Genetic Algorithms and Reinforcement Learning to Develop Agents for a Fighting Video Game

Manuel Roberto Matera[1,*,†], Pierpaolo Basile[1,*,†]

[1]University of Bari Aldo Moro, Bari, Italy

Abstract

This work focuses on the development of agents for fighting video games, presenting three distinct approaches. The first agent implements a hybrid strategy, structured hierarchically by applying the Genetic Algorithm and the Monte Carlo Tree Search. The second and third agents are based on Linear Q-Learning, but differ in their learning strategies: the second agent requires a training phase, whereas the third one learns online. Regarding the second agent, we investigate two training strategies: one based on Self-play and another based on a Genetic Algorithm, which evolves a population of Reinforcement Learning agents. The third agent is a customized variant of QDagger, a Policy-to-Value Reinforcement Learning method, which uses Monte Carlo Tree Search as its teacher policy. Our main interest is to propose alternative approaches to traditional AI enemy design, and to investigate how such methods are perceived by players. To this end, we conducted a user test in which participants played against the developed agents and evaluated their experience through validated questionnaires. Results reveal a generally positive outcome, with the third agent emerging as the most promising in terms of player engagement.

Keywords

Video game, Artificial Intelligence, Genetic Algorithm, Reinforcement Learning, Imitation Learning

1. Introduction

Fighting video games represent a domain in which racing against time, dealing with unpredictable opponents, and adapting to new strategies are necessary for delivering a rewarding and engaging gaming experience. In traditional game design, AI enemies have relied on rule-based systems, finite-state machines, and predefined scripts [1, 2, 3]. Although effective in creating predictable and manageable behaviours, this approach inherently limits the adaptability and complexity of AI opponents. As a result, enemy actions are often repetitive and easily anticipated by players. Additionally, the static nature of rule-based AI tends to produce imbalanced difficulty levels, with enemies appearing either trivially easy or frustratingly challenging, and thereby worsening the enjoyment of the gaming experience. These limitations motivate the exploration of learning-based alternatives that can generate richer and more varied opponent behaviors. In this work, we investigate how different approaches to agent design affect the player's gameplay experience. Hence, we develop three agents using Genetic Algorithms (GA) [4, 5, 6] and Reinforcement Learning (RL) [7, 8], each offering distinct mechanisms for decision-making. Accordingly, this work investigates whether learning-based approaches constitute viable alternatives to traditional AI enemy design in fighting video games and whether they generate a rewarding and engaging gameplay experience for players. To address these research questions, we assess the effectiveness of these approaches through user evaluations.

Proceedings of AI4HGI '25, the First Workshop on Artificial Intelligence for Human-Game Interaction at the 28th European Conference on Artificial Intelligence (ECAI '25), Bologna, October 25-30, 2025

*Corresponding author.

†These authors contributed equally.

m.matera51@studenti.uniba.it (M. R. Matera); pierpaolo.basile@uniba.it (P. Basile)

https://github.com/matera02 (M. R. Matera); https://swap.di.uniba.it/members/basile.pierpaolo/ (P. Basile)

0009-0004-3286-3908 (M. R. Matera); 0000-0002-0545-1105 (P. Basile)

2. Related Work

Research on agent development for fighting video games often relies on DareFightingICE [9] as the experimental environment. DareFightingICE is a fighting game platform developed by the Intelligent Computer Entertainment Laboratory at Ritsumeikan University in Japan. This Java-based platform serves as both a research tool and an environment for competitions at several conferences, including the IEEE Conference on Computational Intelligence in Games (IEEE CIG). Agents can access data such as position, velocity, health points, and action states, with a response time of 16.66 ms per frame.

The prevailing trend in agent development is the adoption of the Monte Carlo Tree Search (MCTS) algorithm [10, 11, 12]. The use of MCTS for stochastic simulations to evaluate in-game decision making was first demonstrated in [13]. MCTS is valued for its real-time decision-making capabilities, ease of implementation, and consistent performance in competitive scenarios. It was also observed that simulation accuracy improves when an opponent-modelling mechanism is incorporated. Building on this insight, [14] introduced an action-prediction module, in which the agent maintains and continuously updates an action table reflecting the opponent's play patterns, thereby achieving superior performance. Nevertheless, a notable limitation affecting both approaches [13, 14] is the strong dependence on the initial states: in both cases, simulations are initialized with five random actions, which may result in suboptimal action selection that fails to account for strategic game dynamics. To address this limitation, [15] proposed a hierarchical architecture combining GA with MCTS. While this hybrid approach reduces response times and eliminates the need for domain-specific knowledge, it introduces computational overhead and necessitates careful design of the fitness function. Despite these limitations, the method adopted for implementing our first agent follows this hybrid approach.

In the application of Reinforcement Learning (RL) to fighting agent development, there has been a shift towards deep neural architectures. For example, [16] introduced a deep RL framework incorporating a hybrid reward architecture (HRA), in which the overall reward function is decomposed into multiple components and a separate value function is learned for each one; experimental results demonstrate that HRA-based models outperform their non-HRA counterparts. Although this architecture exceeds the complexity our experimental setup can accommodate, its reward decomposition scheme could still help design simpler models, such as those employed for our second and third agents. In [17], a training method combining Self-play and MCTS for deep RL agents is proposed. Given the promising results reported, this methodology will be adopted for training our second agent. Self-play, in particular, offers significant advantages, including the elimination of the need for external data and the ability to enable continuous adaptation. However, it also presents challenges, including convergence to suboptimal strategies and an imbalance between exploitation and exploration, often favouring the former.

