НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__”_____2021 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “ Комп’ютерна інженерія”

спеціальності 123 “Комп’ютерна інженерія”

на тему: «Програмна система глибокого навчання з підкріпленням для гри Geometry Dash»

Виконав: студент __4__ курсу, групи _ІО-71_
(шифр групи)
_____Тутевич Віталій Євгенійович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____д.т.н. проф. Новотарський М.А._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) д.т.н. проф. Сімоненко В.П.___ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2021 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

Спеціальність 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТІРЕНКО
(підпис)

“__”_____2021 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Тутевича Віталія Євгенійовича

1. Тема проєкту *Програмна система глибокого навчання з підкріпленням для гри Geometry Dash*
керівник проєкту д.т.н. проф. Новотарський Михайло Анатолійович ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 квітня 2021 року № 1180-с
2. Термін здачі студентом закінченого проєкту 01 червня *2021 р.*
3. Вихідні дані до проєкту *технічна документація, кадри зі гри “Geometry Dash”.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Опис та аналіз предметної області, розбір теоретичних основ даної області, аналіз сучасних архітектур нейронних мереж, розробка фінальної архітектури.
5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень)
схема формування оцінки функції якості дії, діаграма послідовності кроку тренування, схема алгоритму тренування

6. Консультанти проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	проф. д.т.н. Сімоненко В. П.		
Розділ 1	проф. д.т.н. Новотарський М.А.		
Розділ 2	проф. д.т.н. Новотарський М.А.		
Розділ 3	проф. д.т.н. Новотарський М.А.		
Розділ 4	проф. д.т.н. Новотарський М.А.		
Розділ 5	проф. д.т.н. Новотарський М.А.		

7. Дата видачі завдання 01.09.2020

Календарний план

№ з/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	01.09.2020-10.09.2020	
2.	*Вивчення та аналіз завдання*	10.09.2020-10.10.2020	
3.	*Розробка структури інтерфейсу взаємодії*	10.10.2020-15.11.2020	
4.	*Розробка структури моделі навчання*	15.11.2020-09.01.2021	
5.	*Програмна реалізація системи*	09.01.2021-17.04.2021	
6.	*Оформлення пояснювальної записки*	17.04.2021-28.05.2021	
7.	*Захист програмного продукту*		
8.	*Передзахист*		
9.	*Захист*		

Студент-дипломник _____
(підпис)

Керівник проєкту _____
(підпис)

Анотація

В цьому дипломному проєкті було розроблено програмну систему для гри в існуючу комп'ютерну гру "Geometry Dash" з використанням сучасних архітектур нейронних мереж та алгоритмів штучного інтелекту.

Модель дозволяє вивчити будь-який рівень у грі та проходити його на рівні, порівняному з можливостями людини. Даний проєкт дає можливість оцінити сучасні методи штучного інтелекту для прийняття рішень у ігровому середовищі. Модель була побудована за допомогою мови програмування Python та фреймоврку TensorFlow 2.4.1.

Annotation

This diploma project is devoted to creating a software system to play an existing computer game "Geometry Dash" using cutting-edge neural network architectures and artificial intelligence algorithms.

The model is able to learn any level in the game and manages to play it on a level, comparable to human capabilities. Current project allows us to estimate modern artificial intelligence methods in decision-making in a game environment. The model was built using Python programming language and TensorFlow 2.4.1 framework.

Опис альбому
до дипломного проєкту

на тему: «Програмна система глибокого навчання з підкріпленням для гри Geometry Dash»

Київ – 2021 року

	Формат	Позначення	Найменування	Кільк. аркушів	№ екзем.	Примітка
1			Документація загальна			
			Знову розроблена			
2	А4	ІАЛЦ.467100.001 ТЗ	Технічне завдання	4		
3	А4	ІАЛЦ.467100.002 ПЗ	Пояснювальна записка	84		
4	А4	ІАЛЦ.467100.003 Д1	Структурна схема:	1		
			Схема формування оцінки			
			функції якості дії			
5	А4	ІАЛЦ.467100.004 Д2	Функціональна схема:	1		
			Діаграма послідовності			
			кроку тренування			
6	А4	ІАЛЦ.467100.005 Д3	Принципова схема:	1		
			Схема алгоритму			
			тренування			
7	А4	ІАЛЦ.467100.006 Д4	Лістинг програми	15		

Змн.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467100.001 ТЗ		
Розроб.		Тутевич В.Є.			Програмна система глибокого навчання з підкріпленням для гри Geometry Dash Опис Альбому	Літ.	Арк. 1 / Аркушів 1
Перевір.		Новотарський М.А.					
Н. Контр.		Сімоненко В.П.				НТУУ “КПІ” ФІОТ ІО-71	
Затверд.		Стіренко С.Г.					

Технічне завдання
до дипломного проєкту

на тему: «Програмна система глибокого навчання з підкріпленням для гри Geometry Dash»

Київ – 2021 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Це технічне завдання поширюється на розробку програмної системи для комп’ютерної гри “Geometry Dash”. Область застосування: тестування сучасних архітектур нейронних мереж, а також алгоритмів штучного інтелекту для прийняття рішень у ігровому середовищі.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського проєкту професійної програми “Комп'ютерні системи та мережі” спеціальності 123 “Комп'ютерна інженерія”, затверджене кафедрою Обчислювальної техніки Національного технічного Університету України “Київський Політехнічний інститут імені Ігоря Сікорського”.

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою цього дипломного проєкту є розробка програмної системи, що зможе проходити рівні у комп’ютерній грі “Geometry Dash”, використовуючи сучасні алгоритми машинного навчання, а також пошук способів підвищення їх ефективності у заданому ігровому середовищі.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, публікації в спеціалізованих періодичних виданнях, технічна документація, публікації в Інтернеті на цю тему.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Висока ефективність використовуваної інформації під час процесу навчання.
- Прийняття адекватних рішень у кожній можливій ситуації віртуального середовища.
- Адекватна швидкість прийняття рішень та навчання.

5.2 Вимоги до програмного забезпечення

- Операційна система Linux
- Python 3.8
- CUDA 11.0 та вище
- cuDNN 8.0 та вище
- TensorFlow 2.4.0 та вище

5.3 Вимоги до програмного забезпечення

- Комп’ютер на базі Intel Pentium 4 і вище
- Наявність мульти-ядерного GPU
- Обсяг оперативної пам’яті не менше 8 Гбайт
- Вільний простір твердотільного накопичувача не менше 20 Гбайт

6. ЕТАПИ РОЗРОБКИ

	Дата
6.1 Вивчення необхідної літератури	10.09.2020
6.2 Складання і узгодження технічного завдання	10.10.2020
6.3 Аналіз механізмів взаємодії із грою	17.10.2020
6.4 Аналіз сучасних архітектур нейронних мереж	15.11.2020
6.5 Написання програмної частини	09.01.2021
6.6 Тестування та виправлення помилок	05.04.2021
6.7 Оформлення документації дипломного проєкту	17.04.2021

Пояснювальна записка
до дипломного проєкту

на тему: «Програмна система глибокого навчання з підкріпленням для гри Geometry Dash»

Київ – 2021 року

ЗМІСТ

ВСТУП

Сфера штучного інтелекту активно розвивається у всіх напрямках сучасної науки та активно застосовується як у комерційних проєктах, так і для дослідження інших наукових областей. Завдяки дослідженням таких компаній як OpenAI та DeepMind, використання штучного інтелекту у області прийняття самостійних рішень отримало велику популярність. Такі системи як OpenAI Five та DeepMind AlphaGo довели спроможність штучного інтелекту до підтримки інкрементального процесу прийняття рішень у складних ігрових середовищах. Розробка таких систем стала можливою за допомогою новітнього підходу, що має назву навчання з підкріпленням.

В даній дипломній роботі протестовано можливості комбінації алгоритмів навчання з підкріпленням та глибокого навчання до вивчення оптимальної поведінки у ігровому середовищі на прикладі комп’ютерної гри “Geometry Dash”. Також проведене тестування та додатковий пошук способів пришвидшення процесу навчання з метою підвищення ефективності системи.

У першому розділі проведено аналіз сучасних алгоритмів навчання з підкріпленням, а також їх математичних та алгоритмічних основ. Визначено декілька основних методів навчання, а саме: методи Монте-Карло, TD-Навчання та Динамічного Програмування. В результаті аналізу сформований алгоритм навчання, що поєднує усі переваги попередніх методів — n-крокове Q-Навчання.

У другому розділі висвітлена проблема середовищ, що мають кількість станів що значно перевищують норму. Для вирішення даної задачі введений механізм апроксимації функцій з використанням нейронних мереж. Проаналізовано ефективність нейронних мереж прямого розповсюдження у обробці зображень та введена необхідність використання згорткових нейронних мереж. В результаті аналізу сформована оптимальна архітектура нейронної мережі агента навчання.

У третьому розділі проведене проектування та розробка інтерфейсу взаємодії. Даний інтерфейс дозволяє отримувати інформацію щодо поточного

стану середовища та формує відповідні нагороди за перебування в них, базуючись на відстежених термінальних станах. Окрім отримання інформації інтерфейс виконує функцію емуляції обраних агентом дій у середовищі.

У четвертому розділі виконане проектування конкретної архітектури нейронної мережі моделі-агента, а також процесу її навчання. Окрім архітектури розроблений блок пам’яті агента, що дозволяє підвищити ефективність отриманої від середовища інформації. Архітектура агента та процес його навчання розроблений за допомогою модуля Tensorflow.

У п’ятому розділі проведене тестування розроблених елементів системи. Спочатку протестована можливості інтерфейсу взаємодії до передачі відповідної до поточного стану інформації, а також механізми емуляції. Також перевірений механізм наповнення пам’яті агента та сам процес його тренування. В результаті тестування отримано оптимальні параметри навчання.

РОЗДІЛ 1
АНАЛІЗ НАВЧАННЯ З ПІДКРІПЛЕННЯМ ДЛЯ ЗНАХОДЖЕННЯ ОПТИМАЛЬНОЇ ПОВЕДІНКИ У ЗАДАНОМУ СЕРЕДОВИЩІ

1.1 Проблема навчання. Машинне навчання (Machine/Deep Learning)

З моменту винайдення перших комп'ютерів, людство цікавило питання: чи можуть машини приймати рішення самостійно, без програмування їх на вибір таких рішень напряму? Наприклад чи можливо створити машину, що буде самостійно навчатись на медичних записах пацієнтів щоб знаходити оптимальні ліки для ще не відкритих захворювань[1, 2], чи аналізувати вхідний текст для створення реальних об'єктів у двовимірному чи тривимірному просторі[3, 4], та багато інших застосувань. Зараз усі ці ідеї вже не здаються чимось нереальним, адже з винайденням області Машинного навчання багато з них вже були втілені у реальність. Так існує і є досить популярною технологія розпізнавання людської мови (speech recognition), що використовується в Google Assistant[5], технології виявлення об'єктів (object detection) що імплементовані у всіх сучасних безпілотних автомобілях[6, 7], а також технології розпізнавання облич[8] та ще багато інших спеціалізованих інструментів.

Однак що означає термін навчання у контексті обчислювальних машин? Формально даний процес можна задати наступним чином: система навчається з певного досвіду відповідно до певного класу завдання та показнику її ефективності, якщо даний показник у задачах з даного класу покращується зі збільшенням досвіду. Так ефективність системи, що вчиться грати в шахи[9] може бути виміряна процентом її перемог у грі з реальними людьми. При цьому її тренувальний досвід може бути побудований як на існуючих записаних партіях, так і методом її гри самої з собою. Схожим чином можна описати завдання навчання для певного класифікатора зображень, ефективність якого буде залежати від проценту коректно класифікованих об'єктів, а тренувальний досвід буде складатись з певної кількості зображень та їх класів (маркерів).

Також важливо формально описати ціль навчання, тобто задати певну функцію, що система має вивчити для досягнення своєї мети. На прикладі певного класифікатора зображень, що має ділити вхідні зображення на 2 класи, завданням системи може бути апроксимація певної функції $\hat{Y}$ такої, що:

$$G = \begin{cases} Об'єкт\ належить\ до\ класу\ 0, якщо\ \hat{Y} > 0.5 \\ До\ класу\ 1\ в\ іншому\ разі\ (\hat{Y} \le 0.5) \end{cases}, \quad (1.1)$$

де $\hat{Y}$ – певна функція, що відділяє класи 1 та 0;

G – вихідний клас зображення.

Окрім визначення даної функції також необхідно задати її певне представлення, тобто те, від чого вона буде залежати. У проблемі системи, що навчається грати в шахи, ключову роль у представленні такої функції грали б наявність певних фігур на дошці, а також їх позиції. Таким чином функцію $\hat{Y}$ можна задати за допомогою рівняння регресії:

$$\hat{Y} = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + ... + \omega_n x_n \ , \quad (1.2)$$

де x_1, x_2, ..., x_n — певні ознаки (позиції фігур, наявність певних об'єктів на зображенні тощо), що використовуються системою для апроксимації;

ω_1, ω_2, ..., ω_n — відповідні ваги регресії, що задають важливість ознак.

Таким чином, маючи певний тренувальний набір даних, що містить бажаний набір певних ознак, а також бажаний результат, використовуючи певне представлення бажаної функції, таке як (1.1, 1.2) процес навчання системи формально можна описати наступним чином:

$$E = \sum_{i=0}^{N} (Y_i - \hat{Y}(x_i))^2 \ , \quad (1.3)$$

де x_i — вхідний набір представлень;

Y_i — бажаний результат для набору представлень x_i;

N — довжина тренувального набору даних.

Дана рівність задає помилку E на наборі тренувальних даних, і саме мінімізація даної похибки приводить до підвищення ефективності системи. Даний вид проблем машинного навчання відноситься до класу Supervised,

тобто такого, в якому початково заданий бажаний результат системи. Існують також завдання, що відносяться до протилежного класу — Unsupervised. Їх ціль полягає у знаходженні певних закономірностей у даних не базуючись на жодних попередніх представленнях про них, за допомогою певних алгоритмів кластеризації та групування. Разом ці два класи описують майже повний спектр алгоритмів машинного навчання. Також важливою є галузь машинного навчання, що базується на вивченні формування певних представлень для покращення процесу навчання. Даною областю є глибинне навчання (Deep Learning), вона входить у множину алгоритмів машинного навчання і представляє собою набір алгоритмів та методів для формування ієрархічних концептів з вхідних даних[10, 11] та їх використання для навчання певної системи. Важливим елементом цієї області є нейронні мережі, адже саме завдяки різним їх архітектурам стало набагато легше використовувати машинне навчання в таких проблемах, де формування представлень вручну є трудомістким або не очевидним процесом[12]. Більш детально про архітектуру нейронних мереж на прикладі нейронних мереж прямого розповсюдження та згорткових нейронних мереж наведено у розділі 2.

1.2 Навчання з підкріпленням (Reinforcement Learning)

Даний підхід представляє собою інший погляд на проблему навчання. На відміну від біль популярних та використовуваних підходів, таких як навчання зі вчителем (Supervised) та навчання без вчителя (Unsupervised), основна ідея навчання з підкріпленням полягає у постійній взаємодії моделі з середовищем. Така модель, що навчається діяти найбільш оптимальним способом в даному конкретному середовищі називається агентом (Agent). На відміну від Supervised Learning, агент не має інформації про правильну дію чи вибір в будь-якій ситуації. Натомість, він в кожен момент часу взаємодіє з середовищем, отримуючи при цьому певну нагороду (Reward). Керуючись цим механізмом, основна задача агента полягає в знаходженні дій що

призводять до отримання найбільшої нагороди, що також відрізняє цей підхід від Unsupervised Learning. Більше того, дії можуть призводити не тільки до отримання нагород, а й до зміни ситуації в середовищі, що в свою чергу може призвести до зміни усіх наступних можливих нагород. Таким чином, пошук оптимальної поведінки за допомогою проб та помилок та ідея відкладених в часі нагород є двома основними особливостями навчання з підкріпленням.

Як вже було зазначено вище, в процесі навчання агент постійно взаємодіє з певним середовищем. Для формалізації даної взаємодії використовуються Марковські Процеси Рішень (Markov Decision Processes), за допомогою яких вона може бути представлена у вигляді простої послідовності (рис 1.1).

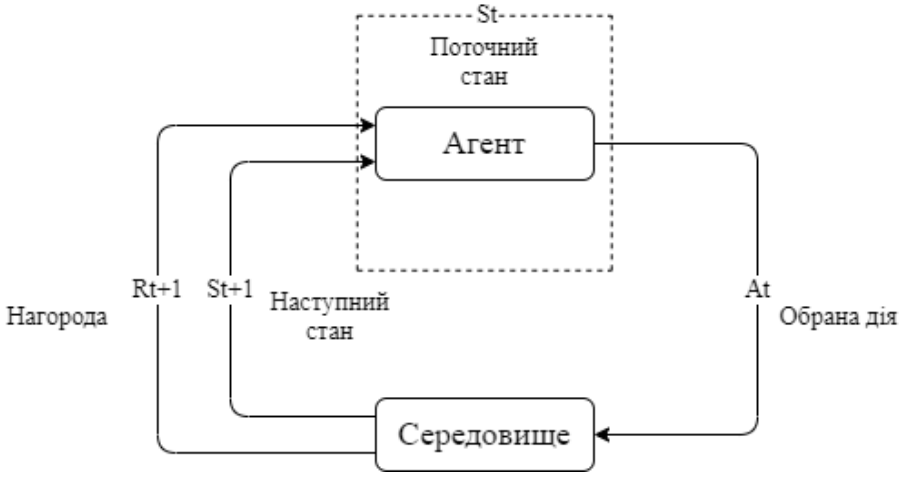


Рис. 1.1. Процес взаємодії агента та середовища в Марковських процесах рішень

Відповідно до (рис. 1.1), агент та середовище послідовно взаємодіють один з одним через певні дискретні проміжки часу. Наприклад, на початку навчання в момент часу t_0 середовище посилає певне представлення свого початкового стану S_0. Агент в свою чергу, знаходячись в цьому стані обирає певну дію A_0 та виконує її. В результаті, середовище віддає нагороду R_1 за виконану дію, а також передає агенту наступний стан S_1. Таким чином, взаємодія може відбуватись нескінченно, доки не буде досягнутий певний термінальний стан S_T. Однак для організації такої взаємодії агента з середовищем існує одне суттєве обмеження. Воно полягає в представленні

станів таким чином, щоб уся інформація про попередні дії агента та їх результати зберігались в його поточному стані. Тобто інформації про поточний стан S_t та дію A_t має бути достатньо для визначення наступного стану S_{t+1} або множини можливих наступних у випадку невизначеного середовища. Це також означає що для будь-якого стану S_t та дії A_t має виконуватись наступна рівність:

$$p(s_1, r_1|s_0, a_0) * p(s_2, r_2|s_1, a_1) * ... * p(s_{(t+1)}, r_{(t+1)}|s_t, a_t) = p(s_{(t+1)}, r_{(t+1)}|s_t, a_t) \quad ,(1.4)$$

де s_t — стан середовища у дискретний проміжок часу t;

a_t — дія, що виконується з поточного стану s_t;

s_{t+1} — наступний стан середовища, отриманий в результаті дії a_t;

r_t — нагорода, отримана в результаті дії a_t.

