The FightingICE game offers three distinct game characters, each with unique skills, but we will focus on the "Zen" character exclusively, while the potential extension of our DDA approach to encompass other characters is envisaged to be a straightforward endeavour.

There are several compelling reasons behind our choice of FightingICE as the test platform to assess the efficacy of our DDA approach. Firstly, the action space within this fighting game is reasonably manageable, comprising a total of 56 distinct actions. While this provides a substantial yet tractable space for evaluating our DDA mechanism, it is important to note that even with 56 actions, the MCTS algorithm still faces the challenge of thoroughly exploring the space within the limited response time of 16.67 ms. That is, the platform offers a judiciously sized action space, neither too large to make the evaluation process hard to control nor too small to evaluate if our proposed approach is conducive to the performance of the MCTS algorithm. Secondly, the FightingICE platform boasts a wealth of AI submissions from previous competitions, thus offering a valuable resource for assessing the performance of our DDA algorithm against a diverse range of AI opponents with varying levels of game proficiency. This enables us to comprehensively test and validate the effectiveness of our DDA mechanism with AI players of different proficiencies, facilitating a robust evaluation of its capabilities.

3.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) algorithm, originally introduced by OpenAI in 2017, has gained widespread popularity and has become one of the most commonly employed reinforcement learning algorithms. PPO has been adopted as the default reinforcement learning algorithm by OpenAI (Engstrom et al., 2020). This is attributed to its superior efficiency and ease of implementation compared to other reinforcement learning algorithms. And it exhibits outstanding performance in various challenging scenarios (Wang et al., 2020). Research by Yu et al. (2021) has shown that the adoption of PPO in multi-agent gaming environments, such as Google Research Football, has resulted in a remarkably strong performance.

PPO falls into the reinforcement learning algorithm, and it is different from supervised learning and unsupervised learning, as the agent does not receive explicit instructions on which actions to take. Instead, the agent interacts with the environment and adjusts its behaviour based on the rewards or punishments it receives from previous actions. Reinforcement learning is particularly well-suited for training agents to learn policies, such as in video games, where supervised learning and unsupervised learning may not be applicable due to the absence of labelled data or inherent patterns in the data (Sutton and Barto, 2018).

Here are several terminologies that are frequently used in deep reinforcement learning:

- **Environment:** The environment is the entity with which the agent interacts,

such as a game setting. The agent provides input in the form of state information and actions, and the environment responds by updating the state and providing a reward based on the action taken by the agent.

- **State, Action and Reward:** The state in a reinforcement learning process represents a snapshot of the environment at a specific time. If we are in a fighting game environment, the state information could be the HP values of players, their energy values, their positions and so on. The agent utilizes the state information to make predictions and decide which action to take. When the agent takes a particular action, the environment provides a reward, which serves as a criterion to evaluate the agent's performance in the environment. The reward value reflects how well the agent performed based on the chosen action.
- **Policy:** The policy pertains to the trained strategy that the agent will utilize for interacting with the environment.
- **Value function:** The value function takes the state or state-action pairs and evaluates how good the state or the action is.

PPO is classified as an on-policy method. On-policy algorithms employ the current policy network to interact with the environment and generate training data and use the trained data to update policy network parameters. Consequently, these algorithms exhibit consistent and correlated training data, which results in reduced variance and enhances training stability. However, on-policy methods necessitate a larger volume of data to learn a versatile policy. Also, as the training data is discarded once the policy is updated, On-policy algorithms typically suffer from low sample efficiency.

PPO is also built upon the policy gradient approach, which involves iteratively updating policy parameters by leveraging the gradient of the objective function to maximize expected rewards (For a more detailed explanation of the policy gradient approach, please refer to Appendix 7.1). In the context of the game environment, the objective function corresponds to the expected rewards obtained from various trajectories. A trajectory represents a sequence of states and actions generated by interacting with the environment using the current policy network, as depicted in Figure 7.1. Mathematically, a trajectory can be defined as follows:

$$\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$$

Thus, the objective function is:

$$\overline{R}_\theta = \mathbb{E}_\theta[R(\tau)] = \sum_\tau R(\tau) p_\theta(\tau)$$

where $R(\tau)$ is the total reward received for performing trajectory τ. The gradient of the objective function can be represented as follows:

$$\nabla \overline{R}_\theta = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau) \nabla log p_\theta(\tau)]$$

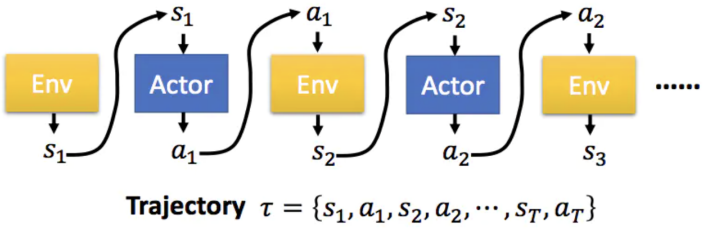


Figure 3.5: Policy gradient Agent-Environment interaction (yi Lee, 2018)

thus, the policy parameter θ can be updated by:

$$\theta = \theta + \alpha \nabla \overline{R}_\theta$$

The traditional policy gradient approach, REINFORCE, approximates the gradient of the objective function by sampling N trajectories. The gradient function can be expressed as follows:

$$\nabla \overline{R}_\theta = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau) \nabla log p_\theta(\tau)] = \frac{1}{N} \sum_{N=1}^{N} R(\tau_n) \nabla log p_\theta(\tau_n)$$

However, the REINFORCE algorithm suffers from several drawbacks. Firstly, the rewards received for each trajectory can exhibit significant variance, which can result in unstable training. Secondly, collecting a large number of trajectories (N) to approximate the gradient can be inefficient and time-consuming, and policy parameters can only be updated after gathering a sufficient number of trajectories. Lastly, selecting an appropriate learning rate for policy gradient methods can be challenging, and the learning rate significantly influences the convergence and stability of the algorithm.

Thus, PPO as an on-policy and policy gradient based reinforcement algorithm is proposed to improve sample efficiency and address the limitations encountered by traditional policy gradient approaches, thereby facilitating effective and stable training procedures while maintaining ease of implementation.(Zakharenkov and Makarov, 2021).

3.2.1 Actor-Critic Structure

To overcome the limitations inherent in the policy gradient approach, PPO incorporates an actor-critic structure comprising an actor (also called the policy network) and a critic (also called the value network). The actor is responsible for generating actions to interact with the environment, while the critic evaluates the performance of the actor by estimating the expected rewards for specific states or state-action pairs. By utilizing the value network, PPO achieves more stable reward estimations for future states compared to using actual rewards from sampled trajectories directly. By leveraging the critic's estimations, policy parameters can be updated in a temporal-difference manner, utilizing

only the current and next state information. This approach improves the training speed and sample efficiency as policy parameters are updated iteratively rather than waiting for the collection of entire trajectories.

3.2.2 Importance Sampling

The sample deficiency is due to the requirement of re-collecting new training data after updating the policy parameter θ. The approach of importance sampling can be employed as a means to alleviate the burden of acquiring new samples following each parameter update iteration.

Importance sampling can approximate the expectation of a function with respect to a probability distribution that is hard to sample directly, by using another distribution. Assuming that we aim to compute $\mathbb{E}_{x\sim p}[f(x)]$, this is equivalent to the following expression:

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = \mathbb{E}_{x\sim q}[f(x)\frac{p(x)}{q(x)}]$$

The technique of importance sampling allows us to leverage data obtained from the q distribution in order to estimate the outcomes of sampling from the p distribution. However, in order to achieve accurate estimation, it is imperative to ensure that the distributions $p(x)$ and $q(x)$ do not exhibit substantial differences. Otherwise, the estimation result may be inaccurate due to the significant disparity between the distributions.

Thus, by incorporating importance sampling into the gradient of the objective function, we can redefine the gradient of the expected reward as follows:

$$\nabla\overline{R}_\theta = \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla log p_\theta(\tau)] = \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}R(\tau)\nabla\log(p_\theta(\tau))]$$

The updated gradient of the expected reward function can be interpreted as utilizing the sampled data from policy parameter θ' to train the policy parameter θ iteratively. This technique eliminates the necessity of collecting new data for each update of the policy parameter, but it imposes a constraint on the magnitude of variation in θ between updates. The trajectory τ in the gradient of the expected reward function can be expanded by using all the state-action pairs within the trajectory, leading to the following updated gradient functions (for a detailed derivation, please refer to the appendix 7.2):

$$\nabla\overline{R}_\theta = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla\log(p_\theta(s_t,a_t))]$$

In the derived expression, $A^{\theta'}(s_t,a_t)$ represents the advantage function, which is an updated version of the reward function. Instead of using the actual rewards received, it is calculated using the value network and provides an estimation of the advantage of taking

action a_t at state s_t compared to the average reward received at state s_t under the policy parameter θ'. The advantage function considers not only the immediate reward but also takes into account the long-term effects of performing the specific action, incorporating a discount factor to indicate that future states further away are less likely to be influenced by the current action choice.

Next, we can formulate the objective function used to train the policy network based on the gradient of the expected reward function $\nabla \overline{R}_\theta$ as follows:

$$J^{\theta'}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t)]$$

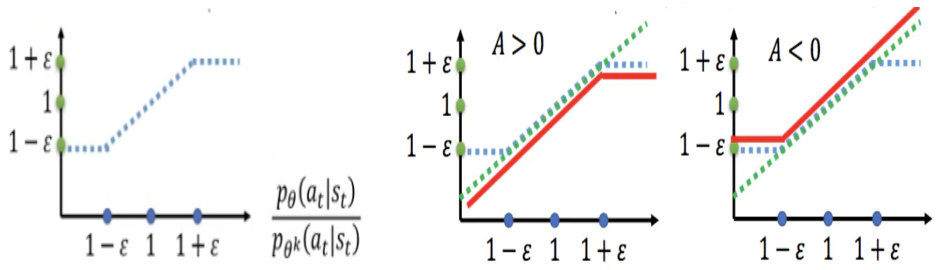
The notation $J^{\theta'}(\theta)$ indicates that we aim to optimize the parameter θ using data sampled from policy parameter θ'.

3.2.3 Clipping

To ensure accurate estimation through importance sampling, it is necessary to constrain the probability distributions of θ and θ' to avoid significant deviations. This is achieved by employing a clipping method. The clipping is applied directly to the probability ratio in the objective function.

So the objective function with clipping can be expressed as follows:

$$J^{\theta'}(\theta) = \sum_{s_t,a_t} \min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t),\quad clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right) A^{\theta'}(s_t, a_t)\right)$$



(a) Figure representation of the clip function

(b) Clipped value times reward

Figure 3.6: Figure representation of how clipping works (yi Lee, 2018)

The "min" function is utilized to select the minimum of two terms, aiming to minimize the change in θ. The first term in the min function represents the original objective, while the second term incorporates the *clip* function. The graphical representation of the *clip* function can be observed in Figure 3.6(a), which constrains the value within the range of $[1-\epsilon, 1+\epsilon]$. By multiplying the clipped value with the advantage value (the advantage

value may be positive or negative depending on how well the action is), as depicted in Figure 3.6(b), we obtain the results. In Figure 3.6(b), the green line represents the first term in the min function, which remains unclipped, while the blue line represents the *clip* function. The red line portrays two scenarios of the second term in the min function. In the left part of the subfigure of Figure 3.6(b), it illustrates the case when the advantage value is positive, signifying that the selected action is favourable and we aim to increase the probability of selecting this action (that is $p_\theta(a_t|s_t)$) as much as possible. However, we also want to ensure that θ does not deviate significantly from θ', hence the maximum change of the new policy output under parameter θ is constrained by $p_{\theta'}(a_t|s_t) \cdot (1+\epsilon)$. Similarly, in the case of a negative reward value, $p_\theta(a_t|s_t)$ is constrained by the minimum value it can attain, which is $p_{\theta'}(a_t|s_t) \cdot (1-\epsilon)$.

By employing the clipping technique, the policy parameter θ is constrained to prevent excessive changes, thereby enabling the reuse of data samples collected using the previous policy parameter θ, consequently alleviating the computational burden associated with data acquisition in each iteration of the optimization process.

In addition to the clipping technique, alternative methods, such as utilizing KL divergence, have been proposed to control the variations in the policy parameter, for example, by using the KL divergence θ (Schulman et al., 2017). However, in our implementation, we specifically emphasize the adoption of the clipping method due to its straightforward implementation while still achieving comparable performance to the KL divergence approach.

3.2.4 Summary

By utilizing importance sampling and clipping methods, the PPO algorithm becomes a versatile deep learning algorithm and makes it a powerful approach to train a proficient AI player to win the game in the fighting game setting. This provides the basis for the successful implementation of our proposed DDA mechanism.

3.3 Monte Carlo Tree Search (MCTS) Algorithm

The Monte Carlo method is a well-known approach that involves utilizing repeated random sampling to estimate the outcome of complex tasks. It has been widely recognized that the Monte Carlo method is particularly effective when obtaining the outcome of a task through other approaches is challenging. This method has been successfully employed in various games, such as POKER and SCRABBLE, as evidenced by previous research findings (Chaslot et al., 2008).

The MCTS algorithm is an extension of the Monte Carlo method and was first introduced by Coulom (2006) in 2006 within the context of a Go program. The notable achievements of MCTS in playing Go have captured significant attention in the research community (Browne et al., 2012). This attention is attributed to the inherent complexity of Go, characterized by an immense number of possible moves and a high branching factor,

which poses challenges for computer programs to make optimal decisions within limited time constraints. Previous attempts to utilize deterministic search algorithms, such as the Minimax algorithm, in Go-playing programs were met with limited success due to the game's overwhelming branching factor and the inherent difficulties in implementing long-term planning strategies.

3.3.1 Four Stages in MCTS Algorithm

The MCTS algorithm follows a repetitive process consisting of four key steps, which are executed iteratively as long as there are computational resources or time available. This iterative process involves the simulation of numerous random games and subsequent analysis of the outcomes to derive an effective strategy. While a single simulation may not provide an optimal result, the accumulation of hundreds of stochastic simulations allows the MCTS to learn and converge towards a relatively optimal strategy (Chaslot et al., 2008). Therefore, after a significant number of simulations, we can anticipate the MCTS algorithm to approach a strategy that demonstrates favorable performance.

An illustration of a single iteration of the MCTS process can be found in Figure 3.7.

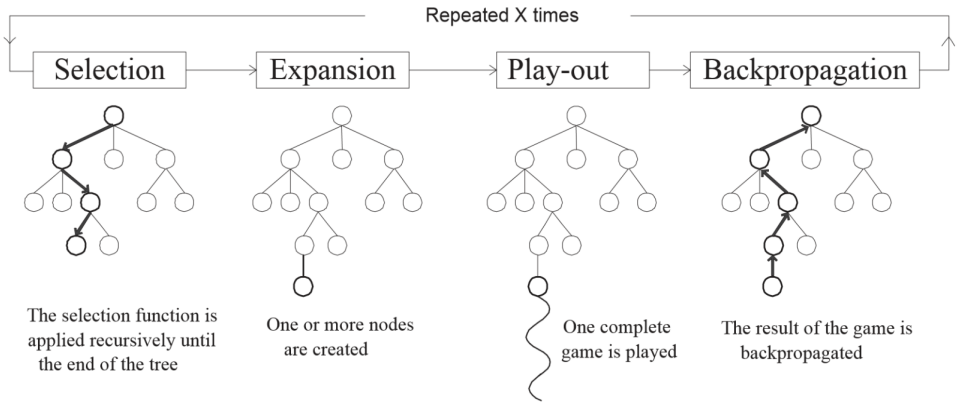


Figure 3.7: The four phases of the MCTS algorithm (Mańdziuk, 2018).