While these approaches have demonstrated strong performance, they typically do not fully account for the player's gameplay experience. In this regard, [18] explores a method to implement an agent based on a variant of MCTS designed to follow specific fighting styles. This study aimed to enhance the experience for both players and spectators by generating personalized gameplay tailored to individual viewer preferences. However, experimental results revealed that the agents struggled to accurately replicate certain fighting styles, indicating the need for further improvements in the evaluation functions used to control agent behavior. In [19] a Dynamic Difficulty Adjustment (DDA) system is introduced using two machine learning agents. The first agent learns the player's behavior through Imitation Learning [20], while the second one is trained via Reinforcement Learning to defeat the first. This combination enables the generation of a personalized level of challenge. Although the study was conducted with a small number of participants, it provided a useful reference that led us to evaluate Imitation Learning as a design component for the third agent. Specifically, we selected DAgger (Dataset Aggregation) [21], as this algorithm leverages no-regret online learning principles to enable progressive strategy optimization in response to opponent moves, delivering continuous performance improvement while adapting to unpredictable adversarial behaviors.

In addition to overlooking users' gameplay experience, the aforementioned RL methods rely on computationally expensive models, both in terms of training and inference. Given the simplicity of a 2D fighting video game, such complexity seems unjustified outside of purely academic interest. Therefore,

we chose to adopt the Linear Q-Learning algorithm [22, 23, 24], which offers a favourable trade-off between computational efficiency and generalization.

3. Methodology

This section reports details about all the methodologies exploited by our work for developing intelligent agents in fighting video games.

3.1. GAMCTS Agent: A Hierarchical Integration of Genetic Algorithm and MCTS

The primary challenge in developing agents for fighting video games lies in ensuring response times within 16.6 milliseconds to match the game's frame rate, while efficiently and exhaustively exploring the state space to learn effective strategies A clear understanding of our approach requires a description of the game environment's action space. The environment defines a total of 56 possible actions, 40 of which are actively selectable by the agent. These actions are partitioned into three categories based on their execution context: 15 air actions, 25 ground actions, and a remaining set of special moves. Since agents must predict both their own actions and opponent responses, the total number of possible action pairs amounts to $40 \times 40 = 1600$ combinations. Given the 16ms time constraint, exhaustive simulation of all combinations is computationally infeasible. For this reason, previous approaches [13, 14] attempted to improve MCTS efficiency by assuming that selecting only 5 actions at random would suffice. Since this small subset of actions has a significant impact on determining the actual action to be executed, we now ask what would happen if, instead, more promising actions were considered from the beginning. To investigate this, we adopt a hierarchical approach to action selection [15]. At each decision point, the process works as follows: we first frame the problem as an optimization task, where the GA identifies the five most promising actions according to a given evaluation criterion; then, these actions are passed to the MCTS to determine the actual action to be executed. This hierarchical structure ensures the GA provides a good starting point for MCTS exploration at every step.

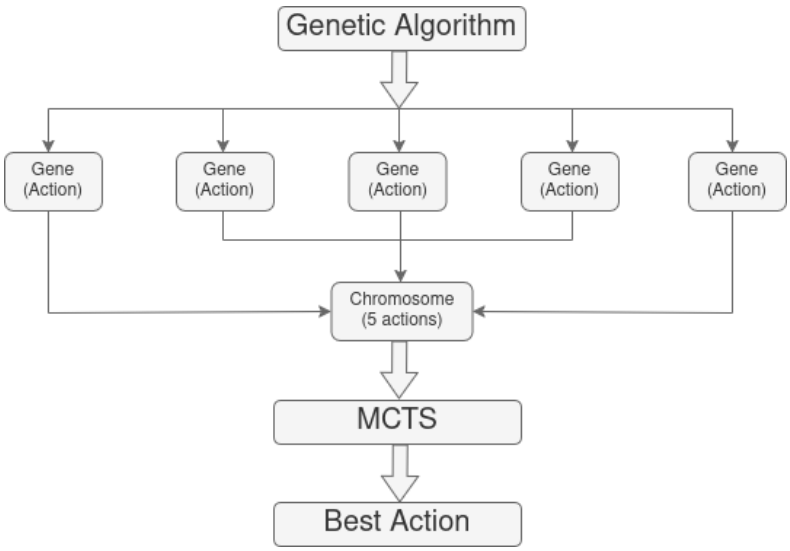

Figure 1: GAMCTS Agent

In Figure 1 we summarize the GAMCTS approach. As shown in Table 1, the selected configuration for MCTS bounds the UCT budget, balancing performance and responsiveness. The UCB1 exploration constant encourages sufficient exploration within the limited UCT iterations, while the node expansion threshold prevents excessive tree growth. The hyperparameter configuration for the GA presented in Table 2 reflects a strategy that prioritizes exploitation over exploration, considering the implications of the Schema Theorem [4, 25]. The reduced population size and limited number of generations represent

Parameter	Value
UCT time limit	16.5 ms
Maximum number of UCT iterations	23
UCB1 exploration constant	3.0
Maximum UCT tree depth	2
Node expansion threshold (UCT)	10
Simulation duration (in frames)	60

Table 1
MCTS Hyperparameters.

Parameter	Value
Gene	Action
Chromosome	Bundle of non-duplicated actions of the same type
Selection method	Tournament selection ($K = 0.95$)
Crossover method	Non-duplicated Two-point Crossover
Population size	10
Chromosome length	5
Mutation probability	0.005
Number of generations	7

Table 2
GA Hyperparameters.

a necessary computational trade-off to ensure algorithm completion within the prescribed temporal frames, though this constrains genetic diversity. The extremely low mutation probability and high tournament selection probability heavily favor exploitation by minimizing random perturbations and systematically selecting higher-fitness individuals. The primary exploratory component resides in the modified two-point crossover configuration, which prevents gene duplication within chromosomes. Despite introducing computational overhead, this constraint ensures that generated action sequences maintain the diversity necessary for effective MCTS simulation. The non-duplicated chromosome structure, which encodes actions of the same type, ensures consistency with the prediction performed by MCTS, by restricting candidate actions to those compatible with the predicted action context. Furthermore, we used tournament selection [26, 27] due to its ease of implementation, noise resistance, and direct control over the selection pressure. To evaluate the quality of candidate solutions, the GA employs a specifically designed fitness function, which is computed from a short-term forward simulation of the game state based on the action sequences encoded in each chromosome. Formally, the function is defined as:

$$Fitness(x) = 2\Delta HP(x) - 1.5D(x) + 10H(x) \tag{1}$$

where x denotes a chromosome, and:

- $\Delta HP(x)$ is the HP difference between the agent and the opponent;
- $D(x)$ is a penalty based on the distance between the agent and the opponent;
- $H(x)$ is the counter of successful hits landed by the agent.

