Reinforcement Learning in Games: Training Smarter Agents with Q-Learning

Victor LEE, Joel TEO, Dylan LAU

1 Introduction

Reinforcement learning (RL) has gained significant traction in recent years as a powerful framework for decision-making in games and interactive simulations. At the heart of this field lies Q-learning, a foundational algorithm that enables agents to learn optimal behavior through trial and error. Whether you're developing stealth AI, dynamic enemy combatants, or procedural puzzle solvers, Q-learning offers an intuitive entry point into autonomous game AI — one that requires no prior model of the environment and yet produces impressively adaptable results.

This article explores Q-learning as both a stepping stone and a key building block of deep reinforcement learning (DRL). We'll break down how Q-learning works, why it's relevant to game developers, and how it scales into more powerful neural-network-driven systems like Deep Q-Networks (DQN). Along the way, we'll cover concepts such as state-action value estimation, policy optimization, and experience replay — but always with a focus on how this applies to practical, real-time gameplay situations. The goal is to equip you with a working understanding of how to implement and extend Q-learning, not just as a classroom algorithm, but as a real design tool for emergent game mechanics and intelligent behavior.

By the end of this article, you should not only understand how to write a Q-learning agent from scratch, but also how to transition that knowledge toward more scalable deep learning approaches that can drive more complex game agents. Whether you're aiming to enhance single-player immersion or introduce adaptive multiplayer bots, Q-learning is a concept every modern game developer should understand — and perhaps even love.

2 What Is Q-Learning and Why Does It Matter in Games?

Q-learning is a foundational algorithm in reinforcement learning, designed to enable agents to learn how to act optimally in an environment based solely on experience. It does this by learning an action-value function (or Q-function), which estimates the long-term expected reward of taking an action in a given state and following the best possible policy thereafter. The agent updates its knowledge through a simple rule (Sodhi, 2017) :

$$Q(s,a) = r + \gamma max_a Q(s',a') \quad (1)$$

This equation (also known as Bellman-Ford algorithm) as explained by Sodhi, expresses how an agent updates its understanding of the value of taking a particular action in a given state (Sodhi, 2017). Specifically, the Q-value (***Q (s, a)***) represents the expected cumulative reward for taking action ***a*** in state ***s***, followed by acting optimally thereafter. The term ***r***

denotes the immediate reward received upon taking that action. The discount factor γ (a value between 0 and 1) determines the weight given to future rewards relative to the immediate one — a lower γ favors short-term gain, while a higher one values long-term outcomes. The variable s' refers to the next state the agent transitions to after the action. Finally, $max_a Q(s', a')$ estimates the highest possible future reward obtainable from state s', assuming the agent continues to act optimally.

This makes Q-learning both model-free and off-policy, which means it can learn optimal strategies without knowing the full environment or following the current best-known policy. In games, Q-learning becomes especially relevant in situations where enemy behaviors, player decisions, or environment dynamics evolve over time. Whether training AI opponents, cooperative bots, or even adaptive level progression systems, Q-learning can help game developers create agents that learn rather than rely on hard-coded rules. As mentioned in an article by Stamper and Moore, classic games such as *Pac-Man, Space Invaders,* and *Frogger* have been extensively used for implementing and evaluating reinforcement learning models (Stamper & Moore, 2019). Especially *Space Invaders*, testers noticed that game AI exhibits patterns that replicate what a human would do. For instance, trained RL agents playing the game will keep a square formation to destroy the right-most enemies in order to slow down advancement (Stamper & Moore, 2019).

3 From Q-Learning to Deep Q-Learning

Traditional Q-learning relies on a Q-table to store state-action values, but this approach breaks down in environments with high-dimensional or continuous state spaces — a common trait in modern video games. To overcome this, **Deep Q-Networks (DQN)** replace the Q-table with a neural network that approximates the action-value function ***Q (s, a)***. This allows agents to generalize across similar states rather than needing to enumerate them all.

