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

ТРЕНУВАННЯ ІГРОВИХ АГЕНТІВ МЕТОДАМИ НАВЧАННЯ З ПІДКРІПЛЕННЯМ

Текстова частина до курсової роботи за спеціальністю “Інженерія програмного забезпечення” 121

Керівник курсової роботи кандидат фізико-математичних наук, доцент
Крюкова Галина Віталіївна

_____

(підпис)

“____”_____2020 р.

Виконав студент
Василенко Олександр Сергійович

“____”_____2020 р.

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій

ЗАТВЕРДЖУЮ
Керівник курсової роботи кандидат фізико-математичних наук, доцент
_____Крюкова Г.В. (підпис)
“____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 4-го курсу факультету інформатики
Василенко Олександру Сергійовичу
ТЕМА: Тренування ігрових агентів методами навчання з підкріпленням
Зміст ТЧ до курсової роботи:

Вступ
1. Навчання з підкріпленням
2. Розробка ігрового агента
Висновки
Список літератури
Додатки

Дата видачі “___” _____ 2020 р. Керівник _____
(підпис)

Завдання отримав _____
(підпис)

Тема: Тренування ігрових агентів методами навчання з підкріпленням

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	листопад 2019 р.	
2.	Огляд літератури за темою роботи.	грудень 2019 р.	
3.	Розробка ігрового агента	січень 2020 р.	
4.	Написання пояснювальної роботи	лютий 2020 р.	
7.	Створення слайдів для доповіді та написання доповіді.	15.03.2020	
8.	Аналіз отриманих результатів керівником.	22.03.2020	
8.	Корегування роботи за результатами аналізу.	29.03.2020	
10.	Остаточне оформлення пояснювальної роботи та слайдів.	5.04.2020	
11.	Захист курсової роботи.	19.05.2020	

Студент **Василенко Олександр Сергійович**

Керівник **Крюкова Галина Віталіївна**

“_____”_____ 2020 р.

Зміст

Анотація

Суть курсової роботи полягає у дослідженні результатів роботи ігрового агента для гри «Snake», тренованого за допомогою навчання з підкріпленням, а саме – Deep Q-Learning. Результатом роботи є програма для тренування ігрового агента, що дозволяє запускати його з різними параметрами, такими як ширина та висота ігрового поля, коефіцієнт знецінювання γ, темп навчання $\propto$ і вірогідність вибору випадкової дії ε. Програму реалізовано мовою Python.

Вступ

Завдяки значному розвитку глибокого навчання в області навчання з підкріпленням, створюється все більше систем, що покращують свої характеристики під час роботи. Серед таких – просунуті ігрові агенти, що можуть вчитися на великій кількості вхідних даних. Натренований ігровий агент може перевершити людину (гравця), і, дослідивши, яким чином програмі вдалось оптимізувати ті чи інші дії, гравець може покращити свої результати, знайшовши новий підхід для досягнення мети. Таким чином ігровий агент може бути своєрідним тренером для гравця або ж стати супротивником для видовищного поєдинку.

Метою розробки є створення ігрового агента для гри «Snake», результати якого можна порівняти з результатами звичайного гравця. Для цього необхідно реалізувати агента та провести ряд випробувань, натренувавши його з різними параметрами навчання.

Робота складається з двох розділів. Перший розділ присвячено визначенню навчання з підкріпленням, принципам роботи Q-Learning та особливостям Deep Q-Learning. У другому розділі наведено дослідження результатів роботи розробленого ігрового агента.

1. НАВЧАННЯ З ПІДКРІПЛЕННЯМ

1.1. Визначення

Перш за все варто визначити, що навчання з підкріпленням (англ. reinforcement learning) — це галузь машинного навчання, у якій агент (система) вчиться взаємодіяти з середовищем таким чином, щоб його дії максимізували деяку довготермінову сукупну винагороду. Реакцією середовища на обрані агентом рішення є сигналами підкріплення, тому таке навчання схоже на частковий випадок навчання з учителем, але у контексті навчання з підкріпленням, учителем є саме середовище або його модель. Навчання з підкріпленням відрізняється від стандартного навчання з учителем тим, що пари правильних входів/виходів ніколи не представляються, а недостатньо оптимальні дії явно не виправляються. Крім того, є акцент на інтерактивній продуктивності, який включає знаходження балансу між дослідженням та використанням. [1] З іншого боку, у випадку з використанням нейронної мережі, деякі сигнали підкріплення базуються на одночасній активності формальних нейронів, тому навчання з підкріпленням можна віднести до навчання без учителя. Саме тому reinforcement learning не відноситься ні до навчання з учителем, ні до навчання без учителя і є окремою гілкою в машинному навчанні.

Формально найпростіша модель навчання з підкріпленням складається з:

1) Множина станів середовища та агента S;

2) Множина дій агента A;

3) $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ вірогідність переходу в момент t зі стану s у стан s’ після дії a;

4) $R_a(s, s')$ скалярна безпосередня винагорода переходу зі стану s у стан s’ після дії a;

5) Правила, що описують, що спостерігає агент в середовищі

У кожен момент часу t агент характеризується станом $s_t \in S$ та множиною можливих дій $A(s_t)$. Обираючи дію $a_t \in A(s_t)$, він переходить у стан s_{t+1} та

отримує винагороду r_{t+1}. Базуючись на такій взаємодії з середовищем, агент, що навчається з підкріпленням, має створити стратегію $\pi: S \rightarrow A$, що максимізує величину майбутньої винагороди $R = r_0 + r_1 + \cdots + r_n$, тому щоб діяти майже оптимально, агент має розуміти довготермінові наслідки дій, аби максимізувати майбутній дохід. [2]

Таким чином навчання з підкріпленням дуже добре підходить для вирішення задач, пов’язаних з вибором між довготерміновою та короткотерміновою винагородою. Воно активно використовується у ряді предметних областей, де необхідно обрати найкращий варіант серед багатьох, аби досягти складної мети за велику кількість кроків, таких як робототехніка, телекомунікації, безпілотні автомобілі.