This fitness function characterizes the agent with an offensive gameplay strategy, where chromosomes exhibiting higher fitness values correspond to strategies that favor frequent close-range attacks. The weight coefficients associated with each component were determined through empirical evaluation to achieve the desired strategic emphasis. Furthermore, being linear in its formulation, this function is well-suited for the maximization problem at hand, which requires a concave function formulation [28].

3.2. Reinforcement Learning Agents

3.2.1. Environment Description

To introduce RL agents, it is necessary to define the environment in which they operate and the reward mechanism that guides their learning process. Its observation space and action space define the environment. The observation space can be schematically represented in Table 3 for both the agent and

Player Feature	**Range / Encoding**	**Projectile Feature** (max 2x)	**Range**
Remaining Frames	[0, 1]	Hit Damage	[0, 1]
Energy	[0, 1]	Hit Area X	[-1, 1]
Position X	[-1, 1]	Hit Area Y	[0, 1]
Position Y	[0, 1]		
Speed X	[-1, 1]		
Speed Y	[-1, 1]		
State	One-hot (4)		
Action	One-hot (56)		

Table 3
Environment – Observation space features.

the opponent. We have divided the table into two sections to represent both the information associated with the player and that related to the launched projectiles. Therefore, the observation space size is 144, as the table represents the information for each entity. Since this size affects the agent's performance and several features are either uninformative or deducible, we applied a feature selection mechanism to reduce the dimensionality of the observation space. The selected features comprise 18 elements distributed as follows, with the number of instances indicated in parentheses: air recovery actions for the agent (2), recovery and position-deducible actions for the opponent (7), transition actions to DOWN state for both entities (2), STAND state for both entities (2), AIR state for the agent only (1), Hit Area Y coordinates for agent projectiles and opponent's second projectile (3), and Hit Damage information for the agent (1). In this fashion, we have reduced the size of the observation space from 144 to 126. As for the action space, the same considerations apply as we discussed in Section 3.1.
In addition, the environment provides the agent with a reward, which we have defined as:

$$R = \frac{\Delta \mathrm{HP}_{\mathrm{opp}} - \Delta \mathrm{HP}_{\mathrm{my}}}{C} + B \tag{2}$$

where:

- $\Delta \mathrm{HP}_{\mathrm{opp}} = \mathrm{HP}_{\mathrm{opp}}^{\mathrm{pre}} - \mathrm{HP}_{\mathrm{opp}}^{\mathrm{now}}$ is the decrease in the opponent's health points.
- $\Delta \mathrm{HP}_{\mathrm{my}} = \mathrm{HP}_{\mathrm{my}}^{\mathrm{pre}} - \mathrm{HP}_{\mathrm{my}}^{\mathrm{now}}$ is the decrease in the agent's health points.
- B is a bonus: $B = +0.01$ if the distance to the opponent has decreased, -0.01 if it has increased.
- The constant $C = 10$ is used for normalization.

This function is inspired by the reward defined in [16]. $\Delta \mathrm{HP}_{\mathrm{opp}}$ represents the reward component that favors the offensive strategy, while $\Delta \mathrm{HP}_{\mathrm{my}}$ values the defensive strategy. In [16], both components are added up to reward both strategies. In our approach, however, we subtract rather than add the defensive component from the offensive one, creating a reward that evaluates the net outcome of combat exchanges. This formulation rewards the agent when its offensive gains exceed its defensive losses, thus encouraging agents to seek favorable combat trade-offs. To prevent excessive reward values, we normalize the difference in the numerator using a constant, denoted as C, determined through empirical testing. Additionally, to encourage proactive engagement with the opponent, we include a bonus factor B that rewards movement toward the opponent. This bonus component provides the primary incentive for offensive positioning and active combat engagement.

3.2.2. Base Model: Q-Learning with Linear Function Approximation

The Reinforcement Learning problem is modeled as a Markov Decision Process (MDP) defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ [29], where $\mathcal{S}$ is the state space, $\mathcal{A}$ the action space, $\mathcal{P}(s'|s,a)$ the transition probabilities, $\mathcal{R}(s,a)$ the reward function, and $\gamma \in [0,1)$ the discount factor. In our environment, the state space $\mathcal{S} \subseteq \mathbb{R}^{126}$ is a 126-dimensional continuous observation space, while the action space $\mathcal{A} = \{1, \dots, 56\}$ consists of 56 discrete actions and the reward function is defined in Equation 2.
The objective in this MDP framework is to learn the optimal action-value function $Q^*(s,a)$, which represents the expected cumulative discounted reward obtained by taking action a in state s and subsequently following the optimal policy. To handle the large dimensionality of the state-action space, the RL agents employed in this study adopt Q-Learning with Linear Function Approximation as their base model [22, 23, 24]. This approach enables generalization across similar states and actions by representing the Q-function as $Q(s,a;\mathbf{w}) = \mathbf{w}^\top \phi(s,a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector and $\phi(s,a) \in \mathbb{R}^d$ a sparse feature vector encoding state-action pairs with $d = 126 \times 56 = 7056$. The weight vector $\mathbf{w}$ is updated using the standard Q-Learning update rule adapted for linear function approximation [7]. An ϵ-greedy policy is adopted for action selection [30, 7].

