EEE 4709 Project Report

Reinforcement Learning in Game Development

Analyzing the application of Q-Learning and Approximate Q-Learning in Python

Project Website: https://rlproject.netlify.app/

TITLE PAGE

Project Title: Reinforcement Learning in Game Development

Group No: C1

Author(s)/Student(s) Names:

Mahdi Kamal (ID: 200021306)

Tahmid Hasan Muttaky (ID: 200021310)

Ragib Yeaser Itmam (ID: 200021324)

Sadman Aster (ID: 200021344)

Institution & Course Name: IUT - Artificial Intelligence and Machine Learning

Instructor Name: Md. Arefin Rabbi Emon

Submission Date: 21 March 2025

ABSTRACT

In this project we analyze the application of Reinforcement Learning (RL) in game development. Among the types of reinforcement learning, we shift our focus on Q-Learning and approximate Q-Learning. We start by analyzing how Q-Learning performs in the well-known game- Pacman. We analyzed how four set of parameter settings reflecting different human mindsets performs in four distinct layouts. Then we expanded our analysis to approximate Q-Learning which introduces generalization. Our target was not only to analyze RL in Pacman, but also to understand and apply the broader scope of it. So we developed a simple game that utilizes the same idea, in a very different environment.

1. INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

- Reinforcement Learning (RL) mimics the most fundamental way how humans learn- to learn from mistakes and the aftermath of past actions.
- Reinforcement Learning is interesting not just because it makes machine adopt and evolve like human, but also because it tells us how different mindsets affect our performance as humans. That makes Reinforcement Learning the most elegant and fascinating form of Machine.
- In games, RL allows automated agents or NPCs to behave intelligently and evolve over playthrough that takes the mechanism to a new horizon.

1.2 PROBLEM STATEMENT

- To implement Q-Learning and Approximate Q-Learning in Pacman game
- To analyze how the agent performs in different layouts and with different learning parameters

1.3 OBJECTIVES

- To analyze how Q-Learning performs with different set of parameters reflecting different mindset in a well-known game of Pacman
- To analyze and compare Approximate Q-Learning with Q-Learning to understand well defined state-based learning vs fewer feature-based learning.
- To study the broader scopes of developing games that evolve with playthroughs
- To develop a new game incorporating multiple types of Artificial Intelligence

1.4 SCOPES AND CONSTRAINS

- **SCOPES:**
  - Reinforcement learning is a versatile model that is not limited to a certain environment or layout. It can be implemented in a wide range of learning agents.
  - Parameters can be optimized to allow better performance and efficiency.
- **CONSTRAINS:**
  - Q-Learning requires exponentially higher number of training to allow better performance, which needs great computational power and time.
  - Policy based learning models can be complex to develop.

2. RELATED WORK

2.1 EXISTING WORK

- A good portion of this project is built upon ‘The Pac-Man Projects’ from UC Berkeley CS188.

2.2 COMPARISON WITH EXISTING WORK

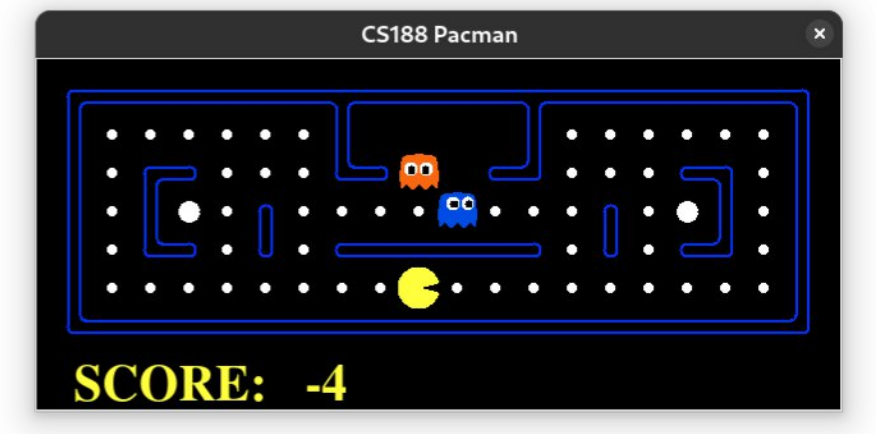
- UC Berkeley CS188 provided an environment with a modular design. Our project implemented Q-Learning and Approximate Q-Learning to that. We analyzed how the different parameters and higher training affects the performance in different layouts. We also found out the limitations and ways to implement it .
- We developed a simple game of our own inspired from the classic snake game, with some additional tweaks

3. SYSTEM ARCHITECTURE / EXPERIMENTAL SETUP

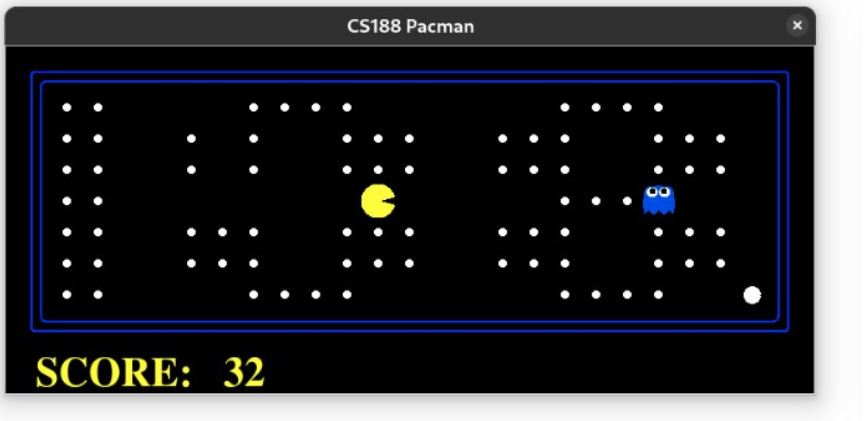
3.1 MODEL DESIGN

- **Layouts**
  - **Small Grid:** A very simple grid

  - **Small Classic:** A smaller version of the classic Pacman



- **Open Classic:** An open area with no walls within boundaries



- **Minimax:** A challenging layout, but with fewer possible states

- **Learning Methods**
  - Q-Learning
  - Approximate Q-Learning

- **States (For Q-Learning)**
  - Layout Information
    - Walls
    - Maze Width and Height
  - Food Information
    - Food dot locations
    - Power pellet locations
  - Agent Information
    - Pacman's position and direction
    - Ghost positions and directions
    - Ghost scared timer states
    - Whether agents are eaten
  - Game Information
    - Win/Loss state

- **Features Extraction (For Approximate Q-Learning)**
  - Closest Food: Normalized distance to the closest food pellet.
  - Number of ghosts 1 step away
  - Food Eaten: 1 when agent eats food and no ghosts are nearby

3.2 HARDWARE AND SOFTWARE REQUIREMENTS

- **Hardware:**
  - A decent configuration computer for basic analysis
  - A highly powerful computer for higher training with big state-space
- **Software and Programming Language:** Anaconda, VS Code, Python

3.3 DATA SOURCES AND PREPROCESSING

- Primary data collected from experiment runs
- Raw outputs was hundreds of pages long, using AI tools (specially Claude) data was converted to a more concise format

4. METHODOLOGY

4.1 THEORETICAL FOUNDATIONS

- **Reinforcement Learning**

Reinforcement Learning is a method where an agent learns to make decisions by interacting with its environment. Over time, the agent figures out which actions bring the best rewards and which ones to avoid.