Навчання з підкріпленням вимагає особливого механізму дослідження, бо випадковий вибір дій дає жахливу продуктивність. На практиці вдаються до простих методів дослідження через нестачу складних алгоритмів, які б довідно добре працювали з великою кількістю станів. Найчастіше вдаються до частково випадкової стратегії, в якій агент передбачає найкращу дію для довготермінового ефекту з імовірністю $1 - \varepsilon$, інакше дія обирається навмання. Параметр іноді змінюється в діапазоні $(0 < \varepsilon < 1)$ або згідно фіксованого розкладу або адаптивно на основі інших параметрів.

1.2. Q-Learning

Q-навча́ння (англ. Q-learning) — це алгоритм безмодельного навчання з підкріпленням. Метою Q-навчання є навчитися стратегії, яка каже агентові, до якої дії вдаватися за яких обставин. Воно не вимагає моделі середовища (звідси уточнення «безмодельного»), і може розв'язувати задачі зі стохастичними переходами та винагородами, не вимагаючи пристосувань.

Q-навчання знаходить стратегію, яка є оптимальною в тому сенсі, що вона максимізує очікуване значення повної винагороди над будь-якими та усіма послідовними кроками, починаючи з поточного стану. Q-навчання може визначати оптимальну стратегію обирання дій за умови нескінченного часу на розвідування та частково випадкової стратегії. Символом Q позначають функцію, яка повертає винагороду, що використовують для забезпечення підкріплення, і про яку можливо сказати, що вона відповідає «якості» дії, обраної в поточному стані. [3] У найпростішому варіант Q-функцію визначають у вигляді таблиці станів та дій, у якій зберігається значення Q для кожної пари стан-дія, таку таблицю називають Q-таблицею. Її формування відбувається за наступним алгоритмом:

6) Ініціалізація. Заповнюємо Q-таблицю початковими значеннями (нулями або навмання). Кожна комірка $Q(s, a)$ – оцінка того, наскільки добре у перспективі обрати дію a у стані s. Це передбачення буде покроково оновлюватись та покращуватись;

7) Спостереження. Агент запам’ятовує попередній стан s_t, обрану дію a_t, новий стан s_{t+1}, у який перейшов агент виконавши дію та отриману винагороду r у результаті переходу в новий стан. Таким чином ми збираємо інформацію про взаємодію агента з навколишнім середовищем;

8) Оновлення корисності (Temporal Difference або TD-Update). Формула оновлення виглядає так: $Q(s_t, a_t) = Q(s_t, a_t) + \propto (r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$, де s_t – поточний стан; a_t – поточна обрана дія; s_{t+1} - новий стан, у якому опинився агент; a_{t+1} - наступна дія з найбільшим значенням Q, яку варто обрати; r – поточна винагорода, реакція середовища у відповідь на застосовану дію; $\gamma (0 \leq \gamma \leq 1)$ – коефіцієнт знецінювання, що визначає важливість майбутніх нагород, чим він більше, тим більше агент прагне довготривалої винагороди (якщо він 0, агент стане надто “короткозорим”); $\propto (0 < \propto \leq 1)$ – темп навчання, чим він більший – тим більше агент «довіряє» новій інформації і вона перевизначатиме стару (якщо

він 0 – агент не буде сприймати нову інформацію) . Це оновлення застосовується на кожному кроці взаємодії агента з середовищем для оцінки значення Q;

9) Обираємо дію ε-жадібним способом. З вірогідністю $1 - \varepsilon$ обирається дія з найбільшим значенням Q і з вірогідністю ε дія обирається випадково;

10) Переходимо на крок 2.

1.3. Deep Q-Learning

Q-Learning створює таблицю або матрицю, у якій є точна відповідність між станом та дією, що максимізує майбутній виграш. Однак у такого підходу є значний недолік і полягає він у тому, що він припускає, що ми знаємо, скільки станів (рядків) у нашому середовищі, але це не завжди так. А з великою кількістю станів цей метод втрачає гнучкість.

Алгоритм Deep Q-Learning полягає у використанні нейронної мережі для оцінки функції Q. Тепер замість оновлення таблиці Q напряму, ми передаємо нейронній мережі стан s, після чого мережа повертає значення Q для кожної дії. Однією з проблем використання нейронних мереж у навчанні з підкріпленням є відсутність гарантій збіжності. Спричинено це тим, що невеликі зміни функції Q можуть значно змінити поведінку агента, розподіл даних та відповідність Q цільовим значенням.

Для покращення збіжності використовується метод збереження попереднього досвіду (experience replay). У кожен момент t ми зберігаємо досвід агента у вигляді $e_t = (s_t, a_t, r_t, s_{t+1})$ у пам’ять, а потім беремо частину досвідів, обраних довільним чином для тренування нейронної мережі. Таким чином experience replay дозволяє зменшити кореляції у послідовності спостережень, ми запобігаємо тому, що мережа навчається лише за поточними результатами, замість цього тренування відбувається за більш різноманітними наборами попередніх досвідів.

2. РОЗРОБКА ІГРОВОГО АГЕНТА

2.1. Гра “Snake”

Гра «Змійка» (Snake) була вперше розроблена для ігрового автомата Huslte у 1977 році компанією Gremlin Industries. Перша версія цієї гри для персональних комп’ютерів мала назву Worm, її було створено у 1978 році для TRS-80. Згодом стали з’являтися нові реалізації гри для різних платформ і однією з найпопулярніших була версія для телефонів Nokia, розроблена у 1998 році. На сьогоднішній день існують сотні варіантів «Змійки», але основа суть гри полягає в тому, що гравець керує продовгуватою, розділеною на квадрати (частини тіла), смужкою, що нагадує за формою змію, по ігровому полю, обмеженому стінами по краям. Змійка має збирати яблука (або інші об’єкти на карті) та уникати зіткнень із стінами або власним хвостом. Гравець управляє лише головою змії, повертаючи вправо, вліво або просуваючись вперед, а її тіло рухається за нею. Головна складність полягає в тому, що коли змійка збирає їжу, вона стає довшою (додається новий квадрат), нова їжа одразу з’являється у довільній точці на полі, а гравець ніяк не може зупинити рух.