The seminal breakthrough came from **Mnih et al. (2015)**, where a convolutional neural network was trained to play multiple Atari 2600 games using only raw screen pixels and game scores as input. The agent surpassed human-level performance in several titles, demonstrating deep reinforcement learning's potential for visually complex and dynamic environments. Two core innovations in DQN enable this:

- **Target Networks**: Instead of using a constantly-updated Q-network to calculate future values (which causes instability), a fixed target network is used temporarily to compute $Q(s, a) = r + \gamma max_a Q(s', a')$ reducing oscillations and divergence.
- **Experience Replay**: Past transitions ***(s, a, r, s')*** are stored in a buffer and sampled randomly during training, breaking the correlation between sequential experiences and improving data efficiency.

These techniques not only improve learning stability but make DQN viable in real-time and visually rich scenarios, such as platformers or arena battles.

A practical example is demonstrated in research by Sodhi where a DQN-based agent was trained to play Archon: The Light and the Dark.

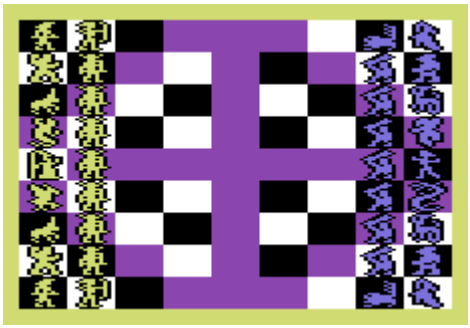

Figure 1 Archon Strategy Mode [4]

In this game, players alternate between strategy mode and a real-time combat mode as shown in Figures 1 and 2 respectively. Sodhi's system used a convolutional neural network (CNN) to interpret raw pixel input from periodic screenshots and learned to control characters using actions like *move*, *fire*, or *defend* — all without explicit game logic. Compared to the slower-converging Q-table approach, the DQN agent learned faster and adapted to the chaotic nature of arcade-style battles, despite running on non-GPU hardware.

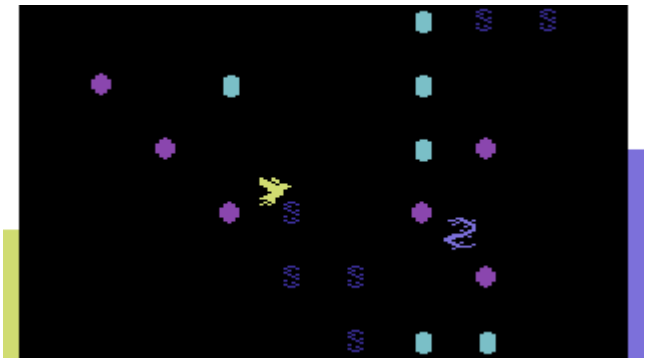

Figure 2 Archon Combat Mode [4]

This evolution — from tabular methods to deep approximators — highlights how reinforcement learning can now handle the unpredictability, partial observability, and complexity found in commercial-grade game environments.

4 Common Pitfalls : Overestimation and Instability

Despite its success, Q-learning is known to suffer from overestimation bias due to the use of during updates. This problem is exacerbated in deep Q-learning where neural approximation adds noise to value estimates. As shown in *Deep Reinforcement Learning with Double Q-Learning*, DQN often predicts unrealistically high action values, which can lead to suboptimal or unstable behavior in games (van Hasselt et al., 2016).

This is especially critical in games where agents must make precise decisions, like dodging projectiles or balancing attack timing. For example, in *Asterix* and *Wizard of Wor*, overestimations in Q-values caused DQN agents to degrade in performance as training progressed (van Hasselt et al., 2016).

5 Our demo : Q-Learning in Arena Battle

5.1. Demo Overview

The “Agent Arena” simulation is a practical demonstration of the Q-learning reinforcement learning algorithm, implemented with the Unity game engine. The objective is to train two autonomous agents to learn two distinct and opposing behaviors simultaneously within a shared environment. One agent is trained to actively chase a player-controlled target, while the other is trained to flee from it.

The simulation serves as a clear and interactive model of core reinforcement learning principles, illustrating how complex, goal-oriented behaviors can emerge from a simple system of states, actions, and rewards, without any explicitly programmed rules for how to achieve the goal.

