Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики

Використання навчання з підкріпленням та генетичних алгоритмів для розробки ігрового агента в стратегії в реальному часі

Текстова частина до курсової роботи за спеціальністю
«Інженерія програмного забезпечення» 121

Керівник курсової роботи
канд. фіз-мат наук Крюкова Г. В.
_____
(підпис)
«___» _____ 2021 р.
Виконав студент
Ларін Д. Л.
«___» _____ 2021 р.

Київ – 2021

Календарний план виконання роботи

Тема: Використання навчання з підкріпленням та генетичних алгоритмів для розробки ігрового агента в стратегії в реальному часі.

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	02.11.2020	
2.	Огляд технічної літератури за темою роботи.	15.11.2020	
3.	Виконати аналіз сучасних методів.	25.11.2020	
3.	Розробка алгоритму .	30.12.2020	
4.	Програмування розробленого алгоритму .	15.01.2021	
5.	Застосування розробленого алгоритму до фреймворку PySC2.	15.02.2021	
6.	Дослідження у мережі наявної інформації про навчання з підкріпленням та генетичні алгоритми	30.03.2011	
8.	Створення слайдів для доповіді та написання доповіді.	22.04.2021	
8.	Аналіз отриманих результатів з керівником, написання доповіді	25.04.2021	
11.	Остаточне оформлення слайдів та доповіді.	16.05.2021	
12.	Захист курсової роботи	3 неділя літнього триместру	

Студент _____

Керівник _____

“_____”_____

Зміст

Анотація

Дану роботу виконав студент Києво-Могилянської Академії 3-го року навчання Ларін Дмитро Леонідович. Метою даної роботи було поглиблення власних теоретичних та практичних знань у сфері навчання з підкріпленням та генетичних алгоритмів. Перший розділ містить теоретичні відомості про Навчання з підкріпленням – основні поняття, методи та невелику історію навчання з підкріпленням як ігрового агенту в іграх, від коротких нард до сучасної відеогри Dota 2. Другий розділ містить теоретичні відомості про еволюційні алгоритми і генетичний алгоритм конкретно, а також сфери і приклади застосування. Третій, останній розділ, містить опис фреймворку PySC2 і процес розробки ігрового агенту у даному середовищі.

Вступ

Проблематика навчання з підкріпленням і генетичних алгоритмів набуває все більшого значення у сучасному світі, але має скромне представлення в україномовних джерелах. З часу, коли ігровий агент OpenAI Five зміг подолати чемпіонів світу з Dota 2, зацікавленість у ігрових агентах різко зросла. Значущість цієї події відмітив навіть Білл Гейтс. Перевага розглянутих у роботі галузей інформатики є величезний потенціал, адже вони разом можуть імітувати людський мозок, а це означає величезний потенціал для розвитку.

Метою є розробити ігрового агента для гри у StarCraft II. Хочеться виділити наступні завдання:

- простежити розвиток навчання з підкріпленням та еволюційних алгоритмів,
- розглянути основні методи роботи,
- систематизувати досягнення навчання з підкріпленням у розробці ігрових агентів.

Об’єкт дослідження: методи навчання з підкріпленням та генетичний алгоритм.

У процесі дослідження був проведений серйозний аналіз літератури: публікації науковців у періодичних виданнях, книжки, веб-сайти, тощо.

Наукова новизна полягає у створення ігрового агенту для гри у StarCraft на основі Q-Навчання.

Розділ 1. Навчання з підкріпленням

1.1 Основні поняття навчання з підкріпленням

1.1.1 Введення

Навчання з підкріпленням (Reinforcement Learning) – одна з галузей машинного навчання, яка вивчає, які дії (actions) мають виконувати агенти (agents) в певному середовищі (environment) для максимізації деякого уявлення про сукупну винагороду (reward). На відміну від навчання з учителем (supervised learning), агент повинен самостійно визначити які дії приносять найбільшу винагороду [1]. Навчання з підкріпленням не варто плутати з навчанням без вчителя (unsupervised learning), яке зазвичай полягає у знаходженні структури чи закономірності у певній інформації.

Метод спроб та помилок (trian-and-error) і відкладена нагорода (delayed reward) – дві найважливіші риси, що відрізняють навчання з підкріпленням від інших галузей машинного навчання. Перша риса полягає у тому, що за відсутності вчителя програма буде проходити через велику кількість різних станів, виконувати різні дії допоки не знайде виграшні стратегії. Друга – у тому, що поточна дія може впливати не лише на поточну винагороду, а й усі наступні ситуації та винагороди з них [1].

Зазвичай, алгоритм навчання з підкріпленням використовує певну функцію, яка визначає нагороду за дію/дії програми. Оскільки галузь навчання з підкріпленням заснована на біхевіористській психології [2], ця задається людиною і включає у себе найважливіші аспекти проблеми, оцінку досягнень алгоритму, винагороду за дії. Як приклад, можна навести гру шахи, де найбільша винагорода досягається при виграші партії, але також система може отримувати винагороду за довжину партії, кількість «живих» фігур в кінці, тощо.

Одна зі складностей, яка виникає у навчанні з підкріпленням, це пошук балансу між розвитком (exploit) та дослідженням (explore). Щоб отримати

більшу винагороду, агент хоче використовувати дії, які він використовував у минулому і знайшов ефективними. Але щоб знайти такі дії, йому потрібно спочатку спробувати щось нове, що він ще не робив. Пошук ідеального балансу між цими діями – дилема математиків та інформатики, яку вони намагаються вирішити вже не одну декаду [3].

1.1.2 Елементи навчання з підкріпленням

Окрім агенту та середовища, виділяють чотири головні елементи навчання з підкріпленням: стратегію (a policy), функція винагород (a reward signal), функція цінності (a value function) і, іноді, модель (a model) середовища [4].

Стратегія визначає дії агенту у поточний момент часу. По-суті, стратегія – це зв’язування певних станів середовища до дій, які мають бути виконані у цьому стані. Іноді, стратегія може бути простою таблицею, де значення А зв’язується з дією Б, а в інших випадках вимагатиме складних обчислень. Загалом, стратегія є стохастичною, визначаючи ймовірність кожної дії для заданого стану.

Функція винагород визначає мету агента. На кожному кроці, середовище надсилає агенту винагороду у вигляді одного числа, а задача агента – максимізувати загальну винагороду за усі кроки. Саме функція винагород визначає, які дії агента чи події в середовищі вважати хорошими чи поганами агентами. Зазвичай, функція винагород є стохастичною функцію від стану середовища та дій, які зробив агент.

Функція цінності, у свою чергу, визначає що є хорошим результатам у довгій перспективі. Іншими словами, цінність – це загальна винагорода, яку може набути агент у майбутньому, починаючи з поточного стану. Нерідко трапляються ситуації, коли поточний стан дає низьку винагороду, але наступні за ним стани дають винагороду високу та навпаки. Функція цінності

дозволяє агенту не бути занадто жадібним і враховувати майбутню вигоду. Вважається, що це найважливіша частина у навчанні з підкріпленням [1].