- **Key Aspects of RL**

- **Agent**: The entity making decisions
- **Environment**: The system the agent interacts with. In game Pacman's maze and ghosts.
- **State (s)**: The representation of the current situation of the agent (Pacman's position, remaining dots, ghost locations).
- **Action (a)**: A possible decision the agent can take like moving up, down, left, or right.
- **Reward (r)**: Feedback from the environment after taking an action
- **Policy (π)**: The strategy the agent follows to choose actions .In game, avoiding ghosts while collecting dots.
- **Value Function (V(s))**: The expected cumulative reward of being in a state like estimating the advantage of different positions in the maze.
- **Q-Function (Q(s, a))**: The expected reward of taking an action in a state, such as whether moving towards a power pellet is beneficial or not.
- **Exploration vs. Exploitation:**
  Instead of always taking the best-known action, the agent follows an ϵ-greedy strategy:
  With probability ϵ, it chooses a random action (exploration).
  Otherwise, it selects the action with the highest estimated reward (exploitation)
  Over time, ϵ is reduced to favor exploitation as more knowledge is gained.

- **Markov Decision Process (MDP) Framework**

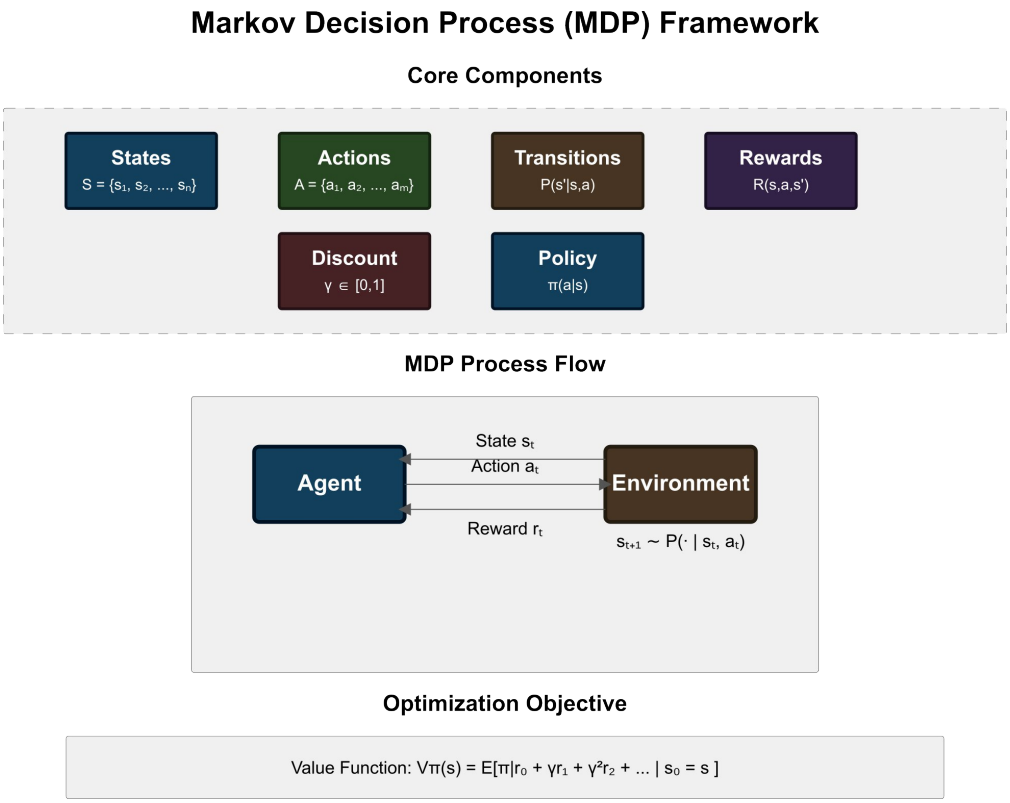
  Markov Decision Processes (MDPs) form the foundation of many reinforcement learning (RL) methods. Here the outcomes are partly random and partly under the control of a decision-maker.

  Properties of MDP Framework:

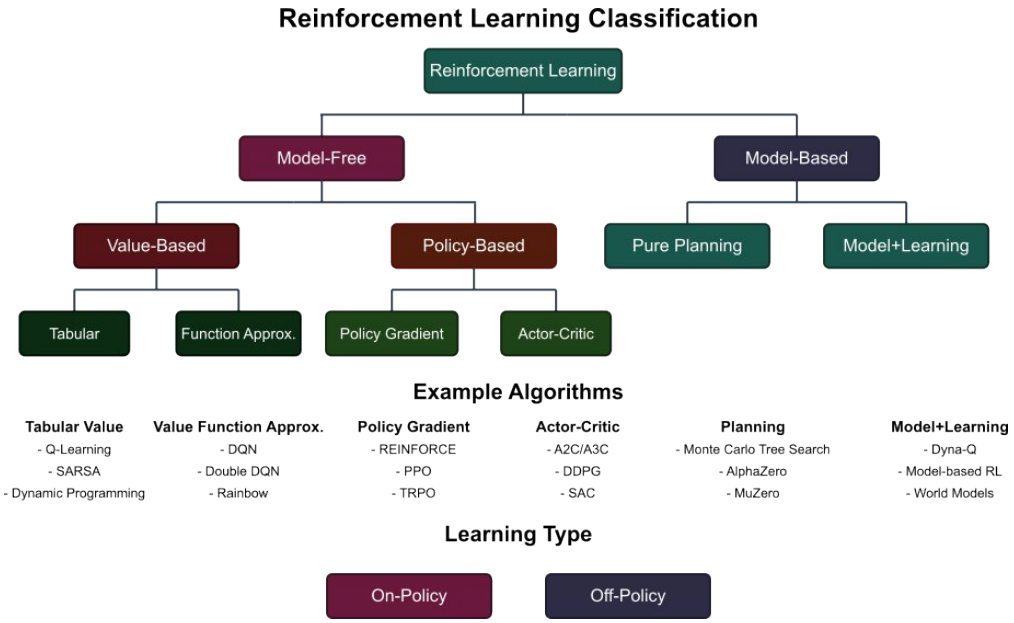
  Markov Property: The future depends only on the present, not the past

  Stationary: Transition and reward functions don't change over time

  Fully Observable: Agent can observe the complete state of the environment



- **Classification of RL Algorithms**



Our project focuses on Q-Learning and Approximate Q-Learning, which are Value-Based methods. These methods are Model-Free, that is, the agent learns without prior knowledge of environment dynamics.

