

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС
"ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ"
НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”
КАФЕДРА СИСТЕМ АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ

Курсова робота

з дисципліни «Об’єктно-орієнтоване програмування»

*на тему: «Дослідження та порівняння деяких технік навчання з
підкріпленням на базі задачі «Лабіринт»»*

РОБОТУ ВИКОНАВ:

студент групи ДА-82

Зайцев К.Ю.

Прийняв:

Булах Б.В.

„___” _____ 2019

ЗМІСТ

ЗМІСТ.....	2
ВСТУП.....	3
РОЗДІЛ І	5
1.1. Класифікація лабіринтів	5
1.2. Алгоритми вирішення лабіринту	7
1.3. Базові концепції навчання з підкріпленням	9
1.4. Глибоке навчання з підкріпленням.....	12
РОЗДІЛ ІІ.....	16
2.1. Розробка програми	16
РОЗДІЛ ІІІ	22
ВИСНОВКИ.....	27
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	30

ВСТУП

Проблема взаємодії з середовищем, маючи неповні відомості про його структуру є активною сферою досліджень, починаючи з другої половини минулого століття. Прийняття рішень, коли агент, який приймає рішення, знає про різні можливі стани оточення, але не має достатньо інформації, щоб зіставити їм відповідну до досягнення мети оцінку, називається прийняттям рішень у стані невизначеності. Рішення в умовах невизначеності - це існування багатьох невідомих і неможливість визначити, що може статися в майбутньому, щоб отримати бажаний результат.

Базовим середовищем для моделювання таких ситуацій виступає лабіринт. Дослідження поведінки model-free агентів, можливість зробити з дискретного лабіринту неперервний, що міститиме надвелику для комп'ютерних обчислень кількість клітин і т.д. – мотивація використання цього середовища для тестування теоретичних здобутків у навчанні з підкріпленням.

Робот робить крок вперед, потім наштовхується на перешкоду. Наступного разу він робить інший крок і здатний просунутися у лабіринті. Робот багато разів пробує різні комбінації варіантів; Врешті-решт, він засвоює правильні кроки, які потрібно робити, і робить їх на шляху до виходу. Цей процес і називається навчанням з підкріпленням. Він безпосередньо пов'язує дію робота з результатом, при цьому робот не повинен засвоювати складний взаємозв'язок між його дією та результатами. Робот вчиться ходити на основі винагороди (перебування на рівновазі) та покарання (падіння). Цей відгук вважається "підкріпленням" для того, щоб робити чи не робити дію.

Ще один приклад навчання підкріплення можна знайти під час гри в гру Go. Якщо агент відкладе свій білий камінь у деякому місці, а потім потрапить в оточення чорних шматочків і втратить цей простір, його карають за такий крок. Після декількох поразок він уникатиме ходів, що будуть ставити власні камінчики в оточення, тим самим зменшивши ризик їх програти.

Спрощеним визначенням навчання з підкріпленням - є вивчення кращих дій, заснованих на винагороді чи покаранні. У зв'язку з ним існує три основні поняття: стан, дія та винагорода. Середовище описує поточну ситуацію. Для

робота, який навчається ходити, стан - це положення його двох ніг. Для програми Go - стан - це позиції всіх камінців на дошці.

Дія - це те, що може зробити агент у кожному положенні. Враховуючи стан або положення двох його ніг, робот може робити кроки на певну відстань. Зазвичай існують обмежені дії у фіксованому діапазоні, які здійснює агент. Наприклад, кроком робота може бути, скажімо, 0,01 метр на 1 метр. Програма Go може зробити хід лише в одній клітинці на всій дошці, розміром 19 x 19.

Коли робот здійснює дію, він отримує винагороду. Тут термін «винагорода» - це абстрактне поняття, яке описує зворотний зв'язок від навколишнього середовища. Нагорода може бути позитивною чи негативною. Коли винагорода є позитивною, вона відповідає нормальному розумінню винагороди. Коли винагорода є негативною, вона відповідає тому, що ми зазвичай називаємо «покаранням».

На прикладі лабіринту ми дослідимо основні положення та практики навчання з підкріпленням та його переваги та недоліки порівняно зі стандартними методами вирішення подібних задач.

За мету для даної роботи було поставлено створення емулятора для моделювання лабіринтів різної складності та агентів, що їх вирішують; імплементацію різноманітних алгоритмів навчання з підкріпленням та їх порівняння з алгоритмами пошуку найкоротшого шляху на графі – структурою даних, якою можна представити лабіринт.

У відповідності з метою роботи поставлені наступні завдання:

- проаналізувати стан проблеми в науковій літературі та Інтернет ресурсах;
- запрограмувати алгоритми з різних сімейств навчання з підкріпленням мовою Python;
- розглянути та впровадити парадигми та основні принципи ООП;
- застосувати низку шаблонів проектування для реалізації взаємодії між компонентами програми;

РОЗДІЛ I

ТЕОРЕТИЧНИЙ АНАЛІЗ НАУКОВОЇ ЛІТЕРАТУРИ ТА ВИХІДНІ ПОНЯТТЯ РОБОТИ

Лабіринт - це сітчастий n -мірний (зазвичай двовимірний) простір довільної форми, як правило, прямокутної. Комірка - це елемент лабіринту. Комірка визначається як обмежений n -мірний простір, як правило, двовимірна область і інтерпретується як позиція агента.

Термін "агент" визначається як сутність, яка здатна рухатися в межах лабіринту. Агент може рухатися в будь-якому довільному напрямку за умови, що простір не зайнятий перешкодою або рух не порушує перешкоду. Таким чином, ми можемо визначити два типи перешкод: перешкоду, яка повністю займає комірку, що називається стінкою, і перешкоду, яка розділяє дві сусідні комірки, називають перегородкою [5].

1.1. Класифікація лабіринтів

У типових проблемах цього типу агент ініціалізується у певній стартовій позиції, з якої він починає шукати вихід. Місце розташування цілі може знаходитися всередині меж лабіринту, наприклад, розташування віртуальної їжі або міфологічного створіння, або може бути точкою виходу з лабіринту. Проблеми лабіринту можуть бути класифіковані відповідно до властивостей лабіринту та властивостей агентів, що його перетинають [5].

Розмірність лабіринту визначає кількість вимірів, які має лабіринт, наприклад, планарний лабіринт використовує два виміри, а кубічний лабіринт - три. Загалом, лабіринт може використовувати будь-яку кількість вимірів, хоча для людини буде складно уявити більш ніж три. Розмір лабіринту - це кількість комірок, що складають лабіринт. Він зазвичай сильно пов'язаний з довжиною шляху, що з'єднує стартову та цільову позиції, або відстань. Відстань визначається також перешкодами в лабіринті, які можна описати або абсолютною кількістю кроків, або, що значно важливіше, щільністю перешкод, тобто відношенням кількості зайнятих комірок до величини лабіринту. Щільний лабіринт може збільшити відстань, але не обов'язково складність. Зокрема, лабіринтом вважається середовище, в якому можливий

лише один звивистий шлях, таким чином обхід лабіринту від початку до кінця може бути тривалим, але не складним.

Перешкоди також надають агентам більше інформації про сенсор у вигляді відмінних форм. Для прикладу, порожній лабіринт не надасть агенту ніякої картографічної інформації, якщо агент не здатний сприймати та обробляти весь простір лабіринту. Складніший випадок, коли агент не в змозі розмежувати між повторюваними патернами у лабіринті, явище, яке називається псевдонімом. Перешкоди також визначають топологію лабіринту [5].

