Dynamic Difficulty Adjustment of Enemy AI in Action Games Using ML-Agents

Yuna Tokuda s1300037 Supervised by Prof. Maxim Mozgovoy

Abstract

Dynamically adjusting game difficulty is essential for providing an engaging experience for players of various skill levels. This paper presents the implementation of the Adaptive Duo-Chromosome (ADC) controller in a 3D action game developed using Unity and ML-Agents. The ADC algorithm employs a genetic algorithm-inspired approach that maintains two independent chromosomes, one encoding "winning" strategies and one encoding "losing" strategies, which are dynamically updated based on game outcomes to adjust NPC behavioral parameters in real-time. This paper compares three types of AI agents: an adaptive agent using ADC, a static agent with a fixed difficulty level, and a random agent with random parameters. The results indicate that the ADC agent achieved a win rate of 42.3% (closest to the balanced target of 50%) and a close game rate of 68.1%, significantly outperforming static (69.3% win rate, 27.0% close game rate) and random (3.6% win rate) agents.
These results validate ADC's effectiveness in providing engaging and balanced gameplay experiences.

1. Introduction

Non-Playable Characters (NPCs) in video games have evolved dramatically in recent years, driven by advances in artificial intelligence (AI) and machine learning (ML) [1]. However, most NPCs in games are predefined and predictable, significantly reducing player engagement. This is particularly acute in single-player games, where interacting with and combating NPCs constitutes a significant portion of the overall gaming experience. Dynamic Difficulty Adjustment (DDA) addresses this problem by automatically adjusting game parameters in real time based on player performance [2]. Unlike manually selecting a difficulty level at the start of a game, DDA continuously monitors player behavior such as win rate, damage dealt, and survival time, and adapts the game's challenge accordingly. This approach has the potential to maintain player engagement by keeping the game neither too easy nor too difficult [4].

Several approaches to DDA have been proposed in the literature.

Reinforcement learning (RL) methods allow agents to learn optimal behaviors through trial and error by receiving rewards or penalties based on their actions [3].

Genetic algorithm-based approaches, inspired by natural selection, use concepts such as chromosomes, mutation, and selection to evolve solutions over generations. These methods can adapt behaviors by encoding parameters as "genes" in a chromosome structure, where each gene represents a behavioral characteristic in the range [0, 1].

1.1 Problem Statement and Motivation

Traditional game AI relies on scripted behaviors with finite states or behavior trees. While these techniques are simple to implement, they result in predictable NPCs that players can easily exploit once they learn the patterns [4]. Recent advances in machine learning have enabled more sophisticated NPC behaviors through reinforcement learning (RL). However, these approaches typically operate with fixed policies after training, lacking the ability to dynamically adjust to individual player skill levels during gameplay. Consequently, policies designed for the average player may be too difficult for beginners or too easy for experts, failing to maintain optimal player engagement. The main goal of this research is to implement an adaptive AI system for a 3D action game.

Tan et al. [2] proposed the Adaptive Duo-Chromosome (ADC) algorithm for dynamic difficulty adjustment in a 2D car racing game. Their work was limited to a simple 2D environment with hand-crafted behaviors.

This research investigates whether ADC can provide effective difficulty adjustment in a more complex 3D action game. We implement ADC in a Unity-based melee combat game where NPCs are controlled by neural networks trained through ML-Agents.

The contributions of this work are: (1) Implementation of ADC in a 3D melee combat environment, (2) Integration of chromosome-based adjustment with deep RL policies trained through PPO, (3) Evaluation against static, random baselines and ADC over 5,000 episodes each.

