Q-Learning in Pac-Man: Feasibility and Constraints Against Intelligent Ghost Agents

Danyal Ahmed
Kaya Shi
Qazi Muhammad Hamza

June 26, 2025

Tabular Q-learning is well-liked for its simplicity, but is it capable of succeeding in highly dynamic arcade games? We utilize Gymnasium with Berkeley's Pac-Man Project to train Q-learning agents against cooperative A* ghosts that share planned paths to create a strong adversary. To reduce the combinatorial complexity of the state space, we design a compact, relative state abstraction, and train the agents under an 8-stage curriculum. After 200,000 episodes, the best model achieves only a 9% win-rate against A* ghosts ($\sim$60% against random ghosts) with 45,000 distinct states visited - indication of severe sparsity and a lack of feature expressivity. Due to the constraints of state exploration, we conclude that tabular Q-learning is poorly-suited to this domain, and that more robust approaches such as convolutional networks are more suited to overcoming large state spaces and the need for a rich observation space.

1 Introduction

Pac-Man's release in 1980 marks a major cultural landmark for modern gaming and pop culture. The deceptively simple game pits the title character against a maze populated with ghosts and pellets, trying to clear the board of pellets while avoiding the ghosts. Today, one would be hard-pressed to find someone unfamiliar with the game's core mechanics.

Our initial objective was to create an adversarial environment for both Pac-Man and the ghosts, hoping to train them either simultaneously as a multi-agent problem, or alternating training cycles in a sequential, call-and-response style of training. Due to time constraints, we elected to focus on the development of a Pac-Man agent that learned against untrained search agents as ghosts.

Existing approaches such as Davide Liu's multi-agent adversarial search [1], use search algorithms such as minimax and alpha-beta pruning to pit both parties against one another. CodeBullet (Evan Gresham) used NeuroEvolution of Augmenting Topologies (NEAT) to leverage genetic algorithms in the creation of artificial neural networks (ANNs) [2]. His implementation modeled ghost behavior in line with the original game's alternating chase and scatter sequences. We saw few fully-formed Q-learning approaches to Pac-Man, and fewer still that attempted to optimize ghost behavior. Our hypothesis was that tabular Q-learning could derive robust policies against search-based ghost agents when provided with an effective exploration strategy and state abstraction.

2 Environment

Berkeley Adaptation

To facilitate our experimentation, we chose to leverage the Berkeley Pac-Man Project [3], used for their CS188 coursework, for our base environment, selected from our initial desire to use multi-agent adversarial training, since few other implementations of Pac-Man include hooks for custom ghost agents. The framework also provides excellent introspection into agent state data and map structure, eliminating the need for additional instrumentation.

The original Berkeley implementation was designed in Python 2 and an deprecated version of OpenAI's Gym. To update it for modern Python and Gymnasium, we leveraged a community-maintained GitHub repository [4] that ported the framework to Python 3, while updating the remaining syntax and structure by hand. We extended the base into a Gymnasium compatible RL environment, enabling training through our own reinforcement learning pipeline while preserving the original mechanics.

Reward Structure

The reward function of the environment was a linear combination of the following: eating a pellet (10), eating a power pellet (25), eating a ghost (200), winning the game (500), dying (-500), and delay, a slight penalty taken at every time step (-1). We also included an exponential penalty that scaled based on the distance from Pac-Man to the ghost:

$$p_{dist} = \beta e^{-0.05 * d}$$

Where d is the Manhattan distance to the nearest ghost and β is a tunable scaling constant.

Of note is the fact that the Berkeley environment defaults to having no reward for power pellet consumption [3], seemingly expecting an agent to find value in power pellet strategies without reward incentive. Experimentation with this resulted in Pac-Man avoiding the power pellets entirely, often clearing boards without touching them at all. As such, we opted to include a moderate reward for their consumption to incentivize those exploration branches.