Лабіринт може бути представлений графіком G , що складається з вузлів, що представляють собою комірки, і ребер, кожне з яких являє собою прохідний шлях, що з'єднує пару комірок. У загальному випадку граф може містити підграфи, що не з'єднані між собою, роблячи таким чином частини лабіринту недоступними для агента. Якщо граф з'єднаний, є два можливих випадки для графічного ведення від початкової точки до мети: лише один шлях з'єднує дві точки, або більше одного шляху з'єднують їх. Перший випадок може бути представлений деревом, а другий - просто графом (рис. 1).

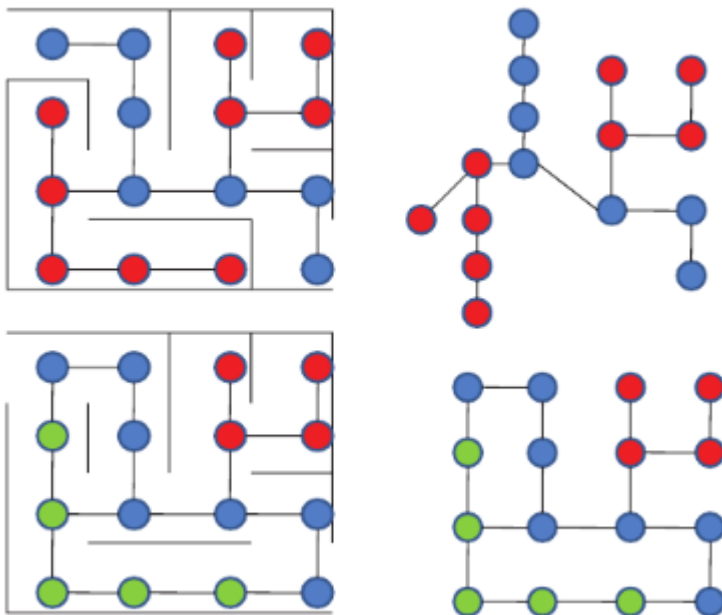


Рисунок 1. Лабіринти представлені деревом (верх) чи графом (низ)

Апріорні знання, які має агент щодо лабіринту, також виступають важливим фактором. Їх об'єм може варіюватися в інтервалі між повною відсутністю та абсолютним знанням.

У першому випадку агент повинен використовувати власні сенсорні можливості для вивчення та картографування навколишнього середовища. Після того як картографування зроблено, агент може спланувати шлях, який веде його від початку до кінця. Якщо агент має абсолютне апріорне знання, він може просто спланувати шлях до мети. Однак, апріорні знання є необхідною, але недостатньою умовою успішного планування шляху, оскільки не всі лабіринти є статичними.

Як правило, в лабіринті є три різні типи змін [5]:

- 1) Зміни структури лабіринту, тобто стіни чи перегородки змінюють власне розташування;
- 2) Зміни місця розташування цілі;
- 3) Зміни, викликані рухом інших агентів у лабіринті, у випадку групи з кількома агентами.

1.2. Алгоритми вирішення лабіринту

У більшості випадків задача звучить наступним чином: дано лабіринт у формі бінарної прямокутної матриці, знайти найкоротший шлях в лабіринті з заданих позицій початку та фінішу. Шлях може складатися з клітин, що мають значення 1, і в одну одиницю часу робиться лише один крок. Валідними кроками є: вгору, вниз, вліво, вправо.

Алгоритм Лі

Одним з можливих рішень для проблеми лабіринту є рішення, що базується на використанні Breadth-First Search. Воно завжди дає оптимальний результат, якщо такий існує, але є повільним та вимагає значних витрат пам'яті. Такі алгоритми мають спільну структуру:

1. Створити пусту чергу та закинути в неї клітину-початок лабіринту, позначити її як відвідану.
2. Доки черга не пуста, повторювати:
 - a. Забрати з черги передній елемент.
 - b. Якщо він є фінішем, повернути дистанцію.
 - c. Інакше для кожної з 4 прилежних до даної клітин, перевірити чи є вони вільними та закинути їх у чергу з

дистанцією +1 та позначкою, що вони відвідані.

- d. Повернути false, якщо всі клітини відвідані та фініш не знайдено.

Часова складність даного алгоритму $O(MN)$, де M , N – параметри лабіринту.

Алгоритм Дейкстри

Алгоритм Дейкстри дозволяє обчислити найкоротший шлях між одним вузлом та кожним іншим вузлом у графі, яким можна представити лабіринт. Вершини – клітини лабіринту, ребра – зв'язок між сусідніми вільними клітинами. Обчислимо найкоротший шлях між вузлом S та іншими вузлами нашого графіка:

1. Позначити початкову вершину та створити масив з відстаней до усіх інших вершин, позначивши всі відстані нескінченністю.
2. Вибрати вершину з найменшою відстанню до неї як поточну.
3. Для кожної сусідньої вершини: додати до поточної відстані відстань до даної вершини. Якщо вона менша за записану в таблиці – оновити таблицю.
4. Позначити поточну вершину як відвідану.
5. Якщо залишились не відвідані вершини – перейти до кроку 2.

Часова складність алгоритму Дейкстри – $O(E+V\log V)$, де E , V – кількість ребер та вершин відповідно. Суттєвим мінусом алгоритму є відсутність будь-якого «відчуття» вірної траєкторії руху, що на практиці часто призводить до серії зайвих проходів по лабіринту.

Алгоритм A^*

Алгоритм Greedy Best-First-Search працює аналогічно алгоритму Дейкстри, за винятком того, що він має деяку оцінку (що називається евристичною) того, наскільки далеко від цілі знаходиться деяка вершина. Замість вибору вершини, найближчої до початкової, вона вибирає вершину, найближчу до мети. Жадібний Best-First-Search не гарантовано знайде найкоротший шлях. Однак він працює набагато швидше, ніж алгоритм Дейкстри, тому що він використовує евристичну функцію для того, щоб

знайти шлях до фінішу дуже швидко.

A* алгоритм об'єднує інформацію, яку використовує Алгоритм Дейкстри (надаючи перевагу вершинам, близьким до вихідної точки), та інформацію, яку використовує Greedy Best-First-Search (надаючи перевагу вершинам, близьким до мети). $g(n)$ являє собою довжину шляху від вихідної точки до будь-якої вершини n , а $h(n)$ являє собою евристичну оцінюючу довжину від вершини n до фінішу. На наведених діаграмах жовтий (h) являє собою вершини, далекі від мети, а teal (g) являє собою вершини, далекі від вихідної точки. A* врівноважує два, рухаючись від початкової точки до мети. В основному циклі алгоритм на кожному кроці обирає деяку вершину n , яка має найменшу відстань $f(n) = g(n) + h(n)$. Алгоритм має наступний вигляд:

- i. Обрати клітину у списку з доступними клітинами з найменшим $f(n)$.
- ii. Додати обрану клітину до списку вершин, що не розглядаються.
- iii. Для кожної сусідньої клітини T :
 1. Якщо T у списку вершин, що не розглядаються – ігнорувати.
 2. Якщо T не у списку доступних клітин – додати T та обчислити $f(T)$.
 3. Якщо T у списку доступних клітин – перевірити поточний шлях до поточної вершини та порівняти його зі шляхом у $T +$ від T до поточної вершини. Якщо відстань менша – оновити $f(n)$ та f усіх батьківських вершин.

Часова складність алгоритму – $O(E)$. Основним його недоліком є використання пам'яті для зберігання додаткових масивів, що рівне $O(b^d)$, де b – середня кількість дочірніх елементів по усім клітинам, d – глибина оптимального шляху.

