Learning Levels of Mario AI Using Genetic Algorithms

Team Members:
Ferran Mirabent, Carlos Bellanco, Roger Pieres,
Martí Lorente, Marc Martínez
Optimization

January 19, 2026

1 Motivation

We selected this paper because we wanted to dive deep into our recent coursework on Genetic Algorithms while combining it with our passion for videogames. The research focuses on the Mario AI Championship benchmark, but instead of training a complex reactive agent, it treats level completion as an optimization problem of finding the perfect timed sequence of actions. This approach allows us to focus deeply on the mechanics of the Genetic Algorithm without getting bogged down in the complexities of real-time vision processing or state-space dimensionality. We would additionally like to explore a Deep Learning approach as an alternative solution.

2 Introduction to the Optimization Problem

The objective is to maximize the score obtained in the Learning Track of *Infinite Mario Bros*, the level used for the Mario AI Championship. Unlike standard gameplay where an agent reacts to the environment, this approach seeks a pre-defined sequence of actions that maximizes the fitness function S (Score).

The constraints and conditions of the problem are:

- **Search Space:** The agent must determine a sequence of inputs (actions) for every time tick/frame of the game.
- **Computational Limit:** The agent is allowed a maximum of $N = 10,000$ games (evaluations) to learn the level.
- **Input Constraints:** The controller simulates a D-Pad (Left, Right, Up and Down) and two buttons (A and B). While there are $2^6 = 64$ combinations, domain knowledge reduces this to 22 feasible distinct actions (e.g., pressing Left and Right simultaneously is impossible, pressing Up does nothing, etc).

The objective function to maximize is the **Score (S)**, defined by the Championship rules:

$$S = D + 64d_f + 58d_m + 58d_{gm} + 42k + 12k_{st} + 17k_{sh} + 4k_f + 1024s + 32m + 24b_h + 16c + 8t' \quad (1)$$

Where:

- D: Physical distance travelled.
- s: Final status (1 if level completed, 0 otherwise).
- t': Time remaining.

- k, k_{st}, k_{sh}, k_f: Enemies killed (total and by specific method).
- d_f, d_m, d_{gm}: Items collected (flowers, mushrooms, green mushrooms).
- c, b_h: Coins collected and hidden blocks found.
- m: Mario's final mode (Small, Big, Fire).

3 Explanation of the Used Algorithms

The authors propose a Genetic Algorithm (GA) to evolve the sequence of actions. The approach is divided into two stages: a naive stage using domain-independent operators and a refined stage using domain-specific knowledge.

3.1 Encoding

The chromosome represents the sequence of actions Mario performs throughout the level.

- **Genes:** Each gene is an integer representing a specific combination of buttons pressed. In the first stage, the search space is reduced to 11 actions (assuming the Run button is always pressed). In the second stage, all 22 feasible actions are used.
- **Chromosome Length:** The length is fixed at 3,000 genes. This derives from the maximum level time (200 seconds) discretized into 15 ticks/frames per second ($200 \times 15 = 3000$).

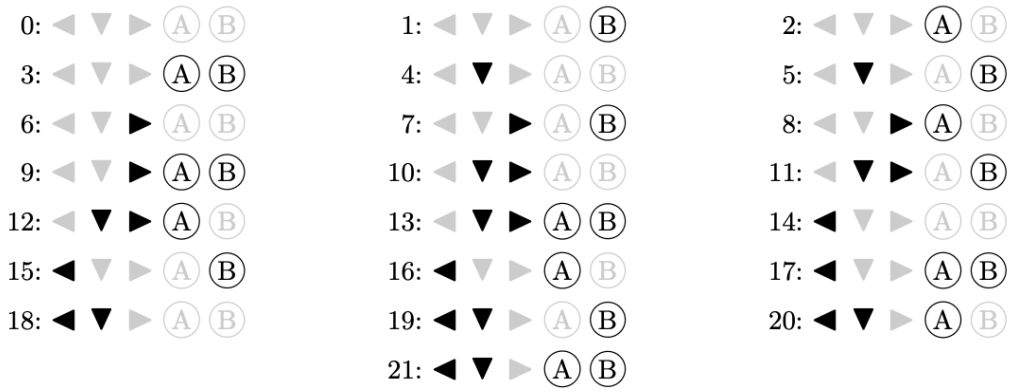

Figure 1: List of actions along with the pressed buttons. [1]