3 A* Ghost Behavior

Original Pac-Man Ghost Behavior

The original Pac-Man ghosts each exhibit distinct behavioral patterns that contribute to their collective threat. Blinky (red) directly chases Pac-Man's current location. Pinky (pink) attempts to ambush by targeting a tile four steps ahead of Pac-Man's current direction, encouraging the player to consider movement predictability. Inky (cyan) introduces more complex logic by using a vector computation, doubling the vector from Blinky's current location to a tile two steps ahead of Pac-Man, resulting in a dynamically shifting and difficult-to-predict behavior. Clyde (orange) exhibits hybrid behavior, chasing Pac-Man until within a range of eight tiles, after which he retreats to the lower-left corner of the map.These behavioral patterns are further structured through the implementation of global ghost modes. During 'chase' mode, ghosts follow their target logic described above. At set intervals, they switch to 'scatter' mode, retreating to map corners, which adds rhythm and pacing to the gameplay. Consumption of a power pellet triggers 'frightened' mode, in which all ghosts flee and may be eaten for bonus points. Some ghosts also contain unique transitions, such as Blinky's shift into 'Cruise Elroy' mode, an aggressive acceleration state activated when few pellets remain.

Pathfinding and Collaboration

Our ghost agents use A* pathfinding to compute optimal movement toward their target tile at each decision step. While the original Pac-Man AI operates on handcrafted logic per ghost, our approach unifies behavior through shared path-planning, enabling dynamic reactivity to Pac-Man's actions. We chose to implement A*-based ghost agents not only because they offer a more challenging opponent, capable of cornering Pac-Man far more efficiently than the original rule-based ghosts, but also because we believed that training Pac-Man against stronger, search-based agents would result in more robust and transferable policies. In contrast to deterministic patterns, A* agents adapt in real-time to Pac-Man's position, encouraging the learning agent to generalize beyond memorized ghost behavior and instead develop tactics that are spatially and strategically informed.

Each ghost computes a new path at every decision step using A*, considering not only its own goal but also the other ghost's planned path. If the ghost is in a 'scared' state (as indicated by a nonzero scared timer), it targets a tile that maximizes distance from Pac-Man. Otherwise, it targets Pac-Man's current position. The ghost then runs a modified A* search that adds a soft penalty for stepping onto tiles occupied by the teammate's path. The selected path is stored in its shared info for use by the teammate on its turn.

We design ghost agents that collaborate indirectly by avoiding each other's planned routes, thus minimizing collisions and improving coverage of the maze. This creates a dynamic threat that adapts to Pac-Man's behavior in real time. Cooperative behavior was implemented by giving each ghost access to the other's most recently computed path via a shared dictionary object. Each ghost then performs A* search toward its target (either Pac-Man or a retreat tile if scared) while incurring a soft penalty for overlapping with the teammate's planned path. This avoids rigid coordination logic while still encouraging spatial separation.

Algorithm 1 Cooperative A* Pathfinding Logic

Require: state (`GameState`), ghost index `self.index`
Ensure: action $\in$ `Directions`

$ghost\_state = state.get\_ghost\_state(self.index)$
$start = ghost\_state.get\_position()$
$pacman\_pos = state.get\_pacman\_position()$
$other\_index = 2$ **if** `self.index` $= 1$ **else** 1
$other\_path = shared\_info[f$'path_{other_index}', []]
if $ghost\_state.scared\_timer > 0$ **then**
    $goal = \_farthest\_legal\_tile(state, start, pacman\_pos)$
else
    $goal = pacman\_pos$
$path = a\_star\_search(state, start, goal)$
if $ghost\_state.scared\_timer = 0 \wedge |path| \geq 2 \wedge |other\_path| \geq 2$ **then**
    if $path[1] = other\_path[1]$ **then**
        for all $n \in Actions.get\_legal\_neighbors(start, state.get\_walls())$ **do**
            if $n \neq other\_path[1]$ **then**
                $path = [start, n]$
                break
$shared\_info[f$'path_{self.index}'$] = path$
if $|path| \geq 2$ **then**
    $next\_pos = path[1]$
    for all $a \in state.get\_legal\_actions(self.index)$ **do**
        $(dx, dy) = Actions.direction\_to\_vector(a)$
        $successor = (\lfloor start_x + dx \rfloor, \lfloor start_y + dy \rfloor)$
        if $successor = next\_pos$ **then return** $action$
$legal = state.get\_legal\_actions(self.index)$
if $legal$ **then**
    return $random.choice(legal)$
else
    `Directions.STOP`