В такому разі стан S_t називається таким, що має Марковську властивість. Прикладом такого представлення станів є розміщення фігур у шахах, адже знаючи поточну позицію усіх фігур на дошці та наступний хід можна точно передбачити їх наступне розміщення.

Окрім формалізації процесу взаємодії агента з середовищем, для розв'язання проблеми навчання з підкріпленням потрібно також формально задати кінцеву ціль агента. Як вже було сказано вище, його основною задачею є пошук оптимальної дії для будь-якого стану середовища. Дана оптимальність задається за допомогою нагород, тобто агент намагається максимізувати свою загальну нагороду впродовж усієї взаємодії з середовищем. Таким чином він фокусується не на поточних нагородах, а на очікуваній нагороді в кінцевому рахунку. Дана величина може бути описана наступною формулою:

$$G_t = R_{(t+1)} + R_{(t+2)} + ... + R_{(T-1)} + R_T \quad , \qquad (1.5)$$

де R_{t+1} – нагорода за дію у проміжок часу t;

T – фінальний проміжок часу, під час якого взаємодія завершується;

G_t — очікувана нагорода, що може бути отримана агентом з моменту t під час взаємодії з середовищем.

Даний підхід до опису цілі агента може бути використаний коли уся його взаємодія з середовищем завжди описується деякою скінченною послідовністю кроків, що називається епізодом (Episode). Якщо ж взаємодія не може бути поділена на епізоди, то зручніше використовувати іншу формулу:

$$G_t = R_{(t+1)} + \gamma R_{(t+2)} + \gamma^2 R_{(t+3)} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{(t+k+1)} \quad , \qquad (1.6)$$

де γ — коефіцієнт дисконтування, що приймає значення від 0 до 1 включно.

Параметр γ відповідає за далекоглядність агента, тобто як сильно він враховує усі наступні нагороди. При значенні γ = 0 агент враховує лише поточні, а при γ = 1 — абсолютно всі нагороди. Також, дуже важливою є властивість формули (1.6), що може бути описана наступним чином:

$$\begin{aligned} G_t = R_{(t+1)} + \gamma R_{(t+2)} + \gamma^2 R_{(t+3)} + \ldots = R_{(t+1)} + \gamma * (R_{(t+2)} + \gamma R_{(t+3)} + \ldots) = \\ = R_{(t+1)} + \gamma * G_{(t+1)} \end{aligned} \quad , \qquad (1.7)$$

Ця формула дає можливість виразити загальну очікувану нагороду рекурсивним чином, що часто використовується в різних алгоритмах навчання.

Також, двома важливими аспектами навчання з підкріпленням є функції якості стану(value function) та політики(policy)[13]. Перша показує на скільки вигідно для агента знаходитись в певному стані. Дана вигода визначається за допомогою очікуваної нагороди G_t, що можливо отримати з даного стану S_t. Більш практичне значення має функція якості дії(action-value function), за допомогою якої можна визначити яку очікувану нагороду можна отримати, виконавши дію a із стану s. Однак обидві ці функції не використовуються без політики. Дана функція показує вірогідність вибору усіх дій з будь-якого стану s. Саме політика задає поведінку агента в середовищі, тому функції якості стану та дії повністю від неї залежні. В загальному випадку значення функцій якості стану та дії можна записати наступними формулами:

$$v_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{(t+k+1)} | S_t = s] \qquad (1.8)$$

$$q_\pi(s,a)=E_\pi[G_t|S_t=s,A_t=a]=E_\pi[\sum_{k=0}^{\infty}\gamma^k R_{(t+k+1)}|S_t=s,A_t=a]\ , \qquad (1.9)$$

де π — політика, функція вірогідності вибору усіх дій з будь-якого стану s;

$E_\pi[G_t | S_t = s, A_t = a]$ - математичне очікування нагороди, що може бути отримана в результаті вибору дії a зі стану s та слідуючи політиці π;

$v_\pi(s)$ — значення очікуваної нагороди, що може бути отримана зі стану s, слідуючи політиці π;

$q_\pi(s, a)$ — значення очікуваної нагороди, що може бути отримана зі стану s виконавши дію a, та слідуючи політиці π.

Таким чином обидві ці функції є математичними очікуваннями нагород, що слідують за перебуванням в стані s або за вибором дії a з даного стану. Іще одним важливим розвиненням формул (1.8) та (1.9) є рівняння Беллмана[14]:

$$\begin{aligned} v_\pi(s)&=E_\pi[G_t|S_t=s]=E_\pi[R_{(t+1)}+G_{(t+1)}|S_t=s]= \\ &=\sum_a \pi(a|s)\sum_{sn}\sum_r p(sn,r|s,a)*(r+\gamma E_\pi[G_{(t+1)}|S_{(t+1)}=sn])= \\ &=\sum_a \pi(a|s)\sum_{sn}\sum_r p(sn,r|s,a)*(r+\gamma v_\pi(sn)) \end{aligned} \qquad (1.10)$$

$$\begin{aligned} q_\pi(s,a)&=E_\pi[G_t|S_t=s,A_t=a]=E_\pi[R_{(t+1)}+G_{(t+1)}|S_t=s,A_t=a]= \\ &=\sum_{sn}\sum_r p(sn,r|s,a)*(r+\gamma E_\pi[G_{(t+1)}|S_{(t+1)}=sn])= \\ &=\sum_{sn}\sum_r p(sn,r|s,a)*(r+\gamma\sum_{an}\pi(an|sn)q_\pi(sn,an)) \end{aligned} \ , \qquad (1.11)$$

де sn – можливий наступний стан досяжний з поточного стану s;

an — дія що може бути обрана зі стану sn;

r – нагорода за перехід в новий стан sn;

p(sn, r | s, a) – вірогідність потрапити у стан sn та отримати нагороду r виконуючи дію a зі стану s;

π(a | s) – вірогідність вибору агентом дії a перебуваючи у стані s.

Воно в більш явній формі задає зв’язок між оцінкою функції якості даного стану s та оцінками усіх наступних станів sn. Проблема навчання з підкріпленням може бути вирішена у вигляді (1.10) або (1.11), однак використання даних формул має один серйозний недолік, а саме необхідність мати функцію вірогідності p(sn, r | s, a), що не завжди можливо. Такі методи

розв’язання часто використовуються в парі з методами Динамічного програмування (Dynamic Programming), але потребують значно більше обчислень для отримання оптимальної функції якості стану ніж методи, що базуються на досвіді агента. Через наявність великої кількості станів у реальних проблемах та необхідності мати функцію p, тобто точну модель середовища, для знаходження оцінок функції якості дії частіше використовується останній варіант.

1.3 Оптимальність агента. Загальний механізм навчання

В незалежності від вибору алгоритмів наближення або обчислення функції якості стану, ціль будь-якого алгоритму навчання полягає в знаходженні оптимальних дій агента в будь-якому стані середовища, тобто оптимальної політики. Щоб знайти найкращу політику потрібно задати певну ознаку порівняння. В даному випадку такою ознакою слугує функція якості стану, адже політика що вважається оптимальною повинна мати найбільшу очікувану нагороду в будь-якому стані середовища серед існуючих політик. Використовуючи значення цієї функції оптимальну політику можна задати наступним чином: політика π* називається оптимальною, якщо для будь-якого стану s вона має більше або рівне значення функції якості стану ніж будь-яка інша політика π. Оптимальні функції якості стану та дії можна записати наступними формулами:

$$v*(s)=max_{\pi} v_{\pi}(s) \quad (1.12)$$

$$q*(s,a)=max_{\pi} q_{\pi}(s,a) \ , \quad (1.13)$$

де max_{π} ... - максимальне значення величини з усіх можливих політик π;

v*(s) — функція якості стану для оптимальної політики π*;

q*(s, a) — функція якості дії для оптимальної політики π*.

Маючи ці дві функції можна легко отримати оптимальну політику, для цього достатньо лише з будь-якого стану вибирати дію, що призводить до стану з найбільшим значенням v*(s), або просто має найбільше значення функції

q*(s, a). Таким чином, будь яка політика, що є жадібною по відношенню до даних функцій, є оптимальною.

Знаючи як можна обчислити значення функцій v*(s) та q*(s, a) а також що будь-яка політика, що є жадібною по відношенню до значень цих функцій є оптимальною, можна звести процес навчання до двох послідовних етапів. Даний метод використовується в Марковських процесах рішень із скінченною множиною станів, однак ця ідея також є основою в апроксимуючих підходах. Нехай ми маємо певні скінченні множини станів S і дій A. Для кожного із станів в певній таблиці буде зберігатися значення функції v(s) (або q(s, a)). Задавши випадкове початкове значення для всіх станів (та дій) у таблиці, випадкову політику π, та використовуючи певний алгоритм обчислення цих функцій можна за допомогою ітеративного методу знайти їх наближені значення з точністю до певної константи. Таким чином для будь-якої політики π ми можемо знайти значення функцій v(s) та q(s, a). Тепер, як вже було сказано раніше, політика є оптимальною якщо вона для будь-якого стану має більше або рівне значення v(s) у порівняння з іншими політиками. Звідси слідує, що для того щоб знайти π* потрібно замінити стару політика на таку, що для кожного стану вибирає вибирає дію a, яка призводить до наступного стану sn з більшим або рівним значенням v(s). Ця ідея має назву Policy Improvement Theorem і є основною в теорії навчання з підкріпленням. Виконуючи послідовно ці два етапи можна отримати оптимальну політику π*. Загальний процес описується діаграмою, як на (рис. 1.2). Цю схему взаємодії процесів обчислення функції якості стану та покращення політики називають Generalized Policy Iteration[15]. За допомогою даної схеми можна описати більшість алгоритмів навчання з підкріпленням, адже майже кожен з них використовує явний процес обчислення або апроксимації функції якості стану та певну політику π. Процеси обчислення та покращення працюють в парі, адже покращення оцінки v(s) дозволяє знайти кращу політику на її основі, а покращення політики створює необхідність обчислення v(s) для всіх станів середовища.

Рис. 1.2. Процес знаходження оптимальних π* та v*(s)

Коли обидва ці процеси стабілізуються, тобто політика та функція якості стану не змінюються, тоді буде отримане оптимальне рішення.

1.4 Алгоритми навчання що базуються на досвіді. Q-Learning.

Головною метою навчання з підкріпленням є пошук оптимальних v*(s) та π*. Однак, як вже було зазначено в пункті 1.1, знаходження цих функцій за допомогою формул (1.10) та (1.11) не завжди можливе, адже для цього необхідно мати функцію вірогідності переходу агента у наступний стан sn з нагородою r – p(sn, r | s, a). Щоб обійти цю проблему та необхідність мати точну модель середовища використовуються методи навчання, що базуються на досвіді агента. Гарним прикладом таких методів є методи Монте Карло[16]. Їх суть полягає у використанні середнього значення суми нагород, що були отримані після перебування в стані s в якості оцінки функції v(s). Таким чином, при достатньо великій вибірці цих значень за правилом великих чисел їх середнє значення буде наближатись до математичного очікування, тобто до (1.8), (1.9). Це означає, що для того щоб отримати оптимальну політику π* нам достатньо лише взаємодіяти з середовищем, отримуючи нові зразки переходів від одного стану до іншого та нагороду за них.

Загальний алгоритм навчання з використанням методів Монте-Карло можна описати наступним чином. Насамперед, для більш зручного формування політики використовується функція q(s, a) замість v(s). Таким

чином для покращення політики достатньо буде обрати дію, що має найбільше значення цієї функції. Далі, для кожного стану s та дії a в певній таблиці буде зберігатись наближене значення функції q. Після завершення кожного епізоду в певному середовищі, будемо отримувати послідовність переходів від початкового стану до останнього (термінального). Йдучи в зворотному порядку цієї послідовності, для кожної пари s, a будемо оновлювати оцінку середнього значення q(s, a) та оновлювати політику так, щоб $\pi(s) = \text{argmax}_a\, q(s, a)$. В такому разі після скінченної кількості епізодів будуть досягнуті оптимальні політика та функція якості стану.

Таким чином існує два кардинально різних підходи для вивчення середовища: методи Динамічного програмування та методи Монте-Карло. При використанні першого варіанту присутня необхідність мати модель середовища, але усе навчання відбувається за формулами (1.10), (1.11), тобто використовуючи оцінки наступних можливих станів. З іншої сторони, для використання методів Монте-Карло необхідна лише достатня кількість зразків переходів та нагород для усіх необхідних для вивчення станів, але для визначення значності певної дії з кожного стану використовується лише середнє значення суми нагород, що можуть бути отримані після її виконання. Однак існує також метод що об'єднує обидва ці підходи. Даний метод називається TD-Навчання[17] (Temporal Difference Learning) і він дає можливість об'єднати навчання з використанням лише досвіду агента як в методах Монте-Карло, а також використання оцінок значень можливих наступних станів середовища, що використовуються в методах Динамічного програмування. Нову формулу для формування функція якості стану з формул (1.7) та (1.10) можна записати наступним чином:

$$v(S_t) = v(S_t) + \alpha(G_t - v(S_t)) = v(S_t) + \alpha(R_{(t+1)} + \gamma v(S_{(t+1)}) - v(S_t)) \quad , \qquad (1.14)$$

де α — параметр, що задає швидкість навчання.

Перша частина формули (1.14) показує процес вивчення функція якості стану за допомогою методів Монте-Карло, а друга частина — її модифікована версія, в якій замість акумульованої суми нагород G_t використовується її

оцінка $v(S_{t+1})$. Таке представлення функції якості стану дозволяє використовувати вже вивчені значення інших станів, а також пришвидшити процес навчання, адже тепер агенту не потрібно чекати завершення епізоду щоб оновити оцінку $v(S_t)$. Даний механізм використання оцінок наступних станів має ще одну додаткову перевагу. При виконанні умови на наявність у представленні станів Марковської властивості, методи TD-Навчання в неявному вигляді будують найбільш вірогідну модель Марківського процесу[18], що в свою чергу дозволяє створювати кращі оцінки станів, для яких було отримано недостатньо зразків. З огляду на ці переваги, для розв'язання проблеми навчання з підкріпленням більше підходить алгоритм TD-Навчання, описаний далі.

Для вивчення агентом заданого середовища частіше за все використовується алгоритм Q-Навчання(Q-Learning)[19]. Його суть полягає в наступному: як і в методах Монте-Карло для зручного формування політики використовується функція q(s, a) замість v(s). Як і минулого разу, для кожного стану s та дії a в певній таблиці буде зберігатись наближене значення цієї функції. Використовуючи досвід взаємодії агента з середовищем, формування оцінок функції q буде відбуватись за наступною формулою:

$$q(S_t, A_t) = q(S_t, A_t) + \alpha(R + \gamma max_a q(S_{(t+1)}, a) - q(S_t, A_t)) \quad , \qquad (1.15)$$

де $\alpha \in (0, 1]$ — параметр, що задає швидкість навчання.

Зміна ж політики залишається такою ж як і в методах Монте-Карло: $\pi(s) = \text{argmax}_a\ q(s, a)$. З формул (1.15) та (1.13) видно, що даний метод напряму апроксимує оптимальну функцію якості дії, що й робить його унікальним. Як і в випадку з методами Монте-Карло для скінченної кількості станів, після певного скінченного числа епізодів Q-Навчання досягне оптимальних $q^*(s, a)$ та π^*[20].

Як вже було розглянуто, серед алгоритмів, що базуються на досвіді агента, найбільш популярними є методи Монте-Карло та TD-Навчання. Обидва ці методи представляють собою граничні випадки іншого методу, що їх об'єднує. Цей підхід називається n-step Bootstrapping[21], і його основна

ідея полягає у введенні додаткового параметру n, що керував би кількістю кроків алгоритму перед використанням оцінки наступного стану середовища. Таким чином можливо комбінувати переваги обох алгоритмів: отримувати більше реальних нагород від середовища, а також не чекати кінця епізоду для початку навчання, а використовувати оцінку стану після n кроків як заміну решті послідовності нагород. Більш наочне порівняння різних параметрів n представлене на (Рис. 1.3):

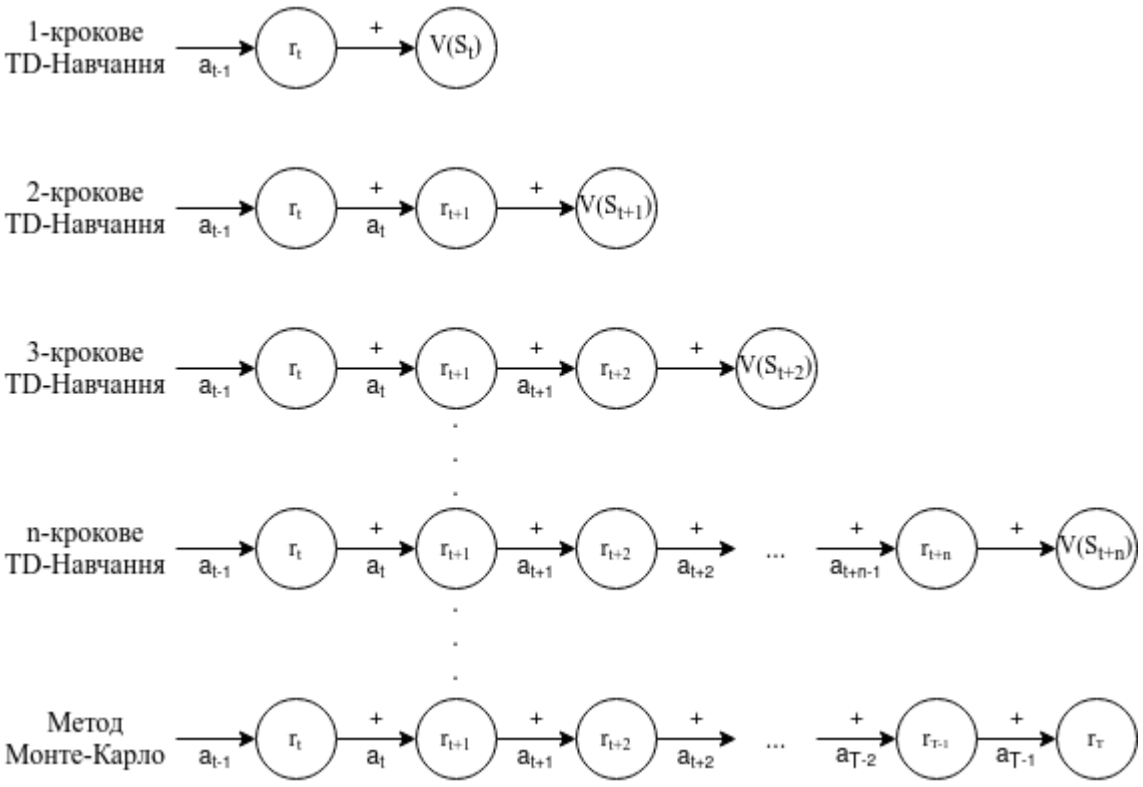


Рис. 1.3. Діаграма послідовності нагород, що використовується для оцінки поточного стану, в залежності від параметру n

Методи навчання, що використовують n кроків в середовищі до використання оцінки наступного стану називаються n-крокове TD-Навчання. Як вже було зазначено, такий підхід дозволяє комбінувати переваги і методу Монте-Карло і Temporal Difference Learning. При цьому алгоритм навчання практично не змінюється. Дійсно, нову формулу для формування функції якості стану можна записати з (1.14) наступним чином:

$$v(S_t) = v(S_t) + \alpha(G_{(t:t+n)} - v(S_t)) = \\ v(S_t) + \alpha(R_{(t+1)} + \gamma R_{(t+2)} + \ldots + \gamma^n v(S_{(t+n)}) - v(S_t)) \quad , \qquad (1.16)$$

де $G_{(t:t+n)}$ — дисконтована сума нагород, отримана агентом в дискретний проміжок часу з t по t + n;

S_{t+n} — стан середовища на дискретний момент часу t+n.