5.2. Core Components and Architecture

The simulation is built on four key script components that work in concert:

- TrainingManager.cs: This is the master controller or "director" of the simulation. It orchestrates the entire learning process, including spawning agents, managing their separate "brains" (Q-Tables), running the main training loop, calculating rewards based on agent actions, and updating the user interface.
- AgentController.cs: This script is the agent's "body." It is responsible for physical actions. It receives a simple command (e.g., "move up") from the TrainingManager and translates that command into movement by applying velocity to the agent's physics component (Rigidbody2D).
- AgentInfo.cs: A simple data container that gives each agent an identity. Its sole purpose is to hold the agent's assigned behavior (Chase or Flee), allowing the TrainingManager to apply the correct logic and use the correct Q-Table for that agent.

- QTable.cs: This class represents the agent's "brain" or knowledge base. It is a data structure (a C# Dictionary) that stores the learned values for every situation an agent has encountered. Each agent has its own independent instance of a Q-Table.

5.3 The Q-Learning Engine

Q-learning is at the heart of this simulation, enabling the agents to learn from trial and error. The implementation can be broken down into the following key concepts:

5.3.1 State Representation

For an agent to make a decision, it must first understand its current situation. This is its State. In this simulation, the state is a simplified snapshot of the environment from the agent's perspective, represented by a unique string: "X_Y_PlayerDirection".

- X and Y Coordinates: The agent's position is discretized into a grid. This simplifies the world, preventing the agent from having to learn a value for every single possible floating-point coordinate. For example, positions (3.1, 4.8) and (2.9, 5.2) might both be simplified to the grid state (3, 5).
- Player Direction: The direction to the player is also discretized into one of eight compass directions (e.g., Up-Left, Down, Right).
- A complete state like 3_5_7 tells the agent: "I am at grid position (3, 5), and the player is somewhere to my Down-Right."

5.3.2 Action Space

The Action Space is the complete set of possible moves an agent can make at any given time. In this simulation, the action space is simple and discrete:

1. Move Up
2. Move Down
3. Move Left
4. Move Right
5. Stay Still

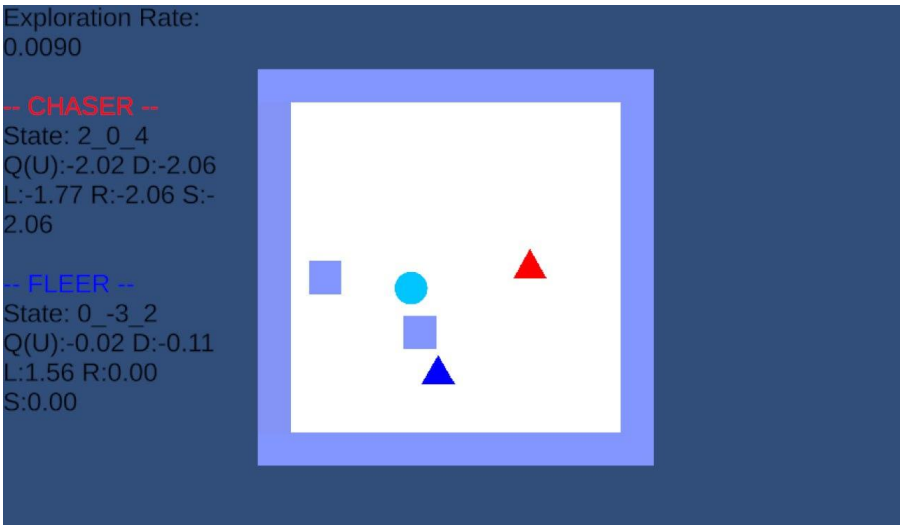

Figure 3 Q-learning demo using our engine

5.3.3. The Q-Table: The Agent's Brain

The Q-Table is the core of the agent's knowledge. It's a lookup table (a dictionary) where the keys are the states (e.g., "3_5_7") and the values are an array of numbers. Each number in the array corresponds to an action and represents the "Quality" or Q-value of taking that action in that state. For example, for state "3_5_7", the Q-Table might store:

- [Up: -1.4, Down: -0.8, Left: -1.5, Right: 0.5, Stay: -1.2]

This tells the agent that from this position, moving Right is the most promising action (it has the highest Q-value), as it will likely lead to the highest future rewards. See figure 3 attached above for an example of the Q-table system on the left.