2.2. Використані технології

Для розробки ігрового агента для гри змійка було використано ряд технологій:

1) Основна логіка гри та Deep Q-Learning агент реалізовано мовою програмування Python. Її обрано за наявність великої кількості інструментів машинного навчання, стабільність, гнучкість та простоту. До того ж, динамічна типізація допомагає простіше оперувати з великою кількістю різних даних.

2) Гра “Змійка” реалізована за допомогою бібліотеки модулей Pygame, що надає простий функціонал для створення 2D ігор. Основною перевагою є те, що розробник може безпосередньо керувати головним циклом гри.

3) Для ініціалізації нейронної мережі та навчання додано бібліотеку TensorFlow. Ця бібліотека надає доступ до високорівневого API Keras, що спрошує процес налаштування і дозволяє сконцентруватися на реалізації основної логіки агента. Також для тренування вона може застосовувати ресурси як CPU, так і GPU.

2.3. Структура програм

2.3.1. Реалізація гри

Логіка гри описана у модулі snakeGame.py, що включає у себе ряд класів, що описують ігрові правила, властивості елементів та їх відображення на екрані (рисунок 2.1). При розробці основним пріоритетом була гнучкість для налаштування при запуску гри для тестування різних варіантів роботи.

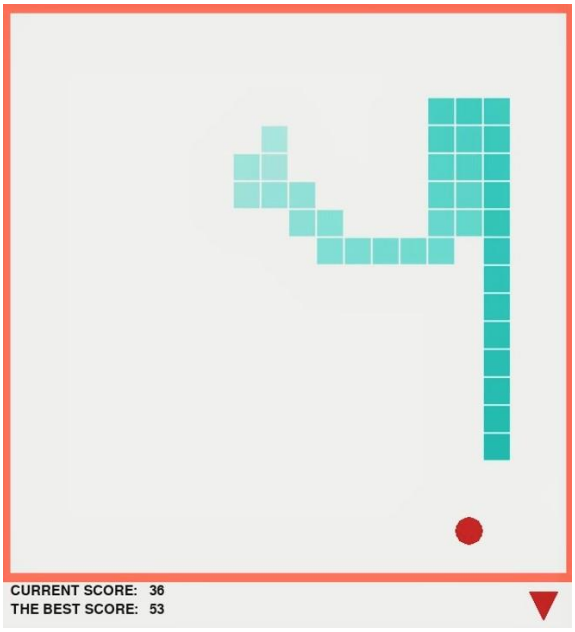

Рисунок 2.1 – Зображення ігрового вікна

За логіку ігрового світу та відображення усіх елементів, що входять до нього, відповідає клас Game. «Змійка» реалізована таким чином, що гравець може тільки рухатися вперед або здійснити поворот ліворуч та праворуч. Кожного разу, коли змійка з'їдає яблуко, її довжина збільшується, а поточний рахунок збільшується, якщо рахунок перевершує рекорд, він змінюється. Рахунок та рекорд відображаються у нижньому лівому куті. Гра закінчується, коли голова змійки перетинає кордон ігрового поля або власний хвіст або коли на протязі певної кількості кроків вона не змогла зібрати жодного яблука. У нижньому правому куті зображена стрілка, що вказує напрямок руху. Тіло змійки зображено у вигляді бірюзових квадратів, колір яких світлішає, чим частина тіла ближча до хвоста, а яблука – у вигляді червоних кіл. Для ініціалізації нового екземпляру гри, необхідно вказати наступні параметри класу Game:

11) Cols – ширина ігрового поля, по якому може пересуватися змійка, ціле число, що вказує кількість стовпчиків;

12) Rows – висота ігрового поля, ціле число, що вказує кількість рядків;

13) Size_snake – розмір квадрата ігрового поля, ціле число, що вказує ширину і висоту клітинок, на які поділене поле;

14) Border_color – колір рамки навколо ігрового поля (усі кольори задаються за допомогою класу pygame.Color);

15) Background_color – колір фону;

16) Food_color – колір яблук, які збирає змійка;

17) Snake_color – колір тіла змійки;

18) Speed – параметр, що вказує швидкість гри. Ціле число, що вказує часовий проміжок між відображенням кадрів. Якщо значення ≤ 0, вікно гри буде порожнім, гра не відображатиметься, що пришвидшує виконання;

19) Max_steps – кількість кроків, через яку змійка помре від голоду, якщо не з'їсть яблуко.

Основною функцією для взаємодії з грою є make_train_step, що приймає на вхід екземпляр ігрового агента та опціональну дію (якщо вона задана, агент не здійснює передбачення наступного кроку). Ця функція здійснює один «крок» гри: зберігає попередній стан гри, здійснює дію (обрану агентом або задану вручну у якості аргументу), зберігає новий стан, записує винагороду та здійснює тренування агента, надавши йому попередній стан, поточний стан, дію, нагороду та чи закінчилась гра, після чого відображає поточний ігровий кадр. Альтернативою є функція make_step, що відрізняється від попередньої лише тим, що тренування агента не здійснюється, це корисно для тестування готової моделі.

У класі Player описується положення частин його тіла, пересування та напрямок руху, а у класі Food – положення їжі. Коли гравець з'їдає яблуко, воно з'являється у випадковій вільній клітинці.