Selection

The Selection process initiates from the root node, representing the current game state. It employs a specific node selection strategy to systematically explore child nodes until a leaf node is reached. The node selection strategy must carefully balance the trade-off between exploitation and exploration. Exploitation involves selecting the most promising node based on current knowledge, while exploration entails selecting unvisited nodes, even if the estimated value of the unvisited node is lower than the current best node. The exploration is essential as node values are estimated through random simulations, and solely focusing on previously explored nodes may overlook potentially more valuable

paths (Chaslot et al., 2008). And choosing the node with the highest estimated value may result in repetitive exploration of the same path, limiting updates to that specific path (Demediuk et al., 2019). Striking an appropriate balance between exploitation and exploration is crucial to fully leverage the potential of the MCTS algorithm. One widely used node selection strategy is the Upper Confidence Bound applied to Trees (UCT) (Kocsis et al., 2006), which we will also employ in our proposed approach.

Expansion

Upon reaching a leaf node during the MCTS simulation process, the node can be expanded if certain requirements are met. The specific requirements may vary depending on the implementation details. For instance, a common requirement is to have a minimum number of visits to the node before expansion is allowed. This condition ensures that the node has been sufficiently explored before further expanding the search tree, thereby preventing premature expansion of nodes that have not yet received adequate attention.

Playout

The playout phase (also called the simulation phase) holds significant importance within the MCTS algorithm. Following the selection of a node for expansion, a playout policy is employed to simulate the game through self-play. In this phase, the game is generally played out until it reaches either the end of the game or a predefined termination condition. The resulting end game state serves as a basis for evaluating the value and goodness of the selected node.

The design of the playout policy in the MCTS algorithm involves trade-offs. Even though we have mentioned one of the advantages of MCTS is that it does not require specific domain knowledge, sometimes, it can be beneficial to incorporate the domain knowledge to advance the MCTS. While combining domain-specific knowledge or heuristics can potentially enhance the strength and reliability of the simulation result, it is still important to take into consideration of the execution speed of the algorithm after introducing extra heuristics. If the added domain-knowledge or heuristics significantly increases the computational cost of the playout strategy, it may negatively affect the overall performance of the simulation phase. This is because the number of simulations that can be executed per unit of time may be reduced, resulting in less efficient exploration and search. Thus, we need to balance the use of domain-specific knowledge or heuristics and the computational cost of the playout strategy to ensure the optimal performance of the MCTS algorithm.

Backpropagation

After the simulation phase, the simulated result is evaluated using an evaluation function that is specific to the game type. For example, in a chess game, the simulation result can be expressed as 1 for a win, 0 for a draw, and -1 for a loss.This evaluated value

is then backpropagated from the leaf node all the way up to the root node of the tree. The processes of selection, expansion, simulation, and backpropagation constitute one iteration of the MCTS algorithm. The algorithm continues to run iterations as many times as possible, continuously updating the node values based on the results of simulations, until a termination condition is met or a maximum number of iterations has been reached.

Once the MCTS process is completed, a game tree is obtained, where each node represents a possible action. The subsequent step involves selecting an action for the agent to play, and various strategies can be employed. One strategy is to choose the node with the highest estimated value, indicating the maximum expected reward. Another strategy is to select the node that has been visited most frequently. The rationale behind this approach is that a higher visit count implies a more thorough exploration of the corresponding action. Alternatively, a mixed strategy can be employed, combining both the estimated value and the visit count of a node in the final node selection Coulom (2006) .

3.3.2 Summary

There are multiple justifications for selecting the MCTS algorithm as the foundational component of our DDA approach. Firstly, the MCTS algorithm can be characterized as **"aheuristic"**, indicating that it does not necessitate explicit tactical domain knowledge. Consequently, the MCTS algorithm can be readily and efficiently applied to different domains with minimal or no modifications. According to Browne et al. (2012), while the full-depth minimax game tree is optimal and desirable from a game theoretic standpoint, the effectiveness of a limited-depth minimax tree largely relies on the quality of employed heuristics. Notably, in the game of Chess, researchers have developed numerous dependable heuristics over several decades, enabling the minimax search algorithm to leverage this domain-specific knowledge and perform admirably. However, in more complex games like Go, which exhibit significantly larger branching factors and pose challenges in formulating effective heuristics, the performance of the minimax search algorithm deteriorates notably

Secondly, the MCTS approach is considered an **"anytime"** method, permitting the algorithm to halt at any point during execution and provide the current optimal value. This capability stems from promptly propagating simulation outcomes back to the root node after each iteration, ensuring continuous updates to the entire game tree. Such adaptability renders the MCTS algorithm well-suited for real-time games, where time constraints necessitate timely decision-making.

Furthermore, the MCTS algorithm generates an **"asymmetrical"** game tree, enabling focused exploration of promising nodes while still affording other nodes the opportunity for exploration. Figure 3.8 demonstrates this asymmetry, with certain branches of the tree exhibiting greater depth, indicative of more frequent node visits. This characteristic holds significance in DDA as it enables the exploitation of specific optimal actions while

simultaneously preserving the ability to explore alternative optimal actions. By doing so, the algorithm can identify the action that best suits the current difficulty level.

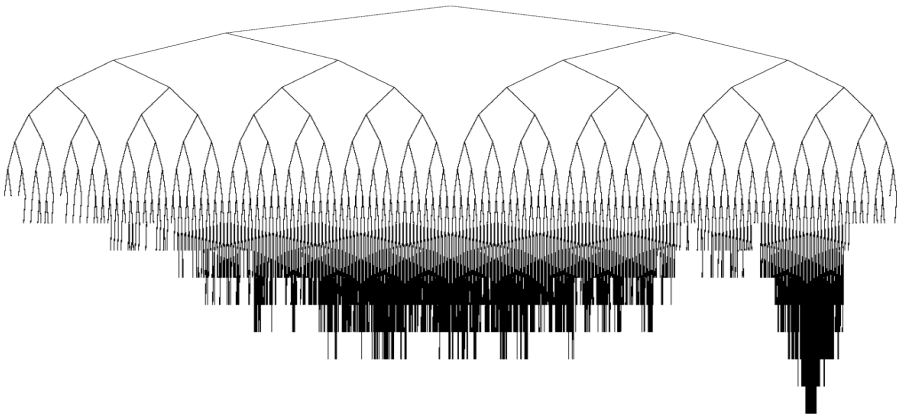


Figure 3.8: An asymmetry generated after 4000 iterations(Coquelin and Munos, 2007).

3.4 Our Proposed DDA Mechanism

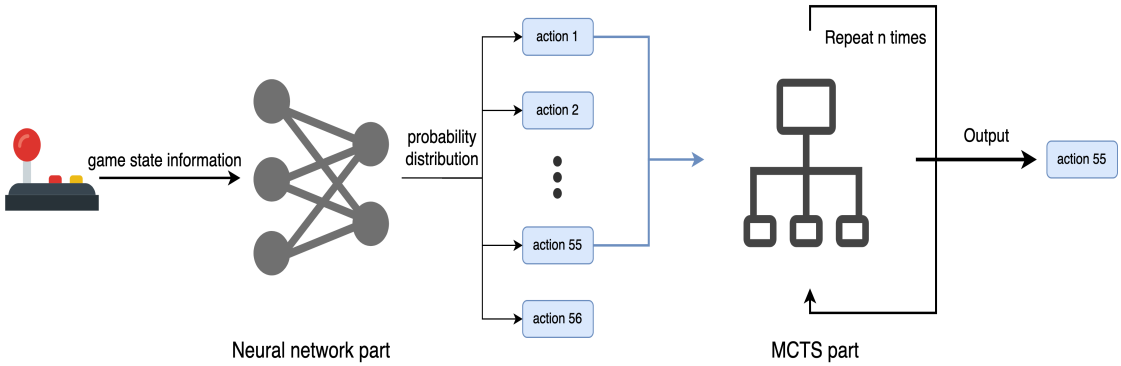


Figure 3.9: The structure of the MCTS-based DDA we proposed

Our proposed DDA approach consists of two primary components (as shown in Figure 3.9): a policy network trained using PPO and the MCTS module. The policy neural network receives the current game state information as input and generates a probability distribution over all possible actions in the action space. Actions with higher probabilities are considered more favorable or advantageous according to the learned policy.

The MCTS component plays a crucial role in implementing the DDA mechanism. Once the action probabilities are obtained from the policy neural network, a subset of actions is sampled and passed to the MCTS algorithm, which effectively reduces the number of actions the MCTS algorithm needs to search. The MCTS algorithm proceeds by iteratively executing the MCTS process among the selected actions multiple times to determine which action suits the current game difficulty level best.

The policy neural network employed in our proposed DDA approach plays a crucial role in guiding the action selection process that subsequently feeds into the MCTS algorithm. Through the utilization of a well-trained neural network, we are able to obtain a probability distribution over all available actions, where the action with the highest probability is deemed the most suitable for the current game state. However, in the context of DDA, our focus lies not in selecting the strongest action, but rather in identifying actions that align with the opponents' skill level. Here, the term "strongest" refers to actions aimed at swiftly defeating the opponent. Concurrently, the output probabilities generated by the neural network can be interpreted as the believability associated with selecting each action within the current game environment. Actions with higher probabilities are regarded as more plausible and less likely to be abnormal behaviour. By eschewing actions with exceedingly low probabilities, the agent mitigates the risk of engaging in abnormal actions, thus enhancing its overall reliability.

Conceiving that the output probabilities serve as an indication of action believability, we can now proceed to rank the actions based on the magnitude of their probabilities.

Rather than inputting all possible actions into the MCTS component, we opt to select the top-k actions from the ranked list and feed them into the MCTS algorithm for further exploration. This approach effectively diminishes the search space of the MCTS algorithm from 56 actions to only k actions. As the number of actions to be explored by MCTS decreases, the likelihood of comprehensively examining each action increases, leading to more reliable outcomes from the MCTS search process. It is noteworthy that this strategy for reducing the search space is solely guided by the outputs of the neural network, without necessitating any supplementary domain-specific knowledge. Instead, it capitalizes on the in-game information obtained from each frame. This advantage highlights the potential transferability of our proposed approach to diverse games. By harnessing the guidance provided by the policy neural network, we anticipate achieving two objectives: mitigating the real-time constraints imposed by reducing the action space explored by MCTS and minimizing abnormal behaviors through the selection of actions with higher output probabilities.

In order to further enhance the effectiveness of our approach, we have incorporated simple heuristics into the decision-making process. Our approach involves categorizing the available actions into two distinct groups: offensive actions and defensive actions. Offensive actions are designed to inflict higher damage on the opponent, while defensive actions prioritize a more conservative approach. The defensive action group consists of 27 actions, while the offensive action group contains a slightly larger set of 32 actions.

Subsequently, after obtaining the probability distribution of actions from the neural network, we select the top-k actions (in our experimental setup, we set $k = 18$) from either the offensive or defensive action group. The selection is based on the current disparity in HP between the DDA agent and the opponent. Specifically, we opt for actions from the offensive action group only when the opponent's HP value surpasses that of the DDA agent, and the HP difference is equal to or greater than 10% of the total HP value (which is 40 in our design). This strategic decision ensures that when the DDA agent finds itself in a disadvantaged state, it exhibits more aggressive behavior to diminish the HP difference and maintain an appropriate level of game difficulty. Conversely, if the HP difference falls within the range of 10% of the maximum HP, indicating an appropriate game difficulty level, actions will be selected from the defensive group to maintain the current game situation.

In our proposed DDA mechanism, we utilized the HP difference between the DDA-controlled player and the opponent player as the indicator of the game's difficulty. Although simply using the HP difference may not fully represent the game difficulty, it is an easy to obtain and straightforward indicator in the fighting game. And in the context of fighting games, many existing DDA mechanisms also employ HP difference as a reliable indicator of game difficulty. There are some other indicators that can also be adapted, such as the remaining time of the game, depending on the game types and requirements.

3.4.1 Neural Network Components in Proposed Approach

In this section, we will discuss the implementation details of the policy neural network part. We will first present the neural network structure of our proposed approach and we will then show the training details of using the PPO algorithm to train our policy neural network.

Neural Network Structure

Layer (type)	Output Shape	Param #
dense (Dense)	(None,800)	115200
leaky_relu (LeakyReLU)	(None,800)	0
dense_1 (Dense)	(None,600)	480600
leaky_relu_1 (LeakyReLU)	(None,600)	0
dense_2 (Dense)	(None,400)	240400
leaky_relu_2 (LeakyReLU)	(None,400)	0
dense_3 (Dense)	(None,200)	80200
leaky_relu_3 (LeakyReLU)	(None,200)	0
dense_4 (Dense)	(None,100)	20100
leaky_relu_4 (LeakyReLU)	(None,100)	0
Total params: 936,500		
Trainable params: 936,500		
Non-trainable params: 0		

Table 3.1: Summary of the neural network structure

The summarized neural network architecture is presented in Table 3.1. In order to account for the real-time nature of the DDA agent, and to minimize processing time, we deliberately designed the neural network to be small in size. This consideration is crucial as the DDA agent operates within a time-constrained environment.

In order for the PPO to work, we have incorporated two additional heads at the end of the neural network structure, as shown in Table 3.2 and Table 3.3. The policy head is designed to output 56 values, with each value representing the probability of selecting a specific action. And the value head generates a single value that signifies the expected reward for a particular state, based on the current policy parameter.

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None,56)	5656

Table 3.2: Policy head

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None,1)	101

Table 3.3: Reward head

Input Features

The input to the neural network is a vector of size 143. The vector contains in-game features that are deemed essential for making accurate predictions. A comprehensive overview of the specific features included in the input vector can be found in Table 3.4.

Feature types	Count
HP	2
energy	2
x, y positions	4
movement directions of x, y	4
absolute speed of x, y	4
current action (one-hot encoded)	112
remaining frame numbers	2
time passed since the game started	1
projectile attack features - projectile damage	4
projectile attack features - projectile x position	4
projectile attack features - projectile y position	4
Total	**143**

Table 3.4: Description of the input features of the neural network, each feature is logged in the order of the DDA agent first, and then the opponent.

Training Data Collection

Rather than training the neural network from scratch, we adopted the neural network parameters from an AI player "BlackMamba" which has demonstrated outstanding performance in the 2020 FightingICE competition and achieved the highest winning rate among all participating AIs. Building upon this pre-trained AI model, we then employed the PPO algorithm for further training. This involved pitting the AI against other AIs with high winning rates from recent years' FightingICE competitions, and iteratively refining its performance through self-play. Through these two processes, our aim was to obtain a more robust and powerful neural network that would outperform other AIs presented in the FightingICE framework.

To implement the PPO algorithm for training the neural network, we logged the following information for each action taken by the agent:

- **state**: Records what the current game state is, including relevant game features and variables that will later be input to the neural network for action prediction.