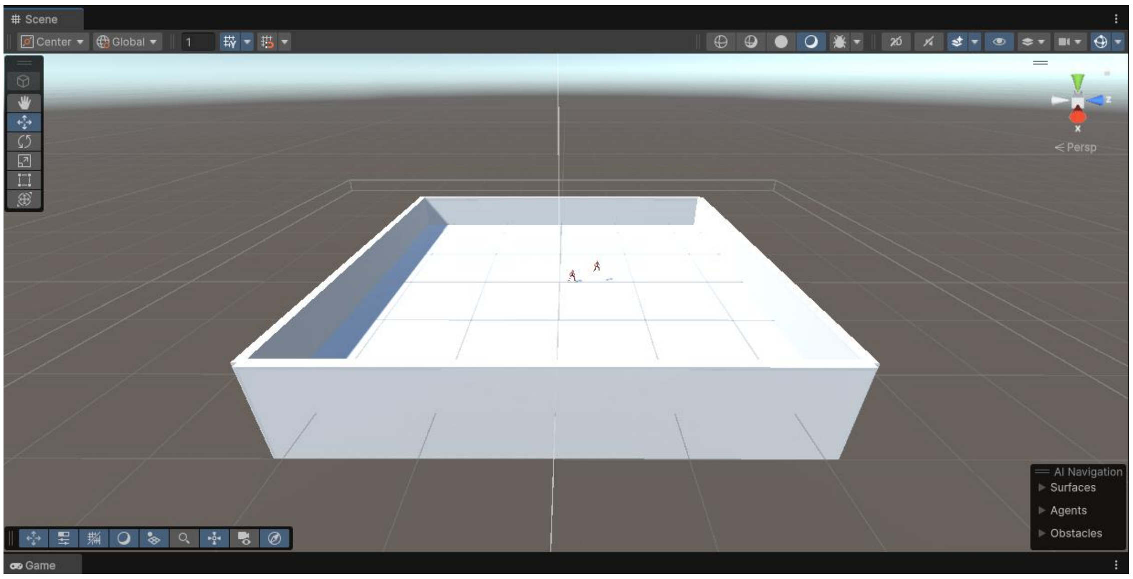

Figure. 1. Game Environment Scene

2. Related Work

Tan et al. [2] proposed two adaptive algorithms for dynamic difficulty scaling: the Adaptive Uni-Chromosome (AUC) and Adaptive Duo-Chromosome (ADC) controllers. Their experiments in a simulated car racing game demonstrated that both algorithms could match opponents in mean score and winning percentage without requiring offline training. Spronck et al. [5] developed dynamic scripting, which uses a weighted rule base that adapts based on game outcomes. Their approach required approximately 50 encounters for effective training. In contrast, the ADC algorithm offers immediate adaptation without a training phase, making it more suitable for real-time applications. Meshram et al. [1] demonstrated the application of Unity ML-Agents for NPC behavior training using reinforcement learning. Their research showed that RL-trained NPCs can learn to navigate environments and adapt their strategies through reward-based learning.

3. Method

3.1 Experimental Environment

This study uses a one-versus-one melee combat simulation environment built on Unity. Figure 1 shows the experimental environment. The experimental field is a flat rectangular area measuring 40m × 40m. At the start of each episode, the initial positions of both the player and enemy agent are randomly determined within the field. If a player falls off the field, the episode terminates immediately.

The maximum duration of each round is set to 120 seconds. For dynamic difficulty adjustment (DDA), five waypoints (evaluation points) are established during each round, with the score difference between both combatants evaluated every 24 seconds. Episodes terminate when any of the following conditions are met: either character's HP reaches 0 or below (knockout), a fall outside the field occurs, or the round time limit is reached.

3.1.1 Enemy Agent Specifications

Enemy agents are implemented as reinforcement learning agents using Unity ML-Agents. This study primarily uses EnemyHybridAgent.cs (adaptive type) and EnemyStaticAgent.cs (fixed parameter type). The agent's observation space consists of 17 dimensions: chromosome parameters (7 dimensions), normalized relative direction vector to player (3 dimensions), distance to player (1 dimension), normalized self-velocity vector (3 dimensions), attack cooldown state (1 dimension), normalized self HP (1 dimension), and normalized opponent HP (1 dimension).

The action space is discrete and consists of three types of discrete actions: Movement action (stationary, forward, backward, right, left), Dash action (normal movement, dash), and Attack action (no attack, side attack, upper attack). Agents can execute two types of attacks: Side attack with base damage 10 × Aggressiveness multiplier and Upper attack with base damage 20 × Aggressiveness multiplier.

3.2 System Overview

The system consists of three main components: (1) ML-Agents trained enemy AI that provides base combat behavior, (2) ADC Controller that manages dual-chromosome difficulty adjustment, and (3) Game

Manager that tracks performance and triggers chromosome updates at waypoints.