- **Q-Learning (Q-L)**

Q-Learning is a model-free RL algorithm used to find an optimal policy by learning the Q-values of state-action pairs. Basically it is a simple method that helps the agent learn which moves are best by updating scores for each move using Bellman equation.

Bellman equation:

$$Q(s, a) = Q(s, a) + \alpha [r + \gamma \max Q(s', a') - Q(s, a)]$$

Here, α is the learning rate (how fast the agent learns) and γ is the discount factor (how much the agent values future rewards). In Pacman, Q-Learning helps the character learn the best directions to move to avoid ghosts and collect dots.

Characteristics:

- Learns from actions outside the current policy.
- Converges to an optimal policy given enough time and exploration.
- Given enough training, it would always reach optimization in a deterministic and limited-state environment. Only problem- enough here can be trillions and zillions!

In Pacman, Q-learning learns the best movements by associating rewards with avoiding ghosts and collecting pellets.

- **Approximate Q-Learning**

Approximate Q-Learning is an extension of basic Q-Learning. Instead of keeping a table of values for every possible state and action this method uses features to estimate the Q-values.

Weight update rule (Gradient Descent):

$$\omega_i \leftarrow \omega_i + \alpha \nabla J(\omega) \times f_i(s, a)$$

where:
ω_i is the weight for feature i,
α is the learning rate,
$f_i(s,a)$ is the value of feature i in state s for action a,
$\nabla J(\omega)$ is an error function between the expected outcome and actual outcome

Characteristics:

Feature Extraction:
Instead of tracking every state exactly, the agent looks at important features. For example, in Pacman, features might include the distance to the nearest ghost or the number of dots left.

Function Approximation:
The agent uses a function to combine these features into an estimated Q-value. Each feature has a weight, and the Q-value is the sum of the products of these weights and the feature values.

Learning Update:
When the agent takes an action and gets a reward, it adjusts the weights based on how good or bad the outcome was. The basic idea is similar to Q-Learning:

- **Comparison of Q-Learning and Approximate Q-Learning**

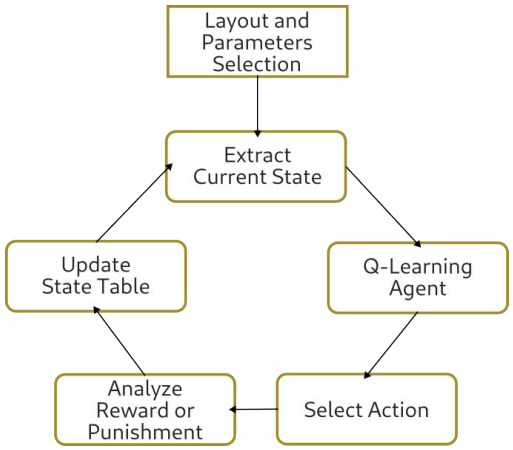
Parameter	Q-Learning	Approximate Q-Learning
Algorithm Type	Tabular RL	Function Approximation
State Space	Discrete & Small	Large/Continuous
Function Approximation	No (Uses Q-table)	Yes (Linear, Polynomial, etc.)
Stability	Stable (Deterministic updates)	Prone to divergence
Key Techniques	Bellman Equation, ε-greedy exploration	Feature Engineering, Linear Regression
Use Cases	Grid Worlds, Simple MDPs	Robotics, Mid-complexity environments
Training Speed	Fast for small tables	Depends on approximation complexity
Memory Requirements	Low (Stores Q-table)	Moderate
Handling Non-linearity	No	Limited
Convergence Guarantee	Yes (Under MDP assumptions)	No (Approximation errors)

4.2 EXPERIMENTAL SETUP / ALGORITHM

Algorithm:

- Initialize Q-table
- Observe initial state
- Select action using ε-greedy strategy or approximate Q-Learning strategy
- Perform action and receive reward
- Update Q-value using
  - Bellman's equation for Q-Learning
  - Gradient Descent Approximate Q-Learning
- Repeat a fixed predefined times

Flow Chart:



1. **Layout and Parameters Selection**

   Choose the game layout and set the Q-Learning parameters such as learning rate α, discount factor γ, and exploration rate ε.

2. **Extract Current State**

   The agent looks at the environment such as position of Pacman, ghosts, pellets, etc. and identifies its current state.

3. **Q-Learning Agent**

   The Q-Learning algorithm processes the current state and refers to its Q-table decide which action might yield the best long-term reward.

4. **Select Action**

   Using an ε-greedy strategy, the agent either chooses the best-known action (exploitation) or picks a random one (exploration).

5. **Analyze Reward or Punishment**

   After the agent acts, the environment gives feedback: a positive reward (for good outcomes) or a penalty (for bad outcomes).

6. **Update State Table**

   The agent updates its Q-values based on the received reward and the estimated future rewards, helping it refine its decision-making for future steps.

4.3 ASSUMPTIONS CONSTRAINTS

- The environment is deterministic.
- The agent has full visibility of the grid layout.
- Training is limited by computational resources.
- The system follows MDP framework for Q-Learning.