- **action**: Denotes the action that has been selected from the probability distribution output by the neural network, based on the provided state information.
- **reward**: Represents the immediate reward associated with taking a particular action in the current game state. In our specific game setting, the reward is calculated based on factors including the difference in HP between players and the distance between them. The formula used to compute the reward is defined as follows:

$$\begin{aligned} reward = &((BeforeHP^{opp} - AfterHP^{opp}) - (BeforeHP^{my} - AfterHP^{my}))/10 \\ &+ (0.01\ if\ (distance^{before}\ >\ distance^{after})\ else\ (-0.01)) \end{aligned}$$

  In our approach, the discrepancy in HP serves as the primary indicator of the action's efficacy, complemented by an additional bonus value represented by the second term in the reward function. The variable *distance* denotes the x-distance between the two players in the game. The inclusion of this bonus term reflects a deliberate preference for actions that minimize the distance between players. This design choice is particularly relevant in the context of a fighting game, as it serves to mitigate situations where two players are being excessively distant from each other, so as to further reduce abnormal behaviours.
- **next_state**: Captures the state information that arises subsequent to the agent's execution of the currently selected action.
- **action_probability**: Records the output of the policy head, which represents the probability of selecting a specific action under the current model parameters.
- **done_flag**: Indicates whether the current game round has concluded or not. It is assigned a value of 1 if the game is ongoing, and a value of 0 if the game has terminated.

These aforementioned features collectively constitute a single training data pair, and consecutive data pairs are recorded over a continuous period of time. Subsequently, these data pairs are utilized to update the model parameters.

Training the Policy Neural Network

The PPO algorithm aims to maximize the following objective function:

$$\begin{aligned} J^{\theta'}(\theta) &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] \\ &= \sum_{s_t,a_t} \min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t),\quad clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)A^{\theta'}(s_t,a_t)\right) \end{aligned}$$

To maximize $J^{\theta'}(\theta)$, it is equivalent to minimizing $-J^{\theta'}(\theta)$. In the equation above, the advantage function $A^{\theta'}(s_t, a_t)$ is utilized to estimate the relative effectiveness of

taking action a_t at state s_t compared to the average action. It is crucial to differentiate the output of the advantage function from the output of the value head. The value head outputs the expected reward received at the current state, while the advantage function evaluates how well the received reward is when undertaking a specific action compared with the average reward received at this state, and it also takes the future rewards received after performing a specific action into consideration, so the output of the advantage function can be a good indicator of the goodness of the action being taken. The advantage function $A^{\theta'}(s_t, a_t)$ is calculated as the following process:

we will first calculate the *td_target* variable by using the following formula:

$$td\_target(s_t) = reward + \gamma \cdot value(s_{t+1}) \cdot done\_flag$$

where the *reward* variable represents the actual immediate reward obtained by taking a specific action a_t, while $value(s_{t+1})$ denotes the expected reward received at state s_{t+1}. The $td\_target(s_t)$ represents the actual expected reward after performing the specific action a_t. It is calculated by adding the actual immediate action reward to the expected reward of the next state. The discount factor γ is used to reduce the importance of future rewards relative to the immediate reward. If the game round has ended, the $done\_flag$ will be set to 0, and the equation above will simply return *reward* as the output, as no future actions are available.

The subsequent step involves the comparison between the disparity of $value(s_t)$ and $td\_target(s_t)$. This disparity serves as an indication of the disparity between the anticipated reward estimated by the value head and the target value that should be produced by the value head. It signifies whether the execution of a particular action will result in a reduction of received rewards. In the event that the disparity is positive, indicating that the action yields greater rewards compared to the average rewards received, the policy network should adapt its parameters to enhance the probability of selecting this action when encountering the same state. Conversely, if the disparity is negative, the policy parameters are updated to diminish the likelihood of choosing this action when encountering similar states. This disparity is denoted as td_error:

$$td\_error(s_t) = td\_target(s_t) - value(s_t)$$

Now we can compute the long-term advantage function $A^{\theta'}(s_t, a_t)$ by utilizing the $td\_error(s_t)$. While $td\_error(s_t)$ solely considers the immediate rewards obtained from the action, $A^{\theta'}(s_t, a_t)$ takes into consideration the long-term effects of executing the action, incorporating a discount factor γ. The formula for calculating $A^{\theta'}(s_t, a_t)$ is as follows:

$$\begin{aligned} A^{\theta'}(s_t, a_t) =& td\_error(s_t) + (\gamma\lambda) * td\_error(s_{t+1}) \\ &+ (\gamma\lambda)^2 * td\_error(s_{t+2}) + \cdots + (\gamma\lambda)^{T-t} * td\_error(s_T) \end{aligned}$$

Note that both γ and λ are hyperparameters, but they play different roles in the PPO algorithm. The γ parameter is used to determine the importance of future rewards. Typically, we consider that the farther away a state is from the current state, the less contribution it should have to the current state's value estimation. The λ parameter is used to balance the trade-off between bias and variance in the estimation of advantage values. When λ approaches 0, the advantage value of the current state is largely determined by the immediate reward of the current state, with little consideration of future rewards. This results in a smaller bias but a larger variance in the advantage estimation, as the estimate is close to the actual reward. However, when λ approaches 1, the advantage estimation is more likely to deviate from the actual reward, but with a smaller variance (Schulman et al., 2015). Here in our training process, we set both λ and γ to be 0.95.

During the training process, the initial step is to load the latest model for our agent. Subsequently, an AI is randomly selected from the AI pools, along with the previously trained model, to serve as components for our AI. And we set the trajectory length to be 30, meaning that a trajectory containing 30 input state-action pairs is collected, and this trajectory is then utilized to train our model. Note that the data in the trajectory is reused three times, effectively mitigating the need for re-sampling new data.

Training the Critic Neural Network

Training the critic network is simpler, and the parameters of the critic network are updated by calculating the mean squared error between the expected rewards outputted by the critic network and the target rewards.

For a detailed description of the training process, please refer to Algorithm 1.

Algorithm 1: Pseudocode for training the network by using PPO algorithm

Require: a trajectory that includes 30 input pairs, and each pair contains (state, action, reward, next_state, action_probability, done_flag). We use s to denote a batch of states, which is the combination of all states in the trajectory, similarly for a, r, $next\_s$, a_prob as well as $done\_flag$.

$epoch \leftarrow 3$
CommentInput the trajectory to the network 3 times
$\gamma, \lambda \leftarrow 0.95$
for i `in` $epoch$ **do**
    $td\_target = r + \gamma * value(next\_s) * done\_flag$
    $td\_error = td\_target - value(s)$
    $advantage\_ls \leftarrow [\,]$ ▷ Advantage list used to store the advantage of each state
    $advantage \leftarrow 0$
    for $error$ `in` $td\_error.reverse()$ **do** ▷ Calculate the advantage in a reverse order
        $advantage = error + \gamma\lambda * advantage$
        $advantage\_ls.add(advantage)$
    end for
    $advantage\_ls.reverse()$ ▷ Reverse the advantage list back
    $a\_prob\_new \leftarrow policy(s)$ ▷ Get the probability of selecting the action under the new policy parameter
    $ratio = \dfrac{a\_prob\_new}{a\_prob}$ ▷ Ratio for important sampling
    $surr1 = ratio * advantage$ ▷ First term in the $J^{\theta'}(\theta)$ function
    $surr2 = clip(ratio, 1-\epsilon, 1+\epsilon) * advantage$ ▷ Second term in the $J^{\theta'}(\theta)$ function
    $policy\_loss = -\min(surr1, surr2)$ ▷ loss value for the policy head
    $value\_loss = loss(value(s), td\_target)$ ▷ loss value for the value head
    $loss = policy\_loss + value\_loss$
    calculate the gradient and update model parameters
end for

Training Results of the Policy Network

By leveraging the pre-trained model, we are able to greatly reduce the training time required. Furthermore, we have improved the performance of the pre-trained model by incorporating the pre-trained model into competitions against advanced AIs that have showcased exceptional performance in recent years' competitions. To further enhance the training process, we adopt a self-training approach by randomly selecting previously trained models as opponents. We train both networks for 25000 epochs using the collected trajectories. These trajectories are utilized as input for training the neural network and are reused three times, thereby reducing the necessity for repetitive data sampling throughout the training phase.

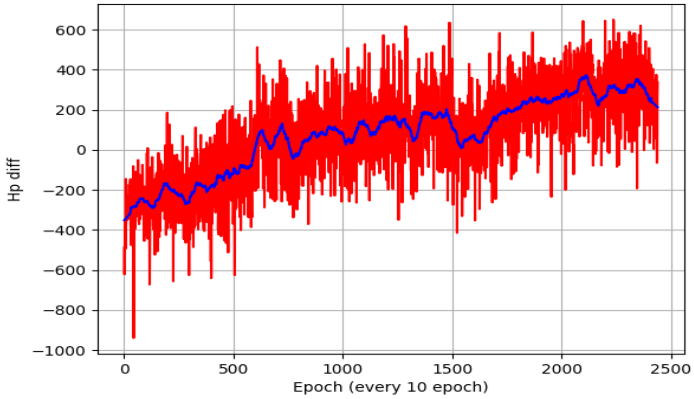


Figure 3.10: The HP difference between DDA-agent and opponent for every 10 epoch

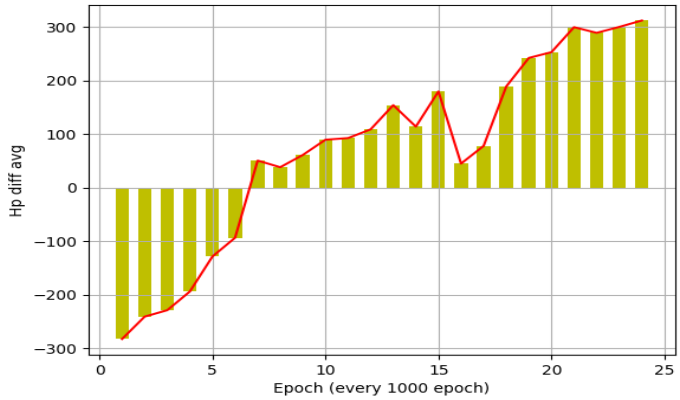


Figure 3.11: The average HP difference between DDA-agent and opponent for every 1000 epoch

Figure 3.10 and Figure 3.11 provide visualizations of the changes in HP differences during

the training process. It is evident from the figures that initially, our model experienced significant losses against the opponent, with a large negative margin in HP difference. However, as the training progressed, the absolute HP difference gradually decreased, and after approximately 7000 epochs, the absolute average HP difference approached zero. And in the subsequent training process, our AI continued to develop robust strategies and become more competitive, leading to a transition of the HP difference from negative values to positive values.

A consistent pattern is observed in Figure 3.12 and Figure 3.13, indicating that our AI successfully acquired a robust policy. Notably, our AI maintained a high winning ratio of approximately 0.95 after completing 20000 epochs, indicating our network acquired effective strategies, and has the capability to outperform in most of the competitions, which provides a solid foundation for the effective implementation of the MCTS component in our approach.

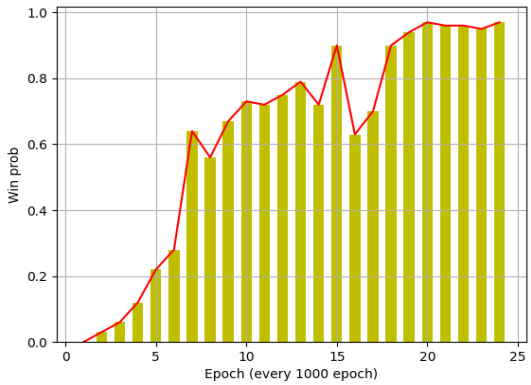


Figure 3.12: How the winning probability changed throughout the training process

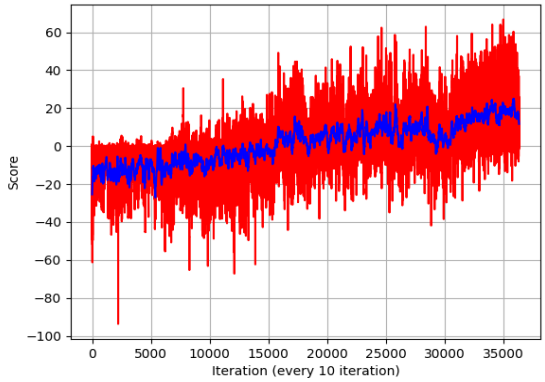


Figure 3.13: The cumulative score for each iteration

In order to minimize time overhead and improve computation speed, we made certain design decisions in our implementation. Although the FightingICE framework supports both Java and Python, and Python programs can dynamically access Java objects in the Java Virtual Machine (JVM), and Java programs can also callback Python objects as well, we chose to implement the MCTS part directly on the Java side due to the framework being predominantly Java-based. This avoids the additional time expense incurred when calling Java methods from Python. Thus, we exported the neural network parameters from Python to Java and implemented the MCTS part directly on the Java side. Moreover, to enhance the computation speed of the neural network, where numerous matrix multiplications are involved, we adopted the Efficient Java Matrix Library (EJML). As shown in Table 3.5, the comparison of computation speeds for using different approaches to make predictions reveals that leveraging EJML results in significant speed improvements in matrix computations. This optimization enables us to allocate more computing time to the subsequent MCTS process, so as to enable the MCTS algorithm to search more nodes.

Model	Time (per prediction)
Python neural network	2.5ms
Java matrix multiplication	1.9ms
Java matrix multiplication with EJML	**0.373ms**

Table 3.5: Comparison of the predication speed of different models

3.4.2 MCTS Component

Having obtained a formidable neural network that consistently outperforms opponents, the next step is to integrate it with the MCTS algorithm to achieve DDA.

Once inputting the current game state into the neural network, the network will output a probability distribution across the entire action space of 56 actions. To alleviate the computational burden of the MCTS algorithm, we selected the top 18 actions with the highest probabilities as inputs for MCTS. By doing this, we can prioritize actions with higher probabilities and focus on exploring these promising actions while disregarding actions that are less likely to be performed.

Hyperparameters of MCTS

Once the actions have been selected, we proceed to construct the Monte Carlo game search tree, with the root node representing the current game state. Subsequently, the MCTS process is applied iteratively to refine the search tree. The MCTS algorithm continues until the time constraint is reached. The specific hyperparameters employed in the algorithm are outlined in Table 3.6.

The hyperparameter UCT_TIME represents the time limit allotted for the MCTS process. Considering that each frame of the FightingICE game corresponds to 16.75ms and

Hyperparameters	Value
UCT_TIME	15.75ms
UCT_C	3
UCT_MAX_TREE_DEPTH	3
UCT_CREATE_NODE_THRESHOLD	6
MAX_NUMBER_OF_CHILD	6

Table 3.6: Description of the hyperparameters used in the model

that each player is required to make a decision within each frame, we subtract the time required for the policy network to make predictions, leaving 15.75ms as the available running time for the MCTS process. The MCTS algorithm is then executed as frequently as possible within this time constraint to effectively explore the search space.

In our proposed approach, we employed the UCT node selection strategy to determine which node in the search tree is the most promising and should be selected for simulation. The selection criteria in UCT are determined by the following formula:

$$a^* = \arg\max_{a \in I} \left\{ \overline{v_a} + UCT_C \times \sqrt{\frac{ln\ n_p}{n_a}} \right\}$$

The UCT formula selects a child node that has the highest UCT value denoted as a^* from node p, where I represents the set of selectable child nodes, $\overline{v_a}$ represents the estimated average value for node a obtained from the simulation result, and n_p and n_a represent the number of visits to node p and child node a, respectively. The hyperparameter UCT_C is used to control the trade-off between exploration and exploitation. The term v_a can be regarded as the exploitation component, while the term inside the square root $\frac{\ln n_p}{n_a}$ can be regarded as the exploration bonus for node a. Notably, as the number of times node p is selected increases, the value of $\frac{\ln n_p}{n_a}$ also increases, highlighting the emphasis on exploration for less frequently visited child nodes. A larger value of hyperparameter UCT_C amplifies the weight given to exploration in the UCT formula. We set the hyperparameter UCT_C to be 3 indicates we place more emphasis on exploration The preference for exploration over exploitation in the MCTS algorithm is based on the idea that trying as many different actions as possible within the limited time for search can help obtain a better understanding of the diversity of strategies employed by the opponent. However, this preference for exploration may come at the expense of simulation accuracy, as it may reduce the number of times each node is visited, potentially sacrificing accuracy. This is because a more thorough exploration of a node through repeated simulations can lead to simulation results that are closer to the true result.

