# Report on Q-Learning

## Link:

**Youtube:**<https://youtu.be/lxyI9S-M0Y8>

**GitHub:**<https://github.com/boot-chuang/q-learning.git>

## Introduction

In recent years,reinforcement learning has emerged as a powerful paradigm for training agents to perform complex tasks through trial and error.Among various reinforcement learning algorithms,Q-Learning has gained significant attention due to its simplicity and effectiveness in solving sequential decision-making problems.This report presents an implementation of Q-Learning applied to a simple 2D space shooter game,where the goal is to train an agent to control a player character and defeat enemy ships using a combination of movement and shooting actions.

The game environment is implemented using the Pygame library,which provides a straightforward way to create graphical interfaces and handle user interactions.The Q-Learning algorithm is used to train the agent to make optimal decisions based on the current state of the game.The state representation is derived from the relative positions of the player and enemies,and the agent can perform four possible actions:moving left,moving right,shooting,or waiting.

This report is organized into several sections:Methodology,Main Function Description,Evaluation Metrics and Result s,Discussion,and Conclusion.Each section provides a detailed explanation of the implementation,evaluation,and analysis of the Q-Learning-based game Al.

## Methodology

### Q-Learning Algorithm Overview

Q-Learning is a model-free reinforcement learning algorithm that aims to learn the optimal policy for an agent by iteratively updating a Q-value table.The Q-value represents the expected cumulative reward for taking a specific action in a given state and transitioning to a new state.The algorithm follows the following update rule:



**Where:**

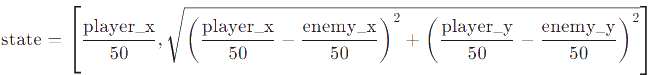
·Q(s,a) is the Q-value for states and action a.

·α is the learning rate,which controls the step size of the update. ·r is the immediate reward received after taking action a in states.

·γ’ is the discount factor,which determines the importance of future rewards. · s' is the new state resulting from taking action a.

### Implementation Details

In this implementation,the Q-Learning algorithm is applied to a discrete state space where each state is represented b y the relative positions of the player and enemies.The state vector is constructed as follows:



The player can take four actions:

1.Move left

2.Move right

3.Shoot

4.Wait

The reward function is designed to encourage the agent to defeat enemies while penalizing excessive movement.Specifically

·A reward of+500 is given for hitting an enemy.

·A reward of+3000 is given for defeating all six enemies.

·A penalty of-50 is applied for each movement action(left or right).

·A penalty of-1000 is applied if the player is hit by an enemy or if any enemy reaches the bottom of the screen.

The exploration-exploitation trade-off is managed using an epsilon-greedy policy,where the agent randomly selects a n action with probability e and chooses the action with the highest Q-value otherwise.The exploration rate e decreases over time according to E=E×EPS\_DECAY,where EPS\_DECAY=0.9998.

## Main Function Description

The main function of the code initializes the game environment,sets up the Q-Learning parameters,and runs the training loop for 20,000 episodes.Below is a high-level description of the main components:

1**.Game Initialization**:The Pygame library is used to set up the graphical interface,load images,and define game objects such as the player,enemies,and bullets.

**2.Stae Representation**:The get\_state()function computes the current state of the game by discretizing the player's position and the relative distances to enemies.

**3.Action Selection:** The agent selects an action based on the epsilon-greedy policy.If a random number is less than e,a random action is chosen;otherwise,the action with the highest Q-value for the current state is selected.

**4.Action Execution:** The player's action is executed,and the game state is updated accordingly.This includes moving the player,firing bullets,and updating the positions of enemies and bullets.

**5.Reward Calculation**:The reward for the current step is calculated based on whether the player hits an enemy,gets hit by an enemy,or defeats all enemies.

**6.Q-Table Update:The** Q-value for the current state-action pair is updated using the Q-Learning formula.

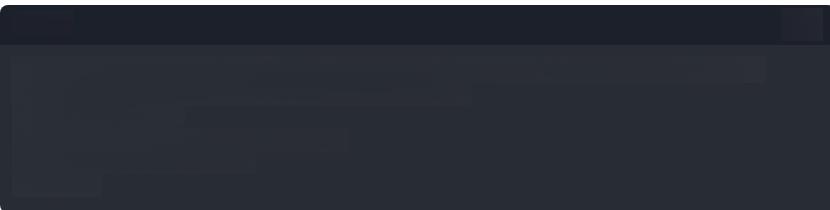
**7.Termination Check:T**he episode terminates if the player is defeated or all enemies are defeated.

**8.Training Loop:The** training loop runs for 20,000 episodes,with periodic updates to track the mean reward over 2000 episodes.

## Evaluation Metrics and Results

### Reward Analysis

The performance of the agent is evaluated based on the cumulative reward obtained over episodes.Figure 1 shows the moving average of rewards over 2000 episodes,which indicates the agent's learning progress.Initially,the agent explores randomly and receives negative rewards due to frequent colisions with enemies.As training progresses,the agent learns to avoid enemies and shoot them effectively,leading to higher rewards.



**Python** 复制

moving\_avg =np.convolve(episode\_rewards,np.ones((update\_steps,))/update\_steps, mode='valid) plt.plot([i for i in range(len(moving\_avg))],moving\_avg)

plt.xlabel('Episode')

plt.ylabel(f'Mean{update\_steps}Reward')

plt.savefig('train\_curve.png') plt.show()

The final mean reward stabilizes at around 2500,indicating that the agent has successfully learned to defeat all enemies consistently.

### Hit Rate

The agent's ability to hit enemies is another important metric.The code tracks the number of hits required to defeat six enemies.In well-trained agents,this number approaches six,meaning every shot hits an enemy.The training process shows that the agent gradually improves its accuracy,reducing wasted shots over time.

## Discussion

The results show that the Q-Learning algorithm can train an agent to do well in this simple game. The agent learns to balance trying new things and using what it knows, and it relies less on random actions as it gains experience. The reward system promotes both attacking enemies and avoiding collisions, resulting in a strong strategy.However, there are limitations to this implementation. First, the state representation is relatively simplistic and may not capture all relevant information about the game environment. For example, the agent does not explicitly track the positions of multiple enemies beyond their relative distances. Second, the reward function could be improved by introducing additional incentives for efficient movement or strategic positioning.

Future work could explore more sophisticated state representations, such as using deep neural networks to process raw pixel data, or implementing hierarchical reinforcement learning to handle complex tasks at different levels of abstraction.

## Conclusion

This report presents a successful implementation of Q-Learning in a simple 2D space shooter game. The agent learns to make optimal decisions by balancing exploration and exploitation, ultimately achieving consistent success in defeating enemies. While there are opportunities for improvement in terms of state representation and reward design, this implementation demonstrates the potential of reinforcement learning in game AI applications.

The results suggest that Q-Learning can be a viable approach for training agents in environments with discrete state spaces and sparse rewards. By carefully designing the reward function and adjusting algorithm parameters, it is possible to achieve stable and effective learning outcomes.