Однак при використанні даної формули необхідно заздалегідь мати нагороди до часу t+n включно, що не є можливим. Для подолання цієї проблеми навчання затримується на n кроків в середовищі щоб зібрати необхідні дані, а також затримується на ту саму кількість кроків при завершенні епізоду. Використовуючи (1.16) а також послідовність дій в методі навчання Q-Навчання, можна отримати його модифіковану версію:

$$q(S_t, A_t) = q(S_t, A_t) + \alpha(R_{(t+1)} + \gamma R_{(t+2)} + \dots \gamma^n max_a q(S_{(t+n)}, a) - q(S_t, A_t)) \quad , (1.17)$$

Даний алгоритм має назву n-крокове Q-Навчання, і він має усі переваги своєї немодифікованої версії, а також переваги підходу n-step Bootstrapping, описані раніше. З огляду на описані причини, даний алгоритм є найбільш оптимальним для вивчення середовища, та знаходження оптимальної політики π^*.

ВИСНОВКИ ДО РОЗДІЛУ 1

В даному розділі розглянуто поняття машинного навчання, а також його розширення у вигляді глибинного навчання та навчання з підкріпленням. Наведено теоретичні основи механізмів роботи навчання та їх популярні застосування.

Для подальшого введення основних алгоритмів вивчення середовища приведено детальний аналіз та розбір теоретичних основ, а також математичне обґрунтування базових концептів, на яких базується навчання з підкріпленням. В якості кінцевої цілі навчання формально задана оптимальна політика π^*, а також процес її отримання.

Розглянуто та проаналізовано два протилежні підходи до даної проблеми, а саме методи Динамічного Програмування та Монте-Карло. У першому випадку наведено, що при наявності точної моделі середовища, а саме доступності функції розподілу $p(s_n, r \mid s, a)$, а також апроксимуючи рівняння Беллмана можливо отримати оптимальну поведінку у заданому середовищі. З іншого боку, методи Монте-Карло дають можливість вивчати середовище за допомогою лише постійної взаємодії з ним та отримання певних нагород. В процесі аналізу наведено алгоритми навчання з використанням даних методів, а також їх недоліки та переваги. В результаті аналізу з'ясовано, що обидва підходи мають свої особливості, методи Динамічного Програмування дозволяють використовувати оцінки наступних станів, а методи Монте-Карло не потребують моделі середовища, тому для їх оптимального використання введено інший метод, що їх об'єднує, а саме — Temporal Difference Learning. Проведено аналіз алгоритмів навчання в даному підході та вибрано найбільш популярний алгоритм навчання: Q-Навчання, адже він дозволяє напряму апроксимувати оптимальну функцію якості дії. В якості розширення даного алгоритму введено ідею, що підсилює швидкість навчання та узагальнює методи Монте-Карло та Temporal Difference Learning, а саме n-step Bootstrapping. За допомогою параметра n, що керує кількістю кроків до використання оцінки наступного стану, отримано оптимальний

алгоритм навчання у заданому середовищі. Даний метод дозволяє вивчати середовище лише взаємодіючи з ним, як методи Монте-Карло, а також заміняє решту траєкторії на її оцінку, як в методах Динамічного програмування. Таким чином, в результаті аналізу отримано модифікацію алгоритму Q-Навчання, що має переваги усіх вище загаданих методів: n-крокове Q-Навчання.

РОЗДІЛ 2
АНАЛІЗ ГЛИБИННОГО НАВЧАННЯ ДЛЯ АПРОКСИМАЦІЇ СКЛАДНИХ ФУНКЦІЙ

2.1 Обчислення action-value функції за допомогою апроксимації.

Лінійна регресія.

До цього моменту усі представлені вище алгоритми навчання з підкріпленням розглядались використовуючи невелику скінченну множину станів середовища. Таким чином, оцінки усіх можливих комбінацій станів та дій можна було зберігати в певній таблиці, та використовувати її для знаходження оптимальної поведінки в будь-якій ситуації. Однак множина задач, в яких можливо використати даний підхід, є досить невеликою, адже існує велика кількість проблем, в яких кількість станів значно перевищує норму. Наприклад гра “Backgammon” налічує 10^{20} станів[22], схожа на неї гра “Go” – 10^{170} станів[23], а проблеми що базуються на представленні станів у вигляді не дискретних величин взагалі мають нескінченну множину станів. Таким чином, для використання усіх визначених вище алгоритмів навчання, що базуються на досвіді агента, потрібно замінити табличне представлення функції якості стану на щось інше.

Такою заміною виступають лінійні та нелінійні методи апроксимації функцій. При їх використанні функція якості стану представляється не у вигляді таблиці, а як певна приблизна функція від вектору ваг $\mathbf{w}$. При такому підході, $\mathbf{w}$ може бути як вагами відповідних величин певного вектора, що представляє собою вектор ознак стану середовища, так і вагами відповідних з’єднань у багатошаровій нейронній мережі. Основна перевага такого підходу полягає у тому, що розмірність вектора $\mathbf{w}$ зазвичай набагато менша за кількість усіх можливих станів середовища, тому представлення функції якості стану в такий спосіб потребує набагато менше пам’яті, а також зміна вагів для певного стану s одночасно змінить оцінку функції для інших станів. Звідси випливає, що зміна оцінки для одного стану впливає на оцінки усіх

інших станів, що одночасно як покращує процес навчання так і робить його більш складним для використання та розуміння.

З введенням апроксимації функцій з'являється необхідність у зміні механізму вивчення функції якості стану для заданої політики π. До цього моменту цього можна було досягти послідовно покращуючи кожну з оцінок станів та дій, адже вони зберігались окремо одна від одної. Однак з використанням апроксимації у вигляді лінійної регресії або нейронних мереж, такий підхід не є доцільним, адже зміна оцінки в певному стані призведе до зміни значення багатьох інших станів, що не дозволяє мінімізувати похибку в кожному можливому стані середовища, до того ж через їх нерідку неподільність неможливо змінити значення одного конкретного стану. Для подолання даної проблеми ціль вивчення функції якості стану v(s) для конкретної політики π задається наступним способом:

$$L(\omega)=\sum_{s\in S}\mu(s)*(v_{\pi}(s)-\hat{v}(s,\omega))^2 \quad , \qquad (2.1)$$

де μ(s) – розподіл, що задає важливість похибки в кожному стані s;

$\hat{v}(s,\omega)$ – приблизне значення функції якості стану, отримане за допомогою апроксимації;

$v_{\pi}(s)$ – її істинне значення;

L(w) — загальна похибка оцінки v(s, w) для всіх станів середовища.

Таке формулювання цілі навчання є більш природним для методів, що використовують апроксимацію функцій, однак також є не досить коректним. Дійсно, основне завдання навчання з підкріпленням — знаходження оптимальної політики π*, однак функція якості стану, що підходить для цього завдання не обов'язково буде мінімізувати (2.1). Не зважаючи на це, через відсутність альтернативних формулювань використання формули (2.1) вважається достатнім для пошуку π*.

Для мінімізації (2.1) використовуються методи, що базуються на методі Стохастичного Градієнтного Спуску[24], адже вони найбільше підходять для постійного навчання в середовищі, а також є найбільш поширеними. На

прикладі лінійної регресії, нехай $\mathbf{w}$ — вектор вагів, тоді процес мінімізації (2.1) можна записати наступним чином:

$$\omega_{t+1}=\omega_t-\frac{1}{2}\alpha\frac{\delta(v_\pi(S_t)-\hat{v}(S_t,\omega_t))^2}{\delta\omega}=\omega_t+\alpha(v_\pi(S_t)-\hat{v}(S_t,\omega_t))\frac{\delta\hat{v}(S_t,\omega_t)}{\delta\omega} \quad ,(2.2)$$

де α — параметр, що задає швидкість навчання;

ω_t – вектор вагів під час певного дискретного проміжку часу t;

$\frac{\delta\hat{v}(S_t,\omega_t)}{\delta\omega}$ - похідна від апроксимуючої функції, відповідно до вагів ω.

Таким чином, маючи дійсне значення функції якості стану $v_\pi(s)$, можливо за допомогою (2.2) отримати ваги, що будуть знаходитись в глобальному мінімумі похибки (2.1). Однак у реальних проблемах отримати $v_\pi(s)$ практично неможливо, тому це значення замінюється на його певну оцінку U. Таким чином (2.2) записується в наступній формі:

$$\omega_{t+1}=\omega_t+\alpha(U_t-\hat{v}(S_t,\omega_t))\frac{\delta\hat{v}(S_t,\omega_t)}{\delta\omega} \quad , \qquad (2.3)$$

де U_t – оцінка $v_\pi(s)$ у певний дискретний проміжок часу t.

Дана оцінка може бути як середнім значенням функції якості стану в стані s як в методах Монте-Карло, так і сформована за допомогою апроксимації в процесі Bootstrapping, як в формулах (1.14) – (1.17). Однак при використанні методів Монте-Карло оцінка U за правилом великих чисел буде постійно наближатись до $v_\pi(s)$, тобто:

$$E_\pi[U_t|S_t=s]=v_\pi(S_t) \quad , \qquad (2.4)$$

де $E_\pi[U_t | S_t = s]$ - математичне очікування оцінки U_t, що може бути отримана в зі стану s та слідуючи політиці π;

Така оцінка називається неупередженою (unbiased) за визначенням, адже оцінка функції якості стану, що формується за допомогою методу Монте-Карло, базується лише на значеннях, отриманих в процесі взаємодії з середовищем. Ця властивість гарантує, що $\mathbf{w}_t$ буде сходитись до певного локального мінімуму функції (2.1), що означає методи Монте-Карло, що базуються на градієнтному спуску, гарантують знаходження локально оптимального рішення[25]. Для методів TD-Learning все не так

однозначно[26], адже при заміні U_t на $\hat{v}(S_t, \omega_t)$ з'являється залежність від вектора ваг $\mathbf{w_t}$. Більше того, при заміні U_t у виразі (2.3) він буде мати вигляд (1.14), однак перехід від лівої частини (2.2) до правої базується на незалежності U_t від вектору ваг $\mathbf{w_t}$, а при використанні таких алгоритмів навчання як Q-Навчання дана умова не виконується. Не зважаючи на це, навчання у вигляді (2.3) використовується і в методах TD-Навчання, однак в такій формі (2.3) враховує лише вплив $\mathbf{w_t}$ на функцію апроксимації $\hat{v}(S_t, \omega_t)$. Через врахування впливу $\mathbf{w_t}$ лише в половині виразу, дані методи навчання дістали назву напів-градієнтних методів. Вони теж сходяться до локально оптимального рішення[27], хоч і не так надійно як градієнтні методи, але використовуються частіше за них через причини, зазначені в пункті 1.4.

2.2 Перцептрон. Нейронні мережі прямого розповсюдження.

Частіше за просту лінійну регресію в задачах апроксимації використовуються нейронні мережі. Нейронні мережі відносяться до класу нелінійних апроксиматорів функцій, що мають певні властивості нейронів в людському мозку. В найпростішому випадку вони можуть складатись з одного нейрону — Перцептрону[28]. В такому вигляді нейрон приймає вектор вхідних значень з множини дійсних чисел, обчислює їх лінійну комбінацію з вектором вагів $\mathbf{w}$, і формує певну функцію результату, що називається функцією активації нейрона. Цей процес можна описати діаграмою, як на (рис. 2.1). Функцією активації може виступати будь-яка нелінійна функція, однак найпопулярнішими є сигмоїдна функція ($a(y)=\frac{1}{1+e^{-\alpha y}}$, де α – параметр нахилу сигмоїдної функції), ReLU-функція ($a(y)=max(0, y)$) та функція гіперболічного тангенсу ($a(y)=\tanh(y)=\frac{e^{y}-e^{-y}}{e^{y}+e^{-y}}$). З них найбільш використовуваною є ReLU-функція, адже доведено, що вона прискорює процес навчання в глибинних нейронних мережах[29].

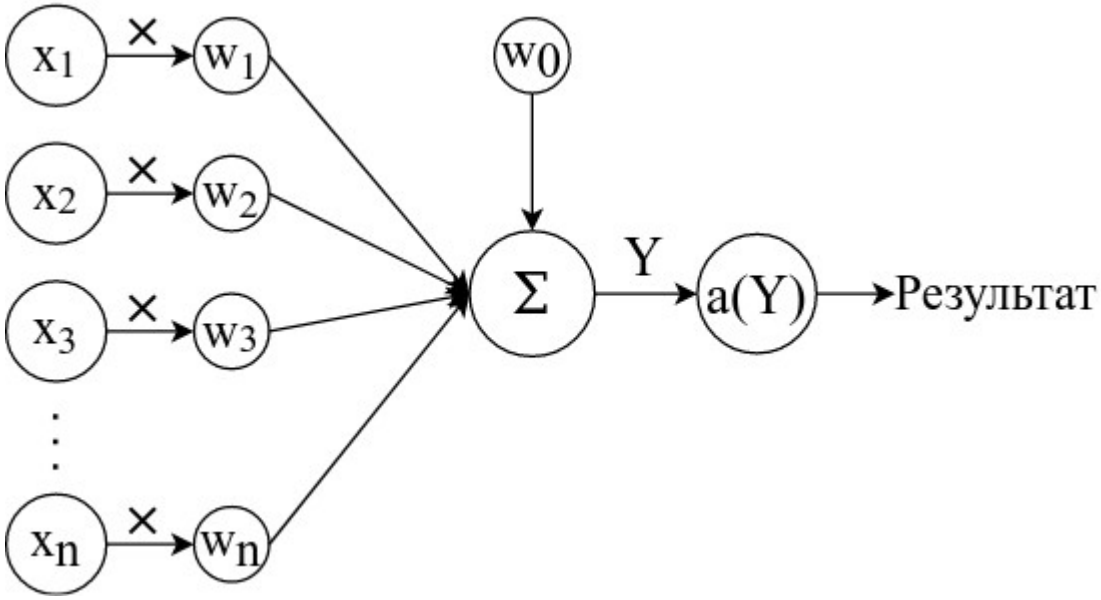


Рис 2.1. Діаграма моделі нейрона (Перцептрона)

Нейронна мережа, що має 1 і більше нейронів як на (рис. 2.1) в одному шарі, а також має 2 або більше шарів називається мережею прямого розповсюдження (Feedforward Artificial Neural Network)[30]. Дана архітектура є найрозповсюдженішою серед інших архітектур нейронних мереж, адже є базовою і часто використовується в парі з іншими їх видами. При використанні нейронних мереж прямого розповсюдження вектор вхідних даних називається вхідним шаром, а останній шар – вихідним. Також, нейронна мережа може мати будь-яку кількість прихованих шарів. Вони не є ні вхідним ні вихідним шаром, і їх функція полягає у обчисленні певних внутрішніх представлень з вхідних даних. Вихідний шар використовує саме ці представлення для апроксимації бажаної функції. Нейрони кожного з шарів мережі підключаються до наступного шару по принципу “кожен з кожним”. Таким чином вихідні дані усіх нейронів одного шару є вхідними даними усіх нейронів наступного. Типову архітектуру нейронної мережі прямого розповсюдження можна представити діаграмою, як на (рис. 2.2).

Доведено, що за допомогою одного лише перцептрону можливо представити більшість булевих функцій[31], а використовуючи архітектуру лише з 1 прихованим шаром та достатньою кількістю нейронів можна апроксимувати будь-яку неперервну функцію в певній області визначення вхідних даних мережі[32]. Не зважаючи на цю властивість, на практиці

апроксимація багатьох складних функцій відбувається набагато легше при використанні декількох прихованих шарів. Як вже було зазначено раніше, кожен прихований шар обчислює певне внутрішнє представлення вхідних даних, при цьому кожен наступний прихований шар обчислює певні абстракції вхідних даних більш високого рівня ніж попередній. Саме обчислення та використання цих абстракцій в прихованих шарах нейронної мережі прямого розповсюдження полегшує процес навчання[33].