Parameter	Value
Discount factor γ	0.95
Learning rate α	0.03
Exploration rate ϵ	0.1

Table 4
Linear Q-Learning Hyperparameters

Table 4 lists the hyperparameters used for Linear Q-Learning. The configuration was selected through preliminary experimentation to ensure reliable convergence and effective performance in the multi-agent fighting environment. The discount factor encourages long-term reward maximization while slightly favoring immediate rewards, contributing to maintaining stability during training. The learning rate was chosen to allow consistent progress without introducing excessive variance in the updates. The exploration rate was set to a commonly used value that balances the exploitation of learned behaviors with sufficient exploration, typically achieved with values in the range $[0.05, 0.2]$ [31].

3.2.3. Two Stage Training with MCTS and Self-play

In the simplest case, the agent could be trained against an opponent that performs random actions. However, this quickly becomes ineffective due to the opponent's predictable behavior, as its actions are discrete and independent. To address this limitation, we employ an MCTS-based opponent that provides strategic behavior through systematic exploration of the state space. While this approach doesn't eliminate the risk of convergence to a local optimum, it offers a more robust foundation for an initial training stage than the previous method. To further improve the agent, our methodology combines MCTS and Self-play [32] training in a structured two-stage approach, drawing from the framework presented in [17]. This combination leverages the complementary strengths of both methods: MCTS provides diverse strategic exposure while Self-play identifies and addresses strategic weaknesses.
Figure 2 provides an overview of the proposed training procedure.
During the first stage, the agent is trained exclusively against an MCTS-based opponent over 500 episodes to establish a baseline level of competency. The MCTS algorithm's broad exploration through simulations exposes the agent to diverse scenarios, enabling effective action filtering through move masking and the development of an accurate value function. After 300 episodes, the learning rate α decreases from 0.03 to 0.01 to facilitate initial rapid learning followed by strategic refinement.
In the second stage, which spans an additional 2,000 episodes, the training alternates between the MCTS opponent and Self-play copies in a 1:3 ratio (MCTS:Self-play). This ratio ensures generalization through MCTS exposure while allowing Self-play to identify and eliminate strategic vulnerabilities.

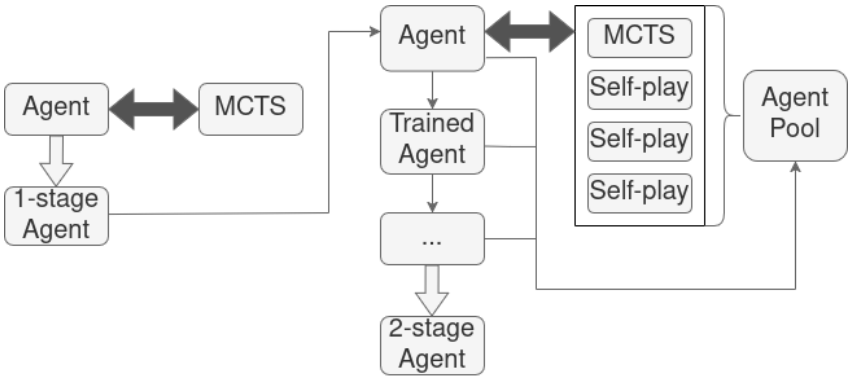

Figure 2: Two-stage Training with Self-play.

Unlike the referenced approach, we expand the agent pool every 500 episodes regardless of win rate, maintaining the fixed ratio due to hardware constraints that prevent parallel environment execution. Each training episode corresponds to a complete match against the designated opponent type. The total estimated training time is approximately 42 hours. To encourage goal-directed behavior, the agent receives win/loss bonuses of $\pm$ 10, supplementing the standard reward function and addressing aspects not explicitly captured by the environment's reward architecture. Self-play agents in the Agent Pool are updated only when the training agent's average reward over a sliding window of five episodes exceeds the previous best by at least 5%. A key limitation of our approach is the risk of convergence to suboptimal solutions, due to the generalization introduced by the linear approximation of $Q(s, a)$ [33]. Unlike in tabular Q-Learning, where state values are independent and local optima are also global, the linear function introduces dependencies between states, so improving performance in one may worsen it in others. Its performance improves when there is a way to escape such local optima [33].

3.2.4. An Evolutionary Approach to Training RL Agents

A common practice for escaping local optima involves stochastic search algorithms based on population methods, where multiple agents are trained in parallel and the best performer is selected based on cumulative reward. This leads to the adoption of Evolutionary Reinforcement Learning (EvoRL), which integrates Evolutionary Computation with RL [34].

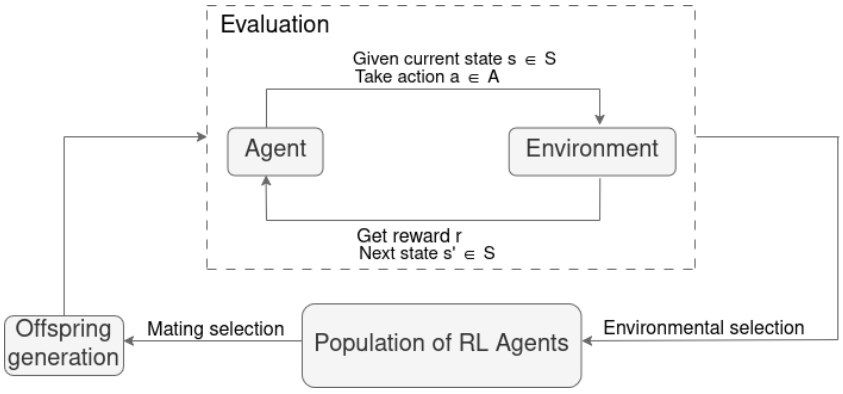

Figure 3: A simple and general framework of EvoRL.