Нарешті, модель допомогає агенту робити висновки про те, як середовище буде реагувати на його дії. Наприклад, для заданого стану і дії передбачити наступний стан і винагороду. Модель – це протилежність до методу спроб та помилок, адже агент вже на початку навчання зможе передбачити коректність чи некоректність своїх дій. Але як вже було зазначено, модель – необов'язкова частина навчання з підкріпленням, і агент може навчатися методом спроб та помилок, хоча зараз навчання на основі моделі вважається більш ефективним [4].

1.2 Огляд методів

1.2.1 Динамічне програмування

Динамічне програмування – це набір алгоритмів для розрахунку оптимальних стратегій у ідеальній моделі середовища. Прикладом такого середовища є Марковський процес вирішування (Markov Decision Process). Класичні алгоритми динамічного програмування дуже обмежені у використанні у навчанні з підкріпленням через їхнє припущення ідеального середовища і великої обчислювальної вартості, але вони все ще важливі теоретично [5].

Ключова ідея динамічного програмування, а також навчання з підкріпленням загалом, полягає у використанні функції цінності для пошуку хороших стратегій. Одним із ключових принципів динамічного програмування є оцінка стратегії (policy evaluation). Це спосіб рахувати стан-цінність функцію v_π для будь-якого значення π, щоб оцінити її якість. У даному контексті вона матимете наступний вигляд:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_\pi(s')],$$

де $\pi(a|s)$ – ймовірність зробити дію a у стані s внаслідок стратегії π, $p(s', r \mid s, a)$ – множина ймовірностей заснована на динаміці станів, дій та винагород, $s \in S, s' \in S'$ де S – множина станів, S' - множина станів + термінальний стан, якщо такий існує, $a \in A$ – множина дій, $r \in R$ – множина винагород, γ – параметр нетерпимості (impatience), який також має назву фактор знижки.

Для послідовності наближень функції цінності v_0, v_1, .., v_k за рівнянням Белмана будемо мати, що $v_k = v_\pi$ у якийсь момент часу. Якщо обрати початкове значення v_0 як завгодно (окрім термінального стану), а кожне наступне наближення рахувати за тим же рівнянням Белмана, то отримаємо, що:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma v_k(s')].$$

Таким чином, метод динамічного програмування дозволяє знаходити оптимальну дію для поточного стану на основі попередніх станів. По-суті, усі інші алгоритми намагаються досягти тієї ж мети, що і динамічне програмування, але у неідеальній моделі або з меншою обчислювальною вартістю. Було наведено один із способів застосування динамічного програмування у навчанні з підкріпленням – оцінка стратегії, серед інших застосувань – покращення стратегії та пошук найоптимальнішої стратегії.

1.2.2 Метод Монте-Карло

Метод Монте-Карло полягає у навчанні системи з досвідом (experience). Досвід – це послідовності станів, дій та винагород зі справжніх або симульованих взаємодій з середовищем. На відміну від Динамічного Програмування, метод Монте-Карло не вимагає ідеальної моделі – потрібна лише генерація епізодів $s_0 a_0 r_1 s_1 a_1 r_2 .. s_t$. Епізод обов'язково має бути повним, тобто закінчуватися термінальним станом.

Уявимо, що цінність стану визначається очікуваним результатом – кумулятивною сумою майбутніх винагород, починаючи з даного стану. Найпростіший спосіб оцінити дану величину – знайти середнє значення результатів після усіх відвідин даного стану. Чим більше буде розвідано таких результатів – тим більше це середнє значення наблизиться до реального значення. Ця ідея лежить в основі усіх методів Монте-Карло.

Якщо застосувати метод Монте-Карло до проблеми оцінки стратегії, яка вирішувалася Динамічним Програмуванням у попередньому підрозділі, то отримаємо наступні міркування. Нехай дана серія епізодів, отримана виконанням певної стратегії π, що проходить через стан s. Кожна поява стану s в епізоді має назву візит (a visit) у s (s може відвідуватися не один раз у епізоді). Назвемо першу появу s у епізоді як *перший візит* s. Тоді *метод Монте-Карло першого візиту* (first visit MC method) оцінює $v_\pi(s)$ як середнє результатів після усіх перших візитів до s [6]. Формулою це описується так:

$$V(S_t) = V(S_t) + \alpha\big(G_t - V(S_t)\big),$$

де G_t – очікуваний результат у час t, α – параметр швидкості навчання, V – оцінка v_π.

Методи Монте-Карло набули широкого використання у навчанні з підкріпленням, адже система – спробуй, зроби висновки, спробуй ще раз з новим досвідом не вимагає ідеального середовища, а також потребує значно менше обчислювальної вартості, ніж методи Динамічного Програмування.

1.2.3 Метод часових відмінностей

Метод часових відмінностей (Temporal Difference Learning, далі TD методи) є комбінацією ідей динамічного програмування та методів Монте Карло. Подібно до Монте Карло, TD методи здатні навчатися з «сирого» експерименту, без моделі динаміки середовища. Подібно до технік

динамічного програмування, TD методи оновлюють оцінку у процесі експерименту на основі попередніх оцінок [7].

В той час, коли метод Монте Карло має дочекатися кінця епізоду щоб визначити інкремент до $V(S_t)$ (бо тільки тоді G_t стає відомим), TD методам достатньо дочекатися лише наступного часового кроку t [1]. У час $t + 1$ вони роблять корисні оновлення використовуючи розвідані винагороду R_{t+1} та оцінку $V(S_{t+1})$. Найпростіший TD зробив би оновлення вигляду:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)],$$

одразу після переходу до стану S_{t+1} і отриманні R_{t+1}. Конкретно цей метод TD навчання має назву TD(0), або однокроковий TD і є особливим випадком TD(λ).

Методи часових відмінностей мають перевагу над динамічним програмуванням через відсутність вимоги моделі середовища. Також, вони добре показують себе перед методами Монте Карло, бо необхідність чекати кінця епізоду та певна нестабільність відсутні у TD методах. Також, практично вставновлено що TD методи доходять до цілі швидше ніж методи Монте Карло с константною α [8].

1.2.4 Підсумки

На Рисунку 1.2.1 наданий рисунок двох основних просторів у методах навчання з підкріпленнях. Ці простори показують який вид оновлення використовується певним методом для оновлення функції цінності.