Таким чином, нейронна мережа в процесі навчання буде самостійно вивчати певні представлення вхідних даних та використовувати їх для прийняття рішень. Як і в випадку лінійної регресії, оптимізація (2.2) відбувається за допомогою методів Стохастичного Градієнтного Спуску. Окрім цього, для навчання прихованих шарів нейронної мережі використовується механізм зворотного поширення помилки[34]. Даний алгоритм є ітеративним та складається з 2 частин: прямого поширення даних від входу мережі до вихідного шару та зворотного поширення помилки від вихідного до вхідного шару. На кожному етапі прямого

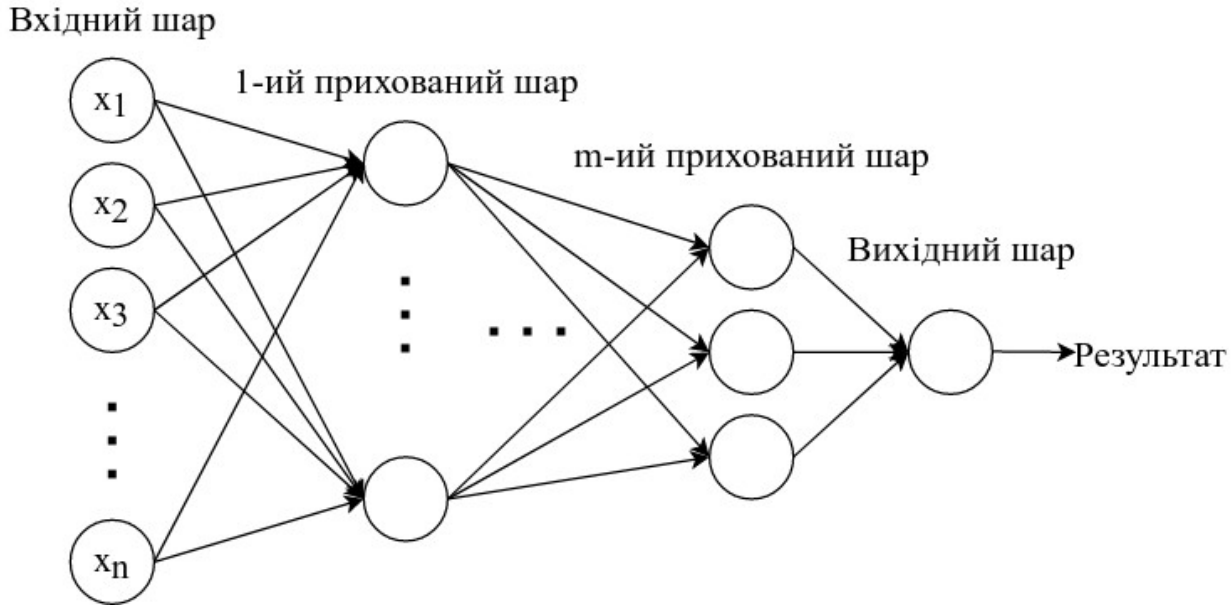


Рис 2.2. Діаграма типової архітектури нейронної мережі прямого розповсюдження

поширення усі нейрони поточного шару обчислюють певну активаційну функцію з вхідних даних, будь то вхідний шар чи вектор активацій

попереднього прихованого шару. В результаті цього етапу, на вихідному шарі формуються певні значення, що у випадку навчання з підкріпленням апроксимацією можуть виступати у ролі значень функції якості стану $\hat{v}(S_t, \omega_t)$. Після виконання прямого поширення, як і у випадку лінійної регресії, обчислюється похибка (2.1), що використовується в наступному етапі. Цим етапом є зворотне поширення цієї помилки, що робиться таким чином, щоб урахувати вплив кожного з вагів мережі за допомогою часткових похідних відносно (2.1). Дану ідея може бути записана у наступному вигляді:

$$\omega_{ij}^{t+1} = \omega_{ij}^{t} + \alpha\left(\frac{\delta L(\omega^t)}{\delta \omega_{ij}}\right) = \omega_{ij}^{t} + \alpha\left(\frac{\delta (U_t - \hat{v}(S_t, \omega^t))^2}{\delta \omega_{ij}}\right) \quad , \qquad (2.5)$$

де ω_{ij}^{t} - значення вагів j нейрону і в певному шарі нейронної мережі у певний дискретний проміжок часу t;

α — параметр, що задає швидкість навчання;

$\frac{\delta L(\omega^t)}{\delta \omega_{ij}}$ - похідна від функції похибки L(ω) відносно значення вагів ω_{ij} у певний дискретний проміжок часу t.

Після певної кількості операцій вигляду (2.5) процес оптимізації похибки зійдеться до рішення, що знаходиться в певному локальному мінімумі (2.1) [35].

Отже, процес знаходження оптимальної політики π* з використанням нейронної мережі в якості апроксиматора можна звести до послідовності наступних етапів. Маючи певну архітектуру нейронної мережі прямого розповсюдження, з вхідних даних, що виконують роль представлення даного стану S_t, на вихідному шарі будуть формуватись значення, що відповідають $\hat{q}(S_t, \omega_t)$ для кожної можливої дії відповідно. На кожному дискретному проміжку часу t будемо отримувати певне представлення наступного стану S_{t+1}, обираючи певну дію a з поточного стану S_t таким чином, щоб $a = argmax(\hat{q}(S_t, \omega_t))$. Після того як було отримано n перших представлень нових станів середовища, за допомогою алгоритму n-крокове Q-Навчання, описаного в пункті 1.4, розрахуємо похибку у вигляді (2.1) та за допомогою

методу Стохастичного Градієнтного Спуску та зворотного поширення похибки обрахуємо вплив кожного з вагів мережі на обчислену похибку L(ω). Дана послідовність дій буде продовжуватись доти, доки не буде отримане рішення, що знаходиться в певному локальному мінімумі (2.1).

2.3 Проблема комп'ютерного зору. Згорткові нейронні мережі.

Як було зазначено в попередньому пункті, для знаходження оптимальної політики π* будуть використовуватись нейронні мережі прямого розповсюдження. Однак для того щоб вони функціонували в якості апроксиматора функції $\hat{q}(S_t, \omega_t)$ необхідно мати певне представлення поточного стану середовища S_t. В ролі такого представлення можуть виступати наприклад, позиції усіх фігур у шахах, дані про поточну швидкість, висоту та кут нахилу до горизонту певного літального апарату, або показники певних приладів на електростанції. Однак існує велика кількість проблем, в яких доступ до таких точних характеристик просто неможливий, або не достатньо очевидно, яким чином було б можливо їх сформувати.

Якраз вирішенням такого виду проблем займається дисципліна комп'ютерного зору. Вона полягає у отриманні представлень високого рівня з цифрових зображень та відео, та намагається зрозуміти та реалізувати механізми, схожі на зір людини. До завдань комп'ютерного зору також входить отримання, аналіз та обробка цифрових зображень з метою формування певних чисельних характеристик, що могли б представляти деякий опис зображуваного. В навчанні з підкріпленням, саме ці характеристики можуть бути використані в якості представлення поточного стану середовища.

Для формування таких представлень з цифрових зображень також можуть бути використані нейронні мережі прямого розповсюдження[36]. В такому разі зображення має бути представлене у відтінках сірого та розгорнуте у певний вектор, в якому кожен з елементів приймав би значення від 0 до 255. Таким чином, пікселі вхідного зображення приймали б участь у

прямому поширенні кожного з нейронів наступного прихованого шару, що показано на (рис. 2.3).

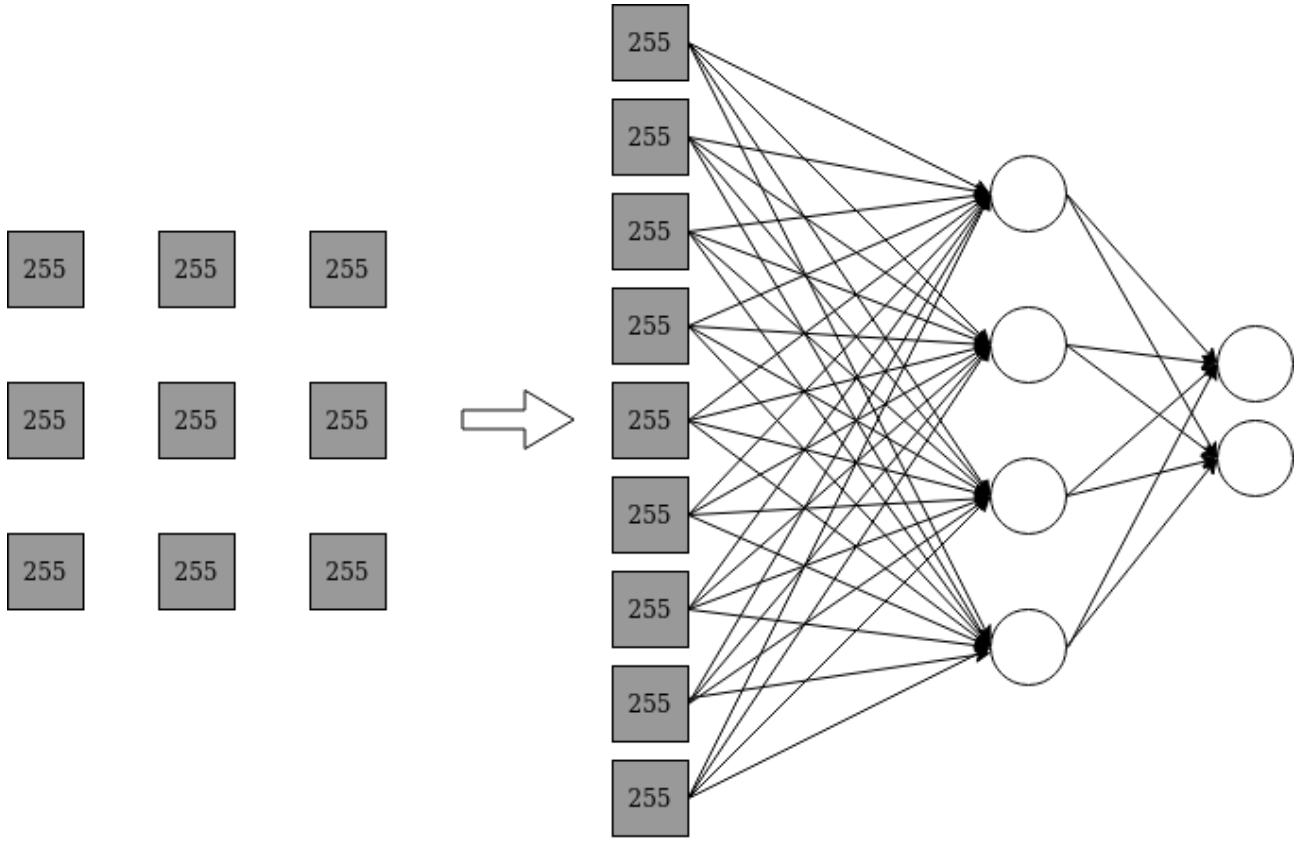


Рис 2.3. Приклад обробки зображення за допомогою нейронної мережі прямого розповсюдження

Однак використання даної архітектури для аналізу стану у вигляді зображення має 2 істотні проблеми. Перша полягає у великих затратах пам’яті на зберігання вагів першого прихованого шару, що взаємодіє з вхідним вектором. На (рис. 2.3) видно, що для того щоб використати зображення розміром 3x3 пікселів, перший прихований шар має мати розмірність вагів 4x9 (4 нейрони на 9 пікселів зображення). У сумі такий шар, в якому ваги нейронів мають тип даних з плаваючою комою розміром 32 біти, займатиме 4 * 9 * 4 = 144 байти. Загалом такий розмір здається невеликим, однак при масштабування на зображення з більшою розмірністю ця проблема стає все більш помітною. Візьмемо зображення розмірністю 84x84 пікселі, в такому разі вектор, що виконує роль вхідного шару мережі буде мати 7056 елементів. У разі використання прихованого шару, що складається з 4 нейронів, ваги цього шару будуть займати 4 * 7056 * 4 = 112 896 байтів, тобто

112 кілобайт. Таким чином зі збільшенням розмірності зображення в 20 разів апаратні затрати на обробку його вхідного вектора зросли в 1000 разів, що означає що затрати пам’яті на зберігання необхідних параметрів становить O(m * n * k), де m x n — розмірність вхідного зображення у відтінках сірого, k — кількість нейронів у першому прихованому шарі. Інша проблема також полягає у великій кількості вагів у мережі, однак має інший ефект — збільшення кількості зв’язків у мережі призводить до того, що модель починає перенавчатись на існуючих даних. Даний процес має назву Overfitting[37], і він має тенденцію виникати частіше в мережах з архітектурою, що має велику кількість шарів. На практиці це означає, що представлення, вивчені нейронною мережею прямого розповсюдження як на (рис. 2.3), можуть виявитись непрактичними у застосуванні, адже при наявності великої кількості нейронів кожен із них буде вивчати певне спеціалізоване на тренувальних даних представлення, що значно погіршує процес узагальнення. Таким чином в найгіршій ситуації, коли кількість вагів більша або дорівнює кількості тренувальних зразків, нейронна мережа може просто запам’ятати увесь тренувальний набір даних, що є поганою апроксимацією необхідної функції.

Іншим підходом до формування необхідного представлення поточного стану S_t є використання Згорткових нейронних мереж (Convolutional Neural Networks)[38]. Дана мережа має іншу архітектуру, що схожа на певну сітку параметрів, яка накладається на вхідні дані. Даний вид нейронних мереж часто використовується для аналізу часових рядів, дані яких можуть бути представлені у вигляди 1-розмірної сітки(вектора), а також аналізу зображень, що представлені у вигляді 2-розмірної сітки(матриці). Причиною їх популярності у даних проблемах є архітектура, адже вона повністю вирішує проблеми, названі вище.

Основа згорткових нейронних мереж базується на математичній операції згортки:

$$c(t)=\int x(a)\,\omega(t-a)\,da \quad , \tag{2.6}$$

де x(a) — певна функція вхідних даних;

ω(t - a) — функція розподілу, що зважує x(a);

a — певний проміжок часу за який відбувається згортка.

Операція згортки також може бути записана наступним чином:

$$c(t)=(x*\omega)(t) \tag{2.7}$$

При використанні згорткових нейронних мереж, x називають вхідними даними, а ω — ядром (kernel) згортки. Результатом цієї операції стають карти збудження (feature map). На практиці, так як усі дані зберігаються у вигляді матриць чи векторів, вони мають певну дискретну кількість елементів. Тому ми можемо записати (2.6) у більш практичній формі:

$$C(i,j)=\sum_m \sum_n X(i+m, j+n)\Omega(m,n) \ , \tag{2.8}$$

де m x n — розмірність ядра згортки;

X — вхідне зображення;

Ω — 2-розмірне ядро, з яким робиться згортка;

C(i, j) — результат операції згортки в рядку i стовпця j.

Операція вигляду (2.8) також має назву крос-кореляції, однак через зручність імплементації в такому вигляді часто подається як операція згортки. Приклад застосування операції згортки до 2-розмірного зображення наведений на (рис. 2.4).

Згорткова нейронна мережа досить схожа на нейронну мережу прямого розповсюдження, однак тут функцію нейронів виконують ядра згортки, що в даній термінології мають назву фільтрів (Filters). Таким чином, один шар згорткової нейронної мережі може мати декілька фільтрів, що накладаються на вхідне зображення як на (рис. 2.4), після чого до результату згортки застосовується одна з функцій активації, які були зазначені в пункті 2.2. В результаті таких обчислень будуть отримані карти збудження, що виконують роль певних представлень з вхідних даних, як і у випадку нейронних мереж прямого розповсюдження. Однак представлення, отримані за допомогою згортки можуть відрізнятися за сенсом від тих, що можуть бути отримані у нейронних мережах прямого розповсюдження[39, 40]. Так один із фільтрів

може бути побудований таким чином, щоб реагувати на наявність горизонтальних чи вертикальних ліній. В такому випадку, відповідний

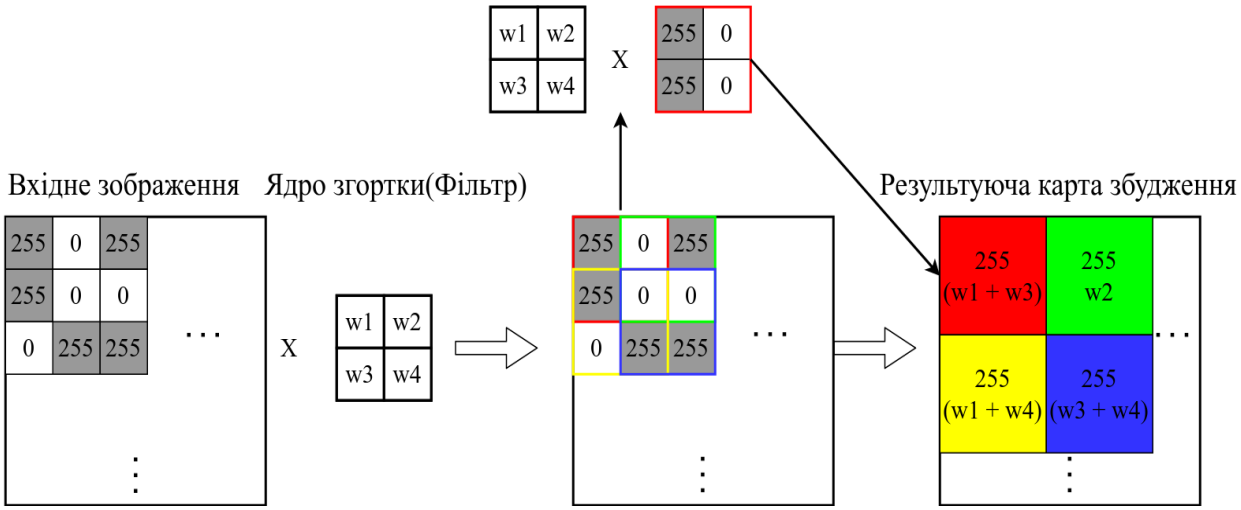


Рис 2.4. Приклад обробки зображення за допомогою операції згортки

елемент карти збудження буде мати більше значення за наявності у рецептивному полі фільтра чогось, що може нагадувати частину лінії, та менше за її відсутності. З використанням більшої кількості шарів згорткової нейронної мережі на відповідних картах збудження будуть формуватись все більш складні представлення, такі як фігури, форми, тощо. Таким чином, за своєю будовою та функціонуванням згорткова нейронна мережа більше нагадує первинну зорову кору людини, що виконує попередню обробку вхідного зображення та формування з нього певних концептів.

Як вже було зазначено раніше, застосування нейронних мереж прямого розповсюдження для аналізу зображень має 2 серйозних недоліки: високі затрати пам’яті та перенавчання. У випадку згорткових нейронних мереж ці 2 проблеми менш помітні, або виключаються повністю через особливості архітектури. Наприклад в нейронних мережах прямого розповсюдження кожен нейрон вхідного шару пов’язаний з кожним нейроном наступного шару, що означає що кожен вхідний нейрон взаємодіє з кожним вихідним. В згорткових нейронних мережах підтримується більш розріджена(sparse) взаємодія, тобто не зважаючи на розмір вхідного зображення розмір ядра згортки залишається таким самим, однак зазвичай набагато меншим за вхідне зображення. Більше того, один і той самий набір параметрів застосовується

до кожного місця в зображенні, що дозволяє знаходити такі прості елементи як лінії навіть у зображеннях з більшою розмірністю. Таким чином для обчислень таких представлень потрібно набагато менше параметрів ніж у нейронній мережі прямого розповсюдження, що дозволяє зменшити потреби до пам'яті. Даний підхід також дозволяє значно покращити процес навчання, адже кожен елемент ядра згортки використовується на кожній позиції вхідного зображення (рис. 2.4), що значно покращує узагальнення.