5.3.4 The Reward System: Shaping Behavior

The agents learn by receiving rewards (positive points) or punishments (negative points) for their actions. The reward system is carefully designed to encourage the desired behavior.

- Primary Objective Rewards (Large & Event-Based):
  - Chaser: Gets a large reward (+10) for touching the player.
  - Flee-er: Gets a large punishment (-10) for being touched by the player.
- Guiding Rewards (Small & Continuous):
  - Chaser: Gets a small reward for decreasing its distance to the player and a small punishment for increasing it.
  - Flee-er: Gets a small reward for increasing its distance from the player and a small punishment for decreasing it.
- Universal Punishments:
  - Wall Collision: Both agents are heavily punished (-5) for hitting a wall, teaching them to avoid obstacles.

- Time Penalty: Both agents receive a tiny, constant negative reward (-0.1) every frame. This "cost of living" incentivizes them to achieve their primary objective as quickly as possible, preventing aimless wandering.

5.3.5. The Training Loop and The Bellman Equation

The agent's Q-Table starts empty. It learns and updates its values using a formula derived from the Bellman-Ford Equation shown above in (1), which is executed every frame in the UpdateQTable function.

5.3.6 Exploration vs. Exploitation

To prevent the agent from getting stuck in a suboptimal strategy, it must balance exploiting the best path it already knows with exploring new, random paths that might be even better. This is controlled by the Exploration Rate (Epsilon / ε).

- At the start, Epsilon is 1.0 (100%), forcing the agent to take completely random actions to build initial knowledge.
- Over time, Epsilon slowly decays, reducing the chance of random actions.
- This allows the agent to gradually transition from pure exploration to confident exploitation of its well-trained Q-Table.

5.4. Multi-Agent Simulation

The simulation architecture elegantly handles two agents by creating two separate Q-Table instances: chaseQTable and fleeQTable. When the TrainingManager processes an agent, it first checks the agent's AgentInfo component to determine its behavior. It then uses the corresponding Q-Table for all calculations (choosing actions, updating values), ensuring that the knowledge for the chaser and the flee-er remain separate. This demonstrates that the same underlying learning algorithm can produce vastly different behaviors simply by altering the reward signals.

6 Conclusion

Reinforcement learning, and specifically Q-learning, represents a paradigm shift in how we approach game AI — moving from rigid rule-based systems to adaptive agents that learn from experience. Through our exploration of core concepts such as value estimation, action selection, and deep approximators, we've seen how these algorithms enable agents to autonomously discover strategies, improve over time, and adapt to changing environments.

While Q-learning provides a simple and intuitive foundation, its deep learning extensions like DQN introduce the scalability required for modern games. However, we have also examined the limitations of these methods, particularly issues like overestimation and instability, and the solutions developed to overcome them — such as Double DQN and experience replay.

Our demo, featuring a learning agent in an arena battle, serves as a practical illustration of these concepts in action. It highlights how reinforcement learning can create emergent behavior, making gameplay feel more dynamic and less predictable. From training bots to adaptive difficulty systems, the applications of Q-learning continue to grow in relevance for professional game developers.

Ultimately, reinforcement learning is not just about algorithms — it's about giving AI the capacity to explore, fail, and grow. As the field matures, we anticipate that more games will embrace learning-based systems, enabling smarter, more believable agents that respond organically to players and environments alike.

7 References

[1] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, pp. 2094–2100, 2016. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389

[2] A. Ramanan and A. Patil, "AI for Classic Video Games Using Reinforcement Learning," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 10, no. 4, pp. 50–57, Apr. 2021. [Online]. Available: https://ijarcce.com/wp-content/uploads/2021/04/IJARCCE.2021.10410.pdf

[3] A. Rashed, M. Saeed, and A. M. Elfatatry, "Reinforcement Learning for Game AI: A Survey," *Frontiers in Artificial Intelligence*, vol. 4, 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frai.2021.550030/full

[4] Screenshot of *Archon: The Light and the Dark*, Lemon64 Commodore 64 game archive. [Online]. Available: https://www.lemon64.com/doc/archon/1 [Accessed: Jul. 16, 2025].

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, ‘‘Playing atari with deep reinforcement learning,’’ CoRR, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602