5. RESULTS AND ANALYSIS

5.1 PERFORMANCE METRICS

Primary Metrics

- Reward from episodes after training
- Win Rate

Secondary Metrics

- Reward from episodes during training
- Computation time

5.2 LEARNING PARAMETERRS

- Quick Learner
  - Learning rate, α=0.95 (Very high learning rate)
  - Discounting factor, γ=0.1 (Prioratizes instant reward)
  - Exploration Rate, ε=0.2 (Explores moderately)
- Explorer
  - Learning rate, α=0.7 (High learning rate)
  - Discounting factor, γ=0.5 (Balanced)
  - Exploration Rate, ε=0.5 (Explores a lot)
- Conservative
  - Learning rate, α=0.5 (Low learning rate)
  - Discounting factor, γ=0.6 (Long term priority)
  - Exploration Rate, ε=0.1 (Explores very low)
- Strategist
  - Learning rate, α=0.6 (Moderate learning rate)
  - Discounting factor, γ=0.8 (Long term priority)
  - Exploration Rate, ε=0.3 (Explores modrately)

5.3 RESULT TABLE

- **Q-Learning**
  - **Q-Learning in Small Grid**

Model Name α | γ | ε	Number of Training	Average Reward During Training	Average Score After Training	Win Rate
Quick Learner 0.95 | 0.1 | 0.2	5000	-102.61	503.7	25/25 (100%)
Explorer 0.7 | 0.5 | 0.5	5000	-489.68	212.1	18/25 (72%)
Conservative 0.5 | 0.6 | 0.1	5000	-51.16	501.9	25/25 (100%)
Strategist 0.6 | 0.8 | 0.3	5000	-412.41	498.7	25/25 (100%)

  - **Q-Learning in Small Classic**

Model Name α | γ | ε	Number of Training	Average Reward During Training	Average Score After Training	Win Rate
Quick Learner 0.95 | 0.1 | 0.2	5000	-410.91	-379.28	0/25 0%
Explorer 0.7 | 0.5 | 0.5	5000	-429.47	-412.6	0/25 0%
Conservative 0.5 | 0.6 | 0.1	5000	-401.40	-399.12	0/25 0%
Strategist 0.6 | 0.8 | 0.3	5000	-417.93	-387.0	0/25 0%

  - **Q-Learning in Open Classic**

Model Name α | γ | ε	Number of Training	Average Reward During Training	Average Score After Training	Win Rate
Quick Learner 0.95 | 0.1 | 0.2	3000	-462.83	-438.44	0/25 0%
Explorer 0.7 | 0.5 | 0.5	3000	-477.97	-430.64	0/25 0%
Conservative 0.5 | 0.6 | 0.1	3000	-456.39	-475.56	0/25 0%

Strategist 0.6 | 0.8 | 0.3	3000	-462.97	-444.12	0/25 0%

- **Q-Learning in Minimax**

Model Name α | γ | ε	Number of Training	Average Reward During Training	Average Score After Training	Win Rate
Quick Learner 0.95 | 0.1 | 0.2	5000	-85.37	434.96	24/25 92%
Explorer 0.7 | 0.5 | 0.5	5000	-381.88	192.6	17/25 68%
Conservative 0.5 | 0.6 | 0.1	5000	-209.55	-197.12	8/25 32%
Strategist 0.6 | 0.8 | 0.3	5000	-271.38	70.24	13/25 56%

- **Approxiamate Q-Learning**

  - **Approximate Q-Learning in Small Grid**

Number of Training	Average Score After Training	Win Rate
1	-17.92	12/25 48%
10	220.24	18/25 72%
100	300.8	20/25 80%
1000	261.32	19/25 76%

  - **Approximate Q-Learning in Small Classic**

Number of Training	Average Score After Training	Win Rate
1	-227.76	0/25 0%
10	838.2	22/25 88%

100	736.84	20/25 80%
1000	833.6	22/25 88%

- **Approximate Q-Learning in Open Classic**

Number of Training	Average Score After Training	Win Rate
1	1239.92	25/25 100%
10	1240.6	25/25 100%
100	1241.16	25/25 100%
1000	1240.2	25/25 100%

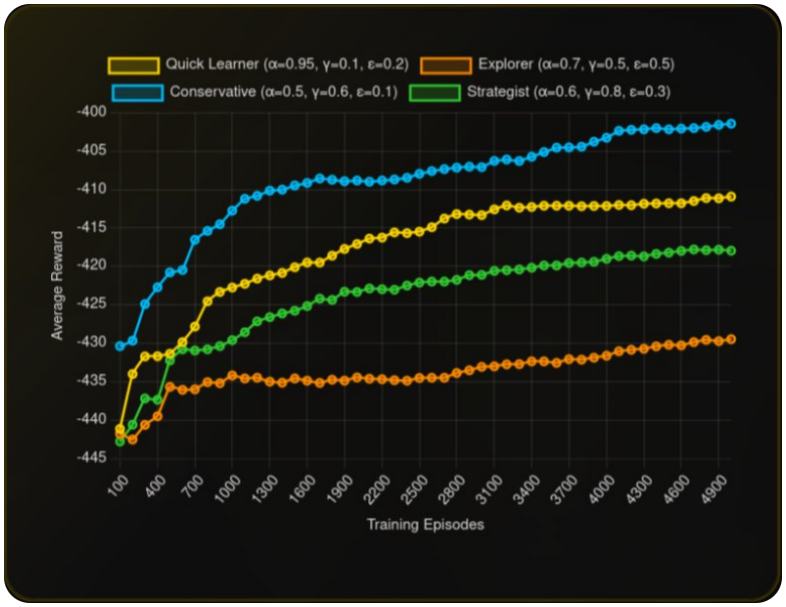
- **Approximate Q-Learning in Minimax**

Number of Training	Average Score After Training	Win Rate
1	271	19/25 76%
10	190.24	17/25 68%
100	271.56	19/25 76%
1000	392.48	22/25 88%