1.3. Базові концепції навчання з підкріпленням

Навчання з підкріпленням визначається як метод машинного навчання, який вивчає, як програмні агенти мають діяти у деякому середовищі. Згідно зі Стівеном Хуангом [7], основні терміни визначаються наступним чином:

- **Агент:** це деяка сутність, яка певним чином діє в середовищі, щоб

досягти заданої мети, отримуючи певну винагороду.

- **Середовище (ϵ):** сценарій, у якому знаходиться агент.
- **Винагорода (R):** міра того, на скільки вдалою чи невдалою була остання дія агента у середовищі.
- **Стан (s):** стан посилається на поточну ситуацію, у якій перебуває середовище.
- **Політика (π):** Це стратегія, за допомогою якої агент приймає рішення як діяти у певній ситуації, і яка застосовується агентом для дослідження середовища.
- **Цінність (V):** очікувана винагорода від дій агента згідно з поточною політикою, починаючи з певного стану середовища.
- **Функція цінності:** вона визначає вартість держави, яка є загальною сумою винагороди. Це агент, якого слід очікувати, починаючи з цього стану.
- **Модель середовища:** сутність, що імітує поведінку навколишнього середовища. Агент вчиться у створеній моделі, вивчаючи її та покращуючи політику дій.
- **Model-based method:** це метод вирішення задач навчання з підкріпленням на основі знань про середовище.
- **Q-значення (або значення дії) (Q):** числова характеристика якості дії у певному стані середовища. Якщо цінність є характеристикою самих станів, Q-значення оцінює усі можливі дії для певного стану на базі його цінності.

Як працює RL?

Базове навчання з підкріпленням включає такі кроки (рис. 1):

- 1) моніторинг середовища;
- 2) вирішення як діяти, використовуючи певну стратегію;

- 3) відповідна дія;
- 4) отримання винагороди чи штрафу;
- 5) навчання на досвіді та вдосконалення початкової стратегії;
- 6) ітерувати до тих пір, поки не буде знайдена оптимальна стратегія;

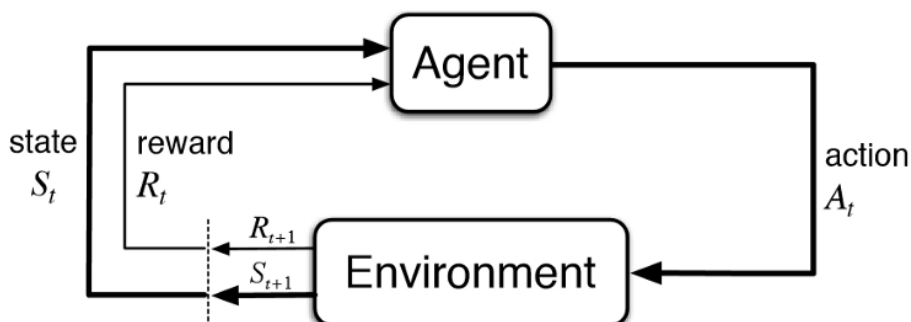


Рисунок 1: Базова схема роботи навчання з підкріпленням

Існує 3 основних підходи, коли мова йде про реалізацію алгоритму RL:

- На основі цінності - в методі навчання з підкріпленням на основі цінності агент намагається максимально використовувати значення $V(s)$. Основний фокус - пошук оптимального значення V^* .
- На основі політики - в методі навчання, що базується на політиці, цільовою є така політика, щоб дія, яка виконується в кожному стані, була оптимальною для отримання максимальної винагороди в майбутньому. Основна увага приділяється пошуку оптимальної політики.
- На основі моделі - у цьому типі навчання з підкріпленням експертом створюється модель для кожного середовища, і агент вчиться діяти в цьому конкретному середовищі.

Markov Decision Process

Процес прийняття рішень Маркова (Markov Decision Process) – формальний опис взаємодії між середовищем та агентом, що певним чином в ньому діє (рис. 2). Даний фреймворк вирізняє залежність не тільки між деяким станом середовища та зчиненою дією, а й вплив попередніх станів середовища з історії станів [6]. MDP можна зобразити набором з (S, A, P, R, γ) :

S – набір станів

A – набір дій

$P(s, a, s')$ – вірогідність, що дія a у стані s у час t приведе до стану s' .

$R(s, a, s')$ – винагорода після переходу у стан s'

γ – discounted factor, використовується для формулювання проблеми з врахуванням майбутніх винагород.

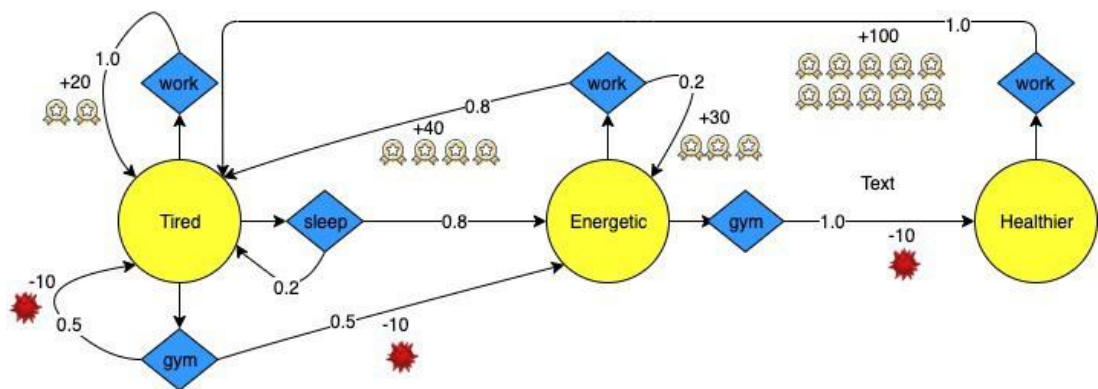


Рисунок 2: Приклад проблеми, що описується як MDP-процес

Рівняння Беллмана

Для визначення state-value функції $V(s)$ існує рівняння у вигляді:

$$V^{\pi}(s) = E_{\pi}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s_t = s] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right]$$

Це рівняння називається рівнянням Беллмана [6] і означає, що цінність у певного стану визначається як очікувана винагорода при слідуванні поточній політиці до кінця епізоду. Значення γ визначає наскільки важливими для агента будуть майбутні винагороди і лежить на проміжку між 0 та 1. Іншими словами, цінність певного епізоду можна визначити за допомогою епізодів після нього, що відкриває двері для ітераційних підходів.

1.4. Глибоке навчання з підкріпленням

Одним з основних недоліків запропонованого фреймворку вирішення проблем є те, що нові проблеми доведеться вирішувати заново [8]. Агент не вчиться загальним правилам роботи середовища, у якому він знаходиться – він вчиться шукати найбільш оптимальний вихід в даній конкретній ситуації.

Це мотивувало прихід до навчання з підкріпленням нейронних мереж (рис. 3), що представили можливість не тільки вчитися узагальнювати отриманий досвід і застосовувати його у нових середовищах, а й відкрили такі напрями як автономне водіння, безпілотні літальні апарати і т.п.