3.2 Genetic Operators

The algorithm uses Tournament Selection, Crossover, and Mutation, tailored differently in the two experimental stages.

3.2.1 Initialization

- **Naive:** Actions are chosen randomly.
- **Hybrid (Domain-Dependent):** Recognizing that moving right is the primary goal, a "guided" initialization favours *Right* and *Right+Jump* actions. The hybrid approach randomly selects between naive and guided initialization for each gene.

3.2.2 Selection

Tournament selection is used, where a subset of individuals (T_s) compete, and the one with the highest fitness becomes a parent. This process is repeated until we have a number of parents equivalent to the population so we can take them in pairs during crossover. A parent might be repeated multiple times if it wins multiple tournaments. The domain-dependent stage adds elitism to preserve the best solutions.

3.2.3 Crossover

- **Single-Point Crossover:** A random point n ($n < 3000$) is selected. Offspring are created by splicing the parents' action sequences at n.
- **Domain-Aware Crossover:** The cut point n is not purely random. The algorithm searches for a point where Mario's physical position (x, y) is similar in both parent simulations (Euclidean distance $< \Delta$). This ensures continuity; if one parent plays well up to point A and the other plays well from point A onwards, splicing them at A (where Mario is at the same location) creates a viable trajectory.

3.2.4 Mutation

- **Random Mutation:** Modifies M genes randomly across the chromosome.
- **Windowed Mutation (Domain-Dependent):** Mutations are targeted specifically at the *end* of the effective action sequence (just before Mario dies or the level ends). This exploits the fact that early actions are likely "correct" if Mario reached that far. The mutation window size (W) doubles dynamically if fitness does not improve, balancing exploration and exploitation.

4 Proposal of Alternative Algorithms

While the Genetic Algorithm effectively optimizes a fixed sequence, other approaches could solve the sequence generation problem or the general gameplay agent problem.

4.1 Q-Learning algorithm

Q-learning is a reinforcement learning algorithm where an agent learns which action to take in each state in order to maximize long-term reward through trial and error.

4.1.1 Steps

At each time step, the agent:

1. Observes the current state s
2. Chooses an action a
3. Receives a reward r received immediately after taking action a in state s
4. Transitions to a new state s'

4.1.2 Q-function and Temporal Difference learning

The agent stores its knowledge in the Q-function $Q(s, a)$, which represents the expected cumulative reward obtained by taking action a in state s and then following the optimal policy. These values are stored in a Q-table, with one entry for each state–action pair.

Q-learning is a model-free reinforcement learning algorithm that learns online using a Temporal Difference (TD) update rule:

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

where α is the learning rate (how much I trust the new information) and γ is the discount factor (controls how much future rewards matter).

As a TD method, Q-learning updates its estimates using the difference between successive predictions, without waiting for an episode to terminate. This incremental update ensures stable learning in noisy environments. Over time, actions leading to higher rewards obtain higher Q-values, and once the values converge, selecting the action with the highest $Q(s, a)$ yields the optimal policy.

4.1.3 States

The state represents what Mario perceives, usually via local features instead of absolute positions:

- Enemy ahead (yes/no)
- Gap ahead (yes/no)
- Mario on ground or in air
- Power-up status

This allows generalization across the level rather than memorizing a fixed path.