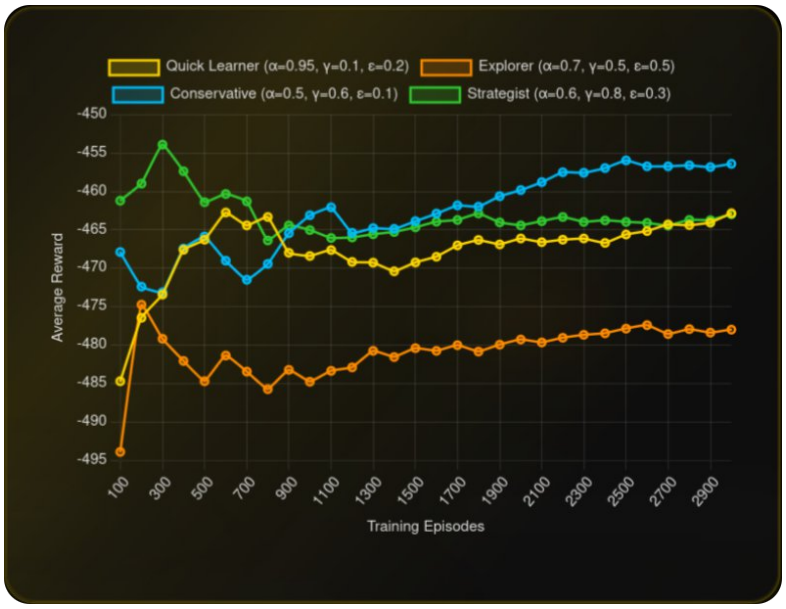
5.4 RESULT VISUALIZATIONS

- **Q-Learning: Learning Curves (During Training)**
  - **Layout: Small Grid**

  - **Layout: Small Classic**



- **Layout: Open Classic**



- **Layout: Minimax**

- **Q-Learning: Average Reward (During Training)**
    - **Layout: Small Grid**

    - **Layout: Small Classic**

- **Layout: Open Classic**

- **Layout: Small Classic**

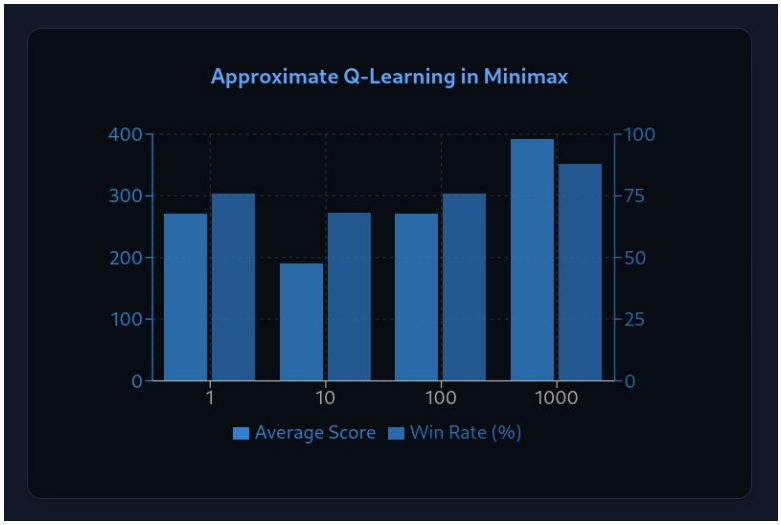
- **Q-Learning: Test Score (After Training)**
  - **Layout: Small Grid**

  - **Layout: Small Classic**

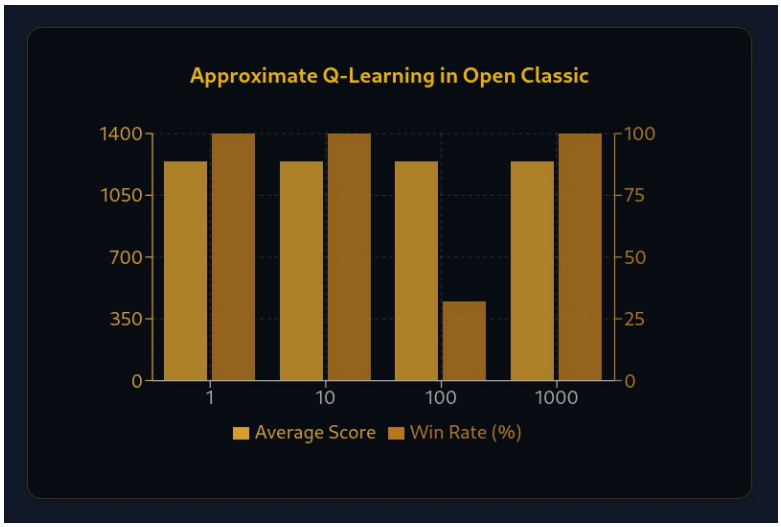
- **Layout: Open Classic**

- **Layout: Small Classic**

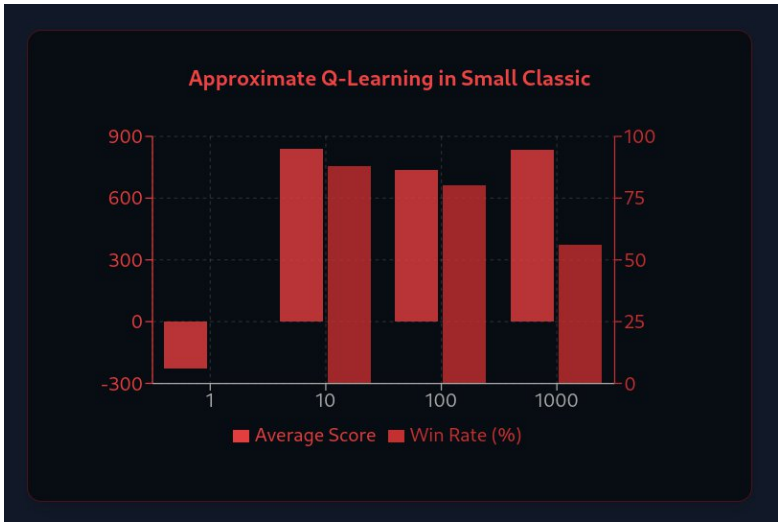
- **Approximate Q-Learning: Performance Graph with Increased Training**
    - **Layout: Small Grid**