Therefore, to mitigate the adverse effects of insufficient exploration of individual nodes, we have introduced three additional parameters, namely $UCT\_MAX\_TREE\_DEPTH$, $UCT\_CREATE\_NODE\_THRESHOLD$ and $MAX\_NUMBER\_OF\_CHILD$. The $UCT\_MAX\_TREE\_DEPTH$ parameter sets the maximum depth of the tree, which

we have limited to 3 in our game setting. This choice favours a wide and shallow tree structure over a narrow and deep one. This is reasonable in the context of a fighting game, where game situations are unstable and non-deterministic, and any action can have a significant impact on subsequent game states. Hence, excessively focusing on actions that are far from the current situation may not be meaningful. The $UCT\_CREATE\_NODE\_THRESHOLD$ parameter restricts node expansion until a certain number of visits has been reached so as to avoid server under exploration. In our case, we have set the threshold to 6. Lastly, the $MAX\_NUMBER\_OF\_CHILD$ parameter limits the number of child nodes that each parent node can have. We have set this value to 6, meaning that instead of expanding all 18 selected nodes at once, we only randomly pick and expand 6 actions, effectively reducing the search space.

Puesdocode of MCTS

Whenever the DDA agent is required to make a decision regarding which action to execute, it utilizes the MCTS algorithm to identify the optimal action that is most suitable for the current game difficulty level (corresponding to line 5 of pseudocode 2). The process of UCT is then iteratively executed as many times as possible within the time constraint (corresponding to lines 6-11 of pseudocode 2).

Algorithm 2: Pseudocode for MCTS algorithm

Require: The output of the probability distribution over the 56 actions.

1: $myActions = select\_top\_k\_actions()$ ▷ select top 20 actions from the probability distribution.
2: $oppActions = set\_opp\_actions()$ ▷ select all actions that the opponent can perform.
3: $rootNode = create\_node()$ ▷ The depth of the root node is 0.
4: $rootNode.create\_node()$ ▷ Expand the root node.
5: $bestAction = rootNode.mcts()$ ▷ Perform the MCTS algorithm from the root node.
6: **function** MCTS( )
7: $start\_time = current\_system\_time$
8: **while** $current\_system\_time - start\_time \leq UCT\_TIME$ **do**
9: $uct()$ ▷ Perform the UCT process as many times as possible.
10: **end while**
11: **return** $highest\_score\_action$
12: **end function**
13:

```
14: function UCT( )
15:     selected_node = find_best_child_node()        ▷ Find the child node that has the highest score.
16:     if selected_node has not visited before then
17:         score = selected_node.play_out()          ▷ Do the simulation process on the selected node.
18:     else
19:         if selected_node does not have child nodes then
20:             if selected_node.depth < UCT_TREE_DEPTH then
21:                 if selected_node.games ≥ UCT_CREATE_NODE_THRESHOLD then
22:                     selected_node.create_node()
23:                     score = selected_node.uct()
24:                 else                              ▷ Node has not been fully visited.
25:                     score = selected_node.play_out()
26:                 end if
27:             else                                  ▷ Exceed the maximum tree depth limitation.
28:                 score = selected_node.play_out()
29:             end if
30:         else
31:             if selected_node.depth < UCT_TREE_DEPTH then
32:                 score = selected_node.uct()       ▷ As the selected node is not the leaf node, perform the UCT on this selected node.
33:             else                                  ▷ If the maximum depth reached
34:                 score = selected_node.play_out()
35:             end if
36:         end if
37:     end if
38:     selected_node.games+ = 1                      ▷ Update the number of visits
39:     selected_node.score+ = score                  ▷ Cumulate the score
40:     return score
41: end function
```

The UCT process starting from line 14 of the pseudocode, aims to identify the most valuable leaf node for conducting a single simulation. The process commences from the root node and iteratively selects the best child node from all available child nodes until a leaf child node is reached. If the selected leaf child node satisfies the two expanding conditions: (a) the depth of the current node is less than the *UCT_TREE_DEPTH* limitation and (b) the node has been visited more than the minimum limit threshold, that is greater than the *UCT_CREATE_NODE_THRESHOLD*., then the selected leaf node can be expanded. Subsequently, a random playout or so-called simulation

process is performed on one of the expanded nodes.

The simulation stage is performed after the best child node is selected, in which random simulations are performed on that node (as indicated by the *play_out*() function in the pseudocode). Unlike in board games where the simulation mechanism involves randomly making moves for both players until the game ends and a winner is determined, our fighting game environment has a different approach. We do not require the simulation to go all the way to the end, but instead, we focus on real-time and short-term rewards. Before each simulation, our DDA agent will select 5 actions from the 18 available actions, while for the opponent, as their strategy is unknown, we randomly select 5 viable actions for the simulation process. The score of the node is then evaluated based on the results of the simulation. The simulation score for the simulated action is calculated as follows:

$$score = -(|HP_{diff}| - HP_{interval})^{+}$$

where $(\cdot)^{+}$ is a ramp function that returns 0 if the inside value is negative, and behaves as an identity function if the inside value is positive. $|HP_{diff}|$ represents the absolute value of the HP difference between the two players, and $HP_{interval}$ defines an interval of the HP difference within which the HP difference will be ignored. In this setting, HP_{interval} is set to 15, meaning that actions that cause an HP difference within the range of $[-15, 15]$ will be assigned a score of 0. Actions that cause HP differences outside the defined interval of $[-15, 15]$ will be assigned a negative score, discouraging the DDA agent from preferring actions that cause large HP differences. This enables the DDA agent to prioritize actions that result in smaller HP differences, potentially leading to a more balanced decision-making process.

The final stage of the MCTS algorithm is the backpropagation stage. After obtaining the score of the leaf node as a result of the simulation process, the score is then propagated back all the way to the root node in the tree. Additionally, the number of visits for each node is incremented by 1. These operations correspond to lines 37-39 in the pseudocode.

The aforementioned MCTS process is executed repeatedly until the time limit is reached. And we will choose the node that has the highest evaluation score as the action to be performed by our DDA agent.

During the experiment, we observed that the MCTS process repeated 137.3 iterations within each time constraint on average. As depicted in Figure 3.14, an example tree was generated within a single time frame, and in this example, the MCTS process was repeated 140 times, as indicated by the label near the root node. The tree exhibited a depth of 3, with non-leaf nodes being expanded only after they were fully visited. Each parent node had 6 child nodes. For further detailed information on this tree, including the score value associated with each node, please refer to appendix 7.3.

It was observed that, even within the time constraint of 15.75ms, every leaf node at the depth of 3 in the generated tree still had a chance of being visited 2-3 times on average. This indicates that every child node had been visited multiple times, ensuring improved

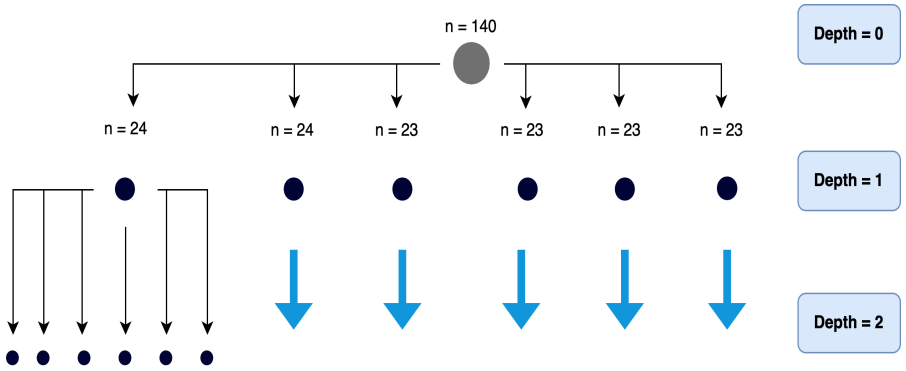


Figure 3.14: An example of the constructed tree within a single time frame

simulation accuracy. This is crucial because if some of the generated child nodes had not been visited, the simulation results for the corresponding parent node would not be accurate.

Chapter 4

Experiment and Evaluation

To assess the efficacy of our proposed DDA approach, we will commence by evaluating the proficiency of our policy neural network. This assessment aims to ascertain whether the policy network exhibits sufficient strength to establish a robust foundation for our proposed DDA framework. Subsequently, we will proceed to evaluate the effectiveness of our DDA approach by subjecting it to competitions involving AI players with different levels of competitiveness. Furthermore, we will conduct an examination of the action distributions manifested by the DDA-controlled AI, aiming to elucidate the manner in which these distributions vary across diverse gaming scenarios. This comprehensive evaluation serves to further appraise the efficacy of our proposed DDA mechanism.

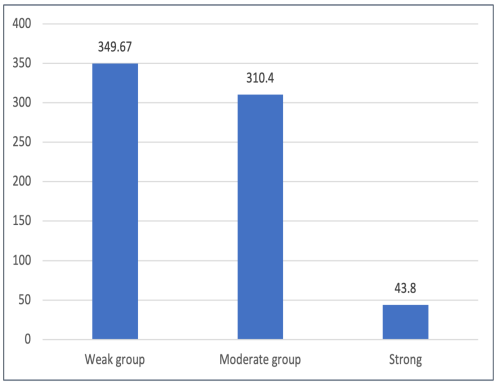
4.1 Evaluating the Neural Network Performance

In this section, we will initially evaluate the performance of the neural network without incorporating the MCTS component. To gauge the competitiveness of our designed AI, we leveraged the availability of multiple AIs with diverse competitive levels from previous FightingICE competitions. This enabled us to engage in matches against these AIs and assess the performance of our trained neural network policy.

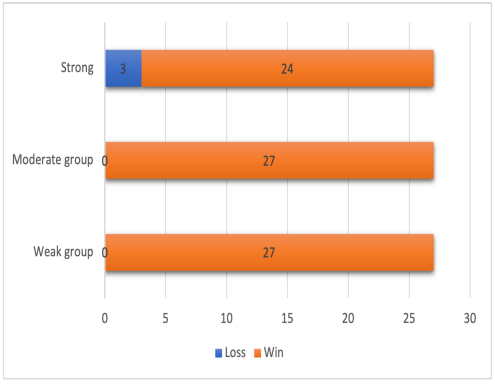
The existing AIs in the FightingICE competition can be roughly classified into three categories. The first category is the "Strong AI group", which comprises the top three AIs in the annual competition. These AIs are considered to have more advanced strategies and a higher probability of winning. The second category is the "Weak AI group", which is formed by the last three AIs in the competition. The last category is the "Moderate AI group", which consists of three AIs randomly selected from the remaining AIs excluding the top three and last three AIs. For our evaluation, we selected AIs that appeared in the competitions from 2019 to 2021, resulting in each group containing 9 AIs.

In our experiment, we set the maximum HP for both agents to be 400. We then had our

neural network-controlled AI agent play against AIs from each of the three categorized groups (Strong, Weak, and Moderate) three times, while recording the outcomes of each game in terms of wins and losses, as well as the HP difference at the end of each round. The results of the HP difference are presented in Figure 4.1(a). It is evident that our neural network-controlled agent consistently achieved a positive HP difference compared to the opponent AIs, particularly demonstrating significant HP advantages in battles against the Weak and Moderate AI groups. Furthermore, Figure 4.1(b) indicates that our neural network agent achieved a 100% winning rate when facing AIs from the Weak and Moderate AI groups, with a slightly lower winning rate when facing AIs from the Strong AI group. Based on these results, we can conclude that our neural network policy exhibits strong competitiveness, serving as a solid foundation for the subsequent MCTS process.



(a) The average HP difference of using the neural network agent to fight against AIs from the three groups



(b) Number of wins and losses

Figure 4.1: The performance of the neural network in comparison with other AIs

4.2 Evaluating the MCTS Performance on DDA

To evaluate the effectiveness of integrating the MCTS process into our neural network agent for adapting to different competence levels of game players, we selected one AI from each of the three categorized groups (Strong, Weak, and Moderate) as opponents for our DDA agent. We then had these AI opponents play against our DDA agent to assess the efficiency of our DDA approach.

4.2.1 Fighting against an AI from the "Easy AI Group"

We selected "SimpleAI" as our opponent AI from the Weak AI group, which was ranked 8th out of ten participating AIs in the 2018 FightingICE competition. We primarily used the HP difference between our DDA agent and "SimpleAI" as an indication of the efficiency of our DDA approach during the evaluation.

We first recorded how the HP difference between our DDA agent and "SimpleAI" changes when our agent only uses the strong network policy without including the DDA mechanism. The plot of the HP difference can be found in figure 4.2. The x-axis represents the number of frames that have passed since the beginning of the game, and the y-axis represents how the HP difference varies during the course of the game. The green shaded area represents the range of HP difference ($[-40, 40]$) that our DDA agent strives to maintain. Without surprise, the strong policy neural network agent quickly gained the upper hand in the game and took a significant lead in HP value over the opponent. The HP difference never decreased to a negative value, indicating that the strong policy agent was dominating the game. Moreover, the strong policy agent aimed to finish the game as fast as possible. From the figure, we can see that in the game against SimpleAI, the game ended with the strong agent winning at around 1100 frames.

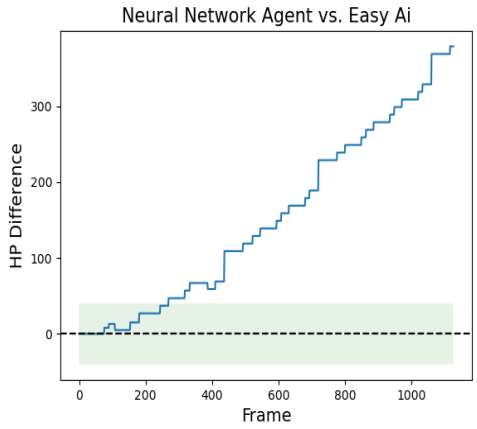
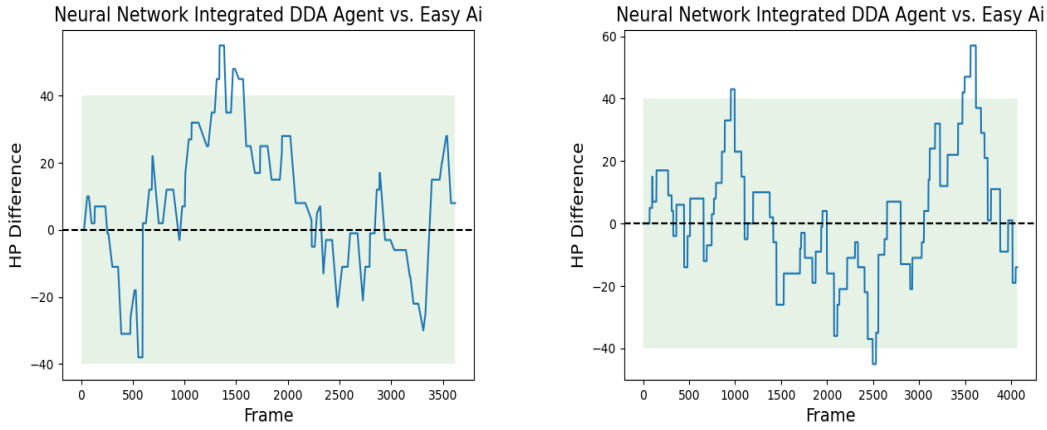


Figure 4.2: The plot shows how the HP difference changes when having a combating between the purely strong policy network-based agent with the "SimpleAI"

Now, let us examine the impact of incorporating the DDA mechanism into the strong policy neural network agent on its performance. We conducted multiple matches between the two agents and obtained two types of results, as illustrated in Figure 4.3.