We illustrate a simple and general framework of EvoRL in Figure 3.
It consists of two loops: the outer loop governs the EC process, while the inner loop represents agent–environment interactions in RL. Initially, a population of candidate solutions is randomly generated. Offspring candidate solutions are then generated from parents via variation. Each offspring is evaluated by performing a RL task to obtain its fitness value. A new population is selected for the next

iteration by replacing the entire current population with the offspring generated through recombination and mutation. While this is a basic example, EvoRL encompasses several research areas. Among them, Policy Search aims to find policies that maximize cumulative reward. One technique that can be adopted within this context is Neuroevolution [35], which evolves neural network weights and architectures without relying on gradients. Research, such as [36], demonstrates this integration, utilizing GA to evolve a population of neural networks, each represented by its weights.
Inspired by this approach, we developed an alternative training methodology for our agent. By evolving a population of RL agents, our goal is to discover the optimal weights for the linear approximation of $Q(s, a)$. In essence, we reformulate the training process as an optimization problem, where the objective is to find a weight configuration that maximizes the cumulative reward. We adopt a GA where each chromosome is encoded by the weights of a corresponding RL agent. Moreover, this approach removes the need for an initial training phase to generate a baseline experience for the agent, as we start from a population of agents, each initialized with a random configuration of weights. Due to limited computational resources, the approach was not parallelized, and each agent in the population is trained sequentially for a fixed number of episodes. Agents that achieve higher fitness, measured as the cumulative reward over their available episodes, are considered better candidates and are therefore more likely to survive and propagate their parameter configurations to the next generation. This evolutionary mechanism enables the exploration of multiple weight configurations, allowing the algorithm to escape suboptimal solutions.

Parameter	**Value**
Individual	RL Agent
Selection method	Tournament selection ($K = 0.95$)
Crossover method	Two-point Crossover
Population size	10
Tournament size	3
Mutation probability	0.01
Episodes per individual	5
Episodes per generation	50
Number of generations	10

Table 5
GARL Hyperparameters.

In Table 5, we report the parameters used for the GA employed in training, referred to as GARL. The parameters were selected empirically. A tournament size of 3 promotes a moderate level of elitism, avoiding excessive selection pressure. The number of episodes per individual was set to balance learning speed and computational cost, limiting the total number of episodes per generation. The training required 500 episodes in total, with an estimated time of approximately 8 hours. While the implementation could benefit from parallelization, this training approach proved more efficient, requiring only about 19% of the time needed by the previous method. Since the agent trained with this methodology outperformed the one trained with the previously discussed approach in preliminary tests, we selected it as our second agent for user evaluation and will refer to it as RLAgent.

3.2.5. Cutting Training Time with QDagger

Since training an RL agent from scratch is computationally expensive, we aim to find a method that can accelerate this process. To address this, we adopt Policy-to-Value Reinforcement Learning (PVRL), which transfers a suboptimal policy to a value-based agent, enabling efficient training regardless of the original model. A proper PVRL algorithm must satisfy three key desiderata: teacher-agnosticism, ensuring that the student does not depend on the teacher's architecture or learning algorithm; weaning support, involving a progressive reduction of the student's reliance on the teacher policy as training proceeds; and computational efficiency, meaning that this method must incur lower cost than training

from scratch. Hence, our third agent is inspired by the QDagger algorithm [37], which combines DAgger [21] with n-step Q-learning. Building on this approach, we propose a modified variant tailored to our specific setting with linear function approximation. The proposed approach differs from [37] in two key aspects: it employs Linear Q-Learning and uses a mean squared error loss (MSE) between the teacher and student Q-value vectors for policy distillation [38]. Furthermore, given the absence of prior data in our setting, we operate exclusively in online mode, bypassing the first training phase outlined in [37]. The loss function combines temporal difference learning with policy distillation:

$$\mathscr{L}_{\text{QDagger}}(D; \mathbf{w}_S) = \underbrace{\mathscr{L}_{\text{TD}}(D; \mathbf{w}_S)}_{\text{TD Loss}} + \underbrace{\lambda_t \cdot \mathscr{L}_{\text{MSE}}(D; \mathbf{w}_S)}_{\text{Distillation Loss}} \tag{3}$$

where D represents a generic replay buffer, $\mathbf{w}_S$ are the student's weights and λ_t is the distillation coefficient at time step t. To collect expert data we employ a buffer defined as $\mathscr{D}_{\text{Expert}} = \{(s_i, \mathbf{q}_i^{\pi_T})\}_{i=1}^{B}$ where s_i is the observed state, $\mathbf{q}_i^{\pi_T}$ is the Q-value vector estimated by the expert policy for state s_i, and $B = 3$ is the buffer size, with FIFO replacement upon reaching capacity. At each update iteration, the agent applies a standard Q-learning update, inserts the current tuple into the buffer, and then updates the weights by minimizing the MSE loss over all buffer entries, while ensuring teacher-agnosticism and computational efficiency. To provide support for weaning, we introduce the distillation coefficient λ_t, which allows QDagger to deviate from the teacher's suboptimal policy π_T, rather than converging toward it. We selected MCTS as our teacher policy because it is applicable to MDPs [12]. The absence of prior domain knowledge precluded training a classifier-based policy, while MCTS offers several advantages within fighting game environments that make it particularly suitable as a heuristic policy. As suggested in [37], the distillation coefficient can be decayed linearly over time to allow for a smooth transition from reliance on the expert policy to student autonomy. In our case, we instead opted for exponential decay to accelerate this transition, updating it at each time step t as $\lambda_{t+1} = \max(\lambda_t \cdot \lambda_{\text{decay}}, 10^{-3})$ with $\lambda_{\text{decay}} = 0.99$. This choice is motivated by both the teacher's reliability and the desire to minimize the risk of overfitting to it. To preserve a minimal influence from the teacher, we set a lower bound of 10^{-3}, ensuring that λ_t does not fall below this threshold.

3.3. Experimental Details

All experiments were carried out in a consistent hardware configuration to ensure reliability of performance in all testing scenarios. The experimental setup is based on an Intel Core i7-8650U processor with 16.0 GB of RAM. All algorithms are implemented in Java from scratch.