    - **Layout: Small Classic**



- **Layout: Open Classic**



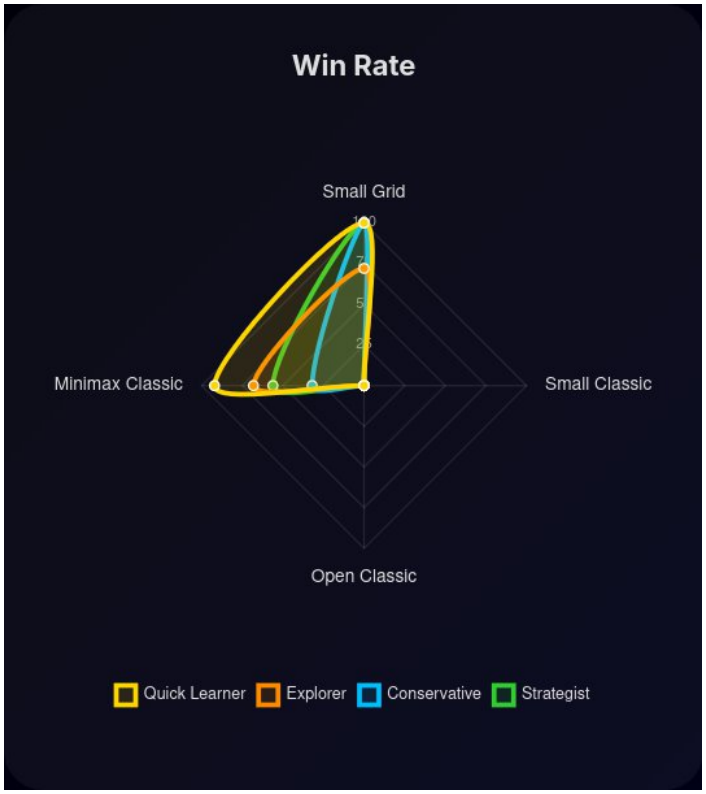
- **Layout: Small Classic**

5.4 PERFORMENCE COMPARISON

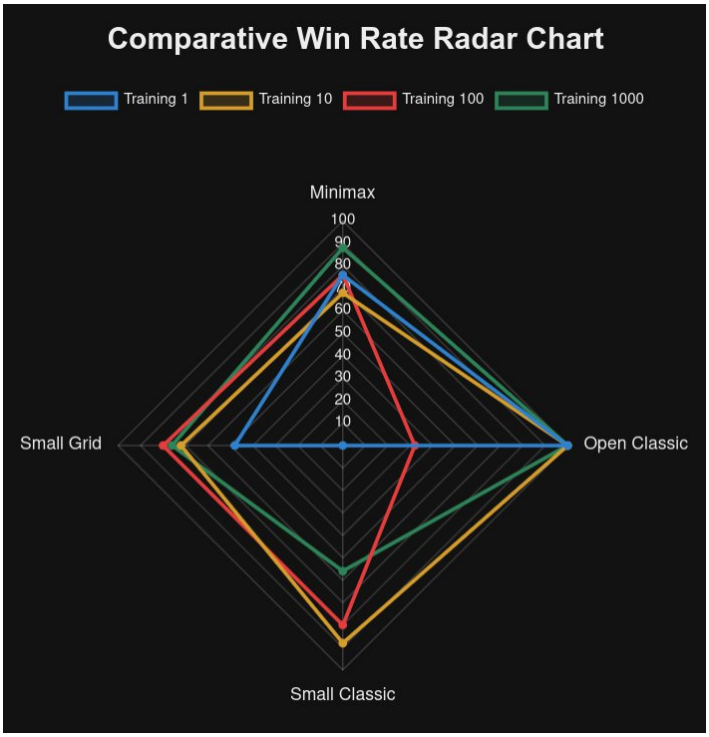
- **Performance Comparison Table for Q-Learning**

Model Name α | γ | ε	Layout	Average Reward During Training	Average Score After Training	Win Rate
Quick Learner 0.95 | 0.1 | 0.2	Small Grid	-102.61	503.7	100%
	Small Classic	-410.91	-379.28	0%
	Open Classic	-462.83	-438.44	0%
	Minimax	-85.37	434.96	92%
Explorer 0.7 | 0.5 | 0.5	Small Grid	-489.68	212.1	72%
	Small Classic	-429.47	-412.6	0%
	Open Classic	-477.97	-430.64	0%
	Minimax	-381.88	192.6	68%
Conservative 0.5 | 0.6 | 0.1	Small Grid	-51.16	501.9	100%
	Small Classic	-401.40	-399.12	0%
	Open Classic	-456.39	-475.56	0%
	Minimax	-209.55	-197.12	32%
Strategist 0.6 | 0.8 | 0.3	Small Grid	-412.41	498.7	100%
	Small Classic	-417.93	-387	0%
	Open Classic	-462.97	-444.12	0%
	Minimax	-271.38	70.24	56%

- **Spider Chart of Q-Learning Performance**



- **Spider Chart of Approximate Q-Learning Performance after Different Number of Training**



- **Q-Learning Performance Analysis**

Layout-wise Analysis

- **Small Grid:** The grid being tiny, the agents reaches optimized performance with Q-Learning after 5000 training. However, the explorer shows a 28% failure among all the successes, that shows 5000 training runs does not explore all states.
- **Small Classic:** A 0% success rate for all agents, suggesting for a little larger layout requires exponentially higher training.
- **Open Classic:** This time however, due to resource limitation, we could allow 3000 training. The result is another complete failure, with even worse score. Other than the limited training, another major factor here could've been the open space allowed for a much larger state space.
- **Minimax Classic:** The most interesting results are from Minimax. Agents perform drastically different. They also show much improvement over training rewards vs actual performance.

Agent-wise Analysis

- **Quick Learner:** Shows excellence in all environments, including the simpler and comparatively complex ones. But there is the consideration of number of training. Because, all our simulations were with upto 5000 training, which isn't many. Which is why naturally a quick short-sighted learner would do better.
- **Conservative:** For a similar reason, conservative learner was expected to perform worse. With low learning rate, low exploration, and eye on the future- it needs huge number of training to reach a optimized performance. It is designed to do good with a higher training in a long-term, but we have little scope to properly analyze that.
- **Explorer:** The explorer model shows an interesting characteristic. It performs poor in training period. But after training, it performs decently- that is it makes a great improvement. This is because the model explorers a lot in the training period, relying less on what it already knows. But that knowledge obviously stays in it. Which is reflected when exploration rate is released after training period.
- **Strategic:** Utilizes an even long term planning. And like conservative agent, it does not perform well overall within our test runs. However, strategist share some characteristics of the explorer. It explores and learns more during training period, which allowed it to perform better than conservative, even with eye at even far distance reward.

- **Approximate Q-Learning Performance Analysis**

    - Approximate Q-Learning showed a very interesting insight, it didn't necessarily improve with higher training.
    - In some cases, higher learning made it act more dumb. This is probably because the AI only knows the elements near it, not the full environment. So even when the AI see same features, the state and best action might be different. So more learning can lead to a confused agent.
    - We see the agent do better in open environments. In maze-like environments, it suffers. This is because it do not have information of the walls.
    - In larger and open-natured environments, Approx-Q can be optimized in 1 run given an optimal feature extractor- as it may explore most possible. In smaller layouts, it requires more training, because i grid layouts, it would suffer even with thousand

- **Comparative Analysis: Q-Learning vs Approximate Q-Learning**

    - Approximate Q-Learning did better in large open environments, where Q-Learning failed. And where Q-Learning shined, in grid environments, Approximate Q-Learning didn't perform as good.
    - Higher training always improves Q-Learning, but Approximate Q-Learning shows unstable performance.
    - With a lot of time and extreme hardware resource available, and more optimized resource management, Q-Learning can be the better performer for comparatively large environments as well. On the other hand, with constrained resources, Approximate Q-Learning would do better in most cases- but not all.

- **Additional Analysis:**
    - With the very different outcomes involved, to have a deeper understanding, we did some further simulation with minimaxLayout, as it showed the most interesting outcome. The findings again shows interesting results.