4.1.4 Actions

Typical actions include:

- Move right
- Jump
- Move right + jump
- Shoot (if powered up)

4.1.5 Rewards

Rewards guide learning:

- Small positive reward for moving right
- Positive reward for defeating enemies or collecting items
- Large positive reward for finishing the level
- Large negative reward for dying

4.2 Deep Q-Learning (DQL)

Deep Q-Learning (DQL) works by combining the Q-learning update rule with a convolutional neural network that approximates the Q-function. Instead of storing Q-values in a table, the agent uses a neural network to estimate the value of each action for a given state.

At each time step, the agent observes the current state of the environment. This state can be a vector of features or even raw pixels from the screen, as in Mario Bros. The state is fed into a convolutional neural network, which outputs one Q-value for each possible action. These values represent the agent's current estimate of how good each action is in that state.

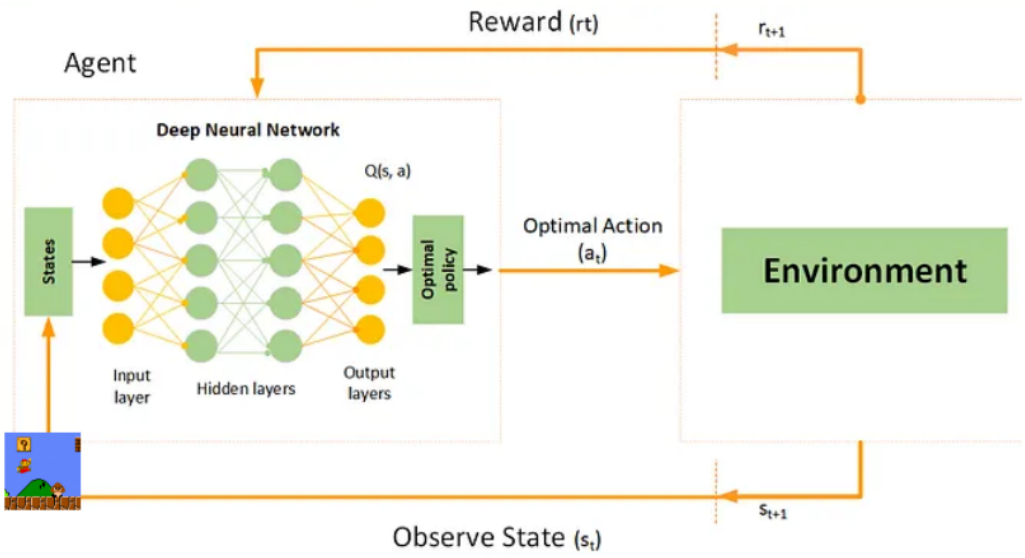

Figure 2: Deep Q-Learning architecture

4.2.1 Exploration–exploitation strategy

To decide what to do, the agent follows an exploration–exploitation strategy, usually ε-greedy. With a small probability ε, it chooses a random action to explore new behaviours. Otherwise, it selects the action with the highest predicted Q-value. If Mario always chooses the best action he knows, he may never discover a better action. If Mario always tries random actions, he never uses what he has learned. So we need both. The chosen action is executed in the environment, which returns a reward and a new state.
Hence, ε-greedy balances exploration and exploitation by choosing random actions with probability ε and the best-known action otherwise.
As the number of episodes increases, the agent gains more information about how to progress through the level, and therefore it becomes increasingly likely to choose the best possible action and for the algorithm to converge.

4.2.2 Learning process

The experience consisting of the current state, the chosen action, the received reward, and the next state is stored in a memory called the replay buffer. Instead of learning immediately from the most recent experience, the agent periodically samples random batches of past experiences from this buffer. This breaks the strong correlations between consecutive steps and makes learning more stable.

Learning is performed by updating the convolutional neural network's weights (the filters). For each sampled experience, a target value is computed using the reward and the estimated value of the best action in the next state. To make this target stable, Deep Q-Learning uses a separate target network whose parameters are updated more slowly. The neural network is then trained by minimizing the difference between its current Q-value prediction and this target value using gradient descent.