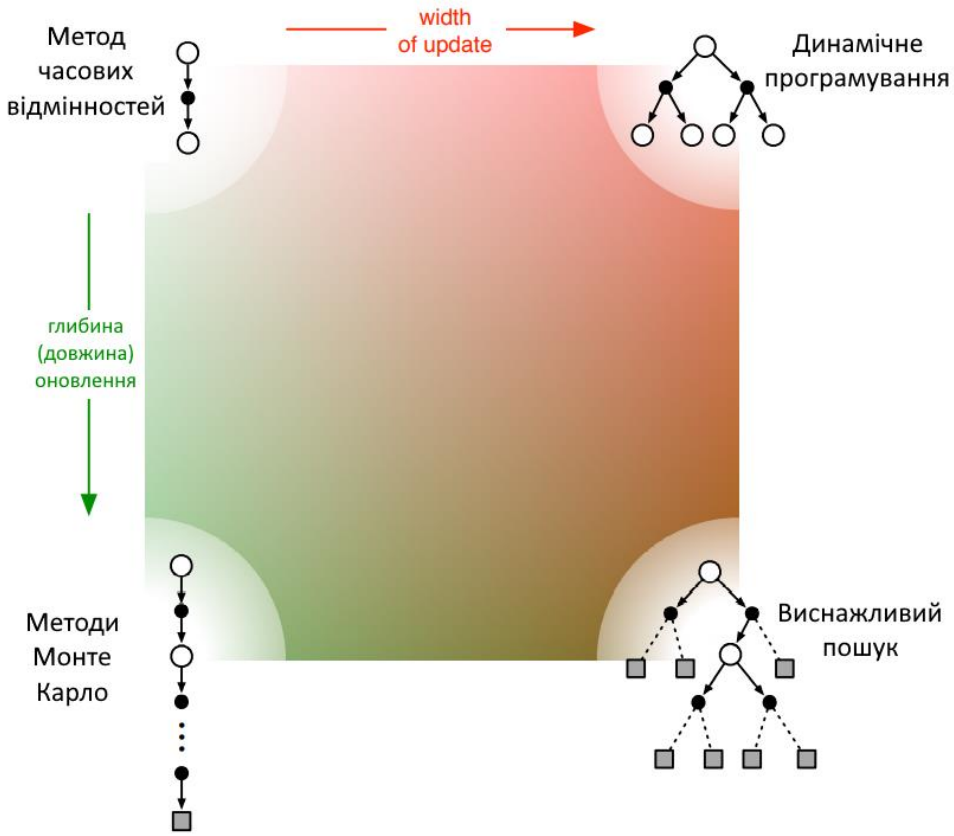

Рисунок 1.2.1

У правому нижньому кутку можна помітити «виснажливий пошук» (Exhaustive search). Це гібридна техніка, яка вимагає надзвичайної кількості ресурсів комп'ютеру для роботи і вважається непрактичною [1].

1.3Практичні застосування

1.3.1 TD-Gammon

TD-Gammon – нейронна мережа, що вчилася грати в короткі нарди, граючи виключно проти самої себе. Програма досягла рівня гри найкращих гравців, а також відкрила тактики, якими не користувалися люди. Завдяки TD-Gammon теорія гри у короткі нарди значно покращилася [7].

В основу ліг алгоритм TD-лямбда, а навчання починалося з випадкової ініціалізації коефіцієнтів. На кожному кроці мережа робила хід, який мав би максимізувати очікуваний виграш [8]. В результаті, програма навчалася на результатах гри з собою, і такий підхід використовувався навіть на початку

навчання, тобто коли ваги мережі були задані випадковим чином і від цього мережа робила випадкові кроки. Через ці обставини (адже випадкову стратегію гри не можна назвати хорошою), мережа навчалася дуже довго – їй потрібні були сотні і тисячі ходів, щоб завершити партію – тоді як людина завершує її за 50-60 [9].

Перша версія гри у якості вхідних даних містила лише базову інформацію про стан дошки – кількість білих або чорних фішек на кожній позиції. Програмі не надавалися якісь попередньо прораховані дані, які допомагають краще грати – такі як сила блокада чи ймовірність отримати удар [9]. Іншими словами, система не мала якихось початкових знань про прийоми хорошої гри у нарди. Тим не менш, ця версія TD-Gammon змогла досягти рівня гри Neurogammon, програми тренованої на основі ходів найкращих гравців свого часу.

Наступна версія алгоритму, яка отримала назву TD-Gammon 1.0, використовувала усі ті ознаки, що й Neurogammon, лишивши концепцію гри-з-собою та TD-навчання. Після проведення 300,000 ігор, вона програвала кращим гравцям з відривом лише в 13 очок за 51 гру. При цьому, дана версія гри враховувала лише виграш від власного поточного кроку, не враховуючи хід суперника та свої наступні ходи.

Наступна версія гри – TD-Gammon 2.0 виправила цей недолік, мережа ходила так, щоб максимізувати виграш, враховуючи власний хід та послідовний хід суперника. Враховувалося усі можливі кидки кубику, і вважалося, що суперник буде ходити оптимально. Такий підхід має назву 2-ply пошук [3]. Щоб полегшити процес обчислень, ходи опонента прораховувалися лише для найкращих кандидатів ходу мережі – у середньому це 5-6 варіантів. Дана версія мережі мала 40 шарів, і після 800,000 тренувальних ігор змогла покращити результат попередньої версії, відстаючи від топових гравців лише на 7 очок за результатами 38 ігор [7]. Після розширення кількості прихованих

шарів до 80 та збільшення кількості тренувальних ігор до 1,500,000 розрив вдалося зменшити до 1 очка [9].

Наступна і остання версія програми, TD-Gammon 3.0, також містила 80 прихованих шарів і проводила ті ж 1,500,000 тренувальних ігор. Але на відміну від попередньої версії, вона використовувала 3-ply пошук [3]. Іншими словами, мережа ходила так, щоб максимізувати виграш, враховуючи поточний хід, послідовний хід суперника, та свій наступний хід. Ця версія мережі змогла обіграти Ніла Казароса, найкращого гравця свого часу [10] з відривом у 6 очок після 20 ігор [3]. Результати ігор та дані про версії програми наведено у Таблиці 1.1.

TD-Gammon стала визначальною епохою у розвитку навчання з підкріпленням, адже остання версія програми змогла набагато покращити результати Neurogammon, яка була заснована на навчанні з учителем. TD-Gammon довела, що концепція гри-з-собою та навчання з підкріпленням мають великі перспективи у створенні штучного інтелекту для ігор, який здатний грати на рівні людини або навіть краще [9].

Програма	Прих. шарів	Тренув. ігор	Опоненти	Результати (очок)	Ігор
TD-Gam. 0.0	40	300,000	-	-	-
TD-Gam. 1.0	80	300,000	Роберті, Магрієль, …	-13	51
TD-Gam. 2.0	40	800,000	Різні Гросмейстри	-7	38
TD-Gam. 2.1	80	1,500,000	Роберті	-1	40
TD-Gam. 3.0	80	1,500,000	Казарос	+6	20

Таблиця 1.1 - Результати ігор та дані про версії програми

1.3.2 AlphaGo Zero

AlphaGo – це перший алгоритм, що зміг перемогти професіонального гравця в Go. Правила гри у Go доволі прості: два гравці (чорний та білий) грають на полі 19x19 і по черзі роблять кроки за допомогою камінців їхнього кольору на пусту клітинку дошки. Задача – оточити камінці суперники своїми по вертикалі і горизонталі, щоб сформувати замкнену площу. Площа вважається «зайнятою», якщо вона оточена камінцями суперника та внутрішня частина має не більше однієї пустої клітинки. Гра завершується тоді, коли обидва гравці «пасують» - найчастіше, коли на дошці не лишилося місця. Перемагає той, хто захопив найбільше ворожих камінців (у поширеному варіанті правил) [11]. На Рис. 1.1 наданий приклад дошки у процесі гри.