2.4 Результуюча архітектура нейронної мережі. Особливості навчання

З викладеного вище можна зробити висновок, що для формування певних представлень доцільно використовувати згорткові нейронні мережі, адже в проблемі аналізу зображень вони є більш ефективними за нейронні мережі прямого розповсюдження. Таким чином використовуючи певну кількість згорткових шарів можливо сформувати концепти вищого рівня, що базуються на вхідних даних. Такий результат може бути використаний в якості представлення поточного стану середовища S_t, після чого переданий на нейронну мережу прямого розповсюдження, що виконує роль апроксиматора $\hat{q}(S_t, \omega_t)$. Таким чином, архітектура нейронної мережі, що побудована з блоків зазначених у пунктах 2.2, 2.3, може виглядати як на (рис. 2.5). Спочатку за допомогою певної кількості згорткових шарів формуються карти збудження, після чого перед першим прихованим шаром нейронної мережі прямого розповсюдження результат останнього згорткового шару розгортається (Flatten) у вектор представлень. Даний вектор буде слугувати у якості вхідного шару нейронної мережі прямого розповсюдження, ціль якої сформувати відповідні значення $\hat{q}(S_t, \omega_t)$ на вихідному шарі. Архітектура такого виду називається Глибинні Q-Мережі(Deep Q-Networks) і є однією з найпопулярніших комбінацій глибинного навчання та навчання з підкріпленням. Однак для коректної роботи даний алгоритм також опирається на два допоміжних механізми навчання.

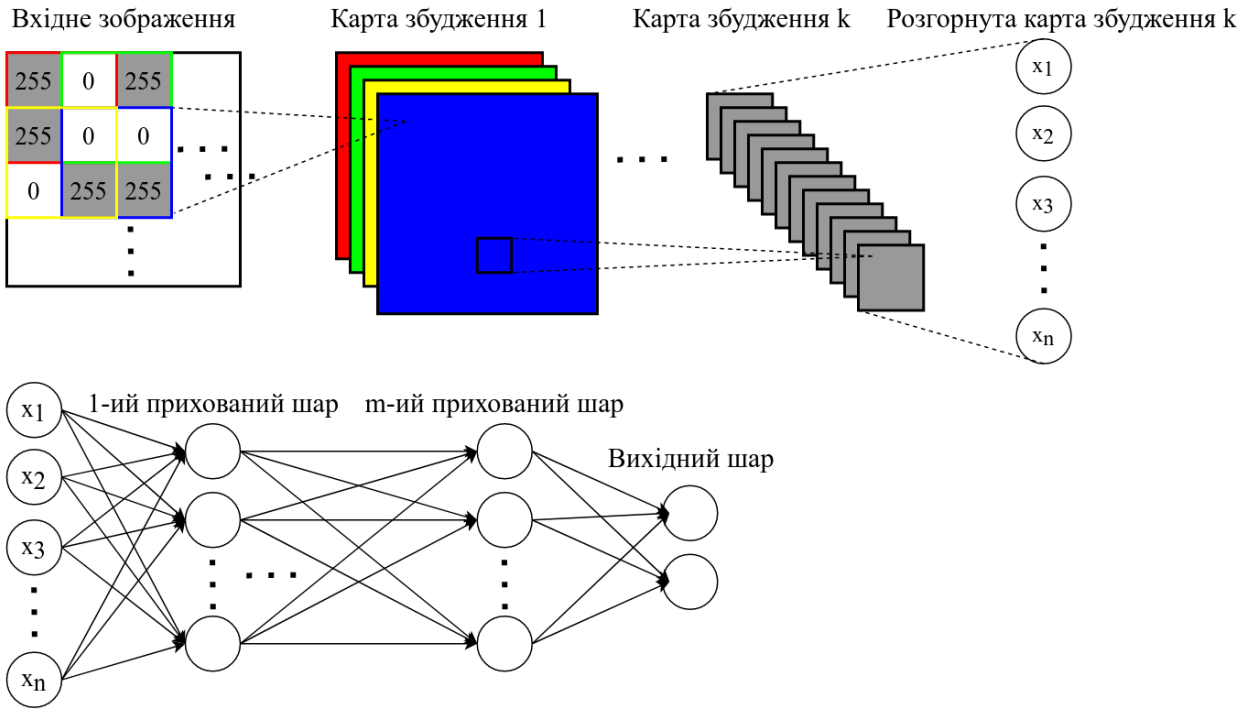


Рис 2.5. Приклад архітектури нейронної мережі для апроксимації функції якості дії

Одним з таких механізмів є перегравання досвіду(Experience replay) [41]. Через використання нейронних мереж в якості апроксиматора функції якості дії $\hat{q}(S_t, \omega_t)$, з'являється необхідність модифікувати процес навчання. Проблема полягає в тому, що у випадку табличних методів зміна оцінки стану s та дії a відбувається одразу після їх переживання агентом у певному середовищі. Однак це означає, що при використанні такого самого способу навчання у випадку апроксимації з'являються наступні проблеми: по-перше, у даних, що використовуються для навчання нейронної мережі з'являються кореляції, адже стани фактично йдуть один за одним, що призводить до збільшення дисперсії у корекціях вигляду (2.5). По-друге, кожна зміна оцінки певного стану s та дії a відбувається лише 1 раз, тобто дані, що отримуються агентом від середовища використовуються не достатньо ефективно. Введення механізму перегравання досвіду дозволяє вирішити ці проблеми за допомогою спеціального буферу пам'яті. Таким чином, після переживання агентом певної комбінації стану та дії, в даний буфер буде записуватись наступна послідовність: поточний стан S_t, дія A_t, що були виконана з даного стану, нагорода R_t, що була отримана за виконання дії

A_t зі стану S_t, та результуючий стан S_{t+1}. В процесі навчання, кожен такий перехід буде доданий у буфер пам'яті, після чого для обчислення похибки (2.1) будуть використовуватись набори(batches) переходів, що обираються випадковим чином. Даний підхід дозволяє стабілізувати процес навчання, адже пошук необхідної поведінки у середовищі відбувається з використанням великої кількості попередніх станів.

Іще одним важливим допоміжним механізмом є використанням цільової мережі(Target network). Як вже було зазначено у пункті 2.1, оптимізація (2.1) у вигляді (2.3) з використанням алгоритмів TD-Навчання вимагає використання оцінки наступного стану в якості заміни істинного значення функції якості стану $v_\pi(s)$. Даний процес схожий за принципом роботи до (1.14) — (1.17), однак має одну суттєву різницю. У випадку табличних методів, кожна можлива комбінація стану та дії має свою окрему оцінку функції якості дії. Однак з введенням апроксимації функцій, така подільність станів не є можливою, адже оцінка кожного з станів середовища отримується за допомогою одного і того самого набору параметрів ω. В такому разі ваги ω використовуються як для формування оцінки поточного стану S_t, так і для наступного стану S_{t+1}. Така ситуація призводить до нестабільності роботи нейронної мережі, адже зміна оцінки $\hat{q}(S_t, \omega_t)$ часто призводить до одночасної зміни $\hat{q}(S_{t+1}, \omega_t)$, що спричинює коливання або навіть призводить до розбіжності політики π. Для подолання цієї проблеми використовується копія поточної нейронної мережі, що залишається незмінною упродовж певної кількості операцій вигляду (2.3). Вона використовується в якості заміни істинного значення функції якості дії замість $\hat{q}(S_{t+1}, \omega_t)$, таким чином вводячи затримку між впливом зміни апроксимуючої функції на формування U_t, що призводить до більшої стабільності процесу навчання.

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі висвітлена проблема використання табличних методів навчання з підкріпленням на задачах з кількістю станів, що значно перевищує норму. В якості вирішення даної проблеми уведений механізм апроксимації функцій якості стану та дії. На прикладі лінійної регресії встановлена нова ціль навчання, що більше підходить для методів апроксимації функцій, а також введений механізм її досягнення за допомогою методів Стохастичного Градієнтного Спуску. Обґрунтована збіжність методів Монте-Карло і TD-Навчання до певного локального мінімуму при використанні лінійної регресії.

В якості більш складного апроксиматора функцій введено поняття нейронних мереж. Наведений принцип їх роботи на прикладі одного нейрона — Перцептрона. Показано типову архітектуру нейронних мереж прямого розповсюдження, а також уведено їх основні властивості, серед яких формування певних внутрішніх представлень різних рівнів абстракції з вхідних даних. Наведено процес навчання нейронної мережі прямого розповсюдження за допомогою механізмів прямого поширення даних та зворотного поширення помилки. За допомогою даної архітектури продемонстровано приклад процесу знаходження оптимальної політики π^*, а також показана необхідність формування певного представлення поточного стану середовища S_t.

Через необхідність отримання таких представлень з зображень, введена проблема комп’ютерного зору в якості отримання певних чисельних характеристик для опису зображуваного. Наведений приклад використання нейронних мереж прямого розповсюдження для обробки цифрових зображень. Показані дві основні проблеми, що виникають у даному випадку: надмірне використання пам’яті, а також схильність до перенавчання (Overfitting). Для подолання даних проблем введене поняття згорткових нейронних мереж. Наведені математичні основи їх роботи, а також продемонстрований процес обробки ними цифрових зображень. В результаті

аналізу обґрунтоване їх використання в якості заміни нейронних мереж прямого розповсюдження для формування певного представлення поточного стану середовища S_t.

В результаті аналізу можна зробити висновок, що для формування функції якості дії та отримання оптимальної політики π^* необхідно використовувати комбінацію з нейронних мереж прямого розповсюдження та згорткових нейронних мереж. З метою подолання проблем, що виникають при використанні апроксимації функцій у алгоритмах TD-Навчання, введені додаткові механізми його стабілізації. Для збільшення ефективності даних, отримуваних від середовища, наведений механізм перегравання пам'яті, що дозволяє повторно використовувати інформацію про кожний пережитий агентом перехід. Також, для зменшення нестабільності у процесі навчання введений механізм цільової мережі, що вводить затримку між впливом зміни апроксимуючої функції та формуванням оцінки наступного стану S_{t+1}.

Таким чином, в результаті аналізу, проведеного в розділах 1 і 2, обґрунтоване використання Глибинних Q-Мереж та алгоритму n-крокове Q-Навчання для пошуку оптимальної поведінки у заданому середовищі.

РОЗДІЛ 3
ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНТЕРФЕЙСУ ВЗАЄМОДІЇ АГЕНТА З СЕРЕДОВИЩЕМ

3.1 Опис інтерфейсу. Основні вимоги

Головною вимогою до системи є її можливість працювати з реальною комп'ютерною грою "Geometry Dash". Таким чином необхідно створити набір інструментів, що буде передавати інформацію про поточний стан середовища, нагороду за перебування в даному стані, належність даного стану до термінального та додаткову інформацію за необхідності. Більше того, окрім передавання інформації від середовища до агента даний набір інструментів повинен забезпечувати передачу та виконання в середовищі дій, що їх обирає агент. Даний процес взаємодії може бути представлений діаграмою, як на (рис. 3.1):

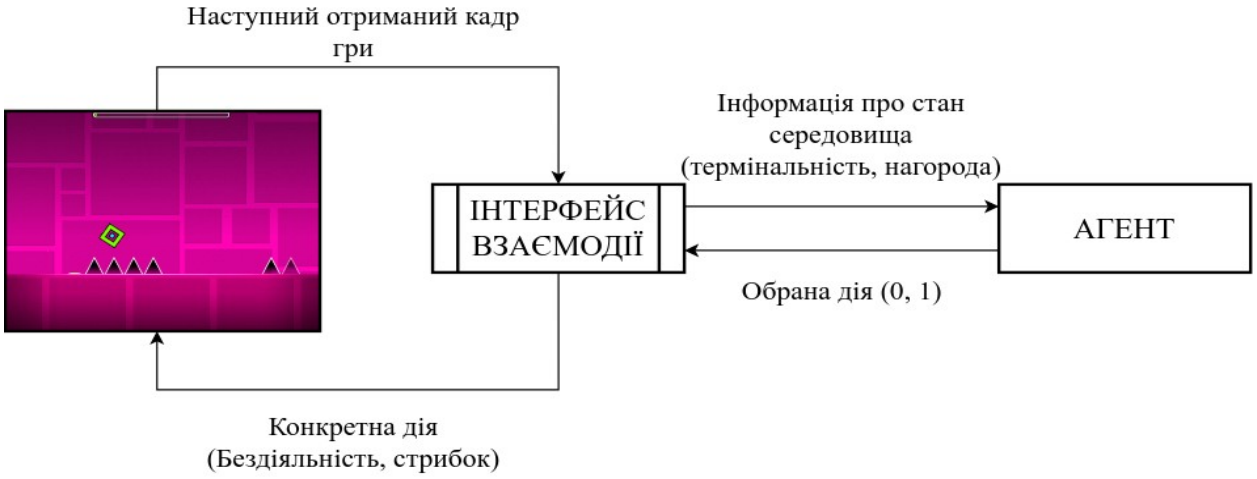


Рис. 3.1. Модель взаємодії агента з грою "Geometry Dash"

Таким чином інтерфейс взаємодії як на (рис. 3.1) повинен мати наступні програмні елементи:

- Програмне забезпечення, що отримує кадри зі гри та виконує їх обробку та аналіз
- Програмне забезпечення, що емулює ввід відповідної дії, обраної агентом

- Програмне забезпечення, що відстежує термінальні стани та передає інформацію про них

Такий інтерфейс реалізуватиме процес взаємодії агента та середовища в Марковських процесах рішень, як на (рис. 1.1). Увесь функціонал такого інтерфейсу можна помістити в один клас, що матиме назву GDenv.

3.2 Формування основної інформації про середовище. Обробка зображень

Перш за все, необхідно запустити “Geometry Dash” у віконному режимі, та вибрати мінімальне допустиме розширення гри, що зменшить подальші витрати на роботу з зображеннями. Далі, для отримання певної інформації зі гри потрібно отримати координати її знаходження на екрані, а також встановити робочу зону, що буде використана для подальших аналізу та перетворення. Для цього завдання підійде модуль PyAutoGui, що дозволяє встановити координати об’єкта на екрані, маючи лише саме зображення об’єкта, як показано на (рис. 3.2).

```
Region = namedtuple("Region", ['x', 'y', 'width', 'height'])
res_dir = normpath(r"/home/scream/GD_RL/res")


class GDenv:
    def __init__(self, resolution, mode="usual"):
        self.region_corner = Region(*pygui.locateOnScreen(join(res_dir, "gd_upper_new_mnjr.png")))
        self.region_main = Region(self.region_corner.x, self.region_corner.y + self.region_corner.height, *resolution)
        self.region_main = dict(zip(('left', 'top', 'width', 'height'), [*self.region_main]))
        pygui.click(self.region_corner)
```

Рис. 3.2. Код знаходження гри на екрані та виділення робочої зони

За допомогою методу *locateOnScreen* та зображення верхнього лівого краю гри, де знаходиться значок гри, можливо в будь-якому її розміщенні на екрані отримати її координати відносно лівого верхнього краю (x, y). Після цього, для отримання робочої зони гри достатньо використати отримані координати наступним чином: координату x потрібно залишити без змін, а y потрібно модифікувати на висоту значка гри, після чого додати до координат вибране у грі розширення. Ці дані зберігатимуться у змінній *region_main*, після чого переводяться у формат, необхідний для інструментів захоплення зображення.

3.2.1 Формування стану середовища

Для захоплення та роботи з зображеннями можна використати популярні бібліотеки Python Image Library(PIL) та OpenCV. PIL це потужна бібліотека для обробки та отримання зображень різних форматів, що також дозволяє отримувати зображення з екрану за допомог модулю *ImageGrab*. Таким чином, використовуючи інформацію про робочу зону гри та даний модуль, необхідні кадри зі гри можуть бути отримані як на (рис. 3.3).

```
frame = np.array(
    ImageGrab.grab(
        bbox=(
            self.region_main['left'], self.region_main['top'],
            self.region_main['left'] + self.region_main["width"],
            self.region_main['top'] + self.region_main["height"]
        )
    ),
    dtype=np.uint8
)
```

Рис. 3.3. Код отримання зображень з робочої зони гри

Змінна *frame* міститиме зображення формату RGB та розширення 640x480. Не зважаючи те що такий розмір є майже мінімальним допустимим для “Geometry Dash”, використання такого формату є дуже затратним для обробки нейронною мережею. Для подолання даної проблеми й використовується бібліотека OpenCV. Це програмна бібліотека з відкритим вихідним кодом, що представляє собою величезний інструментарій для задач комп’ютерного зору, а також машинного навчання. Також, для зберігання та виконання відповідних операцій над зображеннями OpenCV використовує бібліотеку NumPy, що є найпопулярнішою бібліотекою для матричних обчислень. Використовуючи дані бібліотеки, можливо зменшити розмір вихідного зображення, що зменшить як затрати на подальші обчислення так і витрати оперативної пам’яті:

```
self.record_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
new_state = cv2.resize(frame[22:, :], (84, 84))
```

Таким чином, кожен новий кадр гри, що буде далі використаний для побудови поточного стану середовища, передаватиметься за допомогою змінної

new_state у розширенні 84x84 пікселя. Також, для подальшого аналізу додаткових елементів гри та запису відеоряду використовується змінна *record_frame*, що має таке ж розширення як і *frame*, однак використовує інший формат кольору (BGR).

3.2.2 Формування відповідних нагород. Термінальні стани

Окрім безпосереднього представлення стану середовища, для забезпечення навчання з підкріпленням необхідно також задати певний сигнал нагороди (reward signal). Таким чином, агент зможе орієнтуватись у середовищі та формувати оцінки станів виду (1.17). Також, для збільшення ефективності тренування використовується інший ігровий режим, що має назву режим тренування (Practice mode). Таким чином, в разі програшу агент починатиме новий епізод не спочатку рівня, а трохи раніше від місця програшу, що зменшить вірогідність перенавчання. Однак, даний режим також ускладнює відстеження термінальних станів, що є необхідними для формування нагороди. В режимі практики існує 1 термінальний стан, що може бути проінтерпретований як 2 різні: відсутність персонажа на екрані, що означає програш, та відсутність персонажа на екрані при його знаходженні в кінці рівня, що визначається як успіх. В усіх інших випадках персонаж перебуває на екрані, і всі його стани обумовлюються змінною *new_state*. Дана ситуація представлена на (рис. 3.4). Визначення переможного стану можливе, за умови можливості визначення присутності персонажа на екрані, за допомогою шкали прогресу рівня (рис. 3.5).