Table 6 reports the win rates from 11 rounds of matches between agents. Since they use different reward/fitness functions and observation spaces, these results provide a basis for comparing their relative efficiency. The MCTS-based agent is used as a baseline.

Agent A	Agent B	Win Rate A	Win Rate B
GAMCTS	MCTSAI	90.9%	9.1%
GAMCTS	RLAGENT	72.7%	27.3%
GAMCTS	QDAGGER	81.8%	18.2%
RLAGENT	MCTSAI	63.6%	36.4%
RLAGENT	QDAGGER	54.5%	45.5%
QDAGGER	MCTSAI	54.5%	45.5%

Table 6
Win rates from matches between agents.

4. Evaluation

To evaluate the three developed agents (GAMCTS, RLAgent, and QDagger), we conducted a user test involving 20 participants. Each participant played against all three agents in randomized order,

which helped minimize potential bias and ensured that no agent was unintentionally advantaged or disadvantaged. This procedure allowed us to obtain more reliable and objective assessments of both the gameplay experience and the players' perception of each agent. During the study, each participant completed three game sessions, each consisting of three rounds against one of the randomly assigned agents. After every session, participants were asked to complete a questionnaire evaluating their experience. Prior to testing, a short tutorial was provided to familiarize them with the game controls and mechanics.

4.1. Questionnaire

The questionnaire was structured in two sections: the first collected personal information about participants, while the second focused on evaluating their gameplay experience.

4.1.1. Personal information

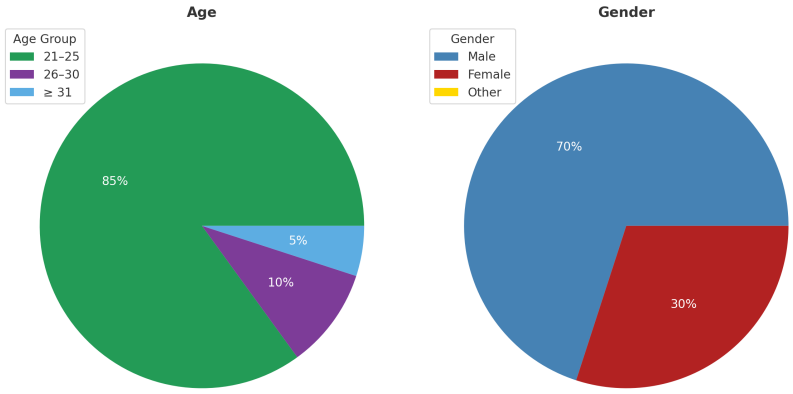

Figure 4: Age and gender of participants.

In the first section, participants were asked to provide their full name optionally, while age and gender were mandatory fields. A summary of this data is presented in Figure 4. A more balanced distribution of participants across age groups and genders would have been preferable for these tests, but this was not feasible due to the difficulty in recruiting volunteers. It is worth noting that, in general, most of the testers were not familiar with this video game genre. This does not undermine the validity of our study; on the contrary, it aligns with one of our primary objectives: to develop agents that can dynamically adapt to the player's skill level. Such adaptability ensures a challenging and engaging experience for both players with prior experience in fighting video games and those new to this genre.

4.1.2. Game Experience Questionnaire

The second part of the questionnaire focuses on assessing the participants' gameplay experience. To this end, we employed the Game Experience Questionnaire (GEQ) [39], a validated instrument commonly used in academic research to measure players' experience both during and after gameplay.
The GEQ consists of three modules: Core Module, Social Presence Module, and Post-Game Module. In our study, only the Core Module and Post-Game Module were employed. In both modules, participants were required to evaluate a set of items reflecting their emotional states, using the following scale:

not at all	**slightly**	**moderately**	**fairly**	**extremely**
0	1	2	3	4
⟨ ⟩	⟨ ⟩	⟨ ⟩	⟨ ⟩	⟨ ⟩

The Core Module is designed to assess how the participant felt during gameplay by averaging the scores assigned to items associated with the following seven components: Immersion, Flow, Competence, Positive and Negative Affect, Tension, and Challenge.
The Post-Game Module provides an assessment of how players felt after finishing the game, by averaging the scores assigned to items related to the following four components: Positive Experience, Negative Experience, Tiredness, and Returning to reality.

4.2. Results and Discussion

	GAMCTS	RLAgent	QDagger
Competence	1.51 (**1.16**)	1.96 (1.08)	**2.53** (0.71)
Sensory	1.84 (1.00)	1.92 (**1.01**)	**2.01** (0.96)
Flow	**1.99** (**1.02**)	1.91 (0.97)	1.96 (0.98)
Tension	**1.27** (**1.22**)	1.03 (1.04)	0.60 (0.75)
Challenge	**2.21** (0.94)	1.90 (**0.97**)	1.74 (0.83)
Affect-	**0.93** (0.71)	0.88 (**0.75**)	0.35 (0.42)
Affect+	2.52 (**0.92**)	2.60 (0.73)	**2.90** (0.68)

Table 7
Core Module scores

	GAMCTS	RLAgent	QDagger
Experience+	1.62 (**1.05**)	2.07 (1.04)	**2.39** (0.75)
Experience-	0.53 (0.50)	**0.67** (**0.58**)	0.50 (0.39)
Tiredness	**1.23** (**1.30**)	0.85 (1.08)	0.80 (1.22)
Returning to Reality	**0.52** (0.50)	0.30 (0.43)	0.45 (**0.58**)

Table 8
Post-Game Module scores

	(GAMCTS, RLAgent)	(GAMCTS, QDagger)	(RLAgent, QDagger)
Competence	-1.33 [1.000]	-4.58 [**0.001**]	-2.00 [0.361]
Sensory	-0.30 [1.000]	-1.20 [0.487]	-0.38 [1.000]
Flow	0.30 [1.000]	0.16 [0.872]	-0.18 [1.000]
Tension	0.72 [1.000]	2.60 [0.087]	1.41 [0.871]
Challenge	1.43 [1.000]	2.13 [0.141]	0.67 [1.000]
Affect-	0.25 [1.000]	3.66 [**0.010**]	2.97 [0.055]
Affect+	-0.30 [1.000]	-2.27 [0.140]	-1.36 [0.871]

Table 9
Paired t-test results for Core Module scores.