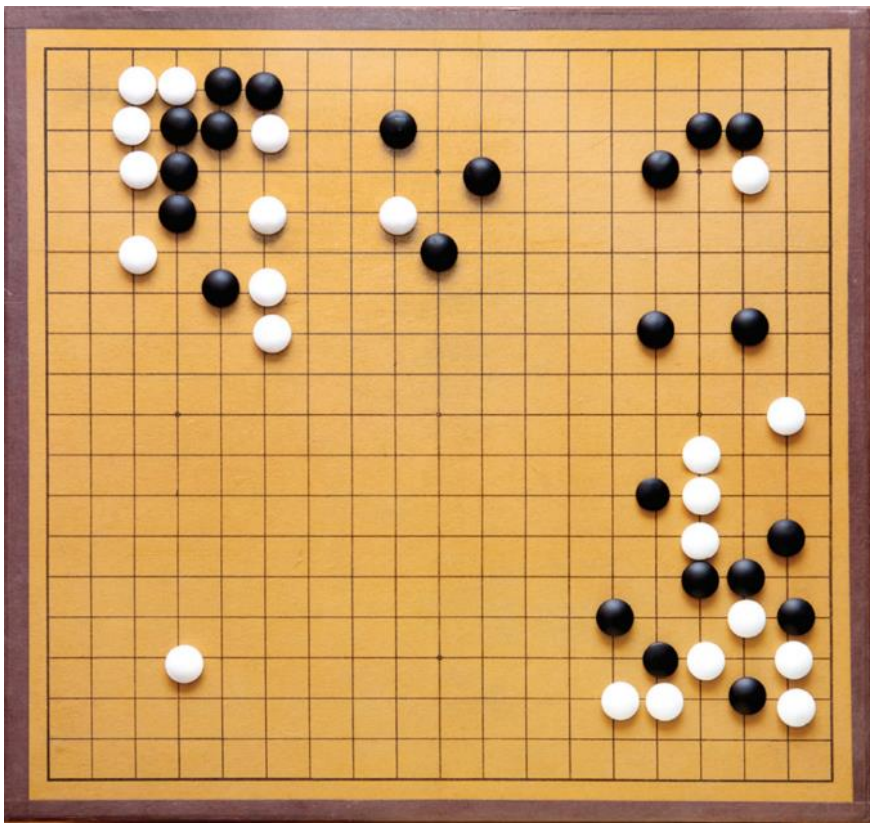

Рисунок 1.1 – процес гри у Go

Перша версія програми AlphaGo Fan була розроблена на основі двох нейронних мереж для оцінки виграшу та стратегії з використанням пошуку Монте-Карло [12]. Виграш – це очікувана майбутня нагорода з поточного стану з припущенням, що гравець грає ідеально. Стратегія – оцінка та пошук найкращої дії у поточній конфігурації дошки. Ця версія була тренована на основі ігор кращих гравців, тобто використовувала метод навчання з учителем.

Наступна версія програми, AlphaGo Zero, є цікавішою з точки зору теми курсової та відрізнялася від попередника декількома важливими аспектами. По-перше, вона тренувалася за технікою гри-з-собою, починаючи з випадкової тактики гри. По-друге, вона використовувала сиру інформацію з дошки – розміщення білих та чорних камінців, без якоїсь додаткової обробки цієї інформації людиною. По-третє, алгоритм був побудований на одній нейронній мережі, замість двох окремих для стратегії та оцінки виграшу. Нарешті, вона використовувала більш простий пошук по дереву,

орієнтований на одну нейронну мережу для оцінки позицій та можливих ходів, без виконання випадкових ходів за методом Монте-Карло [12].

В основі AlphaGo Zero лежить новий алгоритм навчання з підкріпленням. Нейронна мережа f_0 має параметри θ і приймає на вхід s – поточний стан дошки. Вона має два виходи – неперервне значення стану дошки $v_\theta(s) \in [-1; 1]$ з перспективи поточного гравця та стратегію $\overrightarrow{p_\theta}(s)$ – вектор ймовірностей по всім можливим крокам. Під час тренування мережі, в кінці кожної партії гри-з-собою, нейронна мережа надає тренувальні екземпляри у вигляді $(s_t, \overrightarrow{\pi_\theta}, z_t)$, де $\overrightarrow{\pi_\theta}$ – оцінка стратегії зі стану s_t, а z_t – результат гри з перспективи гравця зі станом s_t, приймає значення -1, якщо гравець програв, і +1, якщо виграв. Нейронна мережа тренується для мінімізації наступної функції втрат:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \overrightarrow{\pi_t} \cdot \log(\overrightarrow{p_\theta}(s_t))$$

Усі ідея полягає в тому, що ця нейрона мережа буде навчатися тому, які стани ведуть до перемог (або поразок) [12].

Як і TD-Gammon, AlphaGo Zero не тільки самостійно прийшла до основних ходів у Го, відомих професіоналам, але й винайшла нові стратегії гри. AlphaGo Zero, заснована на навчанні з підкріпленням, не тільки змогла обіграти попередні версії, засновані на навчанні з учителем, але й здолала кращих гравців у світі серед людей. Більше того, щоб набути фундаментальних і не тільки знань про гру, алгоритму необхідно від кількох годин до пари днів, тоді як люди досліджували Го тисячі років з початку її створення [12], не кажучи вже про нові стратегії. AlphaGo Zero остаточно закріпила перевагу навчання з підкріпленням над навчанням з учителем у контексті розробки штучного інтелекту для відеоігор, показавши, що він не тільки креативніший, але й ефективніший.

1.3.3 OpenAI Five

OpenAI Five – проект американської команди дослідників штучного інтелекту OpenAI, метою якого є створення і розвиток штучного інтелекту у такому вигляді, щоб він був безпечним та корисним для людства. Як поле для експериментів вони використовують гру Dota 2, і вже досягли чималого успіху, адже їхні агенти змогли подолати найкращу команду світу з рахунком 2:0 [13].

Dota 2 – це відеогра жанру MOBA (Multiplayer Online Battle Arena), у якій дві команди з п’ятьох гравців у кожній намагаються знищити ворожу головну споруду, в той же час захищаючи свою власну [14]. Десять гравців контролюють одного з 121 доступних на вибір персонажів («героїв»), кожен з яких унікальний: має свої здібності, переваги та недоліки. Цілком пояснити механіку Dota 2 коротко майже неможливо, адже вона складається з величезної кількості аспектів [13].

Але є два особливо важливі аспекти Dota 2, які ускладнюють сприйняття цієї гри штучним інтелектом:

- Довгий час гри – Dota 2 працює на 30 кадрах в секунду, а середня гра йде 45 хвилин. OpenAI робила крок кожний 4-й кадр, що у середньому давало 20,000 кроків. Для порівняння, гра в шахи у середньому йде 80 ходів, а в Го – 150 [13].
- Dota 2 – гра з неповною інформацією. Для кожної команди існує свій «туман війни» - область, інформація про яку відсутня (не вважаючи рельєфу). Люди використовують свої аналітичні здібності, щоб передбачити, що відбувається в таких областях [13].