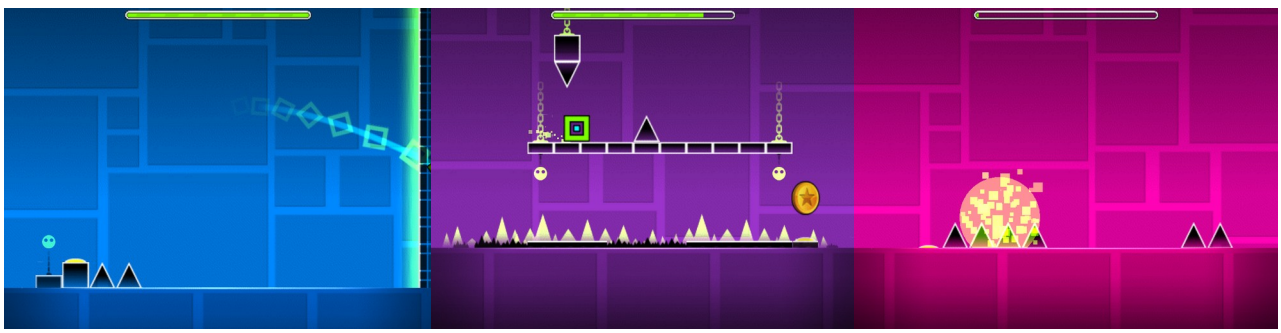


Рис. 3.4. Різні стани середовища (зліва на право: термінальний переможний стан, звичайний стан, термінальний програшний стан)

Рис. 3.5. Шкала прогресу рівня (заповнена)

Використовуючи метод *locateOnScreen* бібліотеки PyAutoGui та зображення повністю заповненої шкали прогресу рівня можливо відстежувати даний стан за допомогою коду, як на (рис. 3.6).

```
def reached_heaven(self, frame):
    return pygui.locate(self.progress_bar_img,
                        cv2.cvtColor(frame[4:22, 228:576], cv2.COLOR_BGR2GRAY), confidence=0.99) is not None
```

Рис. 3.6. Код пошуку повністю заповненої шкали прогресу рівня

Даний метод постійно перевіряє підобласть робочої зони гри *frame* на збіг наявного зображення шкали прогресу рівня з її 100-відсотковим варіантом.

Однак для успішного використання даного методу з метою виявлення переможного стану необхідний також механізм відстеження наявності персонажа на екрані. Дана проблема може бути вирішена за допомогою бібліотеки Tensorflow, а також її інструментарію Tensorflow Object Detection API. Tensorflow це одна з найпопулярніших програмних бібліотек з відкритим вихідним кодом, що дозволяє легко проектувати та використовувати програмне забезпечення з використанням останніх розробок машинного навчання. В свою чергу Tensorflow Object Detection API — фреймворк що дозволяє легко за допомогою готових рішень тренувати та використовувати моделі-детектори. Набір доступних моделей для проблеми object detection представляє Tensorflow Object Detection Model Zoo, що має у наявності велику кількість вже тренованих моделей з різними характеристиками (рис. 3.7). Для вирішення наявної проблеми вирішено використовувати модель SSD MobileNet V1, адже вона має гарну швидкодію, що є одним з найважливіших параметрів для постійної роботи в середовищі.

Насамперед, для використання даної моделі необхідно зібрати тренувальні дані. Це може бути зроблене за допомогою утиліти labelImg, що дозволяє розмічати необхідні області зображення та помічати присутність у

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ☆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssd_mobilenet_v2_quantized_coco	29	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks
mask_rcnn_resnet101_atrous_coco	470	33	Masks
mask_rcnn_resnet50_atrous_coco	343	29	Masks

Рис. 3.7. Набір доступних моделей в Tensorflow 1 Detection Model Zoo

них певного об’єкта (рис. 3.8). На наступному етапі, за допомогою скрипта *generate_tfrecord.py* відбувається перетворення розмічених даних у формат, необхідний для тренування. В результаті за допомогою скрипта *model_main.py* починається процес тренування моделі SSD MobileNet V1 на розмічених даних. Після певної кількості оновлення параметрів нейронної мережі, можна завершити процес тренування. Для легшого відстеження процесу тренування використовується утиліта TensorBoard (рис. 3.9).

В результаті, будуть отримані ваги нейронної мережі, що можуть бути далі імпортовані в коді для подальшого використання (рис. 3.10). Таким чином, змінна *detection_sess* даватиме доступ до натренованої нейронної мережі, а змінні *image_tensor* та *scores* відповідатимуть за вхідні та вихідні дані нейронної мережі відповідно.

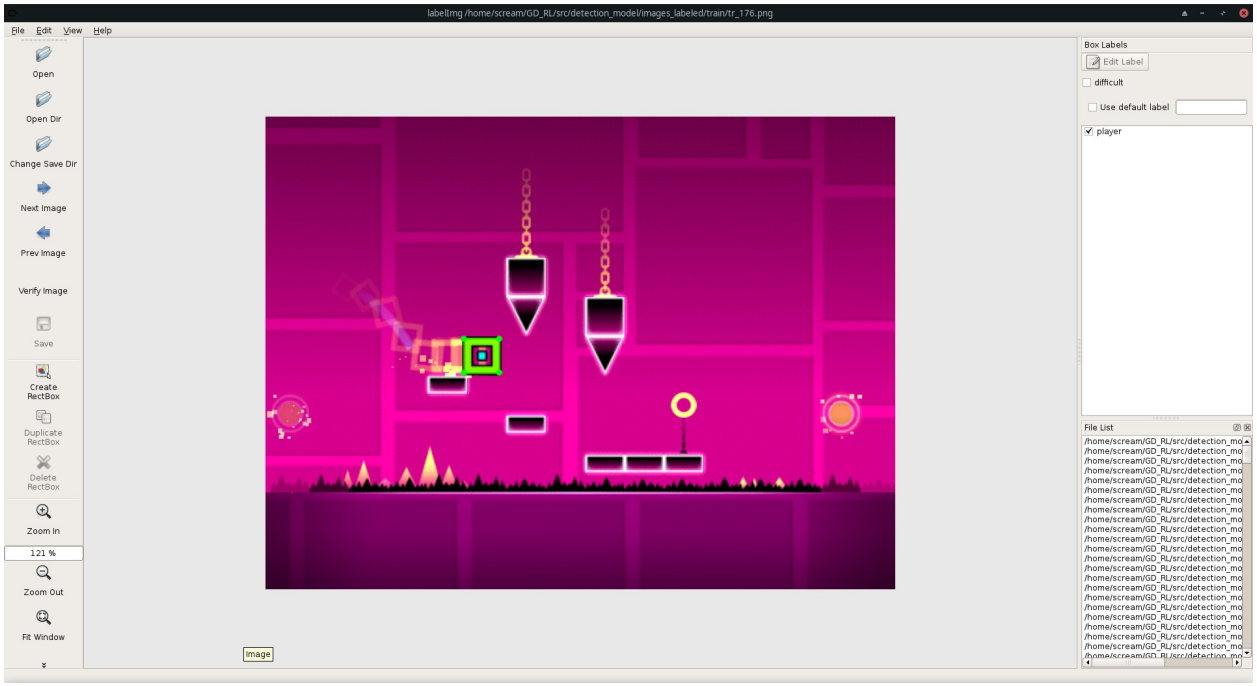


Рис. 3.8. Приклад розмітки зображень за допомогою утиліти labelImg

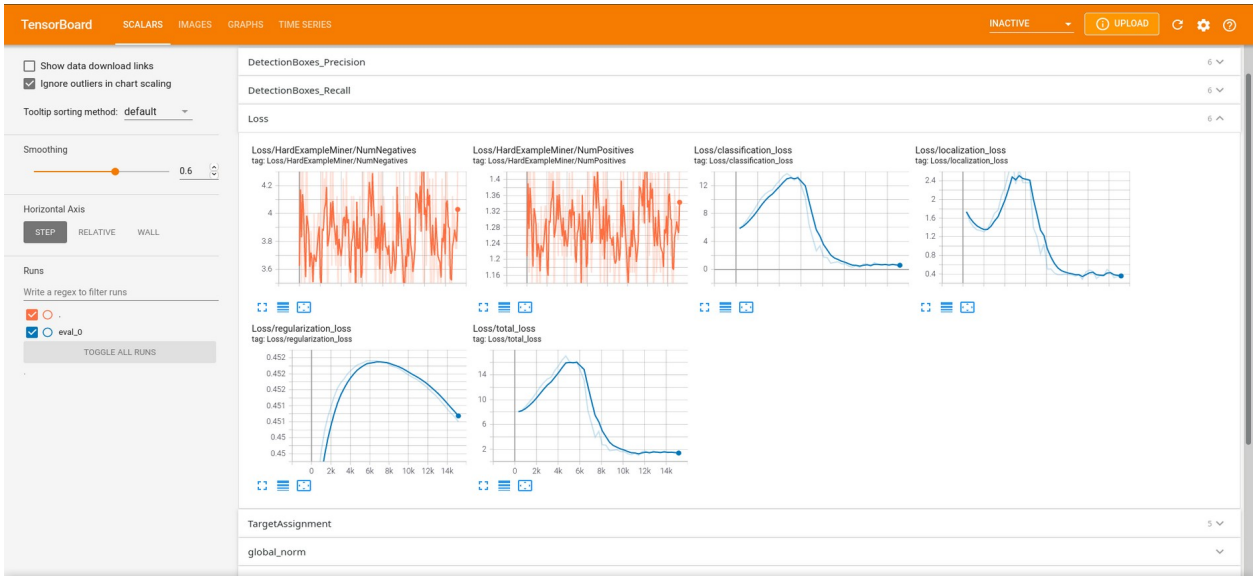


Рис. 3.9. Процес зменшення похибки моделі-детектора у Tensorboard

Використовуючи SSD MobileNet V1 можливо відстежити присутність персонажа на екрані, однак через невелику нестабільність нейронної мережі необхідно також ввести додаткову змінну *agent_absent*. Вона містить невеликий список булевих значень, що приймають значення True, якщо присутність персонажа з певною точністю на даному зображенні встановити не вдалося та False в іншому випадку. Таким чином відсутність персонажа на екрані встановлюється за його відсутності впродовж певної кількості кадрів, тобто коли весь список має значення True (рис. 3.11).

```
MODEL_NAME = '/home/scream/GD_RL/src/detection_model/result_model'
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.compat.v1.GraphDef()
    with tf.compat.v1.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
self.detection_sess = tf.compat.v1.Session(graph=detection_graph, config=config)
self.agent_absent = []
self.image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
self.scores = detection_graph.get_tensor_by_name('detection_scores:0')
```

Рис. 3.10. Код імпорту натренованої моделі SSD MobileNet V1

```
if self.mode == "practice":
    image_np = np.array(frame[22:, :])
    image_np = np.expand_dims(image_np, axis=0)
    scores = self.detection_sess.run(
        self.scores,
        feed_dict={self.image_tensor: image_np})
# new_state = cv2.cvtColor(frame[22:, :], cv2.COLOR_BGRA2RGB)
new_state = cv2.resize(frame[22:, :], (84, 84))
self.reached |= self.reached_heaven(self.record_frame)
if self.mode == "practice":
    if scores.max() < 0.98:
        self.agent_absent.append(True)
    else:
        self.agent_absent.append(False)
    self.agent_absent.pop(0)
    done = True if all(self.agent_absent) else False
else:
    done = self.level_failed(self.record_frame)
reward = 0
if done:
    if self.reached:
        reward = 0
    else:
        reward = -100
```

Рис. 3. 11. Код відстеження термінальних станів середовища

Відсутність персонажа оцінюється за змінною *scores*, що приймає значення у проміжку (0, 1] та відповідає за впевненість моделі у присутності персонажа на екрані в даний момент часу. Після цього дана змінна керує додаванням у список *agent_absent* відповідного булевого значення. Як тільки всі значення списку приймають значення True змінна *done*, що відповідає за термінальний стан, також встановлюється в True. Разом з цим процесом відбувається пошук

повністю заповненої шкали прогресу рівня, що відстежується за допомогою змінної *reached*. На початку кожного епізоду дана змінна приймає значення False, та впродовж усієї взаємодії агента з середовищем виконує код, зазначений на (рис. 3.6). Як тільки знайдене хоча б один збіг, дана змінна приймає значення True, що свідчить про переможний термінальний стан.

Відповідно до заданих термінальних станів необхідно задати певну числову нагороду за знаходження у кожному з них. Наприклад, нехай програш, тобто відсутність агента на екрані, супроводжується нагородою рівною -100, а за виграш та усі інші стани агент отримуватиме нульову нагороду. Такий сигнал нагороди спонукатиме агента до уникнення перешкод, що призводять до його зникнення з екрану і таким чином мотивуватиме його до проходження рівня повністю та без помилок у процесі. Дана ідея зображена в нижній частині (рис. 3.11).

Фінальний алгоритм формування інформації про поточний стан середовища може бути виражений за допомогою блок-схеми, як на (рис. 3.12). Спочатку, відбувається пошук гри на екрані та отримання координат робочого регіону, а також завантаження вагів натренованої моделі-детектора. Разом з цим, на початку кожного нового епізоду скидаються змінні, що відповідають за присутність персонажа на екрані та його досягнення кінця рівня. Отримавши з робочої зони гри новий кадр, відбувається зміна його розміру для подальшого використання у формуванні поточного стану середовища (*new_state*). Далі, за допомогою моделі-детектора вираховується впевненість мережі у присутності персонажа на поточному кадрі. Дане значення порівнюється з необхідним коефіцієнтом точності та додається до відповідного списку (*agent_absent*), після чого термінальність поточного стану вираховується за рівністю значенню True всіх елементів списку (*done*). Разом з цим виконується зміна формату кольору вхідного кадру (*record_frame*) та пошук повністю заповненої шкали прогресу рівня (*reached*). Відповідно до значень змінних, що відповідають за термінальні стани

середовища, встановлюється певна нагорода за перебування в поточному стані (*reward*).

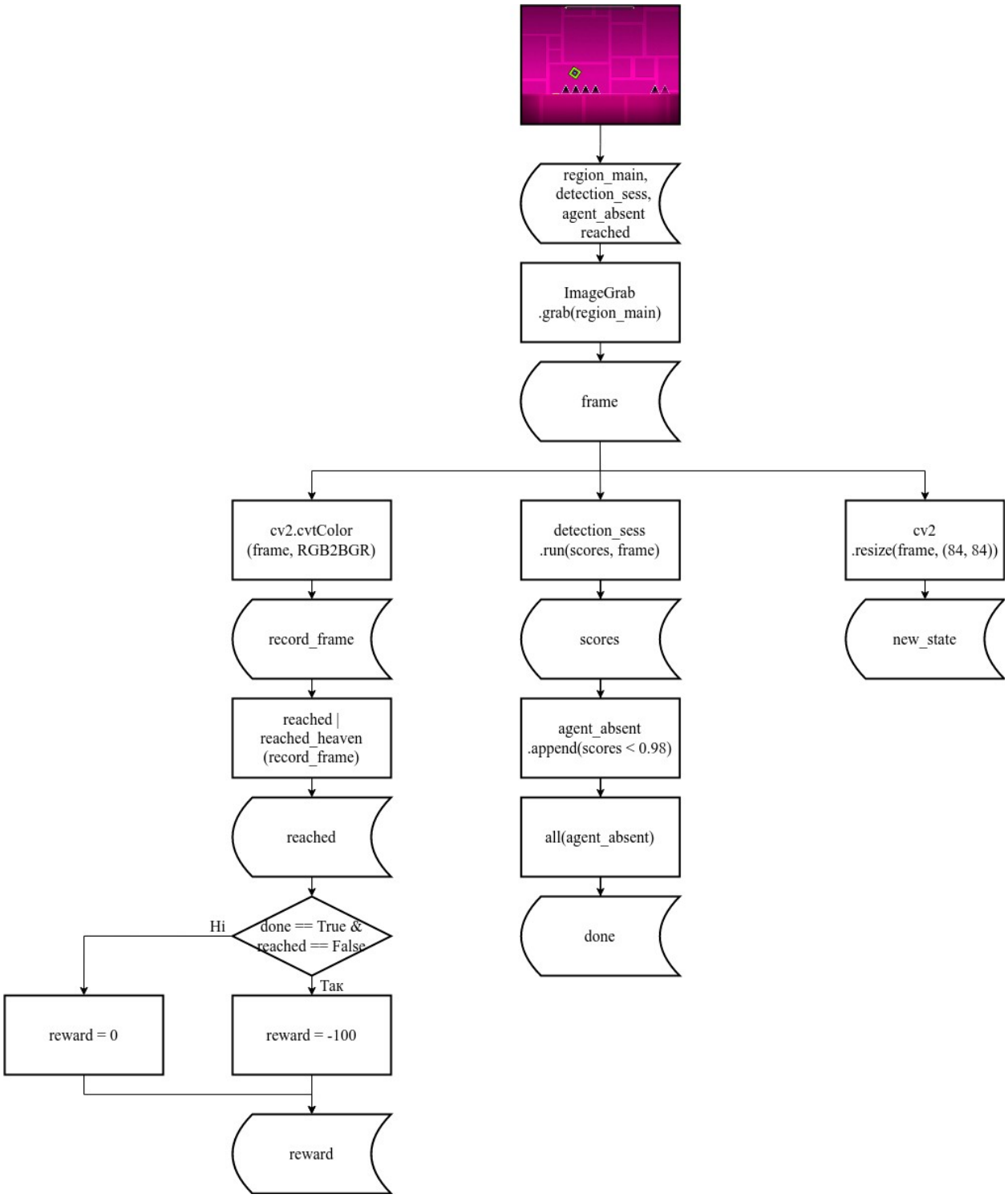


Рис. 3.12. Блок-схема формування інформації про поточний стан середовища

3.3 Контроль над середовищем. Емуляція вводу

Як вже було зазначено вище, окрім отримання від середовища інформації про новий стан, нагороду за перебування в даному стані та його

термінальність, інтерфейс взаємодії також має забезпечувати емуляцію дії, обраної агентом. На операційній системі Linux-типу дану проблему можливо вирішити за допомогою утиліти xdotool. Вона дозволяє програмним чином симулювати ввід с клавіатури та миші, вибравши фокус на відповідному вікні. Для більш зручного використання даної утиліти в скрипті було використано бібліотеку-обгортку python-libxdo, що має той самий функціонал що й оригінальна бібліотека. Так як код на (рис. 3.2) переводить фокус на вікно гри, для отримання об'єкту даного вікна достатньо виконати наступний код:

```
self.xdo = Xdo()
self.window = self.xdo.get_focused_window()
```

Таким чином, змінна *window* міститиме в собі об'єкт вікна відкритої гри, через який в подальшому можливо передавати дії, що обиратиме агент.

