AI Final Report

Train a game player agent to play Text World

Group 5
徐御倫 R06922138
林依蓁 R06944003
陳郁文 B05902071

Work Distribution

徐御倫	林依蓁	陳郁文
● Q-learning agent ● Poster ● English Presentation	● Built the basic agent ● Built the Q-Learning agent ● Presented in Chinese	● Detect how to use API from textworld to make a easier game

Motivation

Text-based games are complex, interactive simulations in which text describes the game state and players make progress by entering text commands.

In this project, we created an agent using Microsoft TextWorld framework to make a sequential decision-making model that takes text information as input and outputs text commands to progress through a game

Textworld environment allows agent to easily observe the environment in the game and retrieve the game state and control the partial observability of the state.

"Text World" - To Generate a game

- Command tw-make custom
  - Generate a customized textworld game
- Try to reduce action : generate a game by ourselves
  - API provided by textworld
  - New_room , connect , record_quest , test
- Difficulty
  - Random generate room , corridor
  - Modify the reward function and the test()

Why is text-based game challenge

Partially observable: Agent can only see the local information such as current room and player’s inventory. Some important detail of environment might miss out

Eg: What’s inside the chest and what kind of key open the chest

Large Space: In textworld game, action space is enormous and finding solution may be challenging.

Exploration and Exploitation: Player must explore the environment, then react to the environment with interaction, such as use, take, etc.

Long-term Credit Assignment: Knowing which actions were responsible for obtaining a certain reward, especially when rewards are sparse, is another fundamental issue in RL

Textworld Solution

TextWorld’s generated text-based games can be represented internally as a Markov Decision Process (MDP) defined by (S, A, T, R, γ),