Over many interactions, rewards obtained at later stages of an episode propagate backward through these updates. The network gradually learns to assign higher Q-values to actions that lead to long-term success and lower values to actions that result in poor outcomes. In a game like Mario Bros, this process allows the agent to learn behaviours such as jumping when an enemy appears or avoiding gaps, even in different levels, because the neural network generalizes across similar states.

4.3 A* Search algorithm

In the context of Mario Bros, the A* algorithm can be used as a planning method to compute an optimal sequence of actions that leads Mario from his initial position to the goal. Each node in the search space represents a possible game state, typically defined by Mario's position, velocity, and whether he is on the ground or in the air. Edges correspond to feasible actions such as moving right, jumping, or performing a running jump, with an associated cost related to time, risk, or energy.

The algorithm evaluates each state using a cost function $f(n) = g(n) + h(n)$, where $g(n)$ measures the cost accumulated so far and $h(n)$ estimates the remaining distance to the level's end, often based on horizontal distance to the goal. By repeatedly expanding the state with the lowest estimated total cost, A* searches for a path that safely reaches the goal while avoiding obstacles and gaps.

Although A* can find optimal paths in deterministic and fully known levels, its applicability in Mario Bros is limited by the dynamic nature of the game, including moving enemies and precise timing constraints, which make real-time replanning computationally expensive. Moreover, Using only the horizontal distance to the goal as the heuristic $h(n)$ in Mario Bros is admissible but weak, because many different paths share similar distances and therefore receive similar heuristic values. As a result, A* is forced to explore a large number of alternative trajectories.

5 Reproduction of Methodology

5.1 Framework Setup

We use version 0.8 of the original Mario AI Framework [2], written in Java. Although the original paper references a specific "Learning Track", this was unavailable in the current repository, so we settled on using the original Level 1 as our benchmark. The framework worked out-of-the-box after setting up the Java environment, but it lacked a crucial component: the scoring function.

We implemented a `getScore` method to replicate the fitness function described in the paper. However, we made two heuristic modifications:

1. We removed the scoring for "hidden blocks" as the current framework does not emit an event when they are discovered. In its stead we increase to 20 the multiplier of the coins,

as hitting a hidden block usually rewards you with a coin.

2. We observed that agents quickly learned to exploit the "time remaining" bonus by dying as quickly as possible to maximize the remaining clock. To prevent this local optimum, we modified the fitness function to only award the time bonus if the level is successfully completed.

5.2 Chromosome Encoding

The framework expects an action at every tick/frame represented as a boolean array of 5 elements: `[Left, Right, Down, Speed, Jump]`. To simplify the search space, we adopted the paper's approach of defining 11 composite actions. An agent is characterized by its chromosome, an ordered sequence of genes (actions):

$$C = \langle g_1, g_2, \ldots, g_L \rangle$$

where L is the chromosome length. The length is determined by the maximum duration of a level and the temporal granularity parameter g, which specifies the number of ticks during which a single action is held constant. Formally,

$$L = \frac{\text{totalTicks}}{g}$$

This granularity models human-like behaviour, as players typically hold actions for multiple frames rather than changing inputs every tick.

Before a run of the level, each gene of the chromosome is expanded to match the total amount of required ticks/frames to cover the full duration of the level. At each tick/frame, the agent selects the corresponding gene and decodes it into a boolean action vector, which is then returned to the environment.

5.3 Genetic Algorithm

Since the JGAP library used in the original paper is no longer available, we implemented the Genetic Algorithm from scratch. We successfully reproduced the first version of the algorithm using:

- **Selection:** Tournament selection (size 15%).
- **Crossover:** Single-point crossover (rate 30%).
- **Mutation:** Random mutation (rate 1%).

Initially, this basic implementation proved too random; best-performing agents were often lost between generations, preventing convergence. To address this, we implemented two domain-specific improvements mentioned in the paper:

1. **Guided Initialization:** Instead of a purely uniform distribution, we biased the initial population to favour moving Right or Right+Jump (62.5% probability), based on the heuristic that forward progress is essential.