- Великий простір і можливі варіанти ходів – Dota 2 грається на великій карті з 10 героями, великою кількістю будівель, непідконтрольних гравцям юнітів, складним рельєфом, тощо. На кожному кроці OpenAI Five оперувала з 16,000 значеннями (дробовими або категоричними) і обирала між 8,000 до 80,000 варіантами ходів. Для порівняння, у Го програма оперувала з 6,000 бінарними значеннями за крок та обирала між 250 варіантами ходів [13].
- Dota 2 – командна гра, де кожен гравець має надати свої 20% для досягнення перемоги. Більше того, комбінація двох-трьох певних героїв може бути набагато сильнішою за таку ж кількість героїв взятих окремо. Професіонали постійно вигадують нові тактики та покращують існуючі, налагоджують комунікацію в команді і особисті навички для досягнення перемоги.

Проект OpenAI Five був заснований у 2016 році, а перше досягнення здобув у серпні 2017 – ігровий агент зміг подолати одного з професіоналів гри у поєдинку 1x1 [15]. Через рік, агент був покращений до гри 5x5 і зміг подолати команду своїх розробників (хоч і з дещо обмеженими правилами). Його перемогу високо оцінив Білл Гейтс, зазначивши, що перемога агенту вимагала навичок командної роботи і взаємодії – і це велике досягнення у розвитку штучного інтелекту [16].

У 2018 році агент програв 8-й по сили команді світу, але з невеликим відривом – гра йшла 50 хвилин (при середній довжині гри між людьми в 45 хвилин) та закінчилися з рахунком 45:41 на користь OpenAI (але їхня головна споруда була знищена) [17].

Нарешті, після збільшення обсягу LSTM та кількості параметрів мережі і декількох інших покращень, 13 квітня 2019 року OpenAI Five отримує чисту перемогу над чемпіонами світу – командою OG (з невеликими обмеженнями, радше на користь людей). Більше того, усього через два місяці новий агент під назвою Rerun досяг 98% перемог проти агента, що грав з OG. Це була

перша перемога штучного інтелекту над чемпіонатами світу у кіберспортивній дисципліні [15].

Цікаво, що ігровий агент винайшов свій стиль гри у Dota 2. Люди використовували стратегію Core-Support, що означало, що 1-2 гравці отримують майже усе золото з карти, тоді як інші гравці їх захищають та виграють час. Ця стратегія завжди була доволі ризикованою, адже вимагала від Core гравців високого рівню здібностей, адже саме вони мали з золотом привести команду до перемоги. OpenAI Five, навпаки, прийшов до того, що усе золото на карті розподілялося рівномірно між усіма героями. Коли один герой ставав досить багатим, а інший – біднішим, перший йшов захищати другого, поки він здобував золото [13]. Таким чином, з ходом гри, кожен герой OpenAI Five залишався небезпечним і боєздатним, тоді як у людей гравці підтримки (Support) з часом ставали слабшими.

OpenAI Five архітектурно влаштована як 4096-unit LSTM, де LSTM (Long Short-Term Memory) – архітектурна рекурентної нейронної мережі. Усі можливі значення гри укомплектовані у єдиний вектор, що подається мережі на вхід. Кожен герой керується копією даної LSTM. Важливою її частиною є функція втрат, яка по суті є набором коефіцієнтів для кожної дії героя. Наприклад, за кожну отриману одиницю золота бот отримує 0,006 очок, а за смерть втрачає очко. Цікавим тут є те, що в функцію втрат також входить параметр, який отримав назву «Командний дух» (Team Spirit). Він усереднює нагороду усіх героїв, щоб кожен окремний персонаж «переймався» нагородою членів своєї команди. І якщо на початку експерименту цей параметр мав вагу 20%, то у кінці досяг 97% [18]. Спрощена модель роботи мережі OpenAI Five надана на Рис. 1.2.

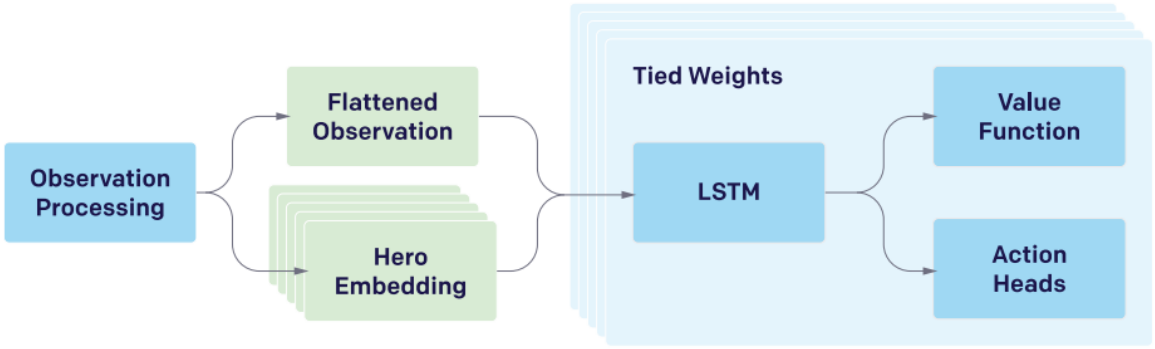

Рисунок 1.2 – Спрощена модель роботи мережі OpenAI Five

Розділ 2. Еволюційні алгоритми

2.1 Основні поняття

Основна ідея еволюційних алгоритмів полягає у наступному: задана популяція індивідуумів у певному середовищі, що має обмежені ресурси. Змагання за ці ресурси провокує природний відбір, тобто виживання найпридатнішого. Це, в свою чергу, збільшує загальну придатність популяції. Для максимізації функції якості популяції, створюється випадкова множина кандидатів. Потім до них застосовується функція кості – чим більше значення буде на виході, тим краще. На основі отриманих даних обираються найкращі кандидати – вони стануть зерном наступного покоління.

У процесі створення нової популяції застосовуються дві техніки. Перша, рекомбінація – застосовується до двох кандидатів (так званих «батьків), створючи одного або більше нових кандидатів («дітей»). Друга, мутація, застосовується до одного кандидата і створює нового. Застосування цих технік веде до створення множини нових кандидатів – «нащадків». Процес повторюється спочатку, допоки не буде отриманий кандидат з достатнім рівнем якості («розв'язок») або не будуть вичерпані обчислювальні потужності.

У еволюційних алгоритмах використовуються дві основні сили, які створюють базис еволюційної системи:

- Варіаційні оператори (рекомбінація та мутація) створюють необхідну різноманітність у популяції, і таким чином провокують новизну.
- Вибірка виступає як сила що збільшує середню якість розв'язків у популяції.

Варто зауважити, що більшість компонентів такого еволюційного процесу є стохастичними. Наприклад, під час вибірки, найкращі індивідууми не обираються зі 100% ймовірністю, а іноді навіть слабкі індивідууми мають шанси стати батьком чи вижити. Під час рекомбінації, вибір які частини батьків будуть використані при створенні потомства обираються випадковим чином. Також і з мутацією, вибір частин для змін обираються випадково.