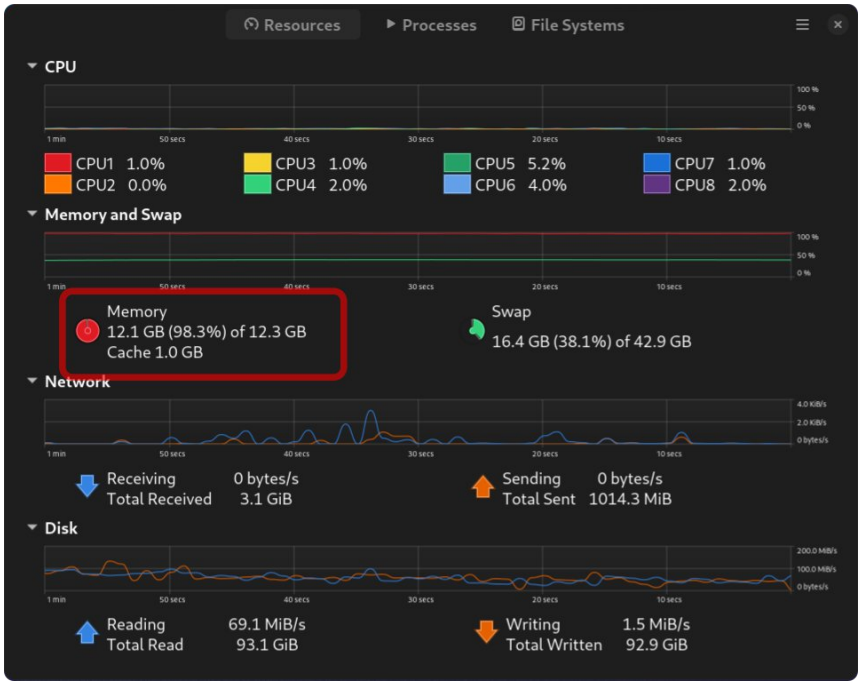
Model	Average Score After Training	Win Rate
Approximate (10000 Training)	54.98	41/75 55%
Quick Learner (100000 Training)	326.94	61/25 81%
Explorer (100000 Training)	366.97	64/75 85%
Strategic (100000 Training)	286.53	58/75 77%

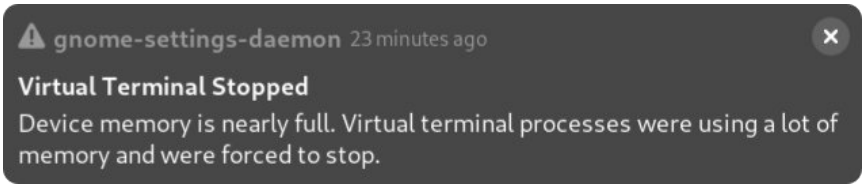
5.5 ERROR ANALYSIS AND LIMITATIONS

- **Error Analysis**
  - As we do not have a previously measured value, or a fixed expected value, there is nothing to analyze errors.
  - The analyzed results indeed show many interesting characteristics. We believe, they are not likely to be due to an error, but due to the very nature of RL.
  - We did face challenges including seemingly problematic data on our way, However, as we analyzed, we found out that there was mistakes in setting parameters or code from our side behind that.

- **Limitations**

  - State space for Q-Learning becomes exponentially larger with larger layouts and number of possible states. Our analysis was very limited due to computing power and time constrains.
  - Limitation in computing power did not allow us to simulate Q-Learning with a considerable number of training in original Pacman. This is why we had to go with the smaller layouts to work with.
  - We did face hardware limitations. With state-space representation expanding, RAM usage jumped up rapidly, resulting in lags and crashes- not allowing us to try extended training numbers.





  - Time constrain was a huge issue as well. Larger layouts took long time to simulate. Also we weren't allowed to simulate higher number of runs after training.

6. THE BROADER SCOPE AND APPLICATION

6.1 RL in Game Development

- Incorporating AI in games doesn't necessarily mean to make it more serious or human-like, the autonomous can agent behave in its way, but differently and intelligently in each runthrough making the game more engaging.
- In competitive games, eg. racing games, sports games- the autonomous agents or NPCs (Non-Playable Characters) are very important. So incorporating intelligent behavior would add a new level to the competition.
- One issue is AI has advantage in ways that is not humanly possible. It can train billions of times, can store and calculate huge amount of data. However, it still provides interesting insights. In the racing game Trackmania, AI has been developed and optimized to beyond human level by a community member Yosh, which has shown some great power of RL.
- However, application of AI in game does not need to be only to make it harder and serious- but AI introduces new ways to make a game creative and fun. Example would include Rain World, an indie 2D survival game, which has been praised for it's lively ecosystem. Although Rain World doesn't use RL, it uses other forms of ML to achieve that.

6.2 Application: The Simple Snake

- We feel these days games are too much about graphics and realism, and less about fun. And RL, while providing a framework to make game more real, also provides a framework to make games more fun and engaging.
- RL in games are greatly practiced in research. But we wanted to use our research knowledge in just simple fun game as well.

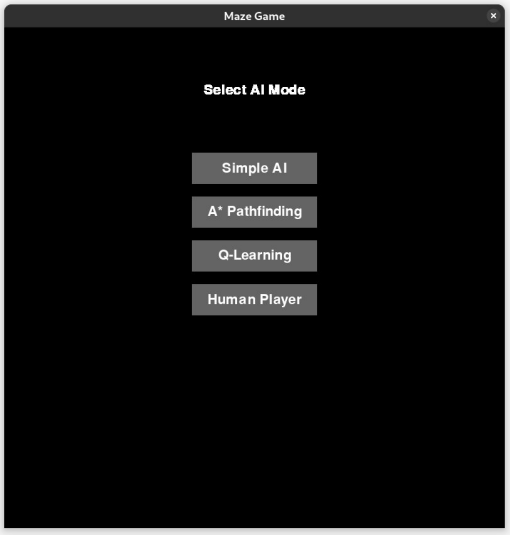
- **The Initial Idea**
- We outlined an idea of a game, that improvises the nostalgic snake game, with some inspiration from a 3d game AxySnake.
- Initial plan:

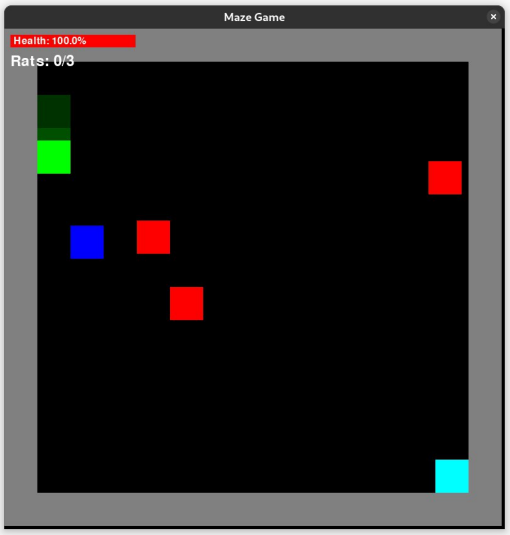
1. The game will have a 2D environment
2. Multi agent -
   Snake: Want to eat rats
   Bee: Creates distraction both rat and snake
   Rat: Wants to flee from snake
3. Mud → Slows down snake and rats,
   Wall → Prevents to move forward,
   Lava → Instant kill,
   Forest → Sensors disabled, state is unclear
4. Going beyond boundary teleports to opposite side
5. Movement -> Left, right, forward (no direct backwards), dies if no possible movement remaims
6. Sensors have heuristic data of wall distance, prey distances, mud distance etc
7. Needs to catch 3 rats in a level
8. 10 predefined level layouts in a seperate layout folder, also random generation options
9. There is one or multiple way out of the maze, if snake goes out before eating 3 rats, rats win. Snake needs to go out after eating 3 rats to win.
10. Game mode where player can play as bee, rat or snake
11. Game mode where all agents autonomous
12: Learning Method: Primarily Reinforcement (Q-learning), also other AI