	(GAMCTS, RLAgent)	(GAMCTS, QDagger)	(RLAgent, QDagger)
Experience+	-1.58 [0.480]	-3.34 [**0.014**]	-1.09 [0.863]
Experience-	-1.02 [0.480]	0.18 [1.000]	1.41 [0.703]
Tiredness	1.44 [0.480]	1.44 [0.499]	0.17 [0.864]
Returning to Reality	1.63 [0.480]	0.46 [1.000]	-0.94 [0.863]

Table 10
Paired t-test results for Post-Game Module scores.

Tables 7 and 8 report the scores obtained for the Core Module and Post-Game Module, respectively, where we use the following abbreviated notation: Mean (Standard Deviation). We report the highest mean and standard deviation values in bold. To assess statistical significance between agents' evaluations, we conducted paired t-tests for each GEQ Core Module and Post-Game Module component across all agent

pairs. Results appear in Tables 9 and 10, showing t-statistics with 19 degrees of freedom and p-values in brackets. To account for multiple comparisons, p-values were adjusted using the Holm-Bonferroni method. Statistically significant p-values ($p < 0.05$) are shown in bold.
The reported scores reveal distinct trends across the three agents, with QDagger receiving higher ratings in terms of user engagement, GAMCTS being perceived as more difficult, and RLAgent scoring between the two extremes. We now turn our attention to statistically significant differences observed between agents. In the Core Module, QDagger significantly outperformed GAMCTS in Competence, indicating that participants felt more skilled when playing against it. In contrast, no significant differences were found between RLAgent and the other two agents. Tension scores across all agents were close to 1, suggesting that tension was perceived as slight rather than moderate, which represents a favorable outcome for all tested agents. QDagger led to significantly lower Negative Affect compared to GAMCTS, indicating that the gameplay experience was not associated with negative emotions. It is worth noting that all Negative Affect scores remain below 1, with QDagger's score approaching zero. For other components, no statistically significant differences were observed. Turning to the Post-Game Module, QDagger achieved significantly higher scores in Positive Experience compared to GAMCTS, in line with the trend suggested by the Core Module scores. Differences in Negative Experience, Tiredness, and Returning to Reality were not statistically significant; however, the consistently low scores reported across all agents indicate that participants generally did not experience strong negative feelings after gameplay, such as guilt or a sense of wasted time, nor did they report difficulty transitioning back to reality.

5. Conclusions and Future Works

We present several strategies for developing AI Agents for a fighting video game, considering a balance between performance and responsiveness. The outcomes offer valuable insights into how participants experienced the gameplay with each agent, which was generally perceived as engaging.
In order to answer our research questions more conclusively, it is important to acknowledge that the absence of a traditional AI baseline limits the strength of our claims. A rule-based agent could serve as such a baseline, although defining effective rules is a nontrivial challenge that initially motivated our focus on learning-based alternatives. Including a rule-based baseline in future work would help better evaluate the benefits of learning-based methods. The proposed methods provide directions for future research on adaptive game AI. For example, GAMCTS could achieve adaptability through the implementation of a difficulty balancing mechanism that strategically selects suboptimal solutions rather than choosing the best ones within the GA. Similarly, RL-based agents could be developed to modify their behavioral patterns dynamically through reward systems that reflect real-time player interactions. In addition, several other directions are being considered. We aim to explore the development of parallel solutions to enhance the performance of GAs employed in our study. Regarding RLAgent, we are interested in investigating its performance if equipped with an online learning mechanism similar to that of QDagger. For QDagger, the idea is to start with a pre-trained policy in a phase preceding the fighting against this agent. In other words, sufficient gameplay data would be collected, for example, by having the user play against other agents. Based on this data, a policy would be trained, which would then be exploited during actual gameplay. These research directions collectively address the need for more sophisticated adaptive AI systems in fighting video games while maintaining the essential balance between computational performance and real-time responsiveness that defines engaging gameplay experiences.

Acknowledgments

We acknowledge the support of the PNRR project FAIR - Future AI Research (PE00000013), Spoke 6 - Symbiotic AI (CUP H97G22000210007) under the NRRP MUR program funded by the NextGenerationEU.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly and Claude Sonnet 4 to improve the writing style. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] D. M. Bourg, G. Seemann, AI for Game Developers, O'Reilly Media, Inc., 2004.

[2] B. Schwab, Ai Game Engine Programming (Game Development Series), Charles River Media, Inc., USA, 2004.

[3] I. Millington, J. Funge, Artificial Intelligence for Games, Second Edition, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[4] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, MIT Press, Cambridge, MA, USA, 1992.

[5] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, USA, 1998.

[6] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, 1996. URL: https://doi.org/10.1093/oso/9780195099713.001.0001. doi:10.1093/oso/9780195099713.001.0001.

[7] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, second ed., The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[8] C. Szepesvari, Algorithms for Reinforcement Learning, Morgan and Claypool Publishers, 2010.

[9] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, R. Thawonmas, Fighting game artificial intelligence competition platform, in: 2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE), IEEE, 2013, pp. 320–323. doi:10.1109/GCCE.2013.6664844.

[10] L. Kocsis, C. Szepesvári, Bandit based monte-carlo planning, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Machine Learning: ECML 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 282–293.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of monte carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in Games 4 (2012) 1–43. doi:10.1109/TCIAIG.2012.2186810.