Маючи об'єкт вікна гри, дії агента можливо передати наступним чином:

```
def step(self, action):
    if not self.pressed and action == 1:
        # PressKey()
        self.xdo.send_keysequence_window_down(self.window, b"Up")
        self.pressed = True
    elif self.pressed and action == 0:
        # ReleaseKey()
        self.xdo.send_keysequence_window_up(self.window, b"Up")
        self.pressed = False
```

Рис. 3.13. Код управління персонажем у середовищі

Так як уся гра управляється лише натисканням однієї кнопки (стрілка вгору), то дії що передає агент зводять лише до натискання чи відпускання відповідної клавіші. Також, для відстеження ситуацій, коли клавіша вже знаходиться у натиснутому стані використовується змінна *pressed*, що приймає значення True якщо клавіша вже натиснута та False в іншому випадку. Це дозволяє краще емулювати ввід з клавіатури, адже в разі коли наприклад клавіша вже натиснута, а агент вибирає дію 1, тобто стрибок, не потрібно зайвий раз передавати інформацію до вікна. Це також дозволяє проходження агентом певних ділянок рівнів, де використання такої механіки є необхідним.

Дана бібліотека дає можливість не тільки управляти діями персонажа під час проходження відповідного рівня, а й керування самою грою. Таким чином за допомогою емуляції вводу та бібліотеки PyAutoGui можливо керувати початком режиму практики:

```
def start_practice(self):
      self.xdo.send_keysequence_window_down(self.window, b"space")
      self.xdo.send_keysequence_window_up(self.window, b"space")
      time.sleep(0.5)
      self.xdo.send_keysequence_window_down(self.window, b"Escape")
      self.xdo.send_keysequence_window_up(self.window, b"Escape")
      time.sleep(1)
      pygui.click(self.practice_pos)
      time.sleep(0.25)
```

Призупиненням та відновленням процесу гри:

```
def pause(self):
      self.xdo.send_keysequence_window_down(self.window, b"Escape")
      self.xdo.send_keysequence_window_up(self.window, b"Escape")
      time.sleep(1)

def unpause(self):
      self.xdo.send_keysequence_window_down(self.window, b"space")
      self.xdo.send_keysequence_window_up(self.window, b"space")
      time.sleep(1)
```

Управлінням контрольними точками (checkpoints), що дозволяє у випадку термінального програшного стану не починати рівень з самого початку:

```
def uncheckpoint(self):
      self.xdo.send_keysequence_window_down(self.window, b"x")
      self.xdo.send_keysequence_window_up(self.window, b"x")
```

Також важливим елементом керування рівнем є управління початком та завершенням епізоду. В режимі практики перехід агента в кожен з термінальних станів супроводжується різними наслідками. В разі програшу, персонаж знову з’являється на екрані після певної кількості секунд, але вже на певній дистанції від місця програшу. У випадку проходження агентом рівня, гра переходить у відповідне меню, що створює необхідність

перезапустити рівень с початку та знову перейти у режим практики. Дана проблема може бути розв'язана за допомогою наступного коду (рис. 3.14, 3.15). Насамперед скидається значення змінної *agent_absent*, що відповідає за відстеження термінального стану програшу. Далі, в залежності від змінної *reached* вибирається необхідна послідовність дій. В разі перемоги у попередньому епізоді, кожні 0.5 секунд на екрані буде шукатись кнопка, що відповідає за перезапуск рівня. Коли дана кнопка буде знайдена на екрані (*restart_visible*) метод *start_practice* перезапустить режим практики.

В разі програшу попереднього епізоду, змінна що відповідає за кількість спроб на проходження рівня *attempt_counter* збільшується на 1,

```
def retry(self):
    self.fr = 0
    if self.mode == "practice":
        self.agent_absent = [False for _ in range(10)]
        if self.reached:
            self.attempt_counter = 0
            restart_visible = False
            while not restart_visible:
                time.sleep(0.5)
                # frame = np.array(self.sct.grab(self.region_main), dtype=np.uint8)
                frame = np.array(
                    ImageGrab.grab(
                        bbox=(
                            self.region_main['left'], self.region_main['top'],
                            self.region_main['left'] + self.region_main["width"],
                            self.region_main['top'] + self.region_main["height"]
                        )
                    ),
                    dtype=np.uint8
                )
                frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
                restart_visible = self.level_failed(frame)
            self.start_practice()
```

Рис. 3.14. Код управління початком та завершенням епізоду, за умови виграшу в попередньому епізоді

а також встановлюється таймер очікування за допомогою змінної *t* та методу *perf_counter* з модулю *time*. Після цього відбувається пошук персонажа на екрані за допомогою моделі-детектора, та встановлюється змінна *appeared* в разі його успішного знаходження, після чого починається новий епізод. У разі

коли неможливо встановити наявність персонажа на екрані протягом 7 секунд відбувається примусовий вихід з циклу а також зняття поточної контрольної точки. Така ситуація може виникнути коли персонаж застряг на поставленій контрольній точці, тому наявність даної перевірки є необхідною.

```
else:
    self.attempt_counter += 1
    appeared = False
    t = time.perf_counter()
    while not appeared and (time.perf_counter() - t) < 7:
        frame = np.array(
            ImageGrab.grab(
                bbox=(
                    self.region_main['left'], self.region_main['top'],
                    self.region_main['left'] + self.region_main["width"],
                    self.region_main['top'] + self.region_main["height"]
                )
            ),
            dtype=np.uint8
        )
        # image_np = np.array(cv2.cvtColor(frame, cv2.COLOR_BGRA2RGB))
        image_np = np.expand_dims(frame, axis=0)
        scores = self.detection_sess.run(
            self.scores,
            feed_dict={self.image_tensor: image_np})
        if scores.max() >= 0.98:
            appeared = True
    if not appeared:
        self.uncheckpoint()
```

Рис. 3.15. Код управління початком та завершенням епізоду, за умови програшу в попередньому епізоді

Загальний алгоритм управління завершенням епізоду може бути виражений блок-схемою, як на (рис. 3.16). Даний алгоритм в залежності від перемоги чи поразки агента в попередньому епізоді (*reached*) або очікує на появу відповідного меню для перезапуску режиму практики(*start_practice*), або запускає таймер очікування (*perf_counter*), після чого за допомогою моделі-детектора (*detection_sess*) виконує пошук персонажа на екрані, в результаті чого починається новий епізод. В разі неможливості знаходження персонажа на екрані виконується скидання поточної контрольної точки (*uncheckpoint*), щоб запобігти застряганню агента.

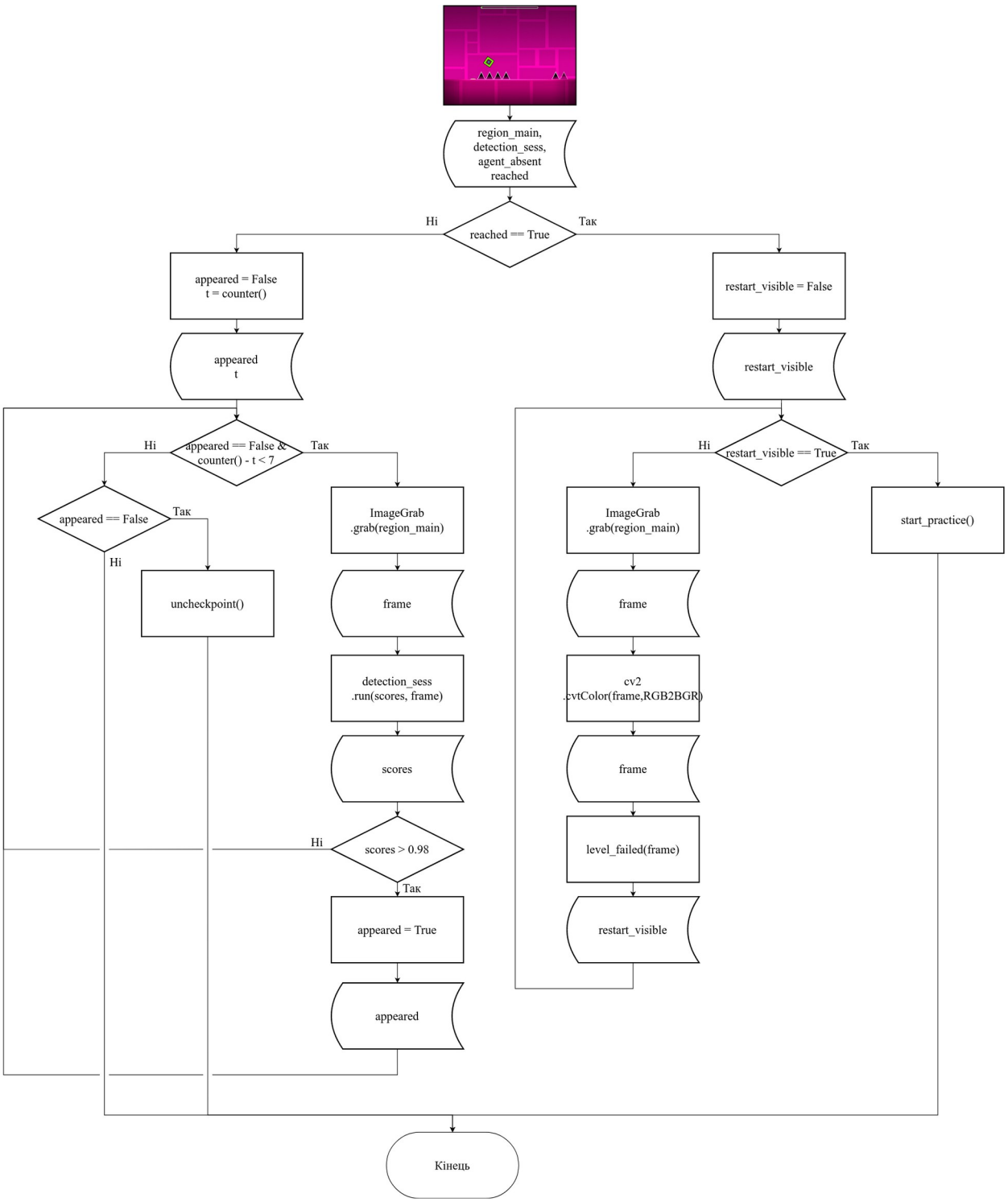


Рис. 3.16. Блок-схема управління завершенням епізоду

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі розроблено інтерфейс взаємодії агента навчання з підкріпленням та середовища гри “Geometry Dash”. До розроблюваного інтерфейсу поставлено 3 основні вимоги, а саме: формування інформації про поточний стан середовища, використовуючи кадри зі гри, емуляція вводу відповідних дій, обраних агентом та відстеження термінальних станів середовища та передача відповідної до них інформації.

Для попереднього знаходження гри на екрані, та виділення її робочої зони використаний модуль PyAutoGui. Даний модуль дозволяє в будь-якому розміщені гри отримати її координати для подальшого формування стану середовища. Для цього завдання використаний модуль ImageGrab бібліотеки Python Image Library, що дозволяє за отриманими координатами та відповідним розширенням гри отримати її зображення. Останнім етапом в побудові поточного стану середовища стала бібліотека OpenCV, що дозволила зменшити розмір вихідного зображення для подальшого використання агентом, а також змінити формат кольору отримуваного зображення для подальшого використання у знаходженні термінальних станів.

Окрім передачі інформації про стан середовища встановлена необхідність задання сигналу нагороди за перебування в кожному з таких станів. З огляду на існуючі механіки обраного режиму гри, встановлено 2 можливих термінальних стани: стан програшу, що виникає у разі відсутності агента на екрані, а також стан перемоги, у разі його досягнення кінця рівня. Задано механізм відстеження термінального стану перемоги за допомогою бібліотеки PyAutoGui та зображення повністю заповненої шкали прогресу рівня. Однак для відстеження даного стану також необхідно встановити присутність персонажа на екрані. Дана проблема розв’язана за допомогою бібліотеки TensorFlow та її інструментарію TensorFlow Object Detection API. Використовуючи модель SSD MobileNet V1, що була натренована на кадрах зі гри “Geometry Dash”, отримано механізм відстеження персонажа на екрані з певною точністю. Використовуючи інформацію про ці два термінальні стани,

впроваджений відповідний сигнал нагороди, що приймає значення -100 у випадку досягнення програшного термінального стану та 0 у всіх інших випадках.

Останньою, але не менш важливою вимогою інтерфейсу взаємодії є емуляція вводу дій агента. Дана проблема розв’язана за допомогою утиліти xdotool та її Python-обгортки python-libxdo, що дозволяють передавати сигнали вводу с клавіатури або миші у сфокусоване вікно. Використовуючи даний модуль, а також елементи бібліотеки PyAutoGui, побудована система що не тільки емулює дії, обрані агентом, а й дозволяє управляти елементами рівня. Таким чином створені механізми, що відповідають за початок та завершення епізоду, а також запобігають застряганню персонажа на певних ділянках рівня.

РОЗДІЛ 4
РОЗРОБКА АГЕНТА ТА МОДЕЛІ ЙОГО НАВЧАННЯ

4.1 Опис моделі навчання агента. Основні елементи

В результаті аналізу, проведеного у розділах 1 і 2, з’ясовано, що для оптимального навчання у заданому віртуальному середовищі необхідно використовувати комбінацію з глибинного навчання та навчання з підкріпленням. В якості алгоритму навчання вирішено використовувати n-крокове Q-навчання, що має сильні сторони як методів Монте-Карло, так і TD-Навчання. Для вирішення проблеми з наявністю кількості станів, що значно перевищує норму, застосовано нейронні мережі в якості апроксиматора функцій. Також з’ясовано, що для вирішення поставленої задачі необхідно використовувати певну комбінацію згорткових нейронних мереж та нейронних мереж прямого розповсюдження (рис. 2.5). Окрім цього, для покращення навчання введено два додаткові механізми стабілізації навчання: перегравання пам’яті та використання цільової мережі. Разом з інтерфейсом взаємодії з середовищем, модель навчання агента може бути задана як показано на (рис. 4.1).

Дана модель передбачає циклічну взаємодію: маючи певне представлення поточного стану середовища, агент обчислює оцінку функції якості дії для кожного можливого варіанту, та передає сигнал про дію, що за цією оцінкою дасть найбільшу винагороду. Далі інтерфейс взаємодії за допомогою емуляції вводу виконує обрану агентом дію та отримує новий кадр з робочої зон гри. В результаті обробки даного кадру інтерфейсом взаємодії агент отримує інформацію про поточний стан середовища, тобто зменшений кадр робочої зони гри, нагороду за перебування в даному стані та його належність до термінального. Агент обробляє дану інформацію та формує перехід, що складається з представлення поточного стану, обраної дії, представлення отриманого стану, нагороди та термінальності отриманого стану. Такий перехід зберігається у пам’яті агента, що на наступному кроці за

допомогою механізму перегравання пам’яті віддає набір(batch) таких переходів для подальшого навчання. Використовуючи даний набір агент формує відповідні оцінки функції якості дії та виконує оптимізацію похибки виду (2.1), що формується за допомогою алгоритму n-крокове Q-навчання (1.17). Даний процес продовжується протягом певної кількості епізодів, що є одним з параметрів навчання.

Рис. 4.1. Модель навчання агента у середовищі

Таким чином, для забезпечення процесу навчання агента потрібно визначити наступні елементи:

- Архітектура нейронної мережі, що формуватиме оцінки фукнції якості дії.
- Модель пам’яті агента, а також механізм перегравання пам’яті.
- Сам процес навчання.

4.2 Архітектура моделі-агента. Реалізація

В результаті аналізу у розділі 2 з’ясовано, що для ефективного навчання у заданому середовищі необхідно використовувати певну комбінацію згорткових нейронних мереж та нейронних мереж прямого розповсюдження.

Конкретною реалізацією даної ідеї, що зображена на (рис. 2.5) є архітектура як на (рис. 4.2).

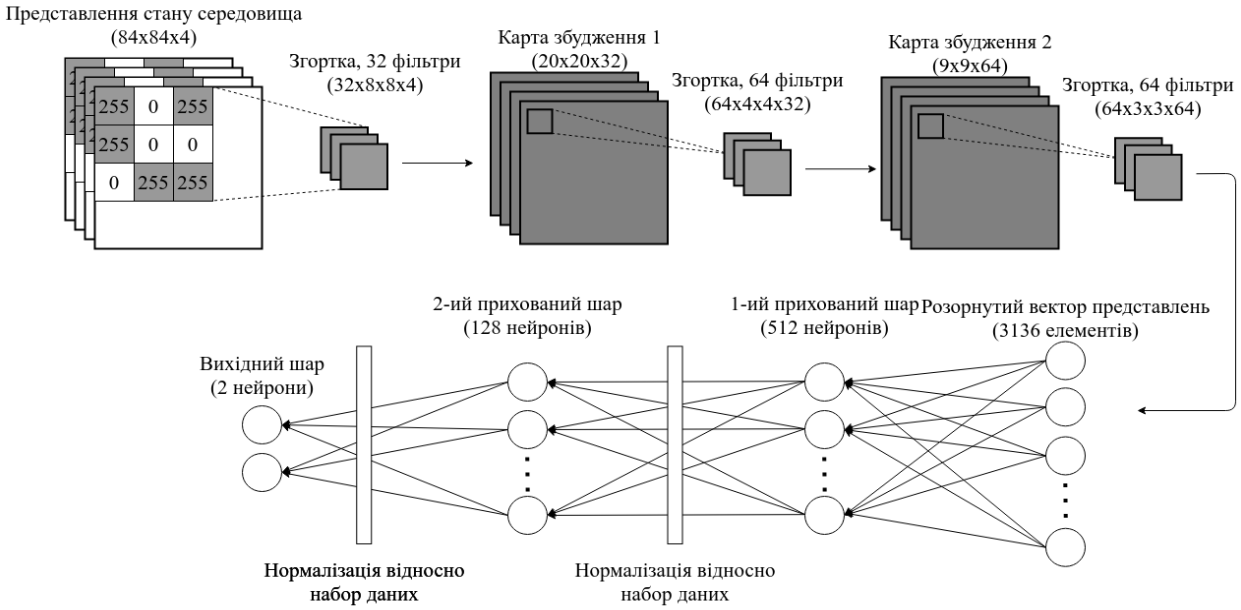


Рис. 4.2. Архітектура нейронної мережі моделі-агента

Для формування внутрішніх представлень кожного зі станів використовується згорткова нейронна мережа з трьома шарами: 1-ий шар складається з 64 фільтрів розміром 8x8 пікселів та використовує зміщення 4 пікселі, два інших шари використовують 64 фільтри розмірами 4x4 і 3x3, а також зміщення 2 та 1 піксель відповідно. Далі вихід останнього згорткового шару розгортається у вектор представлень, що передається на нейронну мережу прямого розповсюдження. Дана обробка розбивається на наступні кроки: перший прихований шар, що має 512 нейронів, оброблює вхідний вектор представлень та пропускає результат обчислень через шар нормалізації відносно набору даних. Даний шар пришвидшує процес навчання та стабілізує його, масштабуючи усі вхідні дані набору до одного числового проміжку. Далі таким самим чином виконується обробка наступним прихованим шаром з 128 нейронами, після чого результат передається на вихідний шар з 2 виходами для формування оцінки функції якості кожної з дій (натиснути кнопку, відпустити).