Overlap Penalty

This version of A* search is a standard implementation with a cost modifier to discourage overlapping paths. Let g(n) be the number of steps from start to node n, and let h(n) be the

Manhattan distance from n to goal. Then the total cost is:

$$f(n) = g(n) + h(n) + \alpha \mathbb{1}\{n \in other\ path\}$$

Where α is a tunable penalty for stepping on the other ghost's path. This encourages the two ghosts to cover separate regions of the map, while preserving flexibility in constrained spaces.

4 Q-Learning Structure and Design

Q-learning is a reinforcement technique that attempts to build a state-action value function that informs a policy for future play. The agent possesses a Q-table, which is a mapping of Q-values, an expected cumulative reward, to state-action pairs (s, a). The agent considers its current state s, and a given action a that takes it to a subsequent state s'. The agent then considers the possible actions that can be taken from s' and notes the greatest reward it can achieve from that state, which is determined by the existing entries in the Q-table. A default value of 0 for previously unexplored states is considered standard for many problem areas. This best reward can be denoted as $V(s')$, or the best reward from state s'.

$$V(s') = \max_{a'} Q(s', a')$$

It then uses this best-case reward to update the Q-table for the state-action pair (s, a), averaging all updates to that state-action pair using the term η:

$$\eta = \frac{1}{1 + \text{no. of updates to } Q(s, a)}$$

Which represents an averaging term across all updates. The update to $Q[(s, a)]$ therefore looks like this:

$$Q_{new}(s, a) = (1 - \eta)Q_{old}(s, a) + \eta(r + \gamma V(s'))$$

Where r is the immediate reward from action a, and γ represents the discount factor, meant to emphasize nearby rewards over distant future rewards.

It is worth noting that the learning process occurs off-policy, as the agent does not need to follow an optimal policy to engage in learning. In a setting such as Pac-Man, we often want to encourage exploration through various states early on to allow for early population of the Q-table that can better inform later iterations. To enable this, we use an ϵ-greedy approach, which uses the following consideration for every iteration:

$$\begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s, a) & \text{with probability } (1 - \epsilon) \end{cases}$$

Where ϵ is multiplied by some decay constant δ after each episode.

State Definition

The key consideration in designing the Q-learning agent was how it should define state for a given frame. For any given moment, our agent must account for the following features:

- Walls, as they prevent movement
- Pac-Man's current position
- Ghost positions
- Scared ghost positions
- Pellet locations
- Power pellet locations

A naive solution would be to express these features as a stack of binary matrices of dimensions $h \times w$, where h and w are the height and width of the game map. Each matrix would represent the presence (1) or absence (0) of a particular feature at each position. For a 20×20 grid, this results in a binary tensor of size $20 \times 20 \times 6$, resulting in a state space of size $2^{2400} \approx 10^{723}$. Even assuming a static wall layout, the combinatorial complexity of the remaining features still leads to an intractable state space for tabular Q-learning.

To address this issue, we defined a compact relative state representation centered on Pac-Man's current position. The revised feature set uses 4 buckets per feature, and significantly reduces the dimensionality of the state space while preserving decision-relevant information:

- Nearest ghost's direction & distance
- Power flag denoting whether the board still contains power pellets
- Nearest ghost's remaining scared time & direction
- Nearest pellet's distance & direction
- Nearest power pellet's distance & direction
- 4-bit vector for walls on each of Pac-Man's sides (16 states)

This relative representation drastically reduces the size of the state space, enabling more efficient learning and gen-

eralization without sacrificing key environmental information. In particular, the use of a power flag allows us to condense states that don't have power pellets on the board, resulting in $4^5 \times 16 = 16384$ states when the flag is 0, and $4^7 \times 16 = 262144$ when the flag is 1.