Both of the subfigures in Figure 4.3 demonstrate similar trends, wherein the HP difference fluctuates around the x-axis. Similarly, the shaded green area in the figures represents the targeted range of HP difference for our DDA agent. Notably, upon incorporating the DDA mechanism, the absolute maximum HP differences were considerably smaller compared to the match without including the DDA. Specifically, the maximum HP differences ranged around 55 and did not exceed the threshold of -40 to 40 by a significant margin.

It is noteworthy that our agent, even with the incorporation of the DDA mechanism, still managed to maintain the crucial elements of tension, challenge, and excitement in a fighting game. This is evident from the fact that the HP difference does not consistently remain either greater than or smaller than 0, but rather constantly oscillates between positive and negative values, as depicted in the right subplot of the figure. Such fluctuation patterns can better stimulate players' desire to win the game compared to



(a) The HP difference between the DDA agent and the "SimpleAI", the DDA agent won at the end of the game

(b) The HP difference between the DDA agent and the "SimpleAI", the DDA agent lose at the end of the game

Figure 4.3: The combating between the DDA agent and the "SimpleAI"

patterns where the HP difference consistently hovers slightly above or below 0. Thus, we can conclude that our DDA agent is able to provide players with an enhanced gaming experience.

The number of frames elapsed serves as an additional valuable evaluation criterion for assessing the effectiveness of our DDA approach. As observed previously, in the absence of the DDA mechanism, the game concluded quite swiftly within approximately 1100 frames. However, after incorporating the DDA mechanism, we can discern a notable increase in game duration, with 3500 frames and 4000 frames respectively, indicating a considerable extension in gameplay duration.

Furthermore, the two subplots also highlight the presence of randomness in our algorithm, as the game outcome is not deterministic, and both agents have chances of winning or losing. This inherent randomness is crucial in maintaining the enjoyable aspect of the game, as it allows players to engage in gameplay without being aware of the existence of the DDA mechanism, while still experiencing an appropriate level of game difficulty tailored to their skill level. As stated in the research conducted by Ikeda and Viennot (2013), their objective was to design an AI system that can provide an entertaining gaming experience for human players in the game of Go. Their findings suggest that the optimal gaming experience for players occurs when the AI either narrowly loses to the player towards the end of the game or secures a victory due to the player's mistakes. These two situations are exactly the cases we obtained in our DDA mechanism as depicted in Figure 4.3.

4.2.2 Fighting against an AI from the "Moderate AI Group"

The subsequent step involves evaluating the performance of our agent against the AI from the "Moderate AI Group". For this purpose, we selected the "SpringAI" as the representative from the "Moderate AI Group", which achieved a ranking of 6th out of 15 participating AIs in the 2020 FightingICE competition. Similarly, we will initially examine how the HP difference would change when the DDA mechanism is not incorporated in our agent. The flow of the HP difference is illustrated in Figure 4.4.

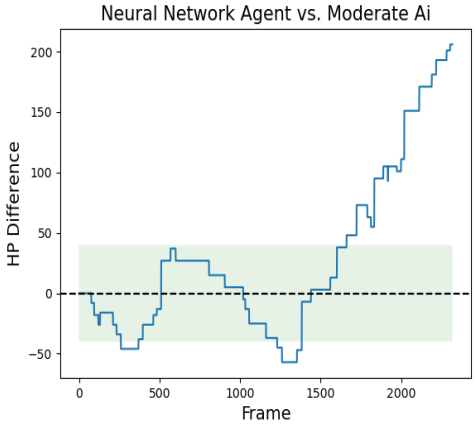
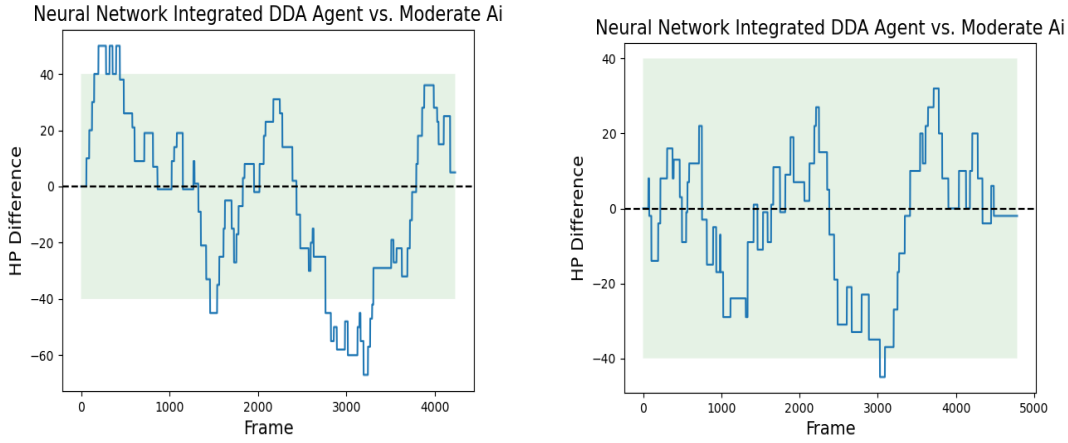


Figure 4.4: The plot shows how the HP difference changes when having a combating between the purely strong policy network-based agent with the "SpringAi"

As depicted in Figure 4.4, although the HP difference between our agent and the opponent exhibited fluctuations in the early stages of the game, our agent gradually gained an advantage in terms of HP as the game progressed, eventually winning with a significant HP advantage over the opponent. Notably, When we raised the competitiveness level of the opponent by selecting "SpringAI" from the "Moderate AI Group", the game duration also extended accordingly, from approximately 1100 frames without utilizing the DDA mechanism when competing with the AI from the easy group to around 2300 frames. This implies that as the opponent's ability increases, it takes longer for our AI to achieve victory. However, as evident from the results, despite the increased challenge, our opponent still faces a formidable task in defeating our agent without the assistance of DDA.

After introducing the DDA mechanism, the flow of the HP difference can be found in figure 4.5, and it can be observed that after introducing the DDA mechanism, our DDA agent was able to maintain the HP difference within the range of -40 to 40 as well, with occasional instances of deviation that were promptly corrected. In subfigure 4.5(a), our DDA agent won the game with a slight advantage in the end, while in subfigure 4.5(b), the opponent AI won the game with a slight advantage in the end.

The game duration also be extended when the DDA is included in our agent, When competing against a simple AI opponent, the implementation of DDA resulted in game durations of 3500 and 4000 frames, respectively. Similarly, when facing an AI opponent of moderate difficulty, the utilization of DDA successfully extended the game duration



(a) The HP difference between the DDA agent and the "SpringAi", the DDA agent won at the end of the game

(b) The HP difference between the DDA agent and the "SpringAi", the DDA agent lose at the end of the game

Figure 4.5: The combating between the DDA agent and the "SpringAi"

from 2300 frames to 4300 and 4800 frames, respectively.

4.2.3 Fighting against an AI from the "Strong AI Group"

Lastly, we will assess the performance of our DDA agent when pitted against the "Strong AI Group". As our representative for the strong AI, we have chosen the "TeraThunder" AI from the 2020 FightingICE competition, which achieved a notable 3rd rank among the 15 participating AIs in that year's competition.

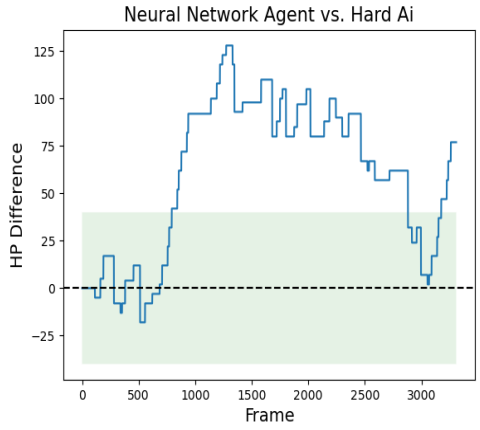
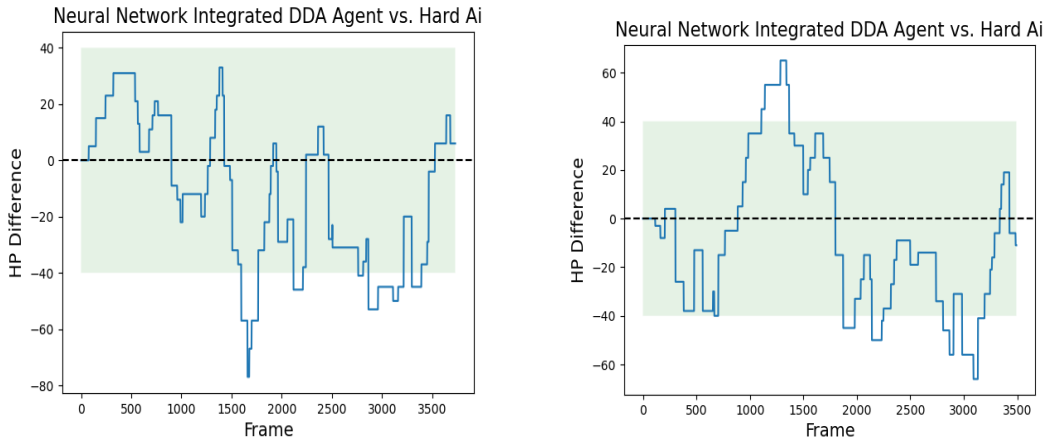


Figure 4.6: The plot shows how the HP difference changes when having a combating between the purely strong policy network-based agent with the "TeraThunder" AI

Similarly, we will initially compare the performance of our pure strong policy neural network agent with the "TeraThunder" AI. Figure 4.6 presents the changes in HP dif-

ference between the two agents over time when our agent does not include DDA. From the figure, it is evident that our strong policy neural network agent did not quickly gain a significant advantage over the opponent, which is different from the cases when facing AI opponents of easy and moderate difficulty levels. Although our agent ultimately emerged victorious with approximately 75 HP remaining, it encountered a close call as the HP difference between the agents almost fell behind the opponent around the 3000th frame.

Upon introducing the DDA mechanism to our agent, the performance of our DDA agent exhibits variability. We will first illustrate two cases where DDA is functioning as intended. In the first case, the DDA agent emerges victorious with a slight HP advantage over its opponent at the end of the game, while in the second case, the opponent prevails with a slight HP advantage over the DDA agent. Both of these cases are depicted in Figure 4.7.



(a) The HP difference between the DDA agent and the "TeraThunder" AI, the DDA agent won at the end of the game

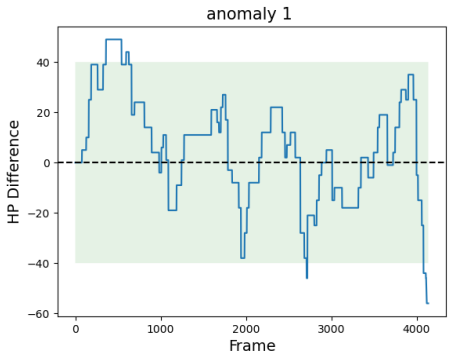
(b) The HP difference between the DDA agent and the "TeraThunder" AI, the DDA agent lose at the end of the game

Figure 4.7: The combating between the DDA agent and the "TeraThunder" AI

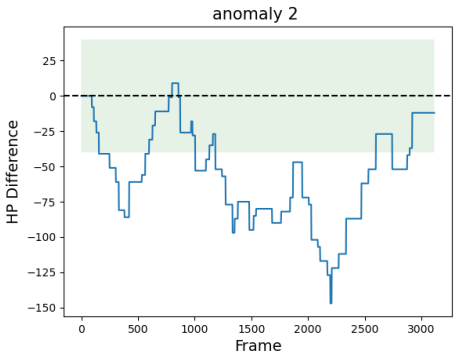
When comparing Figure 4.7 with the previously presented Figure 4.3 and Figure 4.5, we observe that in the scenarios depicted in Figure 4.7, our DDA agent is capable of adapting to the strong opponent. However, the changes in HP during the battle against the strong opponent are more abrupt, as evidenced by the steeper fluctuations in HP shown in the figure. In contrast, the changes in HP were relatively smoother in the previous two cases when combating easy and moderate AI opponents. This may be attributed to the fact that the strong AI opponent is employing more sophisticated, targeted, and aggressive strategies compared to the easy and moderate AI opponents. Consequently, when these attacks inflict damage on the DDA agent, the resulting fluctuations in HP are more pronounced. The opponent's strategy, with its focused and aggressive nature, presents

a challenge to the effectiveness of our DDA, as it may struggle to promptly correct the HP changes or stabilize them within a short timeframe. Meanwhile, the dynamic and aggressive nature of the strong opponent's strategy may cause significant fluctuations in the HP differences, making it challenging for our DDA agents to precisely gauge the opponent's skill level thus leading to unstable performance of the DDA mechanism.

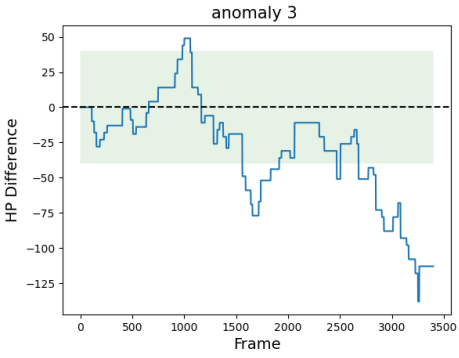
We also observed a list of anomalies during the experiment of our DDA agent against a formidable AI opponent, as depicted in Figure 4.8. Subfigure 4.8(a) demonstrates the consistent performance of our DDA mechanism throughout the game, until the opponent's strategy unexpectedly intensifies in the last 200 frames, swiftly altering the course of the game and causing the discrepancy in HP difference to exceed the anticipated range of -40 to 40, which our DDA mechanism aims to maintain.



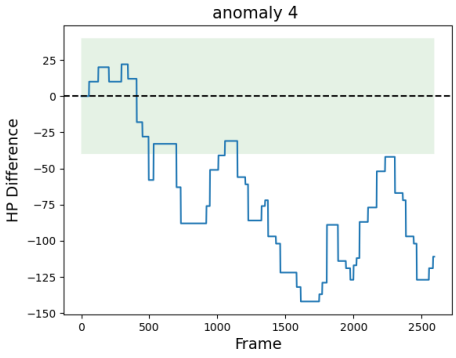
(a) Anomaly 1: In the game's final stages, DDA failed to respond in a timely manner and lost to the opponent with a significant HP disadvantage.



(b) Anomaly 2: Our DDA mechanism did not perform well in constraining the HP difference within the range of -40 to 40. The HP difference exceeded the limit of -40 for a significant period of time.



(c) Anomaly 3: DDA malfunction at the end



(d) Anomaly 4: DDA mechanism did not perform well throughout the entire game

Figure 4.8: Four different anomalies

Subfigure 4.8(b) illustrates another abnormal situation, in which we can observe that the HP difference between the two players exceeded the HP difference limitation during frames between 1200 and 2800, and the absolute HP difference even reaching nearly 150, which is significantly higher than the absolute maximum HP difference value observed in the previous experiments (which are roughly 65 and 45 respectively for easy and moderate AI opponents). However, this does not necessarily imply that our DDA mechanism did not function properly. We can see that between 1200 and 2800 frames, our DDA made several attempts to narrow the HP difference (for instance, the HP difference was close to -40 between 1800 and 2000 frames). This demonstrates one of the drawbacks that existed in our DDA agent. The inability of our DDA agent to maintain the HP difference within the target range of -40 to 40 for nearly half of the game could be attributed to the challenge of adapting to the strong opponent's strategy in a timely manner, and when encountering high competence opponent, it usually requires more time and effort for our DDA agent to correct the HP difference and bring it back to the desired range.