Під час розробки еволюційного алгоритму, має бути визначена певна кількість різних компонентів, процедур та операторів, щоб задати конкретний еволюційний алгоритм. Далі будуть розглянуті найбільш важливі компоненти.

Репрезентація – перший крок у створенні еволюційного алгоритму. Задача – пов'язати задачу «реального» світу до світу еволюційного алгоритму. Зазвичай, це означає спрощення або абстрагування певних частин реального світу для створення гарно-визначеного проблемного контексту, у якому можливі розв'язки можуть з'являтися та існувати. Можливі розв'язки у термінах оригінального світу називаються *фенотипами,* тоді як їхнє представлення у комп'ютері, тобто у світі еволюційного алгоритму, називається *генотипом*. Власне, репрезентація полягає у тому, щоб вказати зв'язок між фенотипом та генотипом.

Наступним кроком йде задання *функції оцінки*, яка представляє собою вимоги до популяції, до яких вона має адаптуватися. Саме ця функція означає, що таке «покращення» у контексті оцінки нової популяції. Значення, що ця функція подає на вихід, мають бути порівнювальними, щоб оцінювати індивідуумів між собою.

Задача *популяції* полягає в тому, щоб містити множину можливих розв'язків. Популяція – це мультимножина генотипів. Саме популяція є основною

одиницею еволюції, адже індивідууми це статичні об’єкти, які ніяк не змінюються чи адаптуються; це робить саме популяція. Варто зазначити, що майже у всіх еволюційних алгоритмах розмір популяції константний – це створює обмеженість ресурсів, яка провокує змагання.

Різноманітність популяції – це міра кількості наявних різних розв’язків. Не існує єдиної міри для різноманітності, найчастіше це просто кількість різних значень функції оцінки після вимірювання популяції.

Важливою складовою еволюційних алгоритмів є ймовірністність – скажімо, мутації чи рекомбінації. Наприклад, якщо для рекомбінації обирати тільки найкращих кандидатів, то система може застрягти в локальному оптимумі. Замість цього, кожен кандидат отримує ймовірність стати батьком, і у гірших кандидатів вона просто нижча, ніж у кращих, але ненульова.

2.2 Сфери застосування

Одна з найпоширеніших сфер застосування еволюційних алгоритмів – оптимізація. Еволюційні алгоритми дуже добре зарекомендували себе у розв’язку задач, які зводяться до повного перебору – іншими словами, NP повних задач. В той час, коли повний перебір усіх варіантів зайняв би надзвичайно багато комп’ютерного часу, еволюційні алгоритми здатні знаходити близький до найкращого розв’язок за доволі короткий час.

Наприклад, генетичні алгоритми використовуються у реляційний базах PostgreSQL та H2 [19] для оптимізації запитів. Один і той же результат може бути отриманий різними способами (планами виконання запиту), деякі вимагають набагато більше комп’ютерних ресурсів, ніж інші. Оскільки пошук оптимального плану – NP повна задача, генетичні алгоритми стають дуже корисними, особливо за великого числа таблиць. Таблиці, які беруть участь у запиті, кодуються в хроносоми. В результаті відбору лишаються лише ті хромосоми, що дають локальний мінімум функції вартості (при цьому вважається, що глобальний мінімум не дає істотних переваг над найкращим локальним мінімумом).

Оптимізація використовується і у термінах реального світу – для індустріального дизайну. Одним із прикладів є пошук оптимальної конструкції тримача для супутникової тарілки. Він має бути стабільним і захищеним від вібрацій. На Рисунку 2.2.1 зазначено традиційний тримач (зліва) у порівнянні з тримачом, знайденим еволюційним алгоритмом

(справа). Результат має показники на 20,000% кращі, ніж традиційна форма, але виглядає досить дивакуватим і несиметричним. Це також показує природу еволюції як дизайнера – вона не враховує таки параметри як естетичність чи якісь загальноприйняті конвенції.

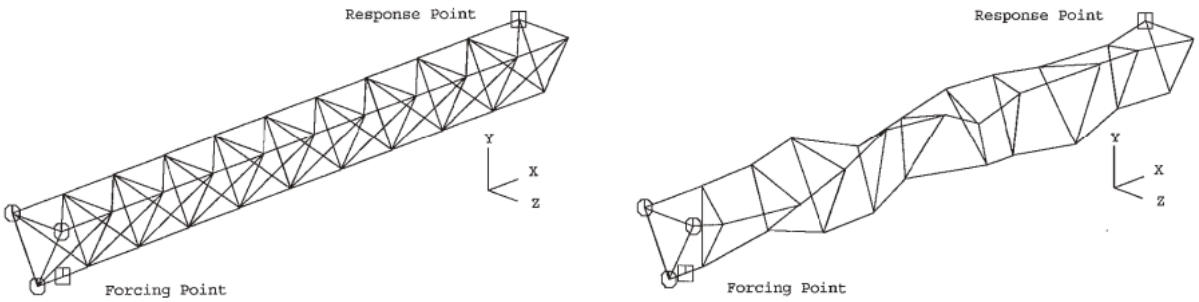

Рисунок 2.2.1 – традиційна форма тримача (зліва) у порівнянні з формою, винайденою еволюційним алгоритмом (справа)

На рисунку 2.2.2 показана винайдена NASA антена, яка також була ефективнішою за традиційні та мала незвичну для людини форму. Такий вид антен навіть отримав спеціальну назву – еволюційні антени (evolved antenas), тобто антени, винайдені еволюційним алгоритмом.

Рисунок 2.2.3 – еволюційна антена, винайдена еволюційним алгоритмом NASA

Очевидним є також використання еволюційних алгоритмів для моделювання біологічних процесів, причому як з існуючими видами, так і вигаданими.

2.3 Генетичні алгоритми

Генетичний алгоритм – найбільш відомий тип еволюційного алгоритму. Класичний генетичний алгоритм має бінарну репрезентацію, вибірку пропорцію до якості, низьку ймовірність мутації та наголос на використанні результатів рекомбінації як спосіб генерації нових кандидатів.

Генетичний алгоритм має фіксований робочий процес: дана популяція μ індивідуумів, вибірка з батьків заповнює проміжну популяцію від μ, дозволяючи дублікати. Потім, ця проміжна популяція випадково тасується, щоб створити випадкові пари для рекомбінації з ймовірністю p_c, діти одразу ж заміщають батьків. Нова проміжна популяція підлягає індивідуальній мутації, де кожен з l бітів у індивідуума модифікується мутацією з незалежною ймовірністю p_m. Результуюча проміжна популяція створює наступне покоління, повністю заміщуючи старе. Варто помітити, що у новому поколінні можуть бути індивідууми, що пережили рекомбінації та мутації без модифікацій, але ймовірність цього досить мала (залежить від параметрів μ, p_m та p_c).