S: Environment State

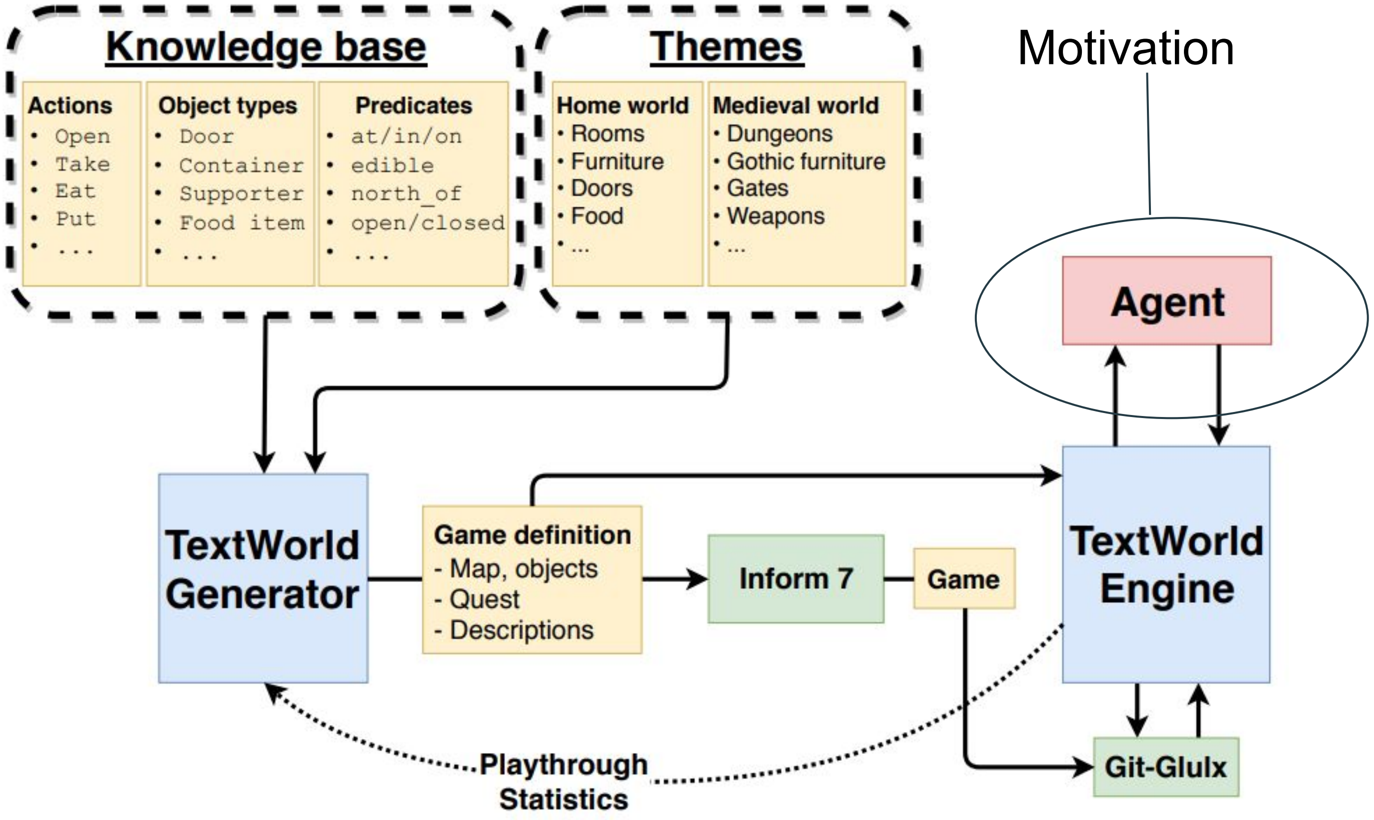
A: Set of Action

T: State Transtion Function

R: Reward

γ: Epsilon discount factor

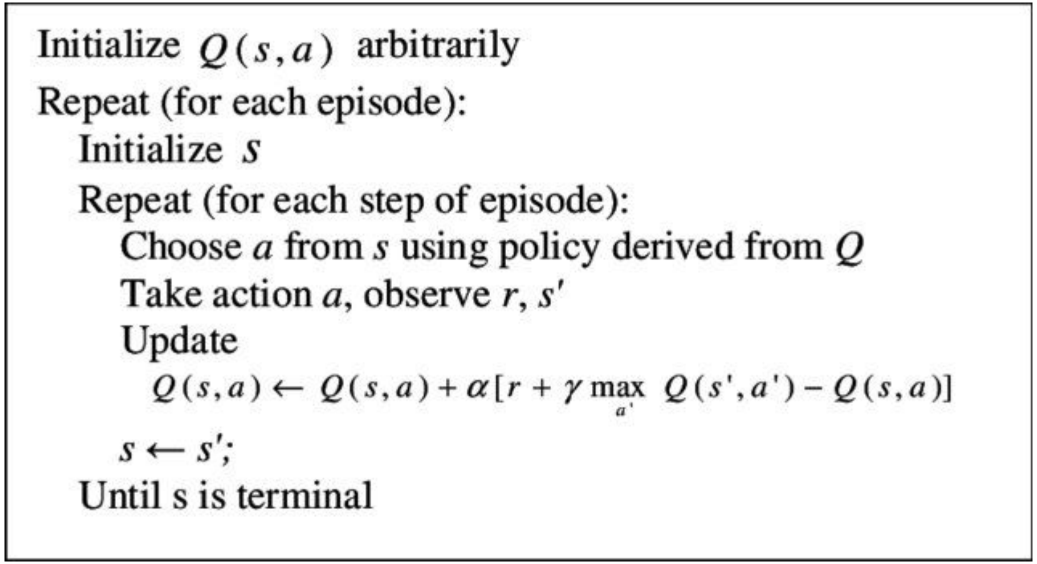
In this project, we implemented Q agent learning model and provide intermediate rewards during training based on environment state transitions and the ground truth winning policy