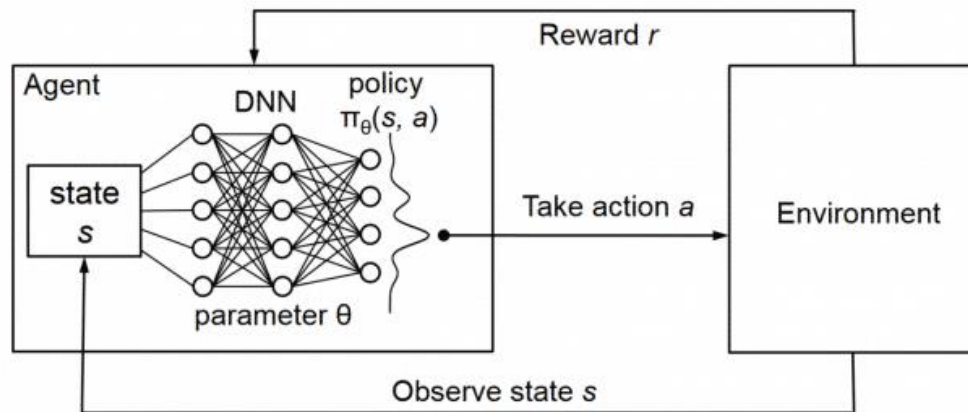


Рисунок 3: Приклад реформатованого під використання нейронних мереж процесу Маркова

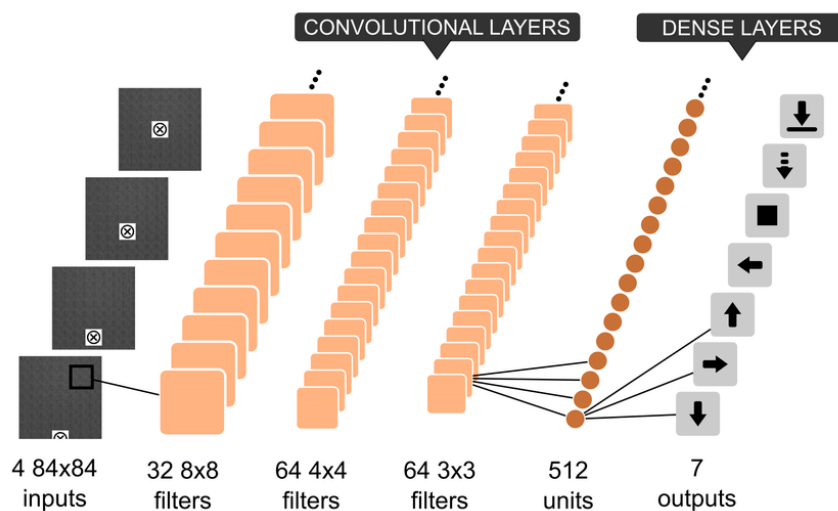


Рисунок 4: Приклад DQN для вирішення однієї з ігор Atari 2600

Виходом такої мережі є вірогідність прийняття тої чи іншої дії у деякому стані середовища, тобто найкраща дія. Тренування полягає у налаштуванні параметрів для кожної дії так, щоб максимізувати очікуване Q-value.

На жаль, сигнал від успішного вирішення задачі зменшується з кожним шаром такої мережі, а тренування сходиться до найкращих параметрів не гарантовано. Це мотивувало розробку таких архітектур як Dueling Networks,

DQN with Experience replay, та ін., в яких за основну мету взято зменшення шуму та збереження сигналу від правильних дій.

В даній роботі основну увагу приділено DQN [10], що є нічим іншим як імплементацію Q-learning, реалізованою за допомогою нейронної мережі. З отриманням нового досвіду з дії, винагороди та наступного стану середовища, агент проганяє нові дані через мережу, виходом якої є найкраща дія для даного стану. Функцією втрат є усереднена квадратична різниця між очікуваним та реальним Q-значенням, яка використовується при зворотному поширенні похибки для покращення відповідних параметрів моделі.

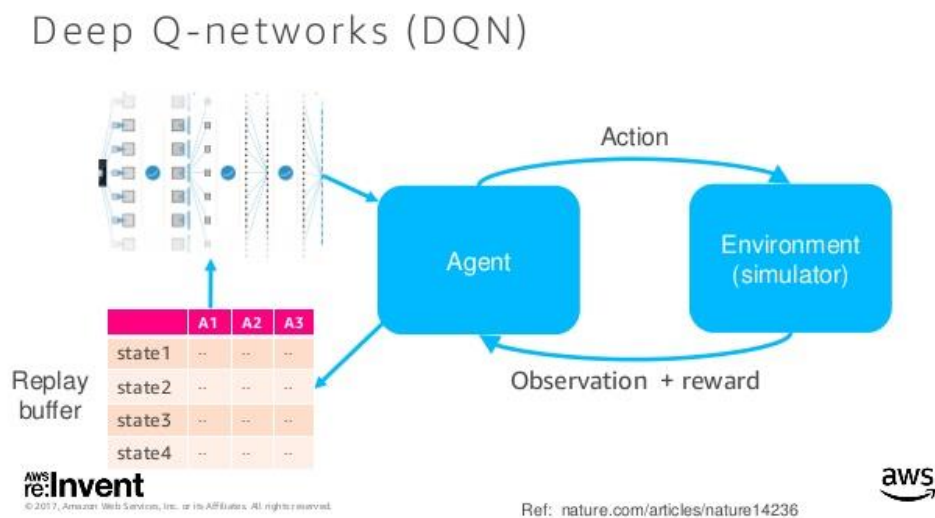


Рисунок 5: Схема роботи мережі з буфером останніх подій

Навчання за допомогою DQN визнано не достатнім для досягнення гарних результатів агентом у нових середовищах, які він не бачив під час тренування. Це спонукало розгляд більш просунутого методу – DQN із використанням буферу дій та станів, з якого вибірково обирається деяка кількість зразків (рис. 5).

Даний метод передбачає тренування двох мереж: основної та тимчасової. Основна мережа відповідає за те, що у стандартному Q-learning називалось Q-таблицею, а саме наближає оцінку дії у певному стані до реальної. Інша мережа тренується на епізодах, зібраних у буфері. Після того, як деяку кількість кроків буде зроблено, агент вчиться, змінюючи параметри основної мережі за правилом м'якої зміни (рухомого середнього):

$$Q_target = \tau * Q_local + (1 - \tau) * Q_target$$

Локальна мережа тренується за рахунок довільно обраних (дія, стан, наступний стан, винагорода), тим самим поступово накриваючи усі стани середовища. Результатом будуть параметри цільової мережі, які потім можна використовувати для проходження подібних лабіринтів.

РОЗДІЛ II

ПОБУДОВА І РЕАЛІЗАЦІЯ ПРОЕКТУ

2.1. Розробка програми

Для моделювання взаємодії між агентом та середовищем було створено декілька модулів, що містять в собі основні класи для реалізації навчання з підкріпленням: тренери, стратегії, політики навчання та модель середовища. Основним класом є *Менеджер*, реалізований на засадах шаблонів проектування Фасад та Одинак. Він збирає задані користувачем типи агентів, середовищ та політик, групуючи в одне ціле об'єкти перелічених класів.

```
def main(strategy, epochs, env, env_size=(10, 10), n_actions=4, seed=42):
    manager = Manager.get_instance(env_size, strategy, epochs, 4)
    manager.set_on_init(Descriptor(env_size, strategy, epochs, 4))
    manager.on_init.execute()

    if env == 'Custom':
        factory = FactoryCustomEnv()
        if 'Sarsa' in strategy:
            strategy_type = 'std'
            trainer = manager.run(strategy_type, factory, manager.env, manager.agent, epochs)
            Q = trainer.train()
            manager.set_on_finish(Summarizer(
                Plotter, trainer, Q))
        else:
            strategy_type = 'dqn'
            torch.cuda.current_device()
            agent = AdvAgents.DQNAgent(state_size=len(env_size), action_size=n_actions, seed=seed)
            env = Environment(env_size)
            trainer = Manager.run(strategy_type, factory, env, agent, epochs)
            scores = trainer.train()
            manager.set_on_finish(Summarizer(
                Plotter, trainer, scores))
```

env – тип середовища, приймає значення ‘Custom’/‘OpenAI Gym’,
agent – програмний агент, *trainer* – об’єкт, що відповідає за процес
 навчання