Apart from the scenarios discussed earlier, we observed more pronounced anomalies during our experiments. For instance, in the latter 1000 frames of Subfigure 4.8(c) (frames 2500-3500), our DDA mechanism appeared to malfunction, with our DDA agent struggling to make appropriate decisions against the opponent's attacks. This issue was even more severe in Subfigure 4.8(d), where the malfunction became apparent as early as 700 frames, ultimately resulting in our DDA agent's defeat with a substantial HP difference of nearly 125.

In order to further assess the performance of our DDA agent against a formidable opponent, we conducted 100 matches between our DDA agent and the "ThunderTera" AI. The outcomes of these matches are presented in Figure 4.9.

In Figure 4.9, we depict the distribution of end-game HP differences based on 100 matches. While a majority of HP differences fall within the range of -40 to 40, there are even some extreme values that fall into the bins of -180 and -200. Notably, a significant portion of HP differences still deviates from the desired range of -40 to 40, primarily concentrated between -140 and -40, indicating instances where our DDA agent did not fully fulfil its intended function. However, it is noteworthy that throughout all 100 rounds of duels, there were no instances where the final HP difference exceeded 40. This indicates that our DDA agent is still functioning as intended to some extent, as it avoids attempting to win against the opponent with a significant advantage in HP difference. The occurrence of HP differences outside the -40 range may be attributed to the adoption of superior strategies by strong opponents, which outperform the AI of weak and moderate opponents. And it is also possible due to the fact that our DDA agents may struggle to adapt to rapidly changing HP differences and respond appropriately in such scenarios, leading to deviations from the desired HP difference range.

To provide a comprehensive analysis, we also conducted 100 matches between our DDA agent and AI opponents set at easy and moderate difficulty levels, as shown in Figure

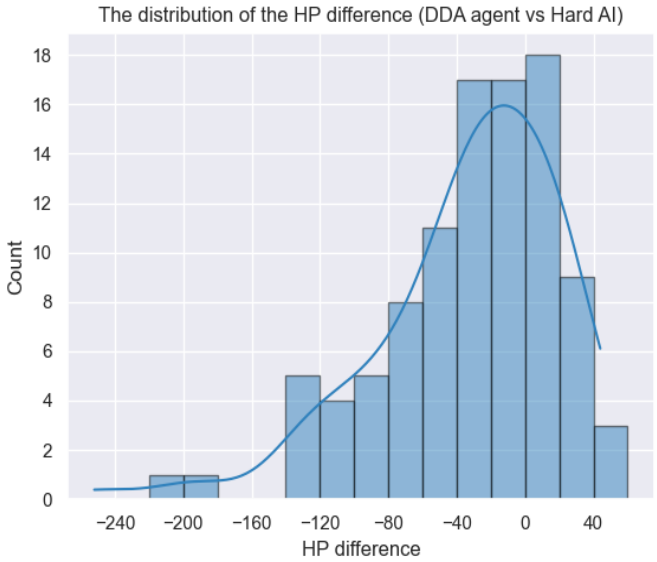
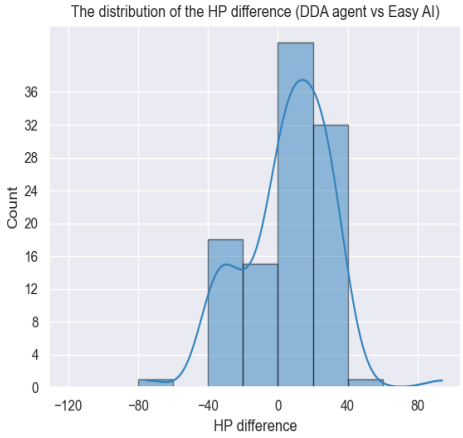
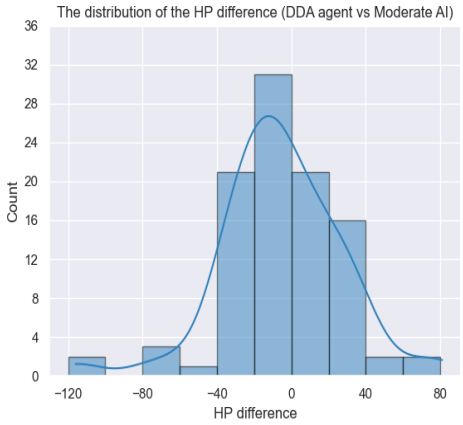


Figure 4.9: The histogram shows the distribution of the HP difference (DDA agent VS Strong AI) at the end of each game, with the total number of 100 games



(a) The HP difference of 100 games when our DDA agent combating with an Easy AI



(b) The HP difference of 100 games when our DDA agent combating with a Moderate AI

Figure 4.10: The combating between the DDA agent and the "SpringAi"

4.10. From the figure, it is evident that regardless of whether our agent played against the easy or moderate AI, the majority of final HP differences fell within the range of -40 to 40, in contrast to when playing against a strong AI where there were many HP differences points often fell outside this range. Subfigure 4.10(a) shows that when playing against

the easy AI, there were noticeably more data points in the range of 0 to 40 compared to -40 to 0. On the other hand, subfigure 4.10(b) displays a more evenly distributed number of data points falling within the range of -40 to 0 and 0 to 40 when playing against the moderate AI. This phenomenon can be attributed to the fact that our DDA agent is searching for the most suitable action from the output of the well-trained strong policy neural network, and when facing a less competitive adversary, blindly matching the difficulty of the adversary may lead our DDA agent to perform many abnormal behaviours, which we would like to avoid as these behaviours are detrimental to the gaming experience, thus leading to a slightly higher winning rate than encountering a moderate gaming ability opponent.

Figure 4.11 presents the relationship between game duration and the HP difference at the end of the game for the 100 matches performed with the strong AI opponent. It is evident from the figure that most of the HP differences outside the -40 to 40 range are associated with a relatively small number of frames, indicating a quicker end to the game. However, as the duration of the game increases, the number of outliers noticeably decreases. This observation suggests that the successful performance of DDA may lead to a longer game duration and that DDA's effects may require a certain amount of time to manifest. A quick end to the game may result in DDA being unable to function properly.

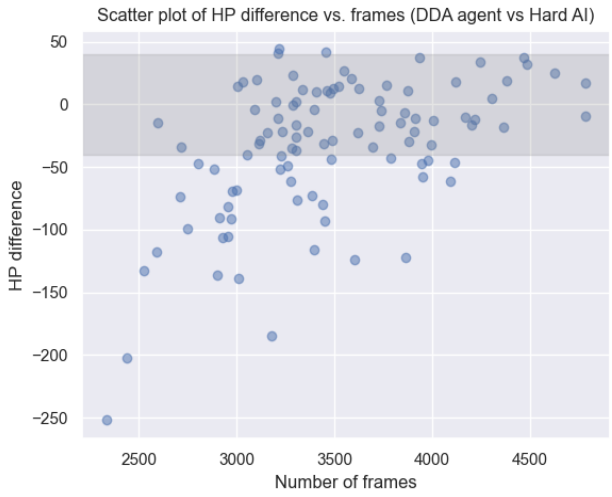


Figure 4.11: The scatter plot shows the relationship between the number of frames and the HP differences, with the total number of 100 games

4.2.4 Analysis of the Action Distribution of our DDA Agent

During the experiments, we also investigated the action distribution of our DDA agent under two different scenarios: (1) when the HP difference between the two players falls

Action Type	Actions
NO_OP	NEUTRAL, STAND, AIR, CHANGE_DOWN, DOWN, RISE, LANDING
OUT_OF_CONTROL	STAND_RECOV, STAND_GUARD_RECOV, CROUCH_RECOV, CROUCH_GUARD_RECOV, AIR_RECOV, AIR_GUARD_RECOV
MOVE	BACK_JUMP, BACK_STEP, FOR_JUMP, FORWARD_WALK, JUMP, DASH, CROUCH
SPECIAL_ATTACK	STAND_F_D_DFA, STAND_F_D_DFB, STAND_D_DB_BA, STAND_D_DB_BB, STAND_D_DF_FA, STAND_D_DF_FB, STAND_D_DF_FC, AIR_F_D_DFA, AIR_F_D_DFB, AIR_D_DB_BA, AIR_D_DB_BB, AIR_D_DF_FA, AIR_D_DF_FB
GUARD	AIR_GUARD, CROUCH_GUARD, STAND_GUARD
NORMAL_ATTACK	**All the remaining actions**

Table 4.1: The 6 different categories of actions

within the range of -40 to 40, and (2) when it exceeds this range. Following the categorization proposed by Lu et al. (2013), we classify the 56 possible actions that can be taken in the game into six major classes, as presented in Table 4.1. The term "NO_OP" indicates that the agent takes no action, meaning it aims to maintain its current state. "OUT_OF_CONTROL" signifies that the agent is currently in a state that can not be controlled by the agent, for example, in a recovery state after being attacked by the opponent. Actions in the "MOVE" group are used to control the movement of the agent. The "SPECIAL_ATTACK" group includes actions that can cause damage to the opponent, while the "NORMAL_ATTACK" group is similar but inflicts less damage and the actions are less complicated to perform compared to the actions in the "SPECIAL_ATTACK" group. Lastly, the "GUARD" group contains three actions that are used for defending against the opponent's attacks, in order to reduce the damage incurred by the agent when hit by the opponent's actions.

By utilizing the data collected from the previous experiments, we categorized the actions performed by our DDA agent into three groups based on the HP difference between our agent and the opponent. The first group consists of actions taken when the HP difference falls within the range of -40 to 40, while the second group comprises actions undertaken by our DDA agent when the HP difference is greater than 40 and the last group contains actions undertaken by our DDA agent when the HP difference is less than -40. Subsequently, we classified the actions in each group into the six categories defined in Table 4.1 and computed their proportions within the respective groups. The resulting action distribution plots are depicted in Figure 4.12.

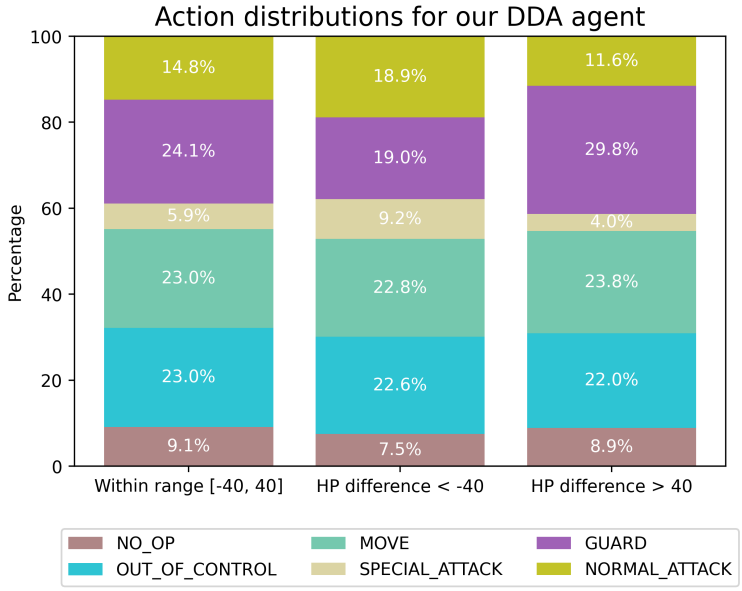


Figure 4.12: The action distribution when the HP difference is within the range of -40 to 40 and out of the range respectively

As depicted in the figure, it is evident that when the HP difference falls within the range of -40 to 40, our DDA agent executed NORMAL_ATTACK actions at a proportion of 14.8%, SPECIAL_ATTACK actions at 5.9%, and GUARD actions at a proportion of 24.1%. Conversely, when the HP difference is less than -40, as shown in the middle column of Figure 4.12, the proportion of NORMAL_ATTACK actions increased to 18.9%, SPECIAL_ATTACK actions increased to 9.2%, and the proportion of GUARD actions decreased to 19.0%. Simultaneously, the proportion of NO_OP actions decreased from 9.1% to 7.5%. These findings suggest that our agent exhibits more aggressive behaviour when the HP difference is less than 40 by reducing the proportion of defensive actions and increasing the proportion of normal and special attacks, aiming to recover from the disadvantaged state as soon as possible. From the rightmost column of Figure 4.12, we can see that when the HP difference is greater than 40, which means the opponent agent is in a more disadvantaged situation. In this case, our agent performed more gently, the proportion of GUARD actions from 24.1% to 29.8%, the proportion of NORMAL_ATTACK actions decreased from 14.8% to 11.6%, and the proportion of SPECIAL_ATTACK actions decreased from 5.9% to 4.0%.

Chapter 5

Discussion

Based on the findings obtained from our experiments, we can infer that our proposed DDA agent demonstrates effective adaptability in its decision-making process when facing opponents with easy and moderate difficulty levels. However, its performance tends to degrade when confronted with strong opponents. This can be attributed to the fact that our DDA agent evaluates the opponent's game level based on the HP difference and aims to maintain the HP difference within a specific range. Strong opponents often employ targeted and aggressive strategies, resulting in higher short-term damage output compared to easy and moderate-difficulty opponents. As a result, our DDA agent's ability to control game difficulty experiences fluctuations, as the opponent's strategy presents challenges in restoring the HP difference to the desired range in a timely manner. It may require more time for our DDA agent to restore the HP difference to the expected range when facing strong opponents compared to easy and moderate difficulty level opponents.

	Mean HP difference	**Standard deviation**
ROSAS	16	67
POSAS	-4	48
Ours(with easy opponent)	**5.7**	**24.8**
Ours(with moderate opponent)	**-9.2**	**23.4**
Ours(with hard opponent)	**-32.3**	**53.9**

Table 5.1: Comparsion of our DDA agent with two benchmarks

In our experiments, we compared our proposed DDA agent with two benchmark DDA agents, named Reactive Outcome Sensitive Action Selection (ROSAS) and Proactive Outcome-Sensitive Action Selection (POSAS), as proposed by Demediuk et al. (2017), as they tested their two agents in the FightingICE platform as well. We collected data on the final mean HP difference and standard deviation for all five DDA agents, as shown

in Table 5.1.

Let us first compare our proposed agent with the ROSAS agent. In matches against easy and moderate-difficulty level opponents, our DDA agent exhibits a significantly smaller absolute value of average HP difference (5.7 and -9.2, respectively) compared to the ROSAS agent, which has an average HP difference of 16. And our proposed agent, achieved standard deviations of 24.8 and 23.4 respectively, which is greatly less than the standard deviation obtained by the ROSAS agent. However, it is worth noting that our DDA agent's performance shows a deviation when facing hard-difficulty level opponents, with a larger absolute value of average HP difference (-32.3) compared to the ROSAS agent. Despite this disparity, our agent's standard deviation (53.9) remains smaller than that of the ROSAS agent (67).

When compared to the POSAS agent, our DDA agent exhibits a larger absolute value of average HP difference in all three cases. However, our agent obtained standard deviations that are generally smaller than those of the POSAS agent, except in the case when our DDA agent confronted a very proficient opponent where it obtained a larger standard deviation than the POSAS agent.