2.3.2. Реалізація ігрового агента

Робота ігрового агента та налаштування нейронної мережі описані у класі SnakeAgent з модуля snakeAgent.py. Для ініціалізації нового агента необхідно передати наступні значення:

1) Epsilon – вірогідність вибору випадкової дії;
2) Gamma – коефіцієнт знецінювання;
3) Alpha – темп навчання;
4) Weights_file – шлях до файлу HDF5 з вагами моделі, збереженого за допомогою save_weights.

Для генерації значення поточного стану є функція get_state, що приймає на вхід екземпляр класу Game, Player та Food. State зберігається у бінарній формі і включає такі значення:

1) Чи є перепона попереду (усі напрямки обраховуються по відношенню до поточного повороту голови);

2) Чи є перепона зліва;

3) Чи є перепона справа;

4) Чи змійка рухається ліворуч;

5) Чи змійка рухається праворуч;

6) Чи змійка рухається вгору;

7) Чи змійка рухається вниз;

8) Чи їжа знаходиться ліворуч (по відношенню до положення голови на карті);

9) Чи їжа знаходиться праворуч;

10) Чи їжа знаходиться згори;

11) Чи їжа знаходиться знизу;

12) Чи змійка наблизилась до їжі.

Для обрахунку винагороди застосовується функція get_reward, що приймає на вхід екземпляр гравця та crash (true, якщо гра закінчилась, інакше false). Винагорода обраховується наступним чином:

1) Якщо гра закінчилась: -10;

2) Якщо змійка з’їла яблуко: 10;

3) Якщо змійка наблизилась до яблука: 0.1;

4) Якщо змійка не наблизилась до яблука: -0.1;

Налаштування нейронної мережі здійснюється за допомогою Keras API. Вона складається з трьох прихованих шарів по 120 нейронів та трьох шарів виключення (dropout) між ними для оптимізації генералізації та запобігання перенавчання. Останній шар використовує Softmax функцію для отримання оцінки трьох можливих дій (прямо, ліворуч, праворуч). Якщо шлях до файлу з вагами вказано, вони завантажуються у мережу.

Для початку тренування необхідно викликати функцію restart, що бере випадкову частину зі збережених досвідів та тренує мережу на них перед новою

грою. Для продовження тренування є метод train, що приймає на вхід попередній стан, новий стан, дію, винагороду та чи гра закінчена.

2.3.3. Запуск програми

Для роботи з ігровим агентом розроблено скрипт snakeAI. У першу чергу він зчитує налаштування з файлу config.json та записує параметри у відповідні змінні. Ці значення використовуються при створенні нових екземплярів ігор та налаштуванні SnakeAgent:

1) field_rows – висота ігрового поля;

2) field_cols – ширина ігрового поля;

3) size_snake – розмір клітинок;

4) border_size – товщина границі ігрового поля;

5) speed – швидкість зміни кадрів;

6) max_steps – кількість кроків до смерті через голод;

7) weights_file – шлях до файлу, з якого зчитуватимуться ваги (якщо рядок порожній – файл не завантажуватиметься);

8) save_file – шлях до файлу, у який наприкінці виконання будуть збережені ваги нейронної мережі;

9) games_number – кількість ігор для тренування агента;

10) epsilon – вірогідність вибору випадкової дії;

11) delta_epsilon – на скільки зменшується epsilon після кожної гри;

12) alpha – темп навчання;

13) gamma – коефіцієнт знецінювання;

Робота програми відбуваєтсья таким чином, що спочатку ініціалізується новий ігровий агент, після чого відбувається стільки епізодів навчання, скільки вказано у games_number. У кожному епізоді створюється нова гра, після чого в циклі виконується метод make_train_step (або make_step, якщо train = false), доки

гра не закінчиться. Записується новий рахунок, результат відображається у консолі та додається у масив рахунків, після чого починається новий епізод. Наприкінці роботи відображається результат у вигляді точкової діаграми, після чого зберігаються ваги нейронної мережі (якщо train = true).

2.4. Результати роботи агента

Для аналізу розробленого агента було проведено ряд тестів для демонстрації ефективності його роботи за різних показників коефіцієнту знецінювання, темпу навчання, розподілу винагород та кількості випробувань. Стандартні налаштування для тренування агента встановлено наступним чином: кількість ігор 200, розмір ігрового поля 20 клітинок у ширину та 20 клітинок у висоту, коефіцієнт знецінювання 0.9, темп навчання 0.0005, вірогідність вибору випадкової дії 0.8.