2. **Elitism:** We preserved the top 5% of the population (minimum 2 agents) unchanged in the next generation. This prevented the loss of the best solutions found so far.

5.4 Results

We conducted our experiments with a population size of 50. Our initial run under championship rules (10,000 level evaluations limit) allowed for only 200 generations.

Initial Reproduction: Our basic GA achieved a maximum score of 308.11 and an average of 132. This failed to match the paper's results (Max: 8551), largely because the agents behaved like random walkers, moving left and right with equal probability and making little forward progress.

Weighted Fitness: We attempted to improve performance by heavily weighting forward movement in the fitness function. While the average score increased to 2488, the actual distance travelled remained low (Best Distance: 310). The agents optimized for local rewards without reaching the goal.

Extra Training (Basic GA): Removing the championship limits and training for longer (10000 generations) and with a bigger population (200) did not help; the lack of elitism meant good solutions were lost as fast as they were found, as we can see from the jagged lines in figure 3.

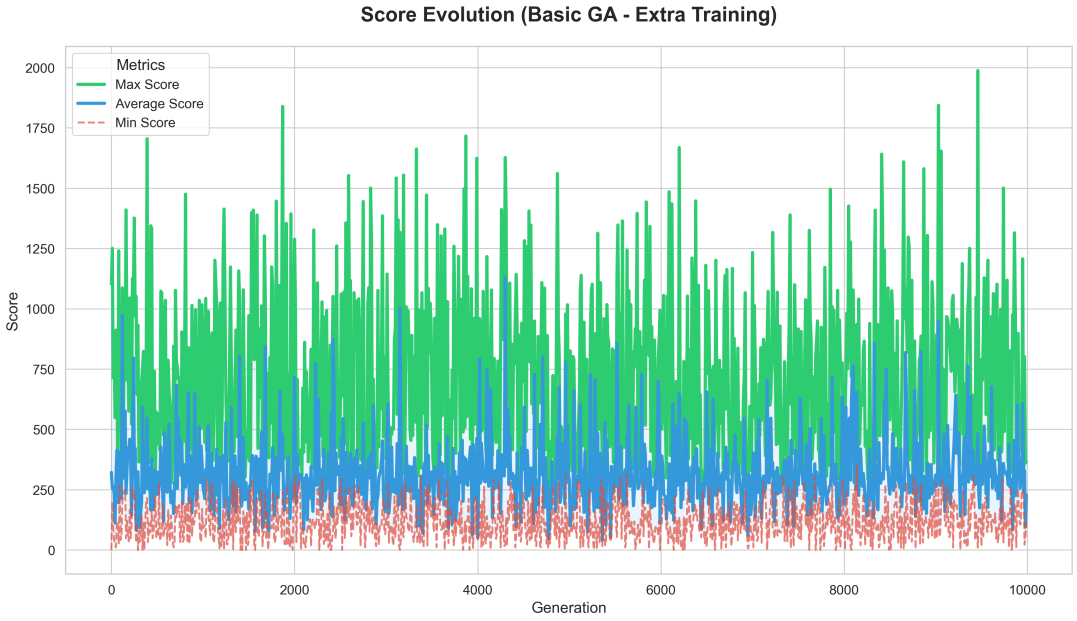

Figure 3: Evolution of fitness over generations during extra training (Basic GA). Note the volatility in the best score due to the lack of elitism.

Improved Algorithm (Guided Init + Elitism): Introducing guided initialization and elitism produced a dramatic improvement. Under championship rules, we achieved a best score of 3169.53 and a distance of 2761. The population average (413) remained low, indicating that while elite agents were succeeding, the general population was still exploring randomly.

Improved Algorithm + Extra Training: By combining the improved algorithm with extended training (200 population, 2000 generations), we finally evolved two agents that successfully completed the level. The best agent achieved a score of 5980 and a distance of 3168

(the end of the level). The emergent behaviour was intelligent: the agent consistently moved right and timed jumps to avoid Goombas and obstacles.