The enemy AI is trained using Proximal Policy Optimization (PPO) [6] through imitation learning against a player AI. The training reward structure includes: +5 for winning (opponent HP reaches 0), -5 for losing, +1 for successful attack hit (with accuracy bonus based on aimAccuracy parameter), -1 for missed attack, +0.05 for being within attack range, -0.02 for being too far from opponent, and -0.001 time penalty per step to encourage decisive action.

3.3 Behavior Chromosome

Agent behavioral characteristics are controlled by a "chromosome" structure inspired by genetic algorithms. Each behavior chromosome consists of seven genes, each representing a behavioral parameter normalized to the range [0.0, 1.0]. These values are mapped to actual game parameters using linear interpolation (Lerp). For example, movement speed is calculated as:

movementSpeed = baseSpeed × Lerp(0.5, 1.5, chromosome.movementSpeed),

resulting in a range of 0.5× to 1.5× the base speed.

3.4 ADC Algorithm

The ADC maintains two separate chromosomes. One is a "winning" chromosome representing strategies that help defeat opponents. The other is a "losing" chromosome representing strategies that allow opponents to win. Unlike the AUC which derives the losing strategy as the complement (1 - value) of the winning chromosome, ADC allows both chromosomes to evolve independently based on actual game outcomes.

Chromosome updates occur at waypoints. When the ADC wins a waypoint, the winning chromosome is updated: gene values are adjusted in the positive direction weighted by the learning rate and mutation. When the ADC loses a waypoint, the losing chromosome is updated instead. This dual-update mechanism allows each chromosome to specialize in its respective role.

The update rules follow the formulation from Tan et al. [2].

If ADC wins:

$$win_i \mathrel{+}= sgn(behavior_i) \times lrnRate \times |mutation|$$

If ADC loses:

$$lose_i \mathrel{+}= sgn(behavior_i) \times lrnRate \times |mutation|$$

Credit assignment is implemented by weighting updates based on relative distances: wins from disadvantaged positions receive higher confidence, while wins from advantaged positions receive lower confidence.

The active chromosome is selected based on the enemy's current win rate relative to the target (default 50%). If the win rate exceeds the target plus tolerance (default ±10%), the losing chromosome is used to weaken the AI. If the win rate falls below the target minus tolerance, the winning chromosome is used to strengthen it. Within the tolerance zone, linear interpolation (Lerp) between the two chromosomes creates a smooth difficulty transition:

activeChromosome = Lerp(losingChromosome, winningChromosome, t),

where t is calculated based on the position within the tolerance range.

According to Tan et al. [2], ADC with learning rate 0.1 and mutation rate 0.1 achieved the best results in nine out of ten evaluation criteria, compared to AUC's seven out of ten. ADC also demonstrated more consistent results across varied opponents and a lower number of drawn games (which are deemed more frustrating than fun), though at the cost of a larger memory footprint (two chromosomes versus one).

4. Implementation

4.1 AI Agent Types

Three distinct AI agent types were implemented for comparative evaluation:

EnemyHybridAgent (ADC): Implements the adaptive difficulty system using the ADC controller. The agent subscribes to chromosome update events via C# delegates (`OnChromosomeUpdated`) and dynamically adjusts parameters through the ApplyChromosomeParameters() method. Two chromosomes (winning and losing) are maintained and updated based on waypoint outcomes.

EnemyStaticAgent: Uses fixed chromosome values corresponding to 0.5. This represents traditional fixed-difficulty game AI and serves as a baseline for comparison.

EnemyRandomAgent: Randomly changes chromosome values at each waypoint without guided learning.

4.2. Training Configuration

Training was conducted using the ML-Agents Toolkit with PPO algorithm. The process consisted of two phases. First, imitation learning was performed where agents learned basic combat behaviors from human demonstrations using the Heuristic() method. Then, reinforcement learning with PPO refined the behaviors through self-play against the trained player AI in the first phase. Training progress was monitored using TensorBoard, with model convergence observed around 40,000 steps.