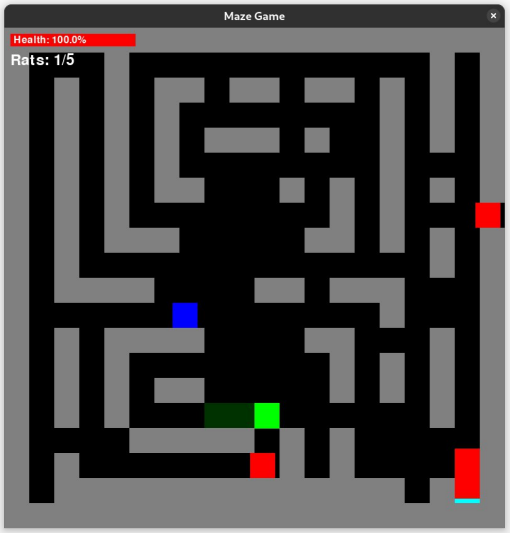
- All together builds a lively environment which has many considerations to choose an action. RL can allow not only to do some pre-training, but also to evolve with each actual playthrough.

- **The Real Implementation**
  - Developing the full game is practically beyond our capability for now, so we made a simplifed version.
    1. 2D grid-based maze game with walls, agents, and objectives.
    2. Core agents:
       - Snake (player/AI-controlled): hunts rats, avoids bee, exits maze after catching all rats.
       - Rats flee using heuristic AI (prioritize directions away from snake).
       - Bee damages snake on contact and respawns randomly.
    3. UI: Level selection, AI mode picker (human/Q-Learning/A*/Simple), health bar, rat counter.
    4. Victory: Exit maze only after catching all rats. Defeat occurs if health reaches 0%.
    5. Obstacles: Walls block movement. Snake loses health on wall collisions (cooldown system).
    6. Progression:
       - Collect all rats to unlock exit.
       - Snake's tail grows with caught rats (visualized via color gradient).
       - Health depletes on bee/wall hits; game ends at 0% health.
    7. AI Modes:
       - SimpleAI (greedy toward closest rat).
       - A* (optimal pathfinding).
       - Approximate Q-Learning (trained with feature-based rewards: target proximity, wall detection).
    8. Predefined levels: 5 mazes (size 15x15 to 30x30) with escalating rat counts and maze complexity.

- **Screenshots**:








```
Training Approximate Q-Learning AI with 10000 episodes...
Episode 0/10000
Episode 1000/10000
Episode 2000/10000
Episode 3000/10000
Episode 4000/10000
Episode 5000/10000
Episode 6000/10000
Episode 7000/10000
Episode 8000/10000
Episode 9000/10000
Approximate Q-Learning training complete.
```

- **How It Works**

SimpleAI Action Selection:

Target Priority:
If any rats remain, move toward the closest one using Manhattan distance. If all rats are caught, move toward the exit.

Direction Logic:
Calculate the difference in X and Y coordinates between the current position and the target. If the X difference is larger, prioritize left or right movement. Otherwise, prioritize up or down.

Try to move in the best direction first. If that is not possible, try the second-best direction. If neither is possible, pick a random valid direction.

AStarAI Action Selection:

Pathfinding:
Use Manhattan distance as a heuristic. Explore possible moves using a priority queue while avoiding walls and tracking movement costs.

Optimal Path:
Determine the shortest path to the target and follow the first step in that path.

QLearningAI Action Selection:

Feature Extraction:
Determine the direction to the target, inverse distance to the target, wall presence in nearby cells, and include a bias term.

Q-Value Calculation:
For each possible move, compute a Q-value based on the extracted features and learned weights.

Action Choice:
Select the move with the highest Q-value. Sometimes, choose a random move that is still biased toward the target direction.

Post-Rat-Catch Behavior (All AIs):
Once all rats are caught, switch the target to the exit. Each AI follows the same logic as before, recalculating paths or updating features based on the new target.

7. DISCUSSION AND INSIGHTS

7.1 REFLECTION OF MINDSET

Off course, human mindset and scenario is much more complex than a very simple game. But it's amazing how it reflects on human behavior. The explorer that was severely punished in it's learning time due to it's unpredictable behavior, still managed to perform well when it had to execute- because it learned.

8. IMPROVENT SCOPES

8.1 ANALYSIS AND DEVELOPMENT

- The project was limited by time and resource constrains. So with more resources, more simulation better analysis can be achieved.
- Research can be utilized to develop actual games, that is the application.

9. ETHICAL CONSIDERATIONS

9.1 ETHICAL ISSUES

- The project analyzes a fundamental learning method. It doesn't involve any major ethical issues.
- There are obviously the general concerns of incorporating human behavior in machines. But as in this project limits it within a game environment, those are not much relevant.

8.2 SUSTAINABILITY

- Regular Q-Learning requires exponentially high amount of training in complex environment, which may not be sustainable in large projects.
- Approximate Q-Learning is more computationally effective, as it approaches optimization with less training.

10. CONCLUSION

10.1 THE NEED OF APPLICATION

- There has been many research works and models based on RL. But interestingly, we do not see as many games intended primarily to be played to utilize this. This shows an issue of modern time, as much as we are invested in research and advancing knowledge and tech- we aren't that much Interested in utilizing what we already have.
- Further research can be worthwhile, applying the knowledge we have already gained may be even more compelling. The amazing aspect of Reinforcement Learning is the possibility of applications is endless. As we focused on game development, this is certainly a part where there's a great room for applications. Specially the way RL allows to design a dynamic world, can hardly be beat by other means. Our snake game is a basic and incomplete prototype for now, but we intend to develop it further.

11.REFERENCES

- UC Berkeley CS188 Intro to AI -- Course Materials
- Lab Manual – CSE 4618 – Artificial Intelligence Lab