The total state space is therefore confined to $16384 + 262144 = 278528$. This localized state definition has the additional benefit of being agnostic to map size, and informs Pac-Man's decision-making using generalized features rather than map-specific features, meaning an agent trained this way should be able to use its training to inform actions on a novel game map.

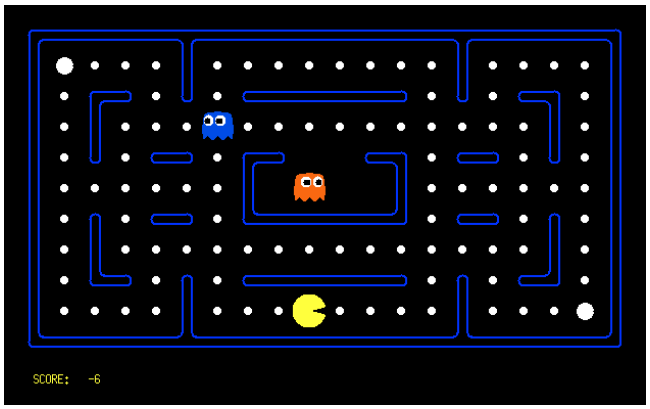

Figure 1: mediumClassic layout

Curriculum Learning and Impetus

Early training was done on the mediumClassic map (shown above) with two A* ghosts. Over 30,000 iterations, the Q-learning agent only completed the map 9 times (a 0.03% success rate), with only 9440 states being explored. The agent struggled to escape the starting area, leading us to conclude that the presence of A* ghosts coupled with the large map size likely introduced high variance in the reward signal, inhibiting early learning. In response, we developed a structured curriculum learning approach, progressively exposing the agent to more complex maps and ghost behaviors. This staged difficulty allowed the agent to first encounter basic navigation and pellet collection before optimizing its reward through more efficient movement and ghost avoidance.

Intended curriculum pattern:

0. Pellet Seeking
1. Introduce Ghost
2. Two Ghosts
3. Larger Map
4. Introduce A*
5. Full Map
6. Intensity Increase
7. Full Target Task

Variations in Q-Learning Structure

The Q-learning regimen had several meta-parameters to consider, including episode count, decay rate, discount factor γ, and maximum/minimum epsilon values. We also implemented a per-episode step limit to prevent infinite loops from stalling learning progress.

For state features, our bucket boundaries also presented an opportunity to optimize decision-making. Distance required a small initial step to allow differentiation between closeby ghosts and ghosts that were on the other side of a wall.

We also experimented with the implementation of an optimistic Q-value, which initialized the Q-value for an unexplored state-action pair to 10.0, rather than 0.0. This was done to encourage the exploration of more unseen states with the intention of broadening the ability to create novel transitions. While the number of total states did increase, 10,515 states to 15,622 in one case, the number of visits-per-states dropped drastically, decreasing from 2.4 to 1.2 visits per state explored in that same example. Due to the large state space, we ultimately reverted to the neutral Q-value initialization, favoring the deep exploration of a more confined subset of states to singular, random exploration of a broader subset.