4.3. Evaluation Metrics

Performance is evaluated based on the following metrics:

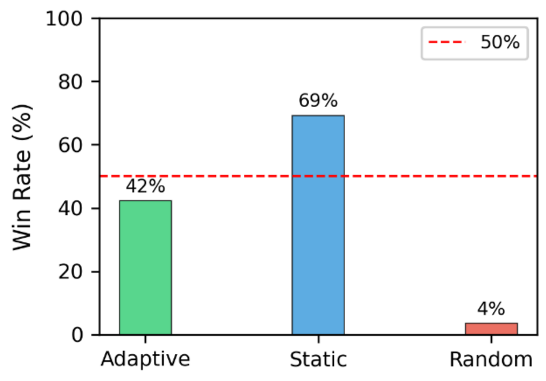

Figure 2. Win rate comparison by AI type

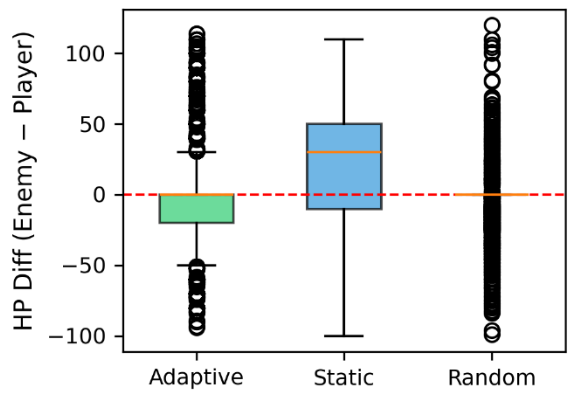

Figure 3. HP difference distribution by AI type. Positive values indicate enemy victory; negative values indicate player victory. The Adaptive AI shows the most balanced distribution centered near zero.

Win Rate: Percentage of games won by each side. The target is 50% for balanced gameplay. Win Rate is calculated as

enemyWins / totalGames × 100

HP Difference is calculated as

|playerHP - enemyHP|

at episode end. Smaller values indicate more competitive matches where neither opponent dominated.
Chromosome Values: Evolution of each gene value over episodes, showing adaptation patterns and convergence behavior for the ADC agent.

4.4 Player Agent

The player agent used in all experiments was trained using imitation learning from demonstrations provided by an experienced player. The player AI learned movement, attack timing, and defensive behavior from

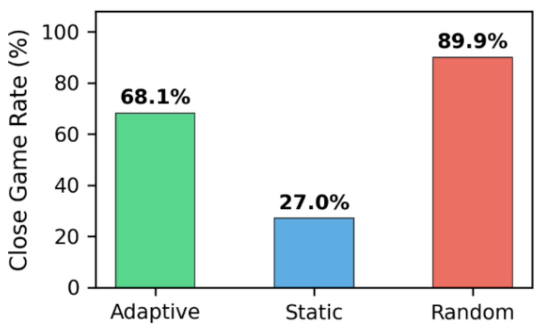

Figure 4. Close game rate by AI type (HP difference ≤ 20).

gameplay recordings. This approach ensured realistic combat patterns.

5. Results

5.1 Win Rate Analysis

Figure 2 presents the win rate comparison across all three AI types. The Adaptive AI using ADC achieved an enemy win rate of 42.3%, which is closest to the target of 50%. The Static AI showed a significantly higher win rate of 69.3%, indicating it was too strong for the opponent. The Random AI had an extremely low win rate of only 3.6%, demonstrating that random parameter changes completely undermine effective gameplay.

5.2 HP Difference

The HP difference at match end serves as an indicator of match competitiveness. Figure 3 shows the distribution of HP differences for each AI type. The Adaptive AI achieved a median HP difference of 10.0 (Mean = 19.5, SD = 23.5), with the distribution centered near zero, indicating balanced matches. The Static AI showed a median of 40.0 (Mean = 42.3, SD = 22.6), heavily skewed toward enemy victories. The Random AI had a median of 0.0 (Mean = 5.6, SD = 12.5), indicating that despite the low win rate, most matches ended with relatively small HP differences.

5.3 Close Game Rate

The close game rate, defined as a game with a difference in HP of 20 or less, indicates a close game and has an impact on game engagement. As shown in Figure 4, the Adaptive AI achieved 68.10% close games, significantly higher than the Static AI's 27.0%. While the Random AI showed 89.9% close games, this high rate is misleading as it primarily reflects consistently narrow losses (with a win rate of only 3.60%), indicating poor gameplay quality despite small HP differences.