The data presented above does not necessarily indicate that our DDA agent performs worse than the baseline agents. It is important to consider the design objectives of our DDA agent, which are specifically aimed at maintaining the HP difference between two agents within the range of -40 to 40. Remarkably, our agent successfully achieved final mean HP differences within this predetermined range in all three cases, despite having slightly higher absolute final mean HP differences compared to the POSAS agent. Considering that our DDA agent was able to consistently keep the final mean HP differences within the target range of -40 to 40, we can still consider that our designed agent is effective in fulfilling its intended design objectives.

In addition, the lower standard deviations demonstrated by our DDA agent when encountering opponents of easy and moderate difficulty levels are noteworthy mentioning. A lower standard deviation suggests that the final HP differences were more tightly concentrated, indicating that our agent's performance is more stable compared to the benchmark agents. In fact, a lower standard deviation is preferable as it signifies that our agent has better control over the game difficulty, thus exhibiting more consistent performance. On the contrary, a higher standard deviation implies inconsistency in performance and potential reliance on inherent randomness in the game, and having a stable performance is more important than controlling the average value around 0.

However, it is worth noting that the stability of our DDA agent is compromised when facing a stronger opponent, as indicated by the increase in standard deviation to 53.9 when competing against a high-difficulty opponent.

There are several reasons for this. Firstly, strong opponents often have more advanced and targeted strategies, which can result in a higher amount of damage inflicted on opponents within a shorter timeframe. This can cause the HP difference to deviate sig-

nificantly from the target range, making it challenging for our DDA agent to readjust the HP difference back into the desired range. Our DDA agent may require more time to adapt and readjust the game difficulty to an appropriate level, but the game may end before the HP difference is fully readjusted, as evidenced by the results in Figure 4.11. As the game duration increases, we observe fewer final HP differences falling outside the target range, indicating that the longer the game duration is, the better performance our DDA agent can achieve. Furthermore, our DDA agent constantly adapts the game difficulty based on the player's skill level during gameplay. However, strong opponents may employ sophisticated strategies to deceive the DDA agent, making it challenging to accurately estimate their true skill level and adapt the game difficulty accordingly. This could lead to our DDA agent underestimating the opponent's ability and not adjusting the game difficulty effectively, resulting in a larger deviation from the desired HP difference range. Moreover, it is also possible that our DDA agent may still have the problem of enabling explore nodes in the game tree enough times, as for strong competent opponents, it might require more simulations before our agent can accurately simulate the opponents' strategy. This could limit the diversity of simulation results and lead to inaccurate estimations, thus leading to the higher standard deviation observed in our agent's performance against stronger opponents.

Now, let us return to our two research questions. The first question aims to investigate potential approaches for reducing the search space of the MCTS algorithm in order to facilitate a thorough exploration of diverse actions. The second question pertains to mitigating the occurrence of abnormal actions by our DDA agent. To address the first research question, we propose utilizing a well-trained neural network as a guidance mechanism to assist the MCTS algorithm in reducing the search space. By leveraging the action probability distribution output by the policy neural network, we can focus only on the actions with higher output probabilities, instead of the whole action space. This allows us to effectively reduce the search space for the subsequent MCTS process, enabling the MCTS process to concentrate on exploring optimal actions within this reduced subspace so as to alleviate the problem of under-exploration of the action space. For the second research question, we believe that under the guidance of the output of the policy neural network, the MCTS process would be less likely to select abnormal actions. Given that the policy neural network is trained to be highly competent, we can trust its output probabilities to guide the agent towards actions that are less likely to be abnormal. By prioritizing and focusing on actions with higher output probabilities, the probability of the DDA agent selecting abnormal actions can be significantly reduced.

Our experimental results, obtained by comparing the performance of our DDA agent with a baseline, revealed superior performance and stability when encountering opponents of easy and moderate difficulty levels. This compelling evidence supports the feasibility and effectiveness of reducing the exploration space of MCTS through the incorporation of a neural network in real-time battle games.

Furthermore, the analysis of our DDA agent's action distribution, as depicted in Figure 4.12, indicates its ability to dynamically adjust the ratio of different actions based on

the HP difference between the characters in a timely manner. Specifically, when facing opponents with a large HP difference, our DDA agent increases the proportion of defensive actions while decreasing the proportion of offensive actions. Even though this may not provide conclusive evidence that our agent will never exhibit abnormal behaviours, it is still good evidence to show the ability of our agent to adapt its action strategies to the opponent's competence level.

As we previously defined abnormal behaviours as actions that contradict the goal of defeating the opponent, such as constantly taking jumping actions, it is intuitive to anticipate that the DDA agents may be more prone to exhibiting anomalous behaviours when playing against easy-level opponents. This is because the DDA agent needs to be downward-compatible with the opponent's capabilities, and repeatedly taking abnormal actions could easily and effectively lower its own abilities. However, fortunately, we did not observe such anomalous behaviours in our DDA agent during our experiments. To provide a more comprehensive understanding, we have included videos of our DDA agent playing against opponents of varying difficulties in the appendix 7.4, with the aim of enabling readers to better appreciate the efforts made by our DDA agent to avoid abnormal behaviours.

However, Table 5.1 reveals a notable trend wherein the efficacy of our DDA agent consistently decreases as the proficiency level of the opponents increases. For instance, the final average HP difference dwindles from 5.7 when pitted against easy-difficulty opponents to -9.2 when against medium-difficulty opponents and plummets further to -32.3 against hard-difficulty opponents. While these values still fall within the -40 to 40 range, the observed negative average HP difference across varying proficiency levels highlights the insufficiency of our DDA proxies in dynamically adapting to stronger opponents.

Multiple factors may account for the observed decline in the performance of our DDA agent. Firstly, prior experiments have demonstrated that our trained neural network is capable of defeating a proficient opponent without the DDA mechanism, albeit gains victory with a small advantage in HP difference(as shown in Figure 4.1). This suggests that the introduction of MCTS to implement the DDA mechanism has excessively lowered our agent's proficiency, particularly when confronting a highly skilled adversary. We hypothesize that this could be attributed to the dependence of the MCTS algorithm on random simulations, which despite we have employed a neural network to focus on a subset of actions, may still suffer problems of the shortage of simulations and an incomplete exploration of all possible scenarios at a given node when confronted with a hard-difficulty opponent, resulting in an inaccurate assessment of the opponent's proficiency. This issue could potentially be mitigated by increasing the number of simulations per unit of time through parallel MCTS. Additionally, another possible reason could be that our policy neural network may still be insufficiently powerful, and our DDA implementation relies on curtailing the network's prowess through the MCTS algorithm. Therefore, the acquisition of a robust policy neural network is essential for the MCTS algorithm to function optimally.

5.1 Limitations

There are a few limitations that we must acknowledge in our work. Firstly, we rely on the HP difference as the primary indicator of the effectiveness of the DDA mechanism as well as the in-game factor to inform when the DDA agent should take action to adjust the game difficulty. However, using the HP difference as a measure and guidance for DDA may have some limitations, as it only considers the difference in HP between the two players and does not take into account other critical factors such as the usage of skills and the complexity of the opponent's strategy. Consequently, solely using HP difference may not be able to fully represent the actual skill gap between the two players.

Despite this, we still adopted the HP difference as a primary measure and guidance in our thesis as it is the easiest in-game metric to obtain and is widely used in the implementation of DDA in HP-based games. To address this problem, there are other game metrics that can be used as the guidance and evaluation criteria of the DDA mechanism. For instance, we suggest the use of opponent behaviour complexity, which refers to the complexity of the opponent's decision-making process and is typically measured using information entropy or KL divergence. In this case, our agent can match the opponent's strategy by analyzing their behaviour complexity. Additionally, we can consider using win rate as another measurement, where the DDA agent's objective now becomes to maintain a 50% win rate with the opponent. However, focusing solely on the win rate may not reflect the actual game situation and it may not accurately reflect how each game was played, as it does not indicate how the agent achieved those wins. For instance, the agent may have won some games by a large HP margin, while in other games, the agent may lose the game purposely with a large HP margin to maintain the total win rate to be 0.5. Therefore, we can consider using one or a combination of these metrics as a comprehensive measure depending on the genre of the game to more accurately assess and guide the DDA mechanism.

The data utilized to train our client may not be comprehensive enough to encompass all types of opponent strategies, particularly those opponent strategies that are much more advanced and sophisticated. Therefore, future research could explore ways to expand and diversify data collection, so as to facilitate better training of our DDA agent and improve its ability to handle a wide range of opponents. As we have mentioned previously, one of the reasons that leading to the degradation of our DDA agent when facing hard-difficulty opponents could be our trained policy neural network is still not strong enough. Thus, if we can train a stronger neural network with more diverse and comprehensive training data, we believe the performance of our DDA agent will increase when encountering hard-difficulty opponents.

Another limitation is that our thesis does not incorporate human experiments to evaluate the effectiveness of our DDA agent. Instead, we assessed the agent's ability to regulate game difficulty by having it play against pre-existing AI opponents with varying skill levels. Although our DDA agent exhibited considerable proficiency in controlling game difficulty against most AI opponents, this may not necessarily reflect its performance

when confronted with human players. The behavioural and decision-making patterns of human players are more diverse and intricate compared to AIs. Therefore, to obtain a more comprehensive evaluation of the adaptability and feasibility of our DDA agent across various player populations, it would be useful to conduct further testing via human experiments.

Chapter 6

Conclusions

In this thesis, we propose a novel approach to the DDA mechanism based on the MCTS algorithm in real-time fighting games. Our approach employs a robust policy neural network that is trained using the PPO algorithm to output probability distributions of all executable actions. Subsequently, a group of actions with higher probabilities is selected and the MCTS algorithm is utilized to explore and select actions that are most likely to match the opponent's current skill level. This approach enables the MCTS process to focus only on a subset of the entire action space, thereby minimizing the risk of encountering search and simulation deficiencies that may arise due to real-time constraints. Meanwhile, our research has also demonstrated that, under the guidance of the strong policy neural network, our DDA mechanism is less prone to executing abnormal actions when adjusting game difficulties, leading to an improved gaming experience for players.

After a series of random simulations, the MCTS process selects the most optimal action that is likely to correspond to the current player's skill level for the agent to execute. In this paper, the HP difference is employed to measure the discrepancy between the skill level of our DDA agent and the player. Our DDA agent is designed to ensure that the HP difference between the two players remains within a range of 10% of the maximum HP, thereby ensuring that the game difficulty is suitable for the opponent. In our study, by subjecting our DDA agent to various AI agents with differing skill levels, we discovered that our DDA agent is capable of matching appropriate difficulty levels to opponents with easy and moderate game skills, and is capable of stably controlling the HP difference within a suitable range. However, when confronted with advanced and complex opponent strategies, our DDA agent demonstrates a degradation in its ability to regulate game difficulty, leading to fluctuations in the HP difference.

Although our proposed DDA mechanism does not perform well when fighting against opponents with strong competency, the approach propose to use neural network output to guide the MCTS algorithm is still meaningful and effective because it effectively re-

duces the search space of MCTS without additional heuristics, thus reducing the burden on MCTS algorithm and enabling MCTS algorithm can explore as many meaningful nodes as possible in a real-time game setting. As this method does not require additional heuristics, it can be easily applied to other different games. Meanwhile, we found that using the neural network output to guide MCTS search can effectively reduce the probability of the DDA agent intentionally exhibiting anomalous behaviour to achieve difficulty adjustment, thereby further enhancing the player's gaming experience.

Despite the fact that our DDA agent exhibits unsatisfactory performance when facing strong-difficulty opponents, the proposed method in this thesis, which involves utilizing neural network output to guide MCTS, remains significant and effective. This is because it provides a way to effectively reduce the search space of the MCTS algorithm, without necessitating the use of additional heuristics and domain knowledge, thus alleviating the computational burden of search and simulation in real-time games and allowing the MCTS algorithm to explore more meaningful nodes. As this approach does not require supplementary heuristics, it can be seamlessly applied to other diverse games. Moreover, our findings reveal that utilizing the neural network output to guide MCTS search can considerably diminish the likelihood of the DDA agent intentionally demonstrating anomalous behaviours to achieve difficulty adjustment, thus further augmenting the player's gaming experience.

6.1 Future Work

There are several improvements that can be done in the future:

- Using the PPO algorithm to further train our agent through self-play to train a stronger policy network and collect more comprehensive and diverse training data, so as to increase the diversity of our agent's policy, enabling it to be compatible with the strong policy opponents.
- To parallelize the MCTS process so as to explore more nodes and simulate more iterations within a given time frame. We believe that the more a node is being simulated, the closer it will be to the actual gameplay scenarios, allowing our DDA agent to make more accurate evaluations and ultimately improving the effectiveness of the DDA mechanism.
- Conducting human experiments to test how our DDA agent will behave when encountering human players.
- Considering if introducing extra domain-specific knowledge specialized for the game to guide the search of MCTS would contribute to the performance of our DDA agent.

Chapter 7

Appendix

7.1 Policy Gradient Method

Prior to delving into a detailed exploration of PPO, it would be beneficial to familiarize ourselves with the Policy Gradient Method, as it is also a policy-based algorithm.

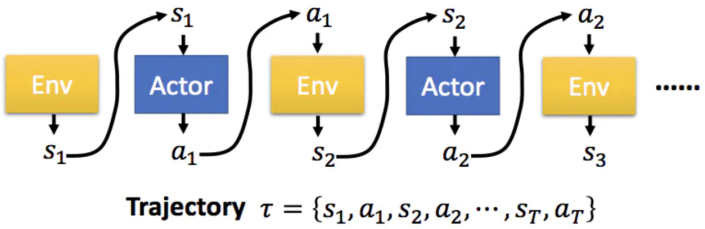


Figure 7.1: Policy gradient Agent-Environment interaction (yi Lee, 2018)

In the Policy Gradient (PG) method, the agent, also referred to as the actor, engages in iterative interactions with the environment, as depicted in Figure 7.1. The agent initiates from an initial state s_1, takes action a_1, and receives a new state s_2 from the environment, continuing this process until the task is completed, constituting an episode. The sequence of states and actions forms a trajectory, which we define as:

$$\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$$

Let us denote the policy parameter of the actor as θ, we can compute the probability of a given trajectory as follows:

$$\begin{aligned} p_\theta(\tau) &= p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_1)p(s_3|s_2, a_2)\ldots \\ &= p(s_1)\prod_{t=1}^{T} p_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t) \end{aligned}$$

In the above equation, it is important to note that the term $p(s_1)$ and the term $p(s_{t+1}|s_t, a_t)$ are determined by the environment, while only the term $p_\theta(a_t|s_t)$ is influenced by the agent's policy. Additionally, for each action taken, a reward is associated with it. Denoting $R(\tau)$ as the total reward for a trajectory τ, the actor's expected reward under the policy parameter θ is defined as follows:

$$\overline{R}_\theta = \sum_\tau R(\tau) p_\theta(\tau) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)]$$

Hence, our objective is to adjust the policy parameter θ in order to maximize the expected reward that the actor can obtain. Utilizing the expected reward function, we can employ policy gradient ascent to iteratively update the policy parameter θ, aiming to find the value of θ that maximizes the expected reward. The gradient of the expected reward, denoted as $\nabla \overline{R}_\theta$, can be calculated as follows:

$$\nabla \overline{R}_\theta = \sum_\tau R(\tau) \nabla p_\theta(\tau) \quad \text{(as } R(\tau) \text{ does not require to differentiate)}$$

$$= \sum_\tau R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} = \sum_\tau R(\tau) p_\theta(\tau) \nabla \log(p_\theta(\tau)) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau) \nabla \log(p_\theta(\tau))]$$

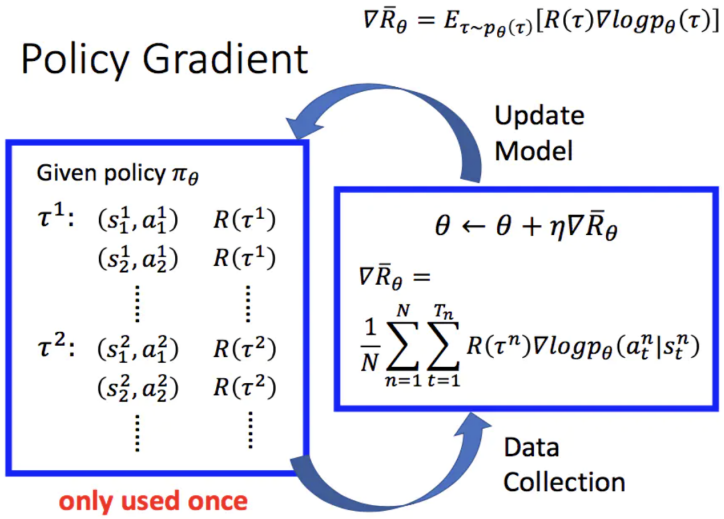


Figure 7.2: Policy gradient policy parameter update process (yi Lee, 2018)

We can update the policy parameter using the equation $\theta = \theta + \eta \nabla \overline{R}_\theta$, where η represents the learning rate. The overall process is illustrated in Figure 7.2. Initially, multiple trajectory samples are collected and fed into the policy model to update the parameter. However, it should be noted that once the policy parameter θ has been updated using the collected data, the data becomes obsolete, and new batches of data need to be collected for continuous model training. This limitation is a prominent drawback of the gradient descent method, as it requires considerable time to re-collect samples for parameter updates.