Knowledge base
Actions
• Open
• Take
• Eat
• Put
• ...
Object types
• Door
• Container
• Supporter
• Food item
• ...
Predicates
• at/in/on
• edible
• north_of
• open/closed
• ...
Themes
Home world
• Rooms
• Furniture
• Doors
• Food
• ...
Medieval world
• Dungeons
• Gothic furniture
• Gates
• Weapons
• ...
Motivation
Agent
TextWorld Generator
Game definition
- Map, objects
- Quest
- Descriptions
Inform 7
Game
TextWorld Engine
Git-Glulx
Playthrough Statistics

Methodology (簡單版)

Q-Learning Algorithm



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$
        Take action $a$, observe $r$, $s'$
        Update
            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    Until s is terminal
```

Source:
https://www.researchgate.net/publication/220776448_Cooperation_Between_Multiple_Agents_Based_on_Partially_Sharing_Policy/figures?lo=1

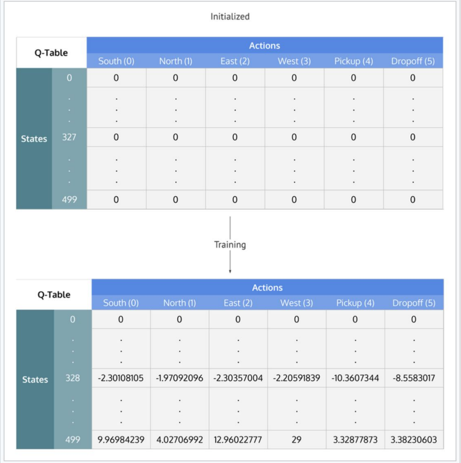
Methodology (複雜版)

The weight for a step from a state Δt steps into the future is calculated as $\gamma^{\Delta t}$. γ (the *discount factor*) is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : S \times A \to \mathbb{R} .$$

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:



Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Q–Learning table of states by actions that is initialized to zero, then each cell is updated through training.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $(0 < \alpha \leq 1)$.

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, Q–learning can also learn in non–episodic tasks.[*citation needed*] If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for state s_f. In most cases, $Q(s_f, a)$ can be taken to equal zero.

來源 : wiki
https://en.wikipedia.org/wiki/Q-learning

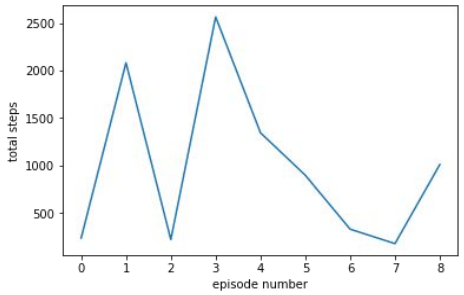
Implementation (Experiment & Result)

Basic Agent

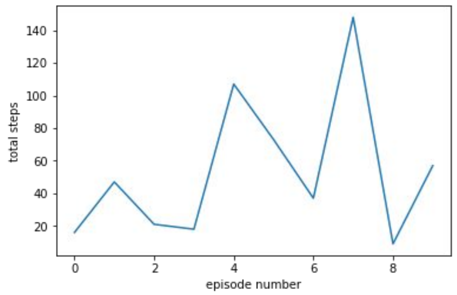
- Experiment
  - Take the room name as the state (there are 5 values).
  - Get the command list with game_state.admissible_commands.
  - Randomly take a command from the command list and pass it into the env.step(command) function.
  - Run 10 episodes / 5000 steps in a episode
  - **Compare the results for the different command list elements.**
- Results
  - Command list contains all admissible_commands
    - avg. steps: 1386.1; avg. score: 0.9 / 1; 110 explored actions
  - Command list only contains the admissible_commands which start with go, insert, take