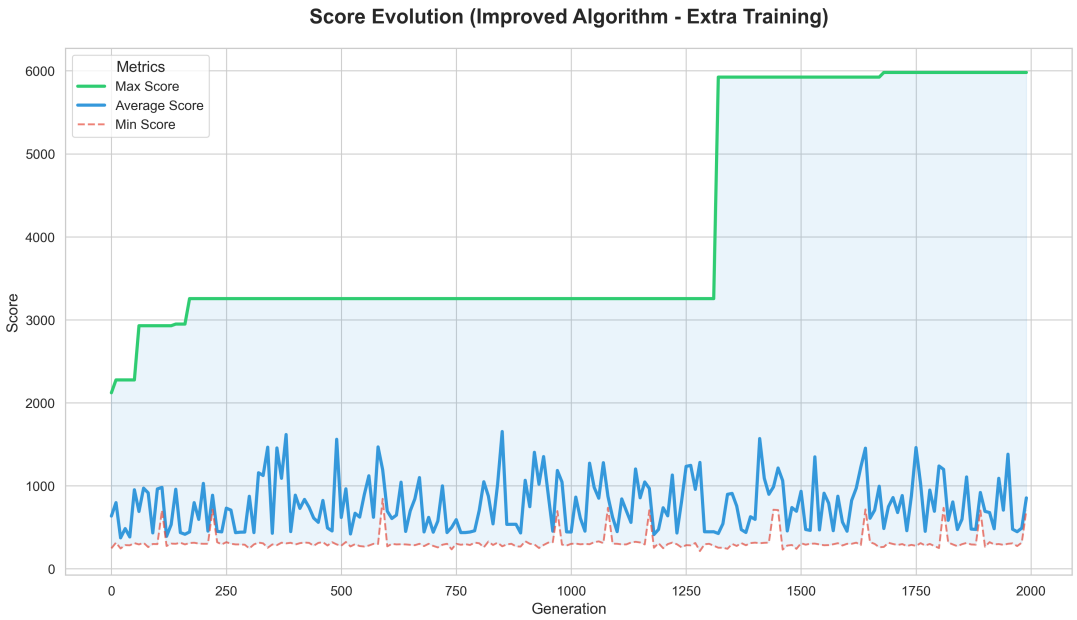

Figure 4: Performance evolution with Guided Initialization and Elitism. The stability provided by elitism allows for consistent improvement, eventually leading to level completion.

While our final score falls short of the paper's 12,000 (likely due to differences in the level length or bonus calculation in the "Learning Track"), the qualitative result is a success. We demonstrated that domain-specific heuristics (Guided Initialization) and stability mechanisms (Elitism) are essential for solving this optimization problem.

6 Conclusions

Our reproduction confirms that Genetic Algorithms are a viable method for solving platformer levels as optimization problems, but "blind" optimization is insufficient. The standard GA failed to converge due to the high dimensionality of the search space. Incorporating domain knowledge, specifically biasing initialization towards forward movement, and ensuring stability via elitism were the critical factors that allowed us to evolve agents capable of beating the level.

A key observation from our experiments is the efficacy of mutating genes based on the agent's failure point. While our current implementation relies on standard random mutation, we believe that an "intelligent mutation" operator that targets genes immediately preceding the agent's death would significantly accelerate training by focusing the search on the critical failure points. This aligns with the paper's findings on the importance of domain-dependent operators.

References

[1] A. Baldominos, Y. Saez, G. Recio, and J. Calle, "Learning Levels of Mario AI Using Genetic Algorithms," in *Conference of the Spanish Association for Artificial Intelligence*, pp. 267-277, Springer International Publishing, 2015.

[2] Ahmed Khalifa, "Mario AI Framework," 2009. [Online]. Available: `https://github.com/amidos2006/Mario-AI-Framework`.