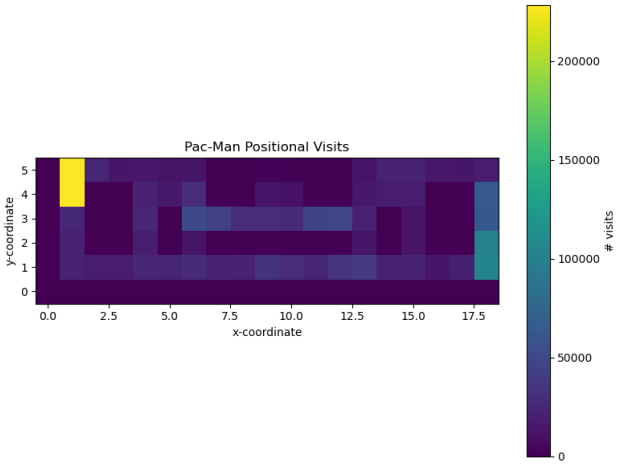

Figure 2: Heatmap of Absolute Agent's Positions over 20,000 episodes

5 Pac-Man Has Behavioral Issues

Constant Oscillation

During early experiments, we observed significant oscillatory behavior in the agent's policy. As Figure 2 illustrates, the agent frequently engaged in a cycle between two states, likely due to a local maximum in its reward landscape coupled with insufficiently detailed state features. This behavior persisted over tens of thousands of training episodes.

To address this, we developed a parallel Q-learning agent that used a relative action space. Rather than absolute actions (North, South, East, West), the new agent's action space was defined as forward, reverse, left, and right, relative to its current heading, inspired by ZuzeloApps' video on neural networks in Pac-Man [5]. This relative action space prevents mirrored states that point to one another, while increasing feature resolution for the agent's Q-table. While this definitively solved the oscillation problem, as seen in Figure 3, it also quadrupled the state space from 278,528 possible states to 1,114,113. The order of magnitude increase represents a significant roadblock in exploration efficiency and policy convergence.

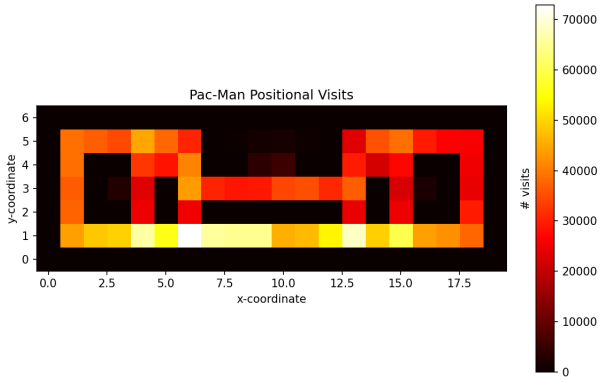

Figure 3: Heatmap of relative agent's positions over 20,000 episodes

Persistent Issues

Even after addressing the oscillation problem, the agent still struggled with decision-making at many points. While able to routinely trace a path to the nearest pellet in the tinyMaze layout, the addition of ghost agents added considerable noise in the reward structure, leading to extreme risk-aversion and hesitation in pursuing pellets. Removing our previous exponential negative reward for Pac-Man's distance did not significantly prevent this behavior. We concluded that the state abstraction used by the agent did not provide enough granularity to differentiate game states that were significantly different from one another. Given the already tremendous size of the state space, we did not think that having the agent utilize more features was tenable, as most explored states would be visited only once or twice, with the majority of states never being explored at all.

6 Results

Training did not progress past stage 4 of our curriculum. The decision to halt training was due to persistent issues that did not meaningfully improve upon retraining or tuning of meta-parameters. During this training period, no model reached above a 9% success rate for any layout that contained an A* ghost agent alongside another agent of any kind. Performance for random ghost agents and single-opponent layouts were near 60%. This result is noticeably higher, but still considerably lower than we'd expect for what should be a straightforward solution.

Upon examination, we observed that the vast majority of states were unexplored, even with the maximum episode count of 200,000. We also observed that the inflated memory required to store the Q-table at runtime caused frequent memory overflows and program crashes at episode counts above 50,000, using WSL with 4GB of allocated RAM. The maximum number of explored states was 45,141, which represents roughly $\frac{1}{6}$ and $\frac{1}{24}$ of the total hypothetical state spaces. Over all runs, the average visits-per-state was 1.88, indicating the vast majority of states were explored only once, assuming early states were visited far more often than only once or twice. Map coverage for the relative agent was fairly consistent, with high visitation near Pac-Man's starting squares. This pattern is reasonable and fairly expected over high episode counts.