    - avg. steps: 53.3; avg. score: 1.0 / 1; 28 explored actions

Implementation (Experiment & Result)

- Experiment 1
- avg. steps: 1386.1
- avg. score: 0.9 / 1



- Experiment 2
- avg. steps: 53.3
- avg. score: 1.0 / 1



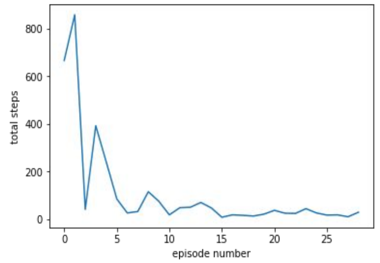
Implementation (Experiment & Result)

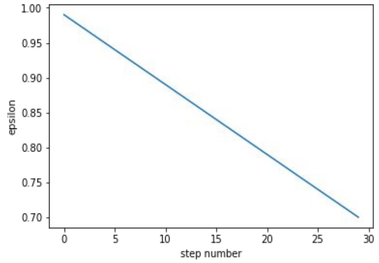
Q-Learning Agent

- Experiment
  - Take the room as the state. (There are 5 values)
  - Get the candidate commands list with game_state.admissible_commands
  - Get the immediate reward with env.compute_intermediate_reward() function, put (state, command, intermediate_reward) data into the Q-table, and keep updating the Q-table.
  - 50% probability to randomly take a command from the command list, 50% probability to take the highest score action from the Q-table, and pass it to the env.step (command) function.
  - Run 30 episodes / 1000 steps in a episode.
  - **Compare the results for different epsilon formulation.**
- Results
  - epsilon = episode - 0.01; avg. steps: 137.6; avg. score: 1.0 / 1.
  - epsilon = math.pow(0.95, episode_num); avg. steps: 68.0; avg. score: 1.0 / 1.
  - epsilon = 1/math.pow(episode_num+1, 2); avg. steps: 18.1; avg. score: 1.0 / 1.

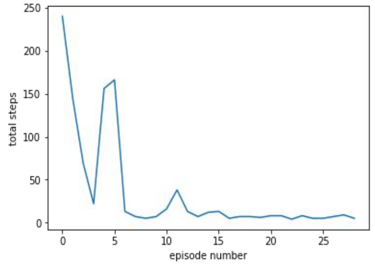
Implementation (Experiment & Result)

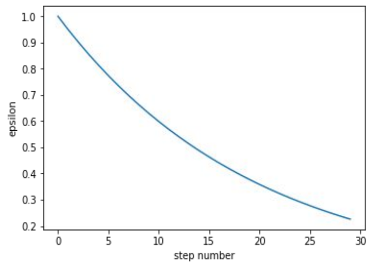
- Experiment 1
- avg. steps: 137.6;
- avg. score: 1.0 / 1.



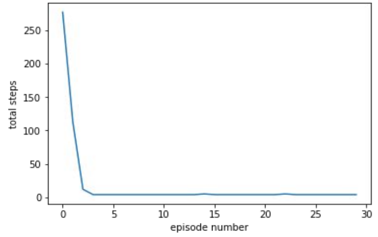


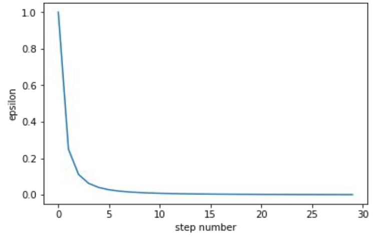
- Experiment 2
- avg. steps: 68.0;
- avg. score: 1.0 / 1.





- Experiment 3
- avg. steps: 18.1;
- avg. score: 1.0 / 1.





Conclusion

- Using the Q-learning algorithm to train the agent performed very well. After about 10 episode training, the number of steps to complete the game can converge to dozens of steps, close to the ability of a human newbie.
- The conclusion is that when the formula of epsilon is 1 / t ^ 2, the performance is the best and the training process is the most stable.

Reference

- https://github.com/Microsoft/TextWorld/blob/master/notebooks/Handcrafting%20a%20game.ipynb
- https://github.com/xingdi-eric-yuan/TextWorld-Coin-Collector
- https://github.com/microsoft/textworld
- https://textworld.readthedocs.io/en/latest/
- Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes,Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada,Wendy Tay, and Adam Trischler. 2018. TextWorld: A Learning Environment forText-based Games.CoRRabs/1806.11532 (2018).