На початку історії генетичних алгоритмів йшло багато історії, які значення мають бути у параметрів μ, p_m та p_c. Ймовірність мутації визначалася між $\frac{1}{l}$ та $\frac{1}{\mu}$, ймовірність рекомбінації між 0.6 – 0.8, а розмір популяції встановлювався від 50 до кількох сотен хромосом. Наразі, ймовірність мутації кодується у кожен ген і може еволюціонувати разом з іншими параметрами. Для рекомбінації використовують альтернативи як рівномірну рекомбінацію та інші. Загалом, класичний варіант генетичного алгоритму зараз використовується у навчальних та порівняльних цілях.

Розділ 3. Розробка ігрового агенту

3.1 Визначення поля домену

StarCraft II – відеогра 2010 року жанру стратегій у реальному часі. Основний акцент у грі робиться на видобутку ресурсів, розбудові бази та численних битвах. Добуваючи ресурси, гравець отримує кошти для будівництва споруд, які можуть виробляти бойові одиниці й надавати вдосконалення для них. Мета гравця – знищити базу та робітників противника.

На початку гри гравець володіє головною спорудою, розташованою біля джерел ресурсів, і кількома робітниками. Робітники добувають ресурси й зводять споруди. В міру розбудови бази відкриваються нові споруди, війська, вдосконалення. Однією з особливостей гри є протистояння трьох рас – кожна з яких має чимало унікальних споруд, бійців та ігрових механік. Це робить StarCraft II грою з асиметричним балансом – в противагу іграм по типу шах чи Го.

На Рисунку 3.1.1 наведений скріншот з гри StarCraft II.

У контексті розробки та дослідження ігрових агентів, StarCraft II цікава тим, що має для цього спеціальний фреймворк – PySC2. Він був розроблений у співробітництві між Blizzard (розробниками StarCraft II) та DeepMind (Google) задля розвитку StarCraft II як середовища для дослідження навчання з підкріпленням. PySC2 надає агентам інтерфейс для взаємодії з грою, тобто можливість робити спостереження та надсилати дії. На рисунку 3.1.2 надано скріншот інтерфейсу фреймворку, який представляє собою дуже спрощу версію гри (що надзвичайно прискорює навчання, адже звільняє ресурси GPU).

Рисунок 3.1.1 – скріншот з гри StarCraft; армія протосів веде наступ на базу теранів (людей)

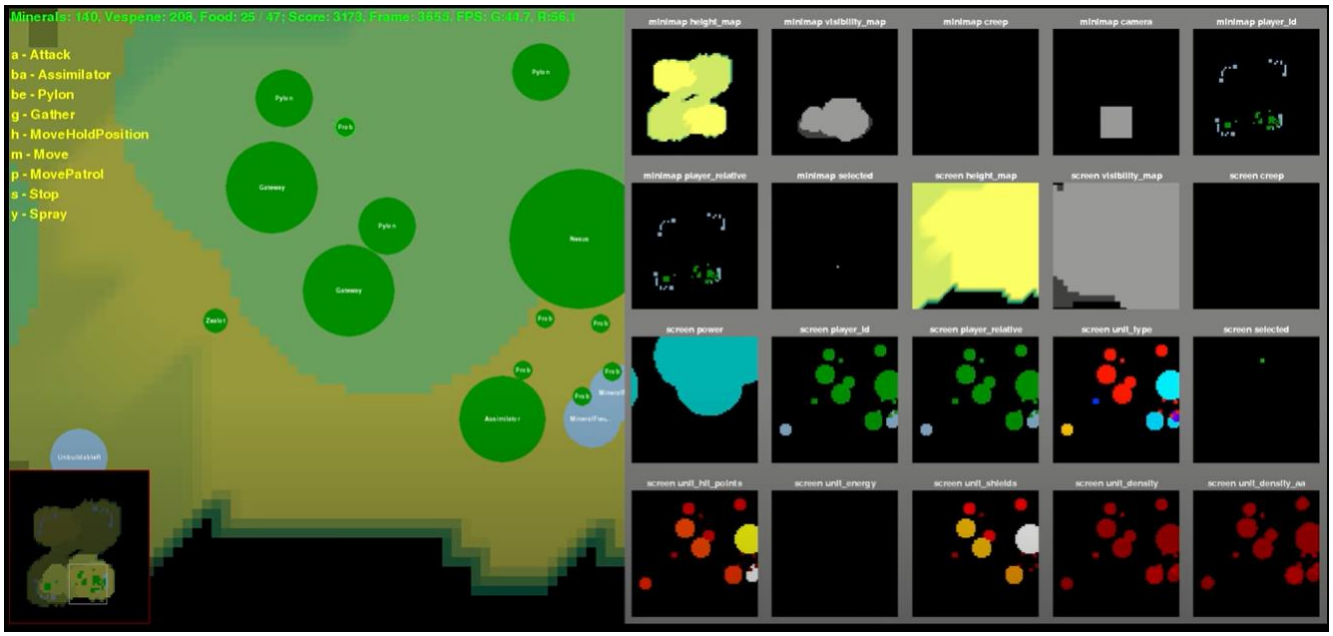

Рисунок 3.1.2 – скріншот інтерфейсу фреймворку PySC2

Розробка бота виконувалося мовою Python, з використанням вбудованих бібліотек random, math, os, а також бібліотек numpy і pandas.

3.2 Ігровий агент на основі Q-Навчання

Поле домену може бути представлене у вигляді Марковського процесу вирішування, що містить усі множини дій, станів, переходів та нагород. Майбутні стані можуть бути передбачені і залежать від поточного стану агенту, зберігаючи усі вимоги процесу Маркова. Головна ціль навчання з підкріпленням – визначити ідеальну множину завдань для заданого Марковського процесу рішень.

Для розробки боту було вирішено використати Q-Навчання, переваги цього підходу будуть наведені пізніше. Формула Q-Навчання має наступний вигляд:

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right),$$

де Q – стратегія винагороди від стану s і дії a, r_t – винагорода, α – параметр швидкості навчання.

В результаті отримуємо Q-таблицю від значень стану і дії. Для кращого розуміння, дивіться Рисунок 3.2.1.

Рисунок 3.2.1 – Q-значення, яке є стратегію винагороди ігрового агенту, обирається з Q-таблиці для відповідних значень стану та дії

Q-таблиця створюється для середовища розмірністю SxA, де S і A – кількість станів та дій відповідно. Є декілька дій для кожного стану, і ймовірність вибору конкретної дії визначається значеннями у Q-таблиці, відомими як оцінка стан-дії (state-action value). Q-таблиця на початку заповнюється нулями. Для кожного стану, обирається дія. Q-значення для цієї стан-дії збільшується, якщо дія надає хорошу винагороду для наступного стану; інакше, Q-значення зменшується.

3.3Деталі реалізації

Простий алгоритм Q-Навчання у Python можна написати у 40 стрічок коду. Він представляє собою клас QLearningTable з методами learn, choose_action, check_state_exists та конструктором __init__. У __init__ відбувається ініціалізація значень класу, а саме: параметр швидкості навчання рівний 0,01, фактор знижки – 0,9, епсілон – 0,9.

У методі choose_action або обирається найкраща дія для поточного стану, або повертається випадкова – у разі відсутності даного стану у Q-таблиці.