Для створення тренерів відповідно до обраного середовища використано патерн Абстрактна Фабрика:


```

class AbstractFactory(ABC):

    @abstractmethod
    def create_trainer_std(self, env, agent, epochs) -> TrainerStd:
        pass

    @abstractmethod
    def create_trainer_dqn(self, env, agent, epochs) -> TrainerDQN:
        pass

class FactoryCustomEnv(AbstractFactory):
    def create_trainer_std(self, env, agent, epochs) -> CustomTrainerStd:
        return CustomTrainerStd(env, agent, epochs)

    def create_trainer_dqn(self, env, agent, epochs) -> CustomTrainerDQN:
        return CustomTrainerDQN(env, agent, epochs)

class FactoryGymEnv(AbstractFactory):
    def create_trainer_std(self, env, agent, epochs) -> GymTrainerStd:
        return GymTrainerStd(env, agent, epochs)

    def create_trainer_dqn(self, env, agent, epochs) -> GymTrainerDQN:
        return GymTrainerDQN(env, agent, epochs)

```

TrainerStd, *TrainerDQN* – тренери для Simple та Advanced агентів,
FactoryCustom/GymEnv – конкретна фабрика для створення об'єктів тренерів.

Модуль *Trainers* відповідає за класи-тренери для відповідного середовища (усіх типів) та методу навчання (усіх методів):

```

class Trainer(ABC):
    @abstractmethod
    def train(self):
        pass

    @abstractmethod
    def step(self, *args):
        pass

class TrainerStd(Trainer):
    def __init__(self, env, agent, n_episodes=10, t_steps=500,
                 eps_start=1, eps_min=0.001, eps_decay=.95):
        self._env = env
        self._agent = agent
        self.n_episodes = n_episodes
        self.t_steps = t_steps
        self.eps_start = eps_start
        self.eps_min = eps_min
        self.eps_decay = eps_decay
        self._history = []

```

Trainer – інтерфейс для тренерів кастомного та середовища Gym, *TrainerStd* – тренер для стандартного (кастомного) середовища.

Тренери покликані забрати відповідальність за навчальний процес у менеджера, середовища та агента і реалізують виключно даний функціонал.

Для зберігання класів агентів створено модулі *SimpleAgents* та *AdvAgents*.

Перший модуль відповідає за агентів з більш простою структурою для реалізації

Монте-Карло та Temporal-difference методів. Просунуті агенти використовують власні виключно їм параметри для реалізації методів DRL:

```
class Agent:
    def __init__(self, strategy, num_actions):
        """
        Params:
            strategy (str): agent's learning strategy
            num_actions (int): number of actions in the environment
        """
        self._strategy = StrategyFactory().init_strategy(strategy)
        self._nA = num_actions
        self.Q = defaultdict(lambda: np.zeros(num_actions))
        # Hyperparameters
        self._alpha = 0.001
        self._gamma = 1.0
        self._eps_decay = 0.999
        self._eps_min = 0.005
        self.eps = 1

class DQNAgent_ExpReplay:
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        =====
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

SimpleAgents (зверху) та *AdvAgents* (знизу)

Клас *Агент* містить інформацію про тип його стратегії, наявну кількість дій у середовищі та низку гіперпараметрів, визначених тим чи іншим типом навчання.

Стратегії, доступні для вибору агентом, розташовані у модулі *Strategies*:

```

class Strategy:

    def __init__(self, policy=None):
        if policy is not None:
            self._policy = PolicyFactory().init_policy(policy)

    @property
    def policy(self) -> Policy:...

    @policy.setter
    def policy(self, policy):...

    @abstractmethod
    def update(self, *args):...

class Sarsamax(Strategy):...

class ExpectedSarsa(Strategy):...

class Dqn(Strategy):...

class StrategyFactory:...

```

policy – методологія вибору дій агентом, *update* – унікальне для кожної стратегії правило, за яким досвід агента переводиться у знання про середовище.

У класі *Менеджер* викликається Фабрика, яка створює ту чи іншу стратегію.

Усі наявні політики реалізовані у модулі *Policies*:

```

class Policy:
    def __init__(self):
        pass

    @abstractmethod
    def get_action(self, *args):
        pass

class Epsilon(Policy):
    def __init__(self):
        super().__init__()

    def get_action(self, Q, state, nA=None, eps=None):...

class EpsilonGreedy(Policy):
    def __init__(self):
        super().__init__()

    def get_action(self, Q, next_state, nA=None, eps=None):...

class PolicyFactory:...

```

get_action – метод вибору дій агентом (простого типу)

Аналогічно до стратегій, створенням об'єктів займається Фабрика.

Для реалізації глибокого навчання з підкріпленням було створено модуль *model.py*, де за допомогою фреймворку Pytorch створено невелику нейронну мережу з декількома fully-connected шарами (тришаровий MLP).

```

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fcl_units=64, fc2_units=64):
        """Initialize parameters and build model.

        Params
        =====
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcl_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fcl_units)
        self.fc2 = nn.Linear(fcl_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

```

Архітектура мережі для задачі «Лабіринт»

Функцією активації обрано RectifiedLinearUnit, оскільки вона диференційовна на всьому проміжку та не так відчутно призводить до затухання градієнтів при зворотному поширенню похибки, як функції на кшталт Sigmoid, Tanh і їм подібні.

В даній роботі основний фокус приділявся алгоритму Q-learning, що на практиці дає кращі результати за такі методи як Sarsa та Expected Sarsa. Результати роботи алгоритму вивчалися за допомогою будування python-словника (state, value) пар та візуалізації найкращих ходів для певних станів середовища засобами бібліотеки Matplotlib. За це відповідає клас Plotter:

```

class Plotter:
    @staticmethod
    def plot_std(trainer, Q):...

    @staticmethod
    def plot_dqn(scores):...

```

Код цієї частини роботи програми організовано за допомогою патерну проектування Команда:

```

class Command(ABC):
    @abstractmethod
    def execute(self) -> None:
        pass

class Descriptor(Command):...

class Summarizer(Command):
    def __init__(self, receiver, trainer, result) -> None:
        self._receiver = receiver
        self._trainer = trainer
        self._result = result

    def execute(self) -> None:
        if isinstance(self._result, collections.defaultdict):
            self._receiver.plot_std(self._trainer, self._result)
        if isinstance(self._result, list):
            self._receiver.plot_dqn(self._result)

```

Сам процес отримання результатів запускається класом *Менеджер*, що викликає Команду *Summarizer*, передавши необхідні компоненти для ініціалізації:

```

    trainer = Manager.run(strategy_type, factory, env, agent, epochs)
    scores = trainer.train()
    manager.set_on_finish(Summarizer(
        Plotter, trainer, scores))

manager.on_finish.execute()

```

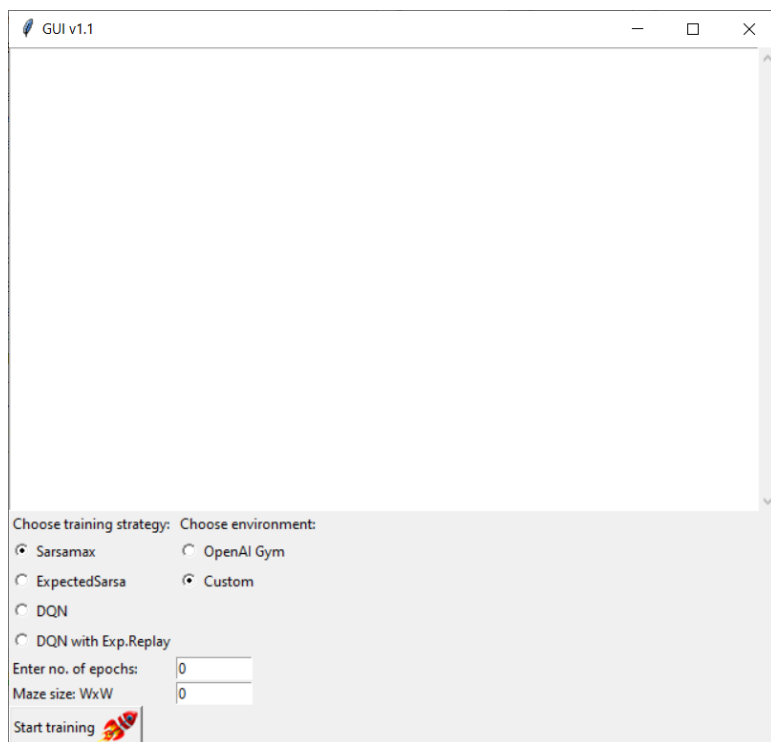
РОЗДІЛ III

ІНТЕРФЕЙС КОРИСТУВАЧА

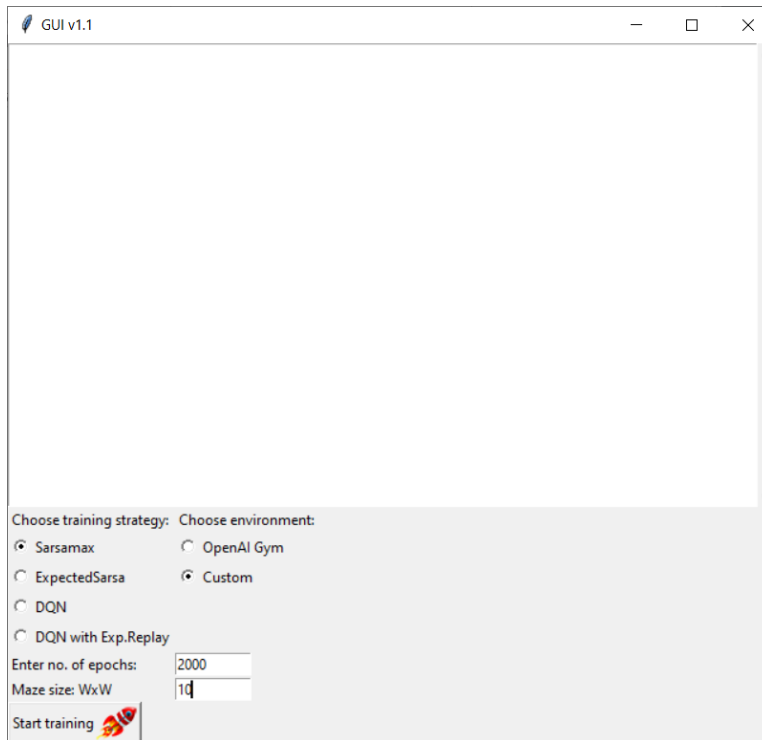
Програму розроблено для роботи у графічному інтерфейсі, який створено за допомогою python-нативної бібліотеки Tkinter.

Перед запуском програми на машині користувача має бути встановлено робочий інтерпретатор мови python версії 3.7х. Для коректної роботи програми з терміналу потрібно перейти до кореневої папки проекту і запустити команду *pip install -r requirements.txt*. Вона встановить усі необхідні пакети з тими версіями, що використовувались при розробці.

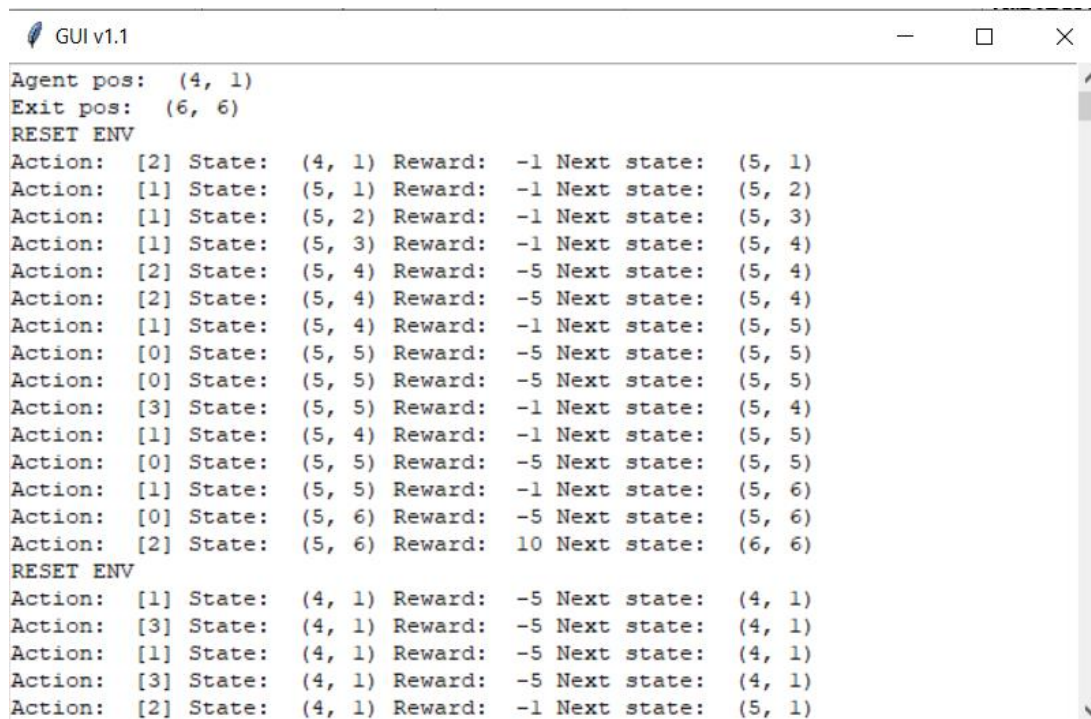
Для запуску програми необхідно ввести у консолі команду *python tkinter_main.py*, після чого користувач потрапляє у наступне вікно:



Наступні кроки здійснюються по введенні користувачем запропонованої інформації: кількості епізодів тренування, параметрів лабіринту, стратегій навчання та типу середовища, де тренування відбуватиметься. Для початку тренування користувач має заповнити всі поля форми ненульовими цілими значеннями:



Після натиснення кнопки *Start training* у фоновому режимі запускається модуль *Manager.py* з параметрами, взятими з полів заповненої форми. Результати виконання програми виводяться у текстовому полі після завершення останнього епізоду. Їх можна відслідкувати до кінця , використавши скроллбар.