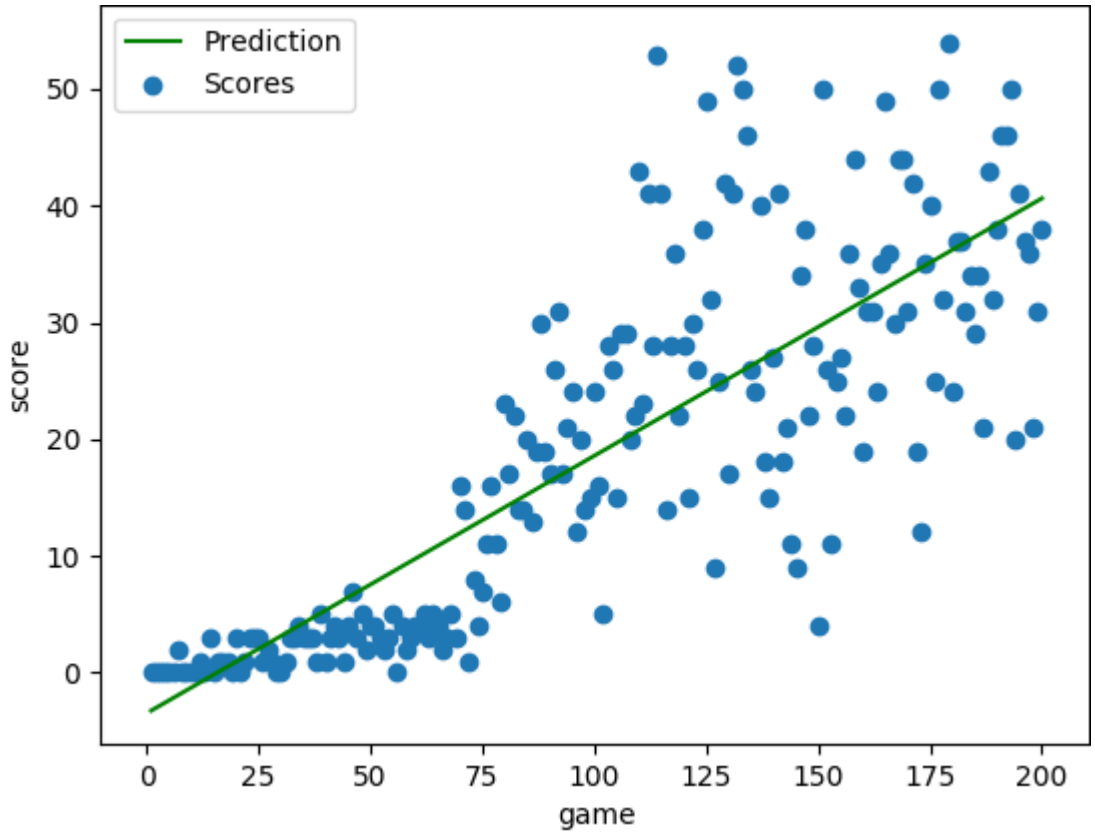

Рисунок 2.2 – Графік результатів роботи агента при 200 іграх

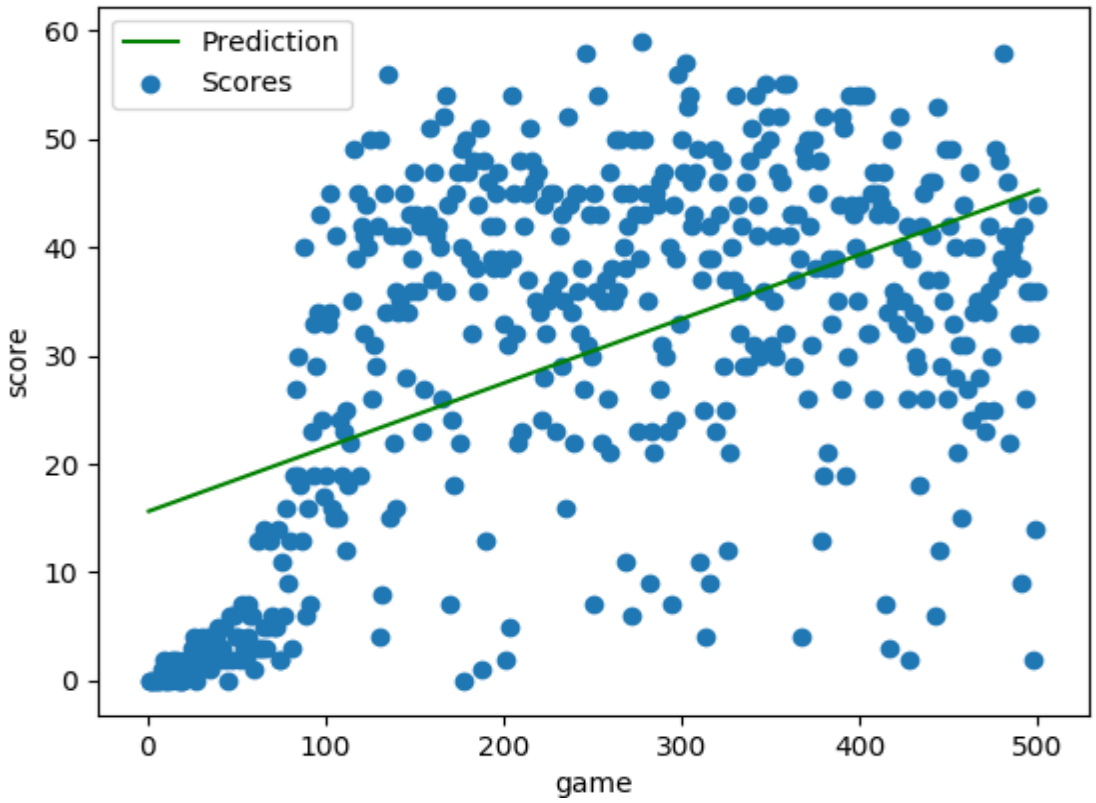

Рисунок 2.3 – Графік результатів роботи агента при 500 іграх

У першу черго варто продемонструвати, як впливає кількість випробувань на ефективність роботи агента, а саме – 200 ігор (рисунок 2.2) та 500 ігор (рисунок 2.3). У обох випадках коефіцієнт знецінювання $\gamma = 0.9$, темп навчання $\propto = 0.0005$, а вірогідність вибору випадкової дії $\varepsilon = 0.8$. Як бачимо, на обох графіках, спочатку приблизно перші 75 ігор агент вивчає навколишнє середовище і дуже часто обирає випадкові дії, через що змійка рухається дуже хаотично та часто помирає. Але після кожної гри вірогідність випадкової дії зменшується на 0.01, а коли вона досягає 0, агент спирається лише на власні передбачення. Після 120 ігор агент вже непогано орієнтується в ігровому полі та може досягти непоганих рекордів, зібравши більше 50 яблук. Після 200 ігор результати змінюються дуже мало, що робить подальше навчання неефективним.