У методі learn для поточного та наступного станів рахується за формулою Q-значення та заноситься у таблицю.

Метод check_state_exists просто перевіряє наявність стану у Q-таблиці.

Далі код містить багато взаємодії з API PySC2, тому опишу основні принципи роботи цієї версії агенту:

- Карта ділиться на чотири квадранти. Так простір дій агента дещо зменшується і йому простіше вчитися.
- Система дій є трьохкроковою. Наприклад, дія Побудувати Бійця складається з трьох кроків – обрати бараки, тренувати бійця, нічого не робити. Це допомогає абстрагувати дії ігрового агента і значно полегшує його навчання. Як видно з прикладу, якщо дія загалом складається з двох кроків, вона доповнюється кроком «нічого не робити», для спрощення коду.
- Версія бота містить 5 дій: Нічого не робити (х3 нічого не робити), Побудувати склад (обрати робітника, побудувати склад, відправити робітника працювати), Побудувати казарми (Обрати робітника, побудувати казарами, відправити робітника працювати), Побудувати Бійця (описано вище), Атакувати(х,у) (обрати армію, атакувати координати, нічого не робити).
- У кінці гри (фреймворк повідомить про перемогу одного з гравців або перевищення максимальної кількості ходів), передаємо класу QLearningTable винагороду – 1 за перемогу, -1 за програш, 0 за ніччю і оновлюємо таблицю.

Дана версія агенту перемагала у 26% відсотках ігор, здобувала ніччю у 24% та програвала у половині. Гра відбувалася зі стандартним ігровим агентом від фреймворку PySC2, створеним для таких порівнянь. Це вже був непоганий результат, але його варто було покращувати:

- Виключення неможливих дій – під час гри агент часто починав виконувати неправильні (неможливі) дії, через що не міг робити корисні дії далі. Такі дії необхідно виключати з Q-таблиці, якщо за умовами середовища вони не дозволяються. Наприклад, агенту немає сенсу намагатися відправити воїнів в атаку, якщо в нього їх нема.
- Додати місцезнаходження підконтрольних одиниць у стани – це дозволяє агенту краще атакувати, адже у попередній версії він не знав, де знаходяться його бійці, і рішення про атаку було ледве не випадковим.

Версія алгоритму з цими покращеннями змогла досягти 50% перемог, 25% поразок і 25% нічиїх.

3.4 Перспективи

До перспектив розвитку ігрового агенту можна віднести додавання можливості грати за раси зергів та протосів (поки що він грає людьми). Також, за допомогою нещодавно анонсованої системі мінігор у PySC2, варто розділити процес навчання на окремі модулі. Процес бою може бути значно покращений.

Висновки

У результаті виконання даної роботи було систематизовано знання з теми навчання з підкріпленням та еволюційних алгоритмів. Було розібрано основні принципи роботи таких методів як Монте Карло, Часових Відмінностей та Динамічного Програмування. Був наданий огляд визначних епох розвитку навчання з підкріпленням у вигляді ігрових агентів, спочатку від відносно простої гри – нарди, до надзвичайно складної – Dota 2.

Систематизовано основні дані про еволюційні алгоритми а також їх сфери застосування.

Розроблено власного ігрового агента для гри в StarCraft II на основі Q-Навчання.

Список джерел

[1] A. G. B. s. e. Richard S. Sutton, Reinforcement Learning: An Introduction, The MIT Press: Cambridge, Massauchsetts, 2018.

[2] F.-A. Fortin, «DEAP: Evolutionary Algorithms Made Easy,» *Journal of Machine Learning Research,* pp. 2171-2175, 7 12 2012.

[3] R. S. S. a. A. G. Barto, Reinforcement Learning: An Introduction, Cambridge, Massachusetts: The MIT Press, 1998.

[4] M. W. v. Otterlo, Reinforcement Learning, -: Springer, 2012.

[5] R. B. B. D. S. D. E. Lucian Busoniu, Reinforcement Learning and Dynamic Programming Using Function Approximators, New York: CRC Press, 2010.

[6] K. S. W. (. D. H. (. W. S. L. (. Yi Wang (NUS), «Monte Carlo Bayesian Reinforcement Learning,» *arXiv.org,* pp. 1-18, 27 6 2012.

[7] G. Tesauro, «Temporal Difference Learning and TD-Gammon,» *Communications of the ACM,* т. 38, № 3, pp. 58-68, 1 March 1995.

[8] G. Tesauro, «Practical issues in temporal difference learning,» *Machine Learning,* т. 8, pp. 257-277, 1 May 1992.

[9] G. Tesauro, «TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play,» *Neural Computation,* т. 6, № 2, pp. 215-219, 1 March 1994.

[10] D. K. Antonio Ortega, Backgammon With the Giants: Neil Kazaross, San Jose, Costa Rica: Editorama, S.A., 2001.

[11] J. X., «The Evolution of Computing: AlphaGo,» *Computing in Science & Engineering,* т. 18, № 4, pp. 4-7, 24 June 2016.

[12] A. H. C. J. M. A. G. L. S. G. V. D. .. D. Silver, «Mastering the game of Go with deep neural networks and tree search,» *Nature,* т. 529, № 7587, pp. 484-489, 1 January 2016.

[13] G. B. B. C. V. C. P. D. C. D. D. F. Q. F. S. H. C. H. R. J. S. G. C. O. J. P. M. P. .. Christopher Berner, «arXiv.org,» 13 12 2019. [Онлайновий]. Available: https://arxiv.org/abs/1912.06680. [Дата звернення: 17 5 2021].

[14] T. McDonald, «A Beginner's Guide to Dota 2: Part One – The Basics,» PcInvasion, 25 7 2013. [Онлайновий]. Available: https://www.pcinvasion.com/a-beginners-guide-to-dota-2-part-one-the-basics/. [Дата звернення: 17 5 2021].

[15] G. B. B. C. V. C. P. D. C. D. D. F. Q. F. S. H. C. H. R. J. S. G. C. O. J. P. M. P. .. Christopher Berner, «OpenAI Five (2016 - 2019),» OpenAI Team, 31 12 2019. [Онлайновий]. Available: https://openai.com/projects/five/. [Дата звернення: 17 5 2021].

[16] B. Gates, «twitter.com,» Twitter, 27 6 2018. [Онлайновий]. Available: https://twitter.com/BillGates/status/1011752221376036864. [Дата звернення: 17 5 2021].

[17] O. Team, «The International 2018: Results,» 23 8 2018. [Онлайновий]. Available: https://openai.com/blog/the-international-2018-results/. [Дата звернення: 17 5 2021].

[18] D. Farhi, «Dota Reward Function,» 21 4 2019. [Онлайновий]. Available: https://gist.github.com/dfarhi/66ec9d760ae0c49a5c492c9fae93984a. [Дата звернення: 17 5 2021].

[19] T. T. A. A. S. K. H. A. G. C. A. L. Aponsoa, «Database Optimization Using Genetic Algorithms for,» *International Journal of Computer,* pp. 23-27, - - 2017.