```

Action: [0] State: (7, 5) Reward: -1 Next state: (6, 5)
Action: [1] State: (6, 5) Reward: 10 Next state: (6, 6)
Best Average Reward over 100 Episodes: -27.9
Resulting table of (state, action) value-pairs:
defaultdict(<function Agent.__init__.<locals>.<lambda> at 0x00000296136E79D8>,
            { None: array([0., 0., 0., 0.]),
              (1, 1): array([-1.44997004, -1.44894924, -1.44963215, -1.44997
1]),
              (1, 2): array([-1.44999   , -1.44895724, -1.44997399, -1.44928
6]),
              (1, 3): array([-1.449994  , -1.44891383, -1.44998001, -1.44912
1]),

```

Результати представлені у вигляді графіку навчання, з якого можна зрозуміти чи покращується усереднена за епізод винагорода з часом (рис. 1). Також продемонстровано створений лабіринт та відповідне рішення у вигляді шляху до виходу, представлене клітинами, позначеними яскравішими кольорами (рис. 2).

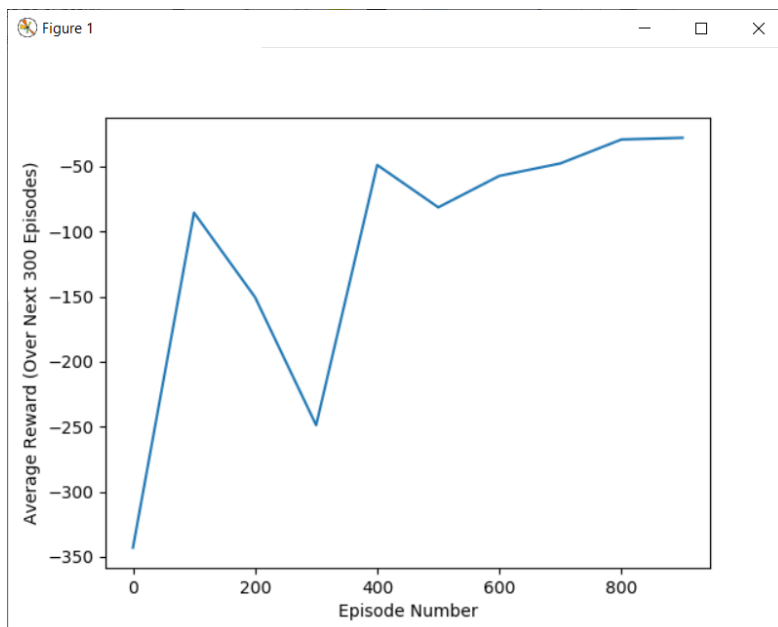


Рисунок 1. Вісь x – кількість пройдених епізодів (початок-фініш), вісь y – середня винагорода за епізод

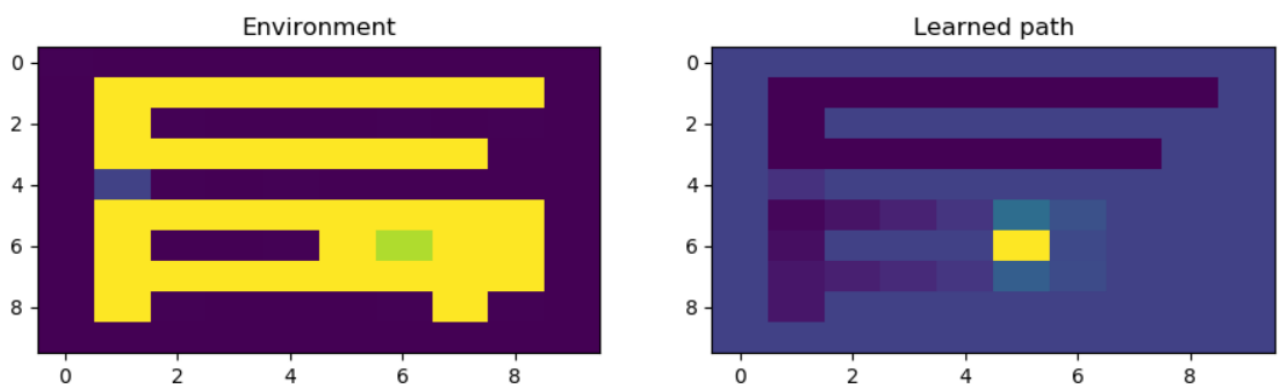


Рисунок 2. Створений лабіринт та найкращий шлях (право) від початку (темна позначка) до фінішу (зелена позначка).

Останнє може бути зображене у вигляді Q-таблиці (рис. 3). Ця таблиця пар значень (положення, дія), яка змінювалась у продовж процесу навчання, де позитивний результат від тієї чи іншої дії зображено кольором від темно-зеленого (погана дія) до червоного (найкраща у положенні).

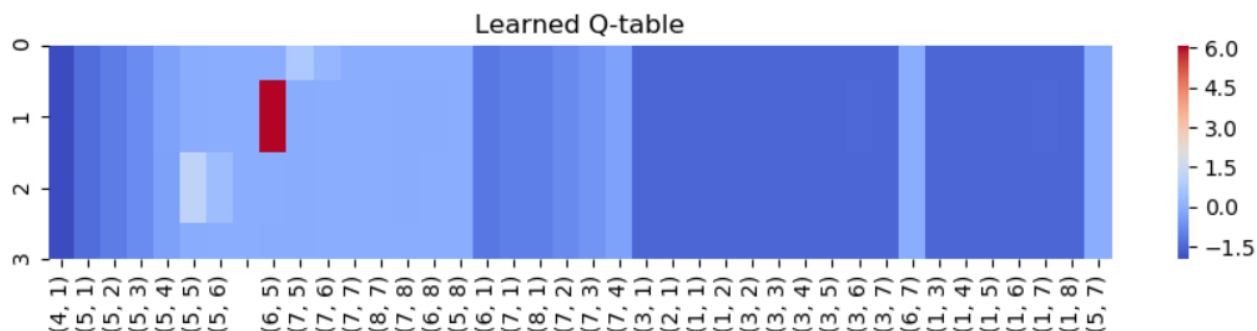
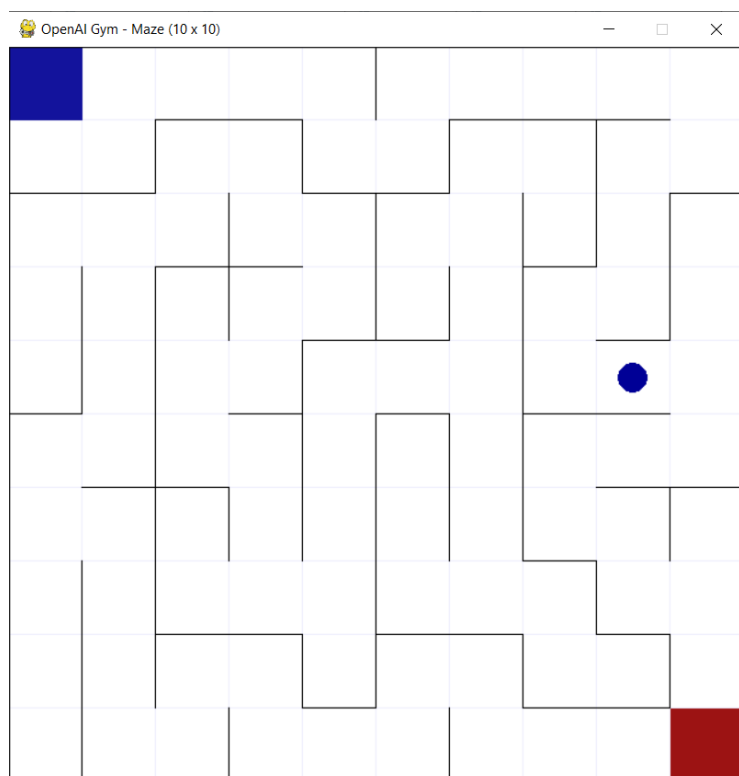


Рисунок 3. Кінцева Q-таблиця

У разі вибору середовища OpenAI Gym, агент потрапляє у попередньо-змодельований лабіринт розмірів (5, 5) або (10, 10). Навчання протікає на тих самих засадах, проте графічне представлення змінюється суттєво:



Кожен крок агента (чорна точка) відображається графічно, а перешкоди, вхід та вихід залишаються статичними до кінця тренування.

Варто зазначити, що при кількості епізодів (epoch) менше 100, результати можна побачити лише у текстовому вигляді, а у головному вікні можна побачити відповідне повідомлення:

```

Action:  [1] State:  (5, 4) Reward:  -1 Next state:  (5, 5)
Action:  [0] State:  (5, 5) Reward:  -1 Next state:  (4, 5)
Action:  [0] State:  (4, 5) Reward:  -1 Next state:  (3, 5)
Action:  [0] State:  (3, 5) Reward:  10 Next state:  (2, 5)

```


Not enough epochs to calculate statistics

Choose training strategy: Choose environment:

☒ Sarsamax ☐ OpenAI Gym
☐ ExpectedSarsa ☒ Custom
☐ DQN
☐ DQN with Exp.Replay

Enter no. of epochs:

Maze size: WxW

Start training 


Закриття всіх вікон поверне до початкового інтерфейсу, де експеримент можна продовжити, задавши нові параметри лабіринту або обравши інші методи навчання з підкріпленням:

Choose training strategy: Choose environment:

☐ Sarsamax ☒ OpenAI Gym
☐ ExpectedSarsa ☐ Custom
☐ DQN
☒ DQN with Exp.Replay

Enter no. of epochs:

Maze size: WxW

Start training 

ВИСНОВКИ

У ході виконання курсової роботи з дослідження методів вирішення задачі «Лабіринт», було розглянуто декілька класичних підходів та апробовано різні техніки навчання з підкріпленням. Застосовуючи методи RL, що не використовують модель середовища для процесу навчання, отримано бажаний результат у формі таблиці пар значень (стан, дія), яка репрезентує найкращий (найкоротший) шлях у лабіринті.

Паралельно дослідивши стандартні методи вирішення даної задачі, можна стверджувати, що методи навчання з підкріпленням мають суттєвий недолік порівняно з ними – часову складність, яка є експоненційною та залежить від розмірів середовища і кількості наявних дій. У свою чергу стандартні алгоритми пошуку найкоротшого шляху на графах, що з успіхом застосовуються у випадку лабіринту, дають час, що лінійно залежить від кількості розмірів лабіринту.

Це дає змогу зробити висновки щодо незастосовності навчання з підкріпленням у сферах, де рішення може бути знайдено відомими методами обчислювальної геометрії, тим паче у разі складності, що не перевищує декількох перших степенів експоненти у останніх. Реальними сферами застосування даної техніки виступають ті, де складність аналізу ситуацій, що з'являються, виходить за можливості аналітики і потребує автоматизованої роботи програмного агента чи деякій їх кількості протягом великої кількості часу.

Підсумовуючи, можна зробити наступні висновки:

- навчання з підкріпленням не має застосовуватись до задач з дискретним середовищем та обмеженими діями;
- збіжність оптимізації шляху методами RL не гарантована для лабіринтів великої розмірності при невдалому виборі гіперпараметрів;
- рівень сигналу, що отримує агент від середовища грає вирішальну

роль у швидкості збіжності методу та збіжності як такої;

- використання квадратних матриць суміжності є неефективним підходом для пошуку шляхів на графах ;
- методи динамічного програмування є найоптимальнішим вибором для вирішення задачі пошуку найкоротшого шляху на графі;
- класичні методи навчання з підкріпленням у дискретних середовищах дають кращий результат та за більш короткий час порівняно з методами, що використовують нейронні мережі, але є незастосовними до нових середовищ;
- Temporal-Difference методи збігаються до найкращого рішення швидше за Monte-Carlo методи у задачах, де існує висока вірогідність не досягти винагороди в межах одного епізоду;
- off-policy методи дають краще рішення за on-policy у разі невеликої кількості епізодів;
- ефективність вирішення задач тим чи іншим алгоритмом має перевірятися на прикладі конкретної задачі, оскільки параметри середовища можуть сильно вплинути на роботу алгоритму;
- для вирішення малорозмірних задач не варто використовувати складні з точки зору обчислень методи, а шукати рішення у стабільно працюючих аналітичних методах;

У процесі проектування програмного забезпечення мовою Python було розроблено низку класів для керування процесом навчання програмного агента. За основу для об'єктно-орієнтованої розробки взято принципи SOLID, KISS та DRY. У програмі використано увесь стек засобів мови Python для імплементації наслідування, поліморфізму, інкапсуляції та ін. парадигм ООП. Для зручності масштабування проекту його розбито на низку модулів, кожен з яких представлений групою класів, що відповідають за певну частину функціоналу програми.

Для легшого масштабування проекту при організації коду використано низку шаблонів проектування, а саме: Одинак, Стратегія, Команда, Фабрика, Абстрактна Фабрика та Фасад. Використано засоби графічної бібліотеки Tkinter для створення графічного інтерфейсу користувача. В розробленому

інтерфейсі можна налаштовувати параметри тренування: максимальну кількість епізодів для проходження деякої варіації лабіринту, стратегію агента, розміри середовища та його тип. Процес навчання продемонстровано інтерактивно засобами бібліотеки OpenAI Gym.

Результатом роботи є програма, яку можна використовувати у цілях дослідження роботи алгоритмів навчання з підкріпленням для складного з точки зору генералізації та отримуваного сигналу винагороди середовища «Лабіринт».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. R. S. Sutton, A. G. Barto, “ Reinforcement Learning: An Introduction” MIT Press, Cambridge, MA, 2018, pp.18-19, 52-55.
2. C. Zhang, S. Bengio, “ A Study on Overfitting in Deep Reinforcement Learning,” ArXiv, arXiv:1804.06893v2 [cs.LG] 20 Apr 2018, Pages 1-4.
3. Yuxi Li, “ Deep Reinforcement Learning: An Overview, ” ArXiv, arXiv:1701.07274v6 [cs.LG], 5 Nov 2018, pp. 11-12.
4. J. Schmidhuber, “ Reinforcement Learning Upside Down: Don't Predict Rewards,” ArXiv, arXiv:1912.02875v1 [cs.AI] 5 Dec 2019, pp. 2-3.
5. E. H. Kivelevitch., K. Cohen, “ Multi-Agent Maze Exploration,” JACIC, Vol. 14, Dec 2010, pp. 391-405.

ІНТЕРНЕТ РЕСУРСИ

6. Deep Reinforcement Learning Tutorial with Open AI Gym.
<https://towardsdatascience.com/deep-reinforcement-learning-tutorial-with-open-ai-gym-c0de4471f368>
7. Алгоритми пошуку шляху в лабіринті.
<https://www.quora.com/How-would-you-design-an-algorithm-that-finds-the-shortest-path-in-a-maze>.
8. RL explained. <https://www.oreilly.com/radar/reinforcement-learning-explained/>
9. 5 речей, які треба знати про RL. <https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html>
- 10.Що не так з навчанням з підкріпленням. <https://habr.com/ru/post/437020/>