А тепер розглянемо, як відрізняються результати роботи агента при різних коефіцієнтах знецінювання: 0.5, 0.8, 0.9, 0.99:

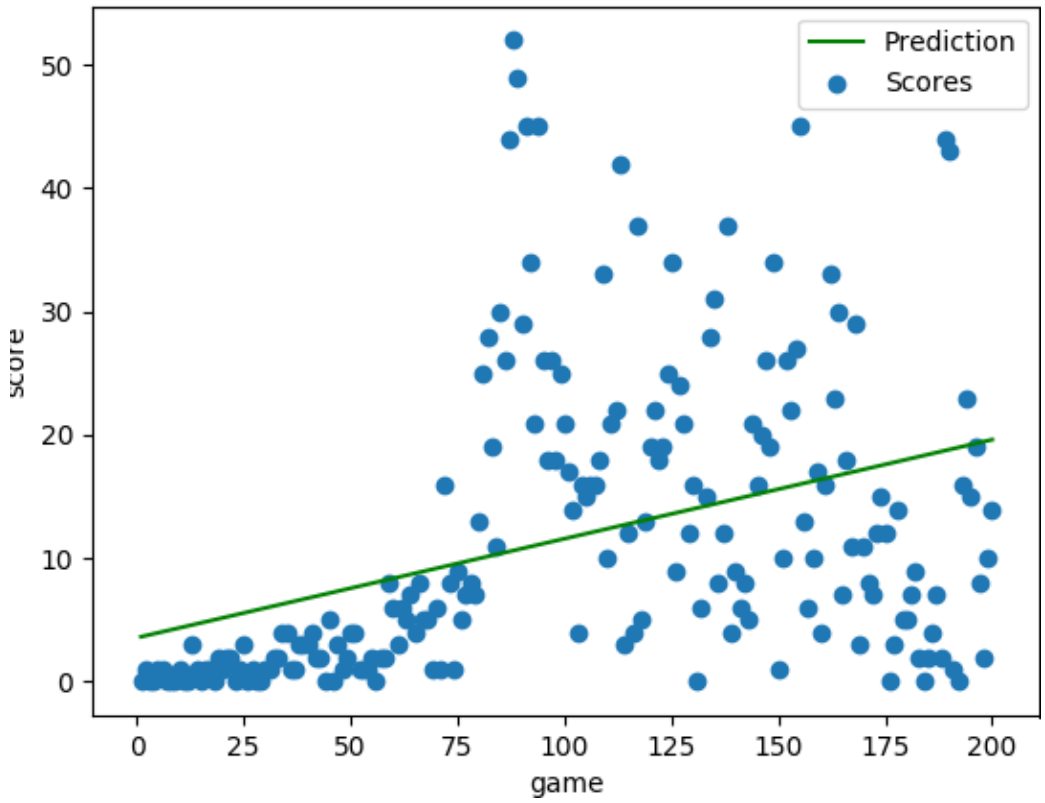

Рисунок 2.4 – Графік результатів роботи агента при $\gamma = 0.5$

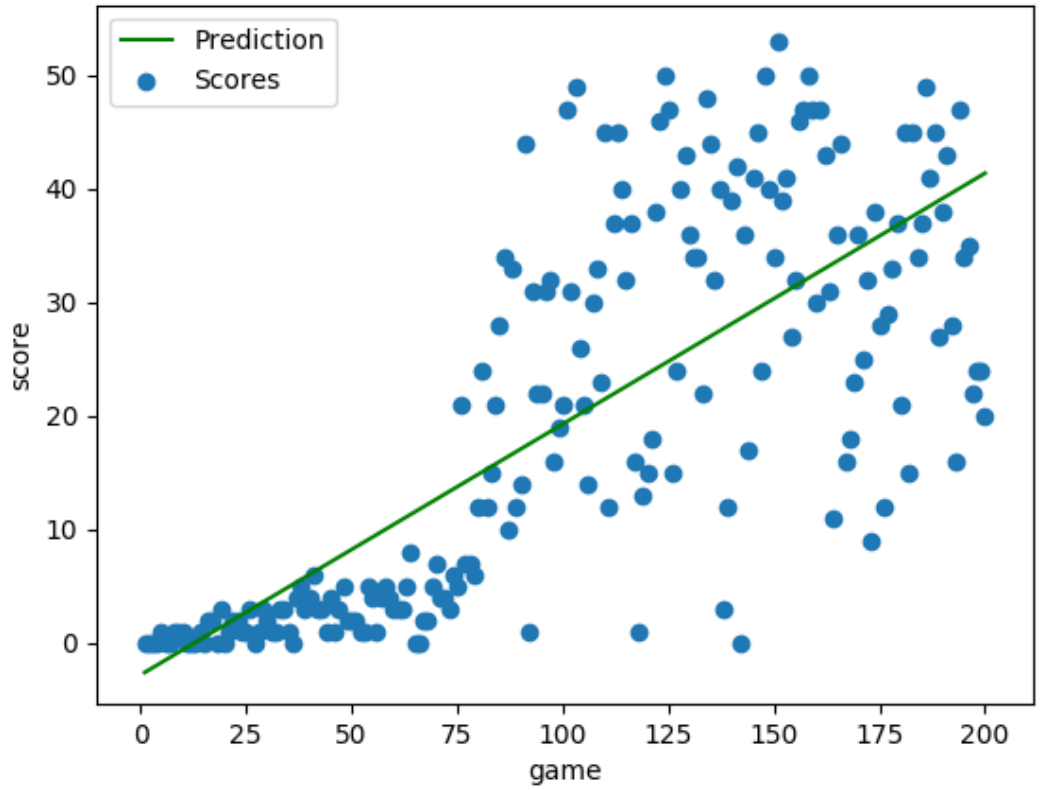

Рисунок 2.5 – Графік результатів роботи агента при $\gamma = 0.8$

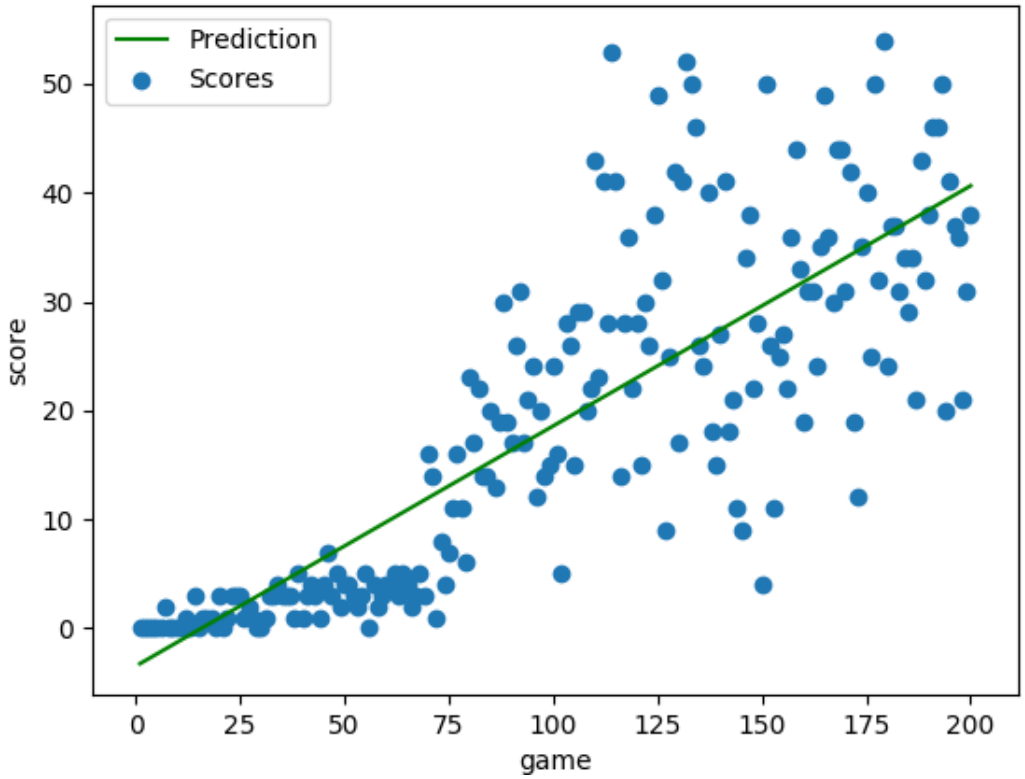

Рисунок 2.6 – Графік результатів роботи агента при $\gamma = 0.9$

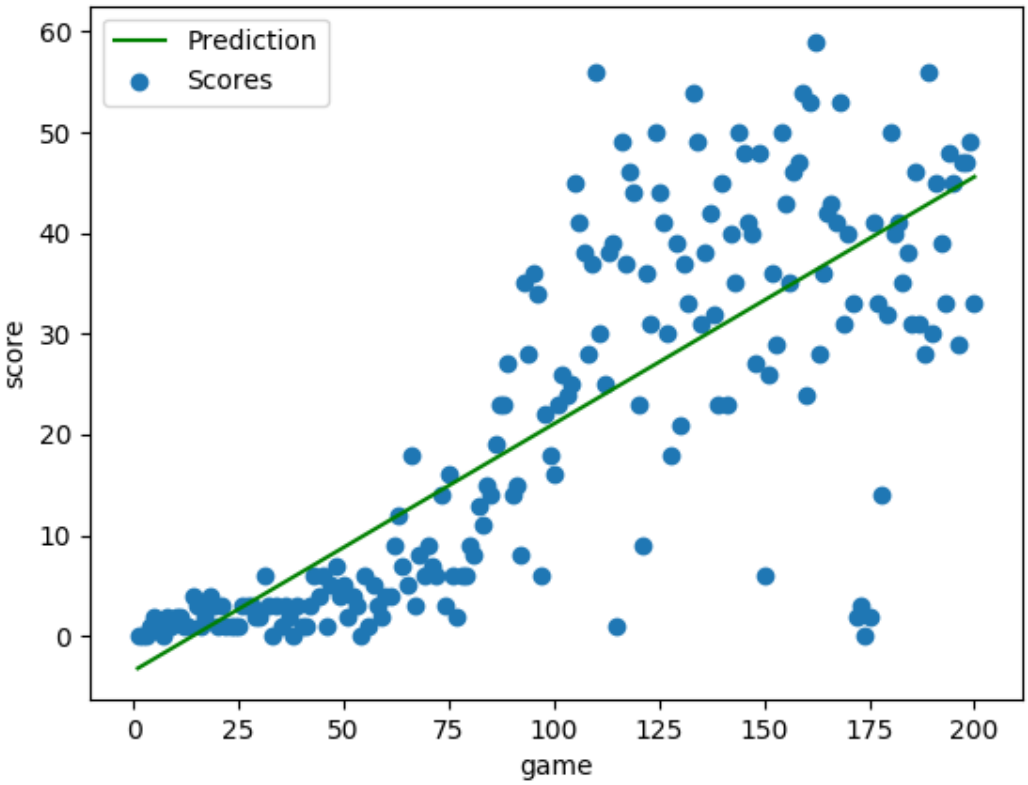

Рисунок 2.7 – Графік результатів роботи агента при $\gamma = 0.99$

1) Одразу варто зазначити, що поведінка змійки при значенні $\gamma = 0.5$ (рисунок 2.4) значно відрізняється від наступних варіантів. Тренування відбувається дуже стрімко (вже через 80 ігор агент досяг рекорду, набравши 54 очка), однак досить швидко ефективність роботи агента погіршується з часом. Спричинено це тим, що він стає “недальновидним” та часто помирає від стінок ігрового поля;

2) При збільшенні γ до 0.8 (рисунок 2.5), ситуація значно покращується, але разом з тим розподіл результатів стає досить великим, через що роботу агента важко назвати стабільною. Час від часу він приймає погані рішення, що призводять до швидкої смерті, однак швидкість тренування на початковому етапі є досить непоганою;

3) Збільшивши коефіцієнт знецінювання до 0.9 (рисунок 2.6), бачимо, що розподіл результатів дещо покращився, змійка набагато рідше здійснює жахливі рішення, але етап дослідження став трохи довшим;

4) Коли значення γ збільшилось до 0.99 (рисунок 2.7), результат став ще кращим. Пройшовши початковий етап тренування, агент досить стабільно набирає більше 30 очок і загальний рекорд сягає 60 очок;

Таким чином можна зробити висновок, що коли значення коефіцієнта знецінювання наближається до 1, стабільність та ефективність роботи агента покращується.