7 Discussion

The observations from the Q-learning agents made it clear that a tabular approach was likely to be insufficient for creating an adept, intelligent Pac-Man agent. The nature of exploration in Q-learning struggles to overcome the immense

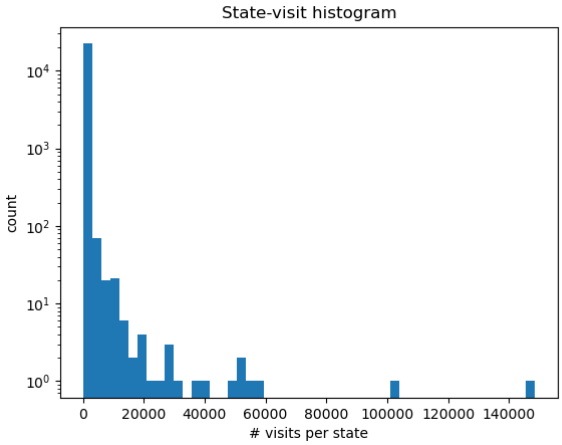

Figure 4: State visit distribution over 30,000 episodes

number of possible states, with state visitation being quite sparse in most instances. Attempts to abstract the state representation resulted in feature sets that lacked the expressivity needed to distinguish numerous distinct game scenarios. Our final agent struggled with both of these drawbacks, failing to balance over-generalized state descriptions that generated conflicting updates to its Q-table with a large state space that resulted in sparse visitation and singular updates. Addressing one of these problems would simply exacerbate the other.

8 Future Improvements

Were we to improve upon the current Q-learning approach, we might explore a few minor adjustments. One can refine the state representation so that our model incorporates both the first and second closest ghosts into its feature set. This would serve to address the boxing-in strategy of the A* ghosts. An alternative improvement would be to give the agent an early training boost, by initializing the Q-table with values based on expert players' games.

Neither of the above variations, however, would be likely to overcome the core issue of state space versus feature expressivity. Were we to continue with our experimentation, we would move forward with a Convolutional Neural Network (CNN). The use of CNNs allows us to sidestep the state space issue by instead using convolutional filters (kernels) to extract feature information without needing to repeatedly explore and tabulate numerous state-action pairs and their respective rewards.

9 Conclusion

It is hypothetically possible to perform Q-learning to train a consistent, capable Pac-Man agent in a non-trivial environment, using a complete state representation over an arbitrarily large number of episodes. Doing so, however, comes with significant computational cost. To hold the resultant Q-table and update counters during runtime would require gigabytes of memory for even a few thousand episodes, let alone the number required to adequately explore the state space. As such, we believe that there likely does not exist an adaptation of tabular Q-learning that could efficiently train an agent to routinely complete boards against aggressive intelligent agents. If such an approach does exist, the computational resources required to train such an agent would almost certainly fail to justify its necessary time and cost, especially compared to any number of other machine learning or deep learning methods. Our recommendation for training a Pac-Man agent to solve practically-sized layouts would be to pursue alternatives to standard tabular Q-learning, such as Deep Q-Networks (DQNs) or Convolutional Neural Networks.

10 Link to Github Repository

Q-Learning Pac-Man Repository

11 References

[1] Code Bullet, AI learns to play PAC-MAN using NEAT, YouTube, Apr. 07, 2018. https://www.youtube.com/watch?v=QpyHYRBKy8U (accessed Jun. 26, 2025).

[2] D. Liu, Playing Pacman with Multi-Agents Adversarial Search, KejiTech, Feb. 13, 2020. https://techs0uls.wordpress.com/2020/02/13/playing-pacman-with-multi-agents-adversarial-search/

[3] Berkeley AI Materials, Berkeley.edu, 2024. https://ai.berkeley.edu/project_overview.html

[4]aig-upf, GitHub UC Berkeley CS188, a refactored version that runs in Python3.X, GitHub, 2024. https://github.com/aig-upf/pacman-projects/tree/main (accessed Jun. 26, 2025).

[5] ZuzeloApps, A.I. Learns to Dominate PACMAN, YouTube, Oct. 03, 2022. https://www.youtube.com/watch?v=3SFgPQtqadk (accessed Jun. 26, 2025).