[12] M. Swiechowski, K. Godlewski, B. Sawicki, J. Mandziuk, Monte carlo tree search: A review of recent modifications and applications, CoRR abs/2103.04931 (2021). URL: https://arxiv.org/abs/2103.04931. arXiv:2103.04931.

[13] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada, R. Thawonmas, Application of monte-carlo tree search in a fighting game ai, in: 2016 IEEE 5th Global Conference on Consumer Electronics, 2016, pp. 1–2. doi:10.1109/GCCE.2016.7800536.

[14] M.-J. Kim, K.-J. Kim, Opponent modeling based on action table for mcts-based fighting game ai, in: 2017 IEEE Conference on Computational Intelligence and Games (CIG), 2017, pp. 178–180. doi:10.1109/CIG.2017.8080432.

[15] M.-J. Kim, C. W. Ahn, Hybrid fighting game ai using a genetic algorithm and monte carlo tree search, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 129–130. URL: https://doi.org/10.1145/3205651.3205695. doi:10.1145/3205651.3205695.

[16] Y. Takano, W. Ouyang, S. Ito, T. Harada, R. Thawonmas, Applying hybrid reward architecture to a fighting game ai, in: 2018 IEEE Conference on Computational Intelligence and Games (CIG), 2018, pp. 1–4. doi:10.1109/CIG.2018.8490437.

[17] D.-W. Kim, S. Park, S.-i. Yang, Mastering fighting game using deep reinforcement learning with self-play, in: 2020 IEEE Conference on Games (CoG), 2020, pp. 576–583. doi:10.1109/CoG47356.2020.9231639.

[18] R. Ishii, S. Ito, M. Ishihara, T. Harada, R. Thawonmas, Monte-carlo tree search implementation of fighting game ais having personas, in: 2018 IEEE Conference on Computational Intelligence and Games (CIG), 2018, pp. 1–8. doi:10.1109/CIG.2018.8490367.
[19] R. Fuchs, R. Gieseke, A. Dockhorn, Personalized dynamic difficulty adjustment – imitation learning meets reinforcement learning, 2024. URL: https://arxiv.org/abs/2408.06818. arXiv:2408.06818.
[20] A. Hussein, M. M. Gaber, E. Elyan, C. Jayne, Imitation learning: A survey of learning methods, ACM Comput. Surv. 50 (2017). URL: https://doi.org/10.1145/3054912. doi:10.1145/3054912.
[21] S. Ross, G. J. Gordon, J. A. Bagnell, No-regret reductions for imitation learning and structured prediction, CoRR abs/1011.0686 (2010). URL: http://arxiv.org/abs/1011.0686. arXiv:1011.0686.
[22] C. J. C. H. Watkins, et al., Learning from delayed rewards, 1989.
[23] C. Watkins, P. Dayan, Technical note: Q-learning, Machine Learning 8 (1992) 279–292. doi:10.1007/BF00992698.
[24] L. C. Baird, Residual algorithms: reinforcement learning with function approximation, in: Proceedings of the Twelfth International Conference on International Conference on Machine Learning, ICML'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, p. 30–37.
[25] A. Eiben, C. Schippers, On evolutionary exploration and exploitation, Fundam. Inform. 35 (1998) 35–50. doi:10.3233/FI-1998-35123403.
[26] D. E. Goldberg, K. Deb, A comparative analysis of selection schemes used in genetic algorithms, in: Foundations of Genetic Algorithms, 1990. URL: https://api.semanticscholar.org/CorpusID:938257.
[27] B. L. Miller, D. E. Goldberg, Genetic algorithms, tournament selection, and the effects of noise, Complex Systems 9 (1995) 193–212. URL: https://www.complex-systems.com/abstracts/v09_i03_a02/.
[28] S. Lenhart, J. T. Workman, Optimal Control Applied to Biological Models, 1st ed., Chapman and Hall/CRC, 2007. doi:10.1201/9781420011418.
[29] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st ed., John Wiley & Sons, Inc., USA, 1994.
[30] T. Lattimore, C. Szepesvári, Bandit Algorithms, Cambridge University Press, 2020.
[31] T. Miller, Mastering Reinforcement Learning, The University of Queensland, 2024. URL: https://gibberblot.github.io/rl-notes/index.html. doi:10.14264/4bf1412.
[32] A. Samuel, Some studies in machine learning using the game of checkers. ii—recent progress, IBM Journal of Research and Development 3 (2000) 206 – 226. doi:10.1147/rd.441.0206.
[33] D. L. Poole, A. K. Mackworth, Artificial Intelligence: Foundations of Computational Agents, 3 ed., Cambridge University Press, 2023.
[34] H. Bai, R. Cheng, Y. Jin, Evolutionary reinforcement learning: A survey, Intelligent Computing 2 (2023). URL: http://dx.doi.org/10.34133/icomputing.0025. doi:10.34133/icomputing.0025.
[35] K. O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evolutionary Computation 10 (2002) 99–127. doi:10.1162/106365602320169811.
[36] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, J. Clune, Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, CoRR abs/1712.06567 (2017). URL: http://arxiv.org/abs/1712.06567. arXiv:1712.06567.
[37] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, M. G. Bellemare, Reincarnating reinforcement learning: Reusing prior computation to accelerate progress, 2022. URL: https://arxiv.org/abs/2206.01626. arXiv:2206.01626.
[38] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, R. Hadsell, Policy distillation, 2016. URL: https://arxiv.org/abs/1511.06295. arXiv:1511.06295.
[39] W. IJsselsteijn, Y. de Kort, K. Poels, The Game Experience Questionnaire, Technische Universiteit Eindhoven, 2013.

A. Online Resources

- Fighting Game AI repo: https://github.com/matera02/Fighting-Game-AI
- DareFightingICE website: https://www.ice.ci.ritsumei.ac.jp/~ftgaic/index.htm
- DareFightingICE repo: https://github.com/TeamFightingICE/FightingICE