Висновки

В результаті проведеної роботи за допомогою розробленого ігрового агента для гри “Змійка” було продемонстровано принцип роботи навчання з підкріпленням, зокрема, Deep Q-Learning.

Розроблена програма для тренування ігрового агента вийшла досить гнучкою завдяки можливості змінити конфігурацію запуску. Це дозволило провести ряд досліджень для перевірки впливу різних параметрів на ефективність тренування. Програмний застосунок реалізує можливість роботи агента як з відображенням самої гри зі встановленою швидкістю зміни кадрів, що дозволяє наочно побачити поведінку агента, так і без візуальної частини, що пришвидшує виконання.

У розробленої системи є деякі перспективи для покращення, наприклад, передавати у якості стану не лише інформацію про наявність небезпеки поблизу, а великої частини ігрового поля навколо голови. Такий підхід може дати кращий результат, однак тренування триватиме набагато довше і вимагатиме більше ресурсів. Також варто зазначити, що створення ідеального ігрового агента для гри “Змійка” (під ідеальним мається на увазі такий агент, що змійка завжди зможе зібрати всі яблука і заповнити своїм тілом все поле) є складною задачею, адже у гри є елемент випадковості, і наступне яблуко може з’явитися у глухому куті, звідки змійка не зможе вийти. Можливо, одним із способів досягти ідеального результату є примусити змійку рухатися виключно по Гамільтоновому графу, кожна вершина якого – клітинка ігрового поля.

Список використаних джерел

1. Навчання з підкріпленням [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Навчання_з_підкріпленням

2. Обучение с подкреплением [Електронний ресурс] – Режим доступу: http://www.machinelearning.ru/wiki/index.php?title=Обучение_с_подкреплением

3. Q-навчання [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Q-навчання

4. Deep Q Learning [Електронний ресурс] – Режим доступу: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/deep_q_learning.html

5. A Theoretical Analysis of Deep Q-Learning [Електронний ресурс] / Jianqing Fan, Zhaoran Wang, Yuchen Xie, Zhuran Yang – Режим доступу: https://arxiv.org/pdf/1901.00137.pdf

Додаток А

Код для запуску

```
def play_train_game(agent):
        game = Game(field_cols, field_rows, size_snake, border_size,
                border_color, background_color, food_color, snake_color,
                speed, max_steps)
        game.make_train_step(agent,[1, 0, 0])
        agent.restart()
        while not game.game_over:
                game.make_train_step(agent)
        return game.score


def play_game(agent):
        game = Game(field_cols, field_rows, size_snake, border_size,
                border_color, background_color, food_color, snake_color,
                speed, max_steps)
        while not game.game_over:
                game.make_step(agent)
        return game.score


pygame.init()
agent = SnakeAgent(epsilon, gamma, alpha, weights_file)
array_score = []
array_num = []
for game_num in range(games_number):
        pygame.display.set_caption('Smart Snake #' + str(game_num+1))
        if train:
                score = play_train_game(agent)
```

```
            agent.epsilon -= delta_epsilon
        else:
            score = play_game(agent)
        game_num += 1
        print('#', game_num, '\tScore -', score)
        array_score.append(score)
        array_num.append(game_num)
show_result(array_num, array_score)
if train:
        agent.model.save_weights(save_file)
```

Додаток Б

Функції для взаємодії з агентом

```
def get_state(self, game):
        a = game.snake.body[-1][0]-game.food.x
        b = game.snake.body[-1][1]-game.food.y
        self.old_to_food = self.new_to_food
        self.new_to_food = math.sqrt(a*a+b*b)
        dangers = self.__get_dir_dangers(game,game.snake)
        state = [
                dangers[0],
                dangers[1],
                dangers[2],
                game.snake.x_speed == -1,
                game.snake.x_speed == 1,
                game.snake.y_speed == -1,
                game.snake.y_speed == 1,
                game.food.x < game.snake.x,
                game.food.x > game.snake.x,
                game.food.y < game.snake.y,
                game.food.y > game.snake.y,
                self.new_to_food < self.old_to_food
                ]
        real_state = [1 if s else 0 for s in state]
        return np.array(real_state)


def get_reward(self,game):
        reward = 0
        if game.game_over:
```

```
                reward = -20
                return reward
        if game.snake.must_grow:
                reward = 10
                return reward
        if self.new_to_food < self.old_to_food:
                reward = 0.2
        elif self.new_to_food > self.old_to_food:
                reward = -0.1
        return reward

def __get_target_reward(self, reward, state, done):
        if done:
                return reward
        else:
                return reward + self.gamma * np.amax(self.get_prediction(state)[0])

def restart(self):
        exp_part = random.sample(self.experiences, min(len (self.experiences),
                self.max_experiences))
        for exp in exp_part:
                self.__train_target(exp)

def train(self, state_old, state_new, action, reward, done):
        exp = Experience(state_old, state_new, action, reward, done)
        self.__train_target(exp)
        self.experiences.append(exp)
```

```
def __train_target(self, exp):
        target = self.get_prediction(exp.state_old)
        target[0][np.argmax(exp.action)] = self.__get_target_reward(exp.reward,
                exp.state_new, exp.done)
        self.model.fit(self.__shape_state(exp.state_old), target, epochs=1, verbose=0)
```