7.2 The Derivation of the Gradient of the Objective Function in PPO

$$
\begin{aligned}
\nabla \overline{R}_\theta &= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} R(\tau) \nabla \log(p_\theta(\tau))] \\
&= \mathbb{E}_{(s_t,a_t)\sim \pi_{\theta'}}[\frac{p_\theta(s_t,a_t)}{p_{\theta'}(s_t,a_t)} A^{\theta'}(s_t,a_t) \nabla \log(p_\theta(s_t,a_t))] \\
&= \mathbb{E}_{(s_t,a_t)\sim \pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} \frac{p_\theta(s_t)}{p_{\theta'}(s_t)} A^{\theta'}(s_t,a_t) \nabla \log(p_\theta(s_t,a_t))] \\
&= \mathbb{E}_{(s_t,a_t)\sim \pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t) \nabla \log(p_\theta(s_t,a_t))] \\
&\text{(assuming } p_\theta(s_t) \text{ and } p_{\theta'}(s_t) \text{ will not differ to much)}
\end{aligned}
$$

7.3 An example of detailed Monte Carlo Tree

Figure 7.3 shows detailed information on a Monte Carlo Tree generated during a single time frame. We can see the root node has been visited 140 times. And the tree has a depth of three, and each node except the leaf nodes has been expanded with 6 child nodes. For each child node, there are several pieces of information related to them, the first one is the index of the child node, and the second one "num_visits" represents the number of visits of this child node, the "node_depth" represents the node belongs to which depth, and the "score" value is used to evaluate the goodness of the node, and the "UCB" value is used to select node during the MCTS selection phase, and the last one is the "actions_selected", which will be performed by the agent if this node is being selected.

The current node has been visited:140 timesnode depth: 0
the children nodes have length: 6
0, num_visited:24, node_depth:1, score:0.0, ucb:1.9237607569154798, actions_selected:
FORWARD_WALK
1, num_visited:24, node_depth:1, score:0.0, ucb:1.9651366895300346, actions_selected:
CROUCH_GUARD
2, num_visited:23, node_depth:1, score:0.0, ucb:1.9651366895300346, actions_selected: STAND_GUARD
3, num_visited:23, node_depth:1, score:0.0, ucb:1.9651366895300346, actions_selected: CROUCH_B
4, num_visited:23, node_depth:1, score:-2.347826086956522, ucb:1.9592671243126434, actions_selected:
AIR_UA
5, num_visited:23, node_depth:1, score:0.0, ucb:1.9651366895300346, actions_selected: AIR_FB

The current node has been visited:24 timesnode depth: 1
parant selected action is: FORWARD_WALK
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: FORWARD_WALK
1, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: AIR_FB
2, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: STAND_FB
3, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: JUMP
4, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: STAND_A
5, num_visited:3, node_depth:2, score:0.0, ucb:5.312198033146199, actions_selected: CROUCH_B

The current node has been visited:24 timesnode depth: 1
parant selected action is: CROUCH_GUARD
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: AIR_UA
1, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: STAND_A
2, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: AIR_DA
3, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: CROUCH_B
4, num_visited:3, node_depth:2, score:0.0, ucb:4.3373915312748625, actions_selected: AIR_DA
5, num_visited:3, node_depth:2, score:0.0, ucb:5.312198033146199, actions_selected: CROUCH_A

The current node has been visited:23 timesnode depth: 1
parant selected action is: STAND_GUARD
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: AIR_GUARD
1, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: THROW_A
2, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: AIR_GUARD
3, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: STAND_FB
4, num_visited:3, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: THROW_A
5, num_visited:2, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: AIR_FA

The current node has been visited:23 timesnode depth: 1
parant selected action is: CROUCH_B
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: STAND_FA
1, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: BACK_JUMP
2, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: AIR_DA
3, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: FORWARD_WALK
4, num_visited:3, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: THROW_A
5, num_visited:2, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: STAND_FB

The current node has been visited:23 timesnode depth: 1
parant selected action is: AIR_UA
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:-3.0, ucb:4.299036278745356, actions_selected: STAND_A
1, num_visited:3, node_depth:2, score:-3.0, ucb:4.299036278745356, actions_selected: FORWARD_WALK
2, num_visited:3, node_depth:2, score:-3.0, ucb:4.299036278745356, actions_selected: CROUCH_FB
3, num_visited:3, node_depth:2, score:-3.0, ucb:4.299036278745356, actions_selected: CROUCH_FA
4, num_visited:3, node_depth:2, score:-3.0, ucb:5.266908220855193, actions_selected: STAND_A
5, num_visited:2, node_depth:2, score:-3.0, ucb:5.266908220855193, actions_selected: STAND_B

The current node has been visited:23 timesnode depth: 1
parant selected action is: AIR_FB
the children nodes have length: 6
0, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: BACK_JUMP
1, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: AIR_UA
2, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: AIR_GUARD
3, num_visited:3, node_depth:2, score:0.0, ucb:4.306536278745356, actions_selected: STAND_FB
4, num_visited:3, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: CROUCH_A
5, num_visited:2, node_depth:2, score:0.0, ucb:5.274408220855193, actions_selected: CROUCH_GUARD

Figure 7.3: An example of the constructed tree within a single time frame with detailed information

7.4 Video examples of our agent playing against different competitive level opponents

Video examples of including DDA mechanism and not including DDA mechanism

Bibliography

Afergan, D.; Peck, E. M.; Solovey, E. T.; Jenkins, A.; Hincks, S. W.; Brown, E. T.; Chang, R.; and Jacob, R. J., 2014. Dynamic difficulty using brain metrics of workload. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 3797–3806. [Cited on page 3.]

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S., 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4, 1 (2012), 1–43. [Cited on page 28.]

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P., 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 4, 216–217. [Cited on pages 28, 29, and 30.]

Chen, J., 2007. Flow in games (and everything else). *Communications of the ACM*, 50, 4 (2007), 31–34. [Cited on pages 1 and 2.]

Cherukuri, A. and Glavin, F. G., 2022. Balancing the performance of a fightingice agent using reinforcement learning and skilled experience catalogue. In *2022 IEEE Games, Entertainment, Media Conference (GEM)*, 1–6. IEEE. [Cited on page 22.]

Coquelin, P.-A. and Munos, R., 2007. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, (2007). [Cited on page 32.]

Coulom, R., 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games: 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers 5*, 72–83. Springer. [Cited on pages 28 and 31.]

Demediuk, S.; Tamassia, M.; Li, X.; and Raffe, W. L., 2019. Challenging ai: Evaluating the effect of mcts-driven dynamic difficulty adjustment on player enjoyment. In *Proceedings of the Australasian Computer Science Week Multiconference*, 1–7. [Cited on pages 6, 12, 13, 14, and 30.]

Demediuk, S.; Tamassia, M.; Raffe, W. L.; Zambetta, F.; Li, X.; and Mueller, F., 2017. Monte carlo tree search based algorithms for dynamic difficulty adjustment. In *2017 IEEE conference on computational intelligence and games (CIG)*, 53–59. IEEE. [Cited on page 63.]

Engstrom, L.; Ilyas, A.; Santurkar, S.; Tsipras, D.; Janoos, F.; Rudolph, L.; and Madry, A., 2020. Implementation matters in deep policy gradients: A case study on ppo and trpo. *arXiv preprint arXiv:2005.12729*, (2020). [Cited on pages 3 and 23.]

Frommel, J.; Fischbach, F.; Rogers, K.; and Weber, M., 2018. Emotion-based dynamic difficulty adjustment using parameterized difficulty and self-reports of emotion. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*, 163–171. [Cited on page 3.]

Ikeda, K. and Viennot, S., 2013. Production of various strategies and position control for monte-carlo go—entertaining human players. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8. IEEE. [Cited on page 52.]

Ishihara, M.; Ito, S.; Ishii, R.; Harada, T.; and Thawonmas, R., 2018. Monte-carlo tree search for implementation of dynamic difficulty adjustment fighting game ais having believable behaviors. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE. [Cited on page 14.]

Khajah, M. M.; Roads, B. D.; Lindsey, R. V.; Liu, Y.-E.; and Mozer, M. C., 2016. Designing engaging games using bayesian optimization. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, 5571–5582. [Cited on page 11.]

Kim, M.-J.; Lee, J.-H.; and Ahn, C. W., 2020. Genetic optimizing method for real-time monte carlo tree search problem. In *The 9th International Conference on Smart Media and Applications*, 50–51. [Cited on page 12.]

Kocsis, L.; Szepesvári, C.; and Willemson, J., 2006. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1 (2006), 1–22. [Cited on page 30.]

Lomas, J. D.; Koedinger, K.; Patel, N.; Shodhan, S.; Poonwala, N.; and Forlizzi, J. L., 2017. Is difficulty overrated? the effects of choice, novelty and suspense on intrinsic motivation in educational games. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, 1028–1039. [Cited on page 1.]

Lu, F.; Yamamoto, K.; Nomura, L. H.; Mizuno, S.; Lee, Y.; and Thawonmas, R., 2013. Fighting game artificial intelligence competition platform. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, 320–323. IEEE. [Cited on pages 21 and 60.]

Majchrzak, K.; Quadflieg, J.; and Rudolph, G., 2015. Advanced dynamic scripting for fighting game ai. In *Entertainment Computing-ICEC 2015: 14th International Conference, ICEC 2015, Trondheim, Norway, September 29-Ocotober 2, 2015, Proceedings 14*, 86–99. Springer. [Cited on pages 7, 8, and 21.]

Mańdziuk, J., 2018. Mcts/uct in solving real-life problems. *Advances in Data Analysis with Computational Intelligence Methods: Dedicated to Professor Jacek Żurada*, (2018), 277–292. [Cited on page 29.]

Moon, J.; Choi, Y.; Park, T.; Choi, J.; Hong, J.-H.; and Kim, K.-J., 2022. Diversifying dynamic difficulty adjustment agent by integrating player state models into monte-carlo tree search. *Expert Systems with Applications*, 205 (2022), 117677. [Cited on pages 15 and 16.]

Ontanón, S., 2016. Informed monte carlo tree search for real-time strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE. [Cited on page 12.]

Ontanón, S., 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58 (2017), 665–702. [Cited on page 12.]

Or, D. B.; Kolomenkin, M.; and Shabat, G., 2021. Dl-dda-deep learning based dynamic difficulty adjustment with ux and gameplay constraints. In *2021 IEEE Conference on Games (CoG)*, 1–7. IEEE. [Cited on page 11.]

Ouessai, A.; Salem, M.; and Mora, A. M., 2020. Improving the performance of mcts-based μrts agents through move pruning. In *2020 IEEE Conference on Games (CoG)*, 708–715. IEEE. [Cited on page 12.]

RITSUMEIKAN, U., 2023. Darefightingice competition (formerly fighting game ai competition). https://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-2a.html. [Cited on pages 3 and 22.]

Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P., 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, (2015). [Cited on page 39.]

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O., 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, (2017). [Cited on page 28.]

Segundo, C. V. N.; Emerson, K.; Calixto, A.; and Gusmao, R., 2016. Dynamic difficulty adjustment through parameter manipulation for space shooter game. *Proceedings of SB Games*, (2016). [Cited on page 9.]

Smith, L., 2021. How to train and learn swordsmanship without a local group. https://historicalfencer.com/how-to-train-and-learn-swordsmanship-without-a-local-group/. [Cited on page 6.]

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E., 2003. Online adaptation of computer game opponent ai. (09 2003). [Cited on page 7.]

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E., 2004. Online adaptation of game opponent ai with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3, 1 (2004), 45–53. [Cited on pages 7 and 8.]

Sutton, R. S. and Barto, A. G., 2018. *Reinforcement learning: An introduction*. MIT press. [Cited on page 23.]

Świechowski, M.; Tajmajer, T.; and Janusz, A., 2018. Improving hearthstone ai by combining mcts and supervised learning algorithms. In *2018 IEEE conference on computational intelligence and games (CIG)*, 1–8. IEEE. [Cited on page 12.]

Szita, I.; Ponsen, M.; and Spronck, P., 2009. Effective and diverse adaptive game ai. *IEEE Transactions on Computational Intelligence and AI in Games*, 1, 1 (2009), 16–27. [Cited on page 8.]

Timuri, T.; Spronck, P.; and van den Heri, J., 2007. Automatic rule ordering for dynamic scripting. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 3, 49–54. [Cited on page 8.]

Tremblay, J., 2011. *A new approach to dynamic difficulty adjustment in video games*. Université du Québec à Chicoutimi. [Cited on page 6.]

Vang, C., 2022. The impact of dynamic difficulty adjustment on player experience in video games. *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal*, 9, 1 (2022), 7. [Cited on page 2.]

Wang, Y.; He, H.; and Tan, X., 2020. Truly proximal policy optimization. In *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, vol. 115 of *Proceedings of Machine Learning Research*, 113–122. PMLR. [Cited on page 23.]

Xue, S.; Wu, M.; Kolen, J.; Aghdaie, N.; and Zaman, K. A., 2017. Dynamic difficulty adjustment for maximized engagement in digital games. In *Proceedings of the 26th International Conference on World Wide Web Companion*, 465–471. [Cited on pages 9 and 10.]

yi Lee, H., 2018. Drl lecture 1: Policy gradient. https://www.youtube.com/watch?v=z95ZYgPgXOY&list=PLJV_el3uVTsODxQFgzMzPLa16h6B8kWM_. [Cited on pages 25, 27, 71, and 72.]

Yu, C.; Velu, A.; Vinitsky, E.; Wang, Y.; Bayen, A.; and Wu, Y., 2021. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, (2021). [Cited on page 23.]

Zakharenkov, A. and Makarov, I., 2021. Deep reinforcement learning with dqn vs. ppo in vizdoom. In *2021 IEEE 21st International Symposium on Computational Intelligence and Informatics (CINTI)*, 000131–000136. IEEE. [Cited on page 25.]

Zohaib, M., 2018. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018 (2018). [Cited on pages 1, 2, and 10.]