5.4 Chromosome Parameter Evolution

Figure 5 shows the distribution of chromosome

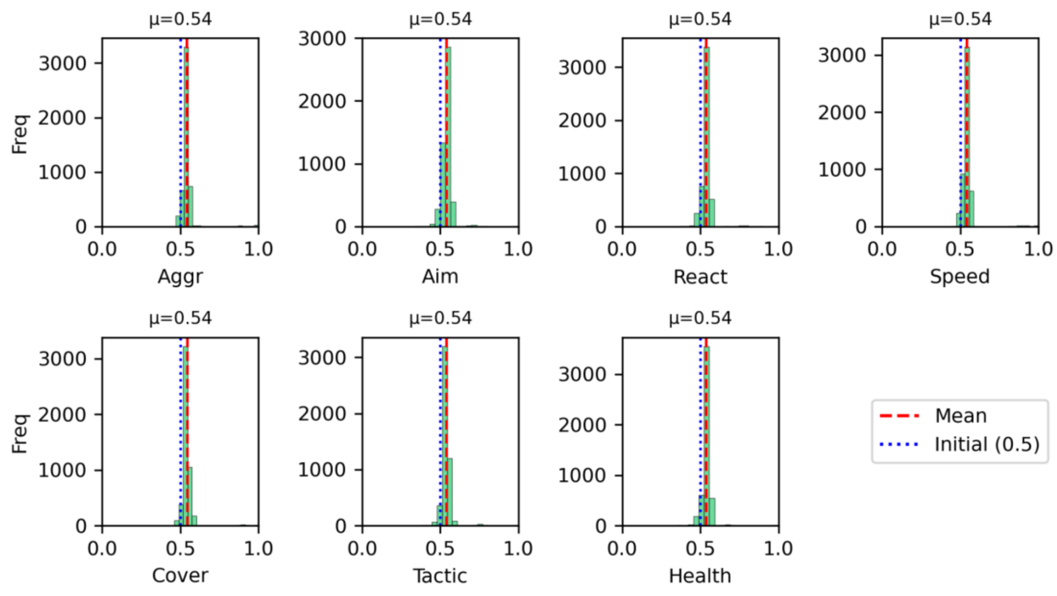

Figure 5. Distribution of chromosome parameters (Adaptive AI).

parameters for the Adaptive AI across all 5,000 episodes. All seven parameters show distributions shifted slightly above the initial value of 0.5, indicating that the ADC learned to strengthen the AI overall to better match the opponent.

6. Conclusion

This paper presented an implementation of the Adaptive Duo-Chromosome Controller (ADC) for dynamic difficulty scaling in a Unity-based 3D action game. Experimental results spanning a total of 15,000 episodes demonstrate that the ADC achieved a win rate of 42.3% and a close match rate of 68.1%, significantly outperforming fixed-parameter and random-parameter approaches. The ADC's dual-chromosome approach provides effective real-time difficulty adjustment, even in action games, resulting in a more engaging gameplay experience. Future work could explore incorporating player feedback for refinement and adding new attack types and movement patterns controlled by additional chromosome parameters.

References

[1] R. Meshram et al., "NPC Behavior in Games Using Unity ML-Agents: A Reinforcement Learning Approach," in Proc. Int. Conf. Automation and Computation (AUTOCOM), 2025, pp. 1519-1523.
[2] C. H. Tan, K. C. Tan, and A. Tay, "Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI," IEEE Trans. Comput. Intell. AI Games, vol. 3, no. 4, pp. 289-301, Dec. 2011.
[3] K. Souchleris, G. K. Sidiropoulos, and G. A. Papakostas, "Reinforcement learning in game industry—review, prospects and challenges," Applied Sciences, vol. 13, no. 4, p. 2443, 2023.
[4] A. Breugelmans, "Implementing machine learning (AI) in game development with Unity," 2021. [Online]. Available: https://www.theseus.fi/handle/10024/496604
[5] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, "Adaptive game AI with dynamic scripting," Mach. Learn., vol. 63, no. 3, pp. 217-248, 2006.
[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347, 2017.
[7] Unity Technologies, "Unity ML-Agents Toolkit Documentation," 2024. [Online]. Available: https://unity-technologies.github.io/ml-agents/