Використовуючи бібліотеки Tensorflow та Keras таку архітектуру можливо реалізувати за допомогою коду, як на (рис. 4.3).

```
self.X_pl = tf.placeholder(shape=[None, 84, 84, 4], dtype=tf.uint8, name="X")
# The TD target value
self.y_pl = tf.placeholder(shape=[None], dtype=tf.float32, name="y")
# Integer id of which action was selected
self.actions_pl = tf.placeholder(shape=[None], dtype=tf.int32, name="actions")
self.lr = tf.placeholder(tf.float32, shape=[])
self.training = tf.compat.v1.placeholder_with_default(True, shape=[])

X = tf.to_float(self.X_pl) / 255.0
batch_size = tf.shape(self.X_pl)[0]

# Three convolutional layers
conv1 = tf.keras.layers.Conv2D(32, 8, 4, activation=tf.nn.relu, kernel_initializer='glorot_normal')(X)
conv2 = tf.keras.layers.Conv2D(64, 4, 2, activation=tf.nn.relu, kernel_initializer='glorot_normal')(conv1)
conv3 = tf.keras.layers.Conv2D(64, 3, 1, activation=tf.nn.relu, kernel_initializer='glorot_normal')(conv2)

# Fully connected layers
flattened = tf.keras.layers.Flatten()(conv3)
self.fc1_l = tf.keras.layers.Dense(512, activation=tf.nn.relu,
                                   kernel_initializer='glorot_normal',
                                   )
self.fc1_o  = self.fc1_l(flattened)
self.bn1 = tf.keras.layers.BatchNormalization()(self.fc1_o, training=self.training)
self.fc2_l = tf.keras.layers.Dense(128, activation=tf.nn.relu,
                                   kernel_initializer='glorot_normal',
                                   )
self.fc2_o  = self.fc2_l(self.bn1)
self.bn2 = tf.keras.layers.BatchNormalization()(self.fc2_o, training=self.training)
self.predictions_l = tf.keras.layers.Dense(len(VALID_ACTIONS), activation=None,
                                   kernel_initializer='glorot_normal',
                                   )
self.predictions = self.predictions_l(self.bn2)
```

Рис. 4.3. Код архітектури моделі-агента

Спочатку задаються спеціальні об’єкти-наповнювачі(placeholder), що грають роль контейнерів для майбутніх вхідних даних. Тут, *X_pl* є контейнером для представлення набору пережитих станів середовища, що агент отримуватиме зі своєї пам’яті, *y_pl* використовуватиметься для вираховування похибки виду (2.1) та її оптимізації, *actions_pl* зберігає дії, виконані агентом з набору станів *X_pl*, а *lr* та *training* відповідають за темп навчання та режим роботи допоміжних шарів нейронної мережі відповідно. Згорткові шари нейронної мережі задаються за допомогою змінних *conv1*, *conv2*, *conv3*, а відповідні шари нейронної мережі прямого розповсюдження та нормалізації відносно набору даних — *fc1_l*, *fc2_l*, *bn1*, *bn2*. Вихідний шар нейронної мережі задається змінною *predictions*.

Представлення поточного стану середовища оброблюється у форматі 84x84x4, що відповідає використанню 4 останніх оброблених кадрів з робочої

зони гри, отриманих від інтерфейсу взаємодії. Таке представлення стану середовища дає можливість відстежити траєкторію руху різних об'єктів на екрані, а також краще відрізняти стани один від одного.

Окрім задання нейронної мережі необхідно також задати процес оптимізації похибки виду (2.1). Дана задача може бути вирішена за допомогою коду як на (рис. 4.4). Тут, спочатку обраховується квадратична різниця між *y_pl* та відповідним результатом обробки нейронної мережі вхідних даних *X_pl*, що в результаті усереднюється до похибки по набору даних (*loss*). Дану похибку мінімізуватиме оптимізатор, що має назву RMSProp, даний оптимізатор є аналогом методу Стохастичного Градієнтного Спуску, що також пришвидшує процес навчання та стабілізує його. В результаті створюється операція мінімізації похибки виду (2.1), що використовуватиметься агентом для відповідної корекції вагів нейронної мережі.

```
# Calculate the loss
self.td_errors = tf.subtract(self.y_pl, self.action_predictions)
self.losses = tf.squared_difference(self.y_pl, self.action_predictions)
self.loss = tf.reduce_mean(self.losses)

# Optimizer Parameters from original paper
self.optimizer = tf.train.RMSPropOptimizer(self.lr, 0.99, 0.0, 1e-6)
self.train_op = self.optimizer.minimize(self.loss, global_step=tf.train.get_global_step())
```

Рис. 4.4. Код мінімізації квадратичної похибки оцінки функції якості дії

4.3 Пам'ять моделі-агента. Механізм перегравання пам'яті

Як вже було зазначено у пункті 2.4, перегравання пам'яті (Memory Replay) є одним з допоміжних механізмів, що стабілізує навчання нейронної мережі моделі-агента, а також робить його більш зручним для реалізації. Пам'ять агента складатиметься з набору певної кількості переходів виду: пережитий поточний стан, обрана дія, отриманий наступний стан, нагорода та термінальність отриманого стану. Даний формат пам'яті дозволяє зручно використовувати історію пережитих агентом станів для вивчення ситуації у середовищі. Також, окрім самої пам'яті необхідно задати метод отримання з

неї переходів. Необхідна інформація може бути отримана й вибираючи їх випадковим чином, однак такий метод не є оптимальним, адже таким чином вважається, що усі переходи в пам’яті мають однакову важливість для агента. Реалізувати такий модуль пам’яті можна за допомогою наступного коду:

```
class RelatedMemoryReplay:
def __init__(self, feature_shape, memory_len=75_000,
      file_name='replay_memory'):
      self.feature_vectors = np.zeros(shape=(memory_len, feature_shape),
      dtype=np.float32)
      self.memory = []
      self.index = 0
      self.MAX_ELEMENTS_VOLATILE = memory_len - self.BASE_MEMORY
      self.v = tf.placeholder(tf.float32, shape=feature_shape)
      self.vs = tf.placeholder(tf.float32, shape=self.feature_vectors.shape)
      normalize_v = tf.nn.l2_normalize(self.v, 0)
      normalize_vs = tf.nn.l2_normalize(self.vs, 1)
      self.cos_similarity = tf.math.divide(tf.math.add(1.0,
            tf.reduce_sum(tf.multiply(normalize_v, normalize_vs), axis=1)),
      2.0)
```

Даний код створює окремий об’єкт пам’яті, що має декілька допоміжних полів. Наприклад змінна *memory* відповідає за безпосереднє збереження у списку переходів, що агент передає до пам’яті, *index* відповідає за те, щоб при перевищенні певної кількості елементів у пам'яті старі переходи перезаписувались на нові. Також задається змінна *feature_vectors*, що зберігатиме певні внутрішні представлення, що будуть отримані з 2-го прихованого шару нейронної мережі прямого розповсюдження (рис. 4.3). Це необхідно для задання певної важливості кожного з переходів для агента, тобто для більшої ефективності використання пам’яті. Дана ціль може бути досягнута за допомогою механізму схожості, тобто для кожного з переходів у пам’яті необхідно задати певну метрику, що дозволяла б отримувати з пам’яті набори переходів, що найменше відрізняються від поточного стану середовища. Таким чином можливо прискорити процес навчання, адже агент отримуватиме інформацію, що найбільше стосується кожної конкретної

ситуації у середовищі. Однією з таких метрик є косинусна схожість (cosine similarity), вона задає схожість між двома векторами наступним чином:

$$similarity = \cos(\theta) = \frac{A * B}{|A||B|} = \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad , \tag{4.1}$$

де A, B — n-вимірні вектори;

|A| — модуль вектора;

θ — кут між векторами.

Таким чином, чим менший кут θ між векторами, тим вони вважаються більш схожими один на одного. Дана метрика приймає значення у проміжку [-1, 1], що не є зручним для використання у коді, тому для вирішення даної проблеми вона може також бути записана наступним чином:

$$metric = \frac{similarity + 1}{2} = \frac{\cos(A, B) + 1}{2} \quad , \tag{4.2}$$

де mertic — результуюча метрика схожості.

Саме таку метрику схожості реалізує код, показаний вище, використовуючи змінні-контейнери *v* та *vs* можливо обчислити косинусну схожість між конкретним вектором представлень та усіма векторами, що знаходяться в пам’яті.

Таким чином потім можна вибрати набір переходів, що мають найбільше значення цієї метрики або задати з їх допомогою вірогідність вибору. Реалізацію даної ідеї може бути продемонстрована використовуючи код, як на (рис. 4.5). Для отримання певної кількості переходів з пам’яті спочатку обчислюється косинусна схожість між векторним представленням поточного стану середовища та історією таких представлень у пам’яті (*similarities*). Далі дана метрика підноситься до коефіцієнту α, що згладжує оцінки схожості, та ділиться на суму усіх значень для отримання вірогідностей. Дані вірогідності й задають важливість вибору кожного з переходів у пам’яті агента, після чого отриманий набір повертається агенту для обчислення похибки та її оптимізації.

```
def sample_like(self, feature_vec, batch_size: int, sess, alpha=0.6):
    similarities = sess.run(self.cos_similarity,
                            feed_dict={self.v: feature_vec, self.vs: self.feature_vectors})
    ixs = np.arange(self.MAX_ELEMENTS_VOLATILE + self.BASE_MEMORY)[:len(self.memory)]
    similarities = similarities[:len(self.memory)]
    similarities = similarities ** alpha
    similarities /= np.sum(similarities)
    ixs = np.random.choice(ixs, batch_size, p=similarities)
    return [self.memory[i] for i in ixs], ixs
```

Рис. 4.5. Код відбору переходів за косинусною схожістю з пам’яті агента

4.4 Процес навчання. Особливості підходу

Іще однією важливою особливістю навчання з підкріпленням з використанням нейронних мереж є використання цільової функції. Як вже було зазначено у пункті 2.4, вона також допомагає стабілізувати процес навчання та вводить затримку між впливом зміни апроксимуючої функції на формування оцінок. Реалізувати даний механізм можливо за допомогою додаткової нейронної мережі, що має таку ж саму архітектуру як і на (рис. 4.3), ваги якої не будуть оновлюватись окрім випадків коли ваги нейронної мережі агента копіюватимуться у дану мережу. Дана ідея може бути реалізована за допомогою наступного коду (рис. 4.6). Даний код за допомогою бібліотеки Tensorflow задає граф, згідно якому усі параметри моделі-агента копіюються у цільову нейронну мережу, після чого даний набір операцій може бути виконаний у будь-який момент часу за допомогою методу *make*.

Для початку процесу навчання необхідно наповнити пам’ять агента певною кількістю початкових переходів. Це може бути зроблене за допомогою коду як на (рис. 4.7). Як вже було зазначено раніше, поточний стан середовища представляється за допомогою 4 останніх оброблених зображень, отриманих від інтерфейсу взаємодії. Для пришвидшення процесу навчання, такі зображення пропускаються через додатковий детектор граней (Canny edge detector), що дозволяє зменшити набір значень у кожному зображенні до 2: 255 при присутності грані та 0 за відсутності.

```python
class ModelParametersCopier:
    """
    Copy model parameters of one estimator to another.
    """

    def __init__(self, estimator1, estimator2):
        """
        Defines copy-work operation graph.
        Args:
          estimator1: Estimator to copy the paramters from
          estimator2: Estimator to copy the parameters to
        """
        e1_params = [t for t in tf.trainable_variables() if t.name.startswith(estimator1.scope)]
        e1_params = sorted(e1_params, key=lambda v: v.name)
        e2_params = [t for t in tf.trainable_variables() if t.name.startswith(estimator2.scope)]
        e2_params = sorted(e2_params, key=lambda v: v.name)

        self.update_ops = []
        for e1_v, e2_v in zip(e1_params, e2_params):
            op = e2_v.assign(e1_v)
            self.update_ops.append(op)

    def make(self, sess):
        """
        Makes copy.
        Args:
            sess: Tensorflow session instance
        """
        sess.run(self.update_ops)
```

Рис. 4.6. Код копіювання вагів з нейронної мережі-агента до цільової

```python
while len(replay_memory) < replay_memory_init_size:
    env.retry()
    done = False
    elapsed_time = time.perf_counter()
    state, _, _, fr, _, _ = env.step(0)
    state = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)
    state = cv2.Canny(state, threshold1=275, threshold2=300)
    state = np.stack([state] * 4, axis=2)

    rewards_window = []
    states_window = []
    actions_window = []
    feature_vecs = []
    for t in count(start=1):
        # Populate replay memory!
        action_probs, _ = policy(sess, state, epsilon, None, training=False)
        action = np.random.choice(VALID_ACTIONS, p=action_probs)
        next_frame, reward, done, fr, _, _ = env.step(action)
        feature_vec = target_estimator.get_feature_vector(sess, state)

        rewards_window.append(reward)
        states_window.append(state)
        actions_window.append(action)
        feature_vecs.append(feature_vec)

        next_frame_g = cv2.cvtColor(next_frame, cv2.COLOR_RGB2GRAY)
        image = cv2.Canny(next_frame_g, threshold1=275, threshold2=300)
        next_state = np.append(state[:, :, 1:], np.expand_dims(image, axis=2), axis=2)
```

Рис. 4.7. Код наповнення пам’яті агента

Далі представлення стану (*state*) використовується агентом для формування відповідної дії за допомогою політики (*policy*), що задає вірогідності вибору кожної з можливих дій. Використовуючи даний набір вірогідностей агент обирає та виконує дію у середовищі (*env.step(action)*). Для забезпечення формування оцінок за допомогою алгоритму n-крокове Q-навчання відбувається збереження отриманої інформації від середовища (*reward_window*, *states_window*, *actions_window*, *feature_vecs*), та обробка нового кадру з робочої зони гри для формування нового стану середовища (*next_state*). Даний процес повторюється до тих пір, поки не буде досягнута певна кількість переходів, сформованих в результаті взаємодії з середовищем, та може бути зображений як на (рис. 4. 9).

Одразу після даного етапу відбувається процес навчання. Протягом даного процесу також відбувається взаємодія з середовищем та наповнення пам'яті агента, як на (рис. 4.7). Разом із цим нейронна мережа моделі-агента обирає відповідні набори переходів для оптимізації на низ похибки виду (2.1). Даний процес може бути представлений за допомогою коду як на (рис. 4.8).

```
# Sample related transitions from memory replay
samples_rlt, ixs = replay_memory.sample_like(feature_vec, batch_size, sess, alpha=0.6)
states_batch, action_batch, reward_batch, next_states_batch, done_batch = map(np.array, zip(*samples_rlt))

# This is where Double Q-Learning comes in!
q_values_next = q_estimator.predict(sess, next_states_batch)
best_actions = np.argmax(q_values_next, axis=1)
q_values_next_target = target_estimator.predict(sess, next_states_batch)
td_targets = np.dot(reward_batch, n_step_discount) + (1 - done_batch) * \
             discount_factor ** n_step * q_values_next_target[np.arange(len(best_actions)), best_actions]

# Learning rate decays until 500k parameter updates
lr = lrs[min(499_999, total_t)]
batch_loss, _ = q_estimator.update(sess, states_batch, action_batch, td_targets, lr)
loss += batch_loss
update_num += 1
total_t += 1
```

Рис. 4.8. Код процесу навчання агента

Тут, агент спочатку звертається до пам'яті, щоб отримати набір переходів найбільш схожий до поточного стану середовища (*sample_like*) за косинусною схожістю (рис. 4.5). Для формування оцінок з отриманого набору

використовується модифікація алгоритму Q-Навчання, що має назву подвійне Q-навчання.

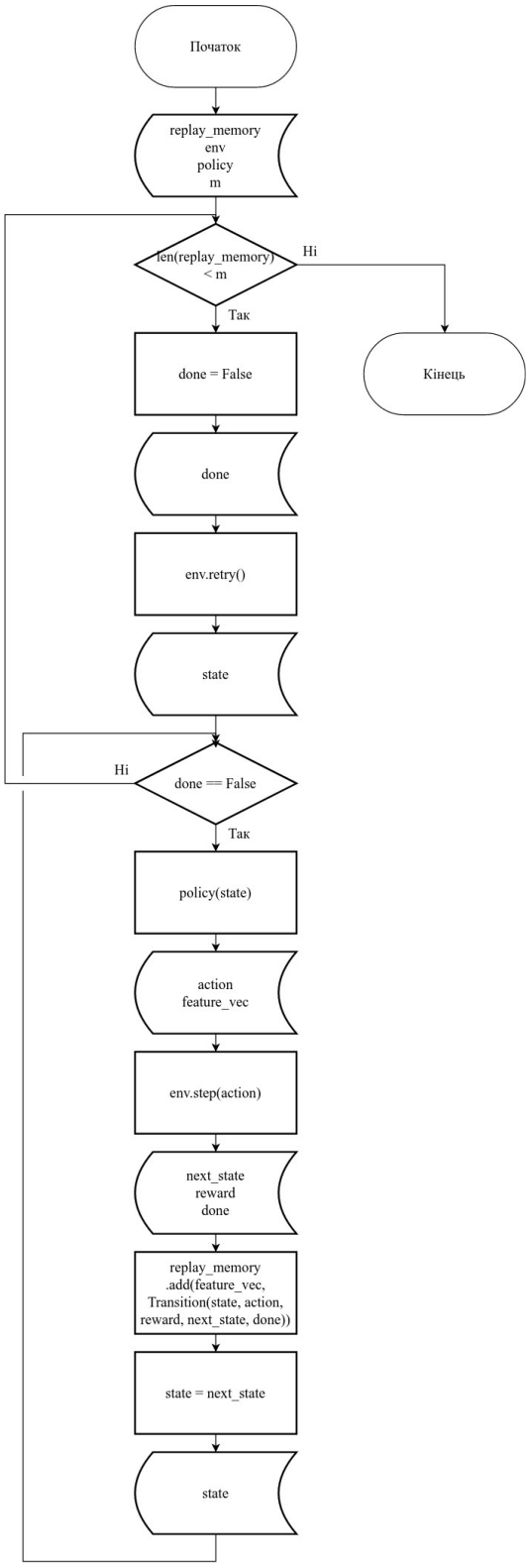


Рис. 4.9. Блок-схема наповнення пам’яті агента

Даний метод полягає у використанні нейронної мережі-агента для формування оцінок функції якості дії, та вибору найкращих дій відповідно до даної оцінки (*best_actions*). Далі використовується цільова мережа для формування самих оцінок обраних дій, що дозволяє підвищити стабільність навчання та покращити точність самих оцінок. На останньому етапі виконується оптимізація похибки виду (2.1) з використанням оцінок як у (1.17) (*q_estimator.update()*), після чого увесь процес повторюється спочатку. Даний алгоритм може бути представлений блок-схемою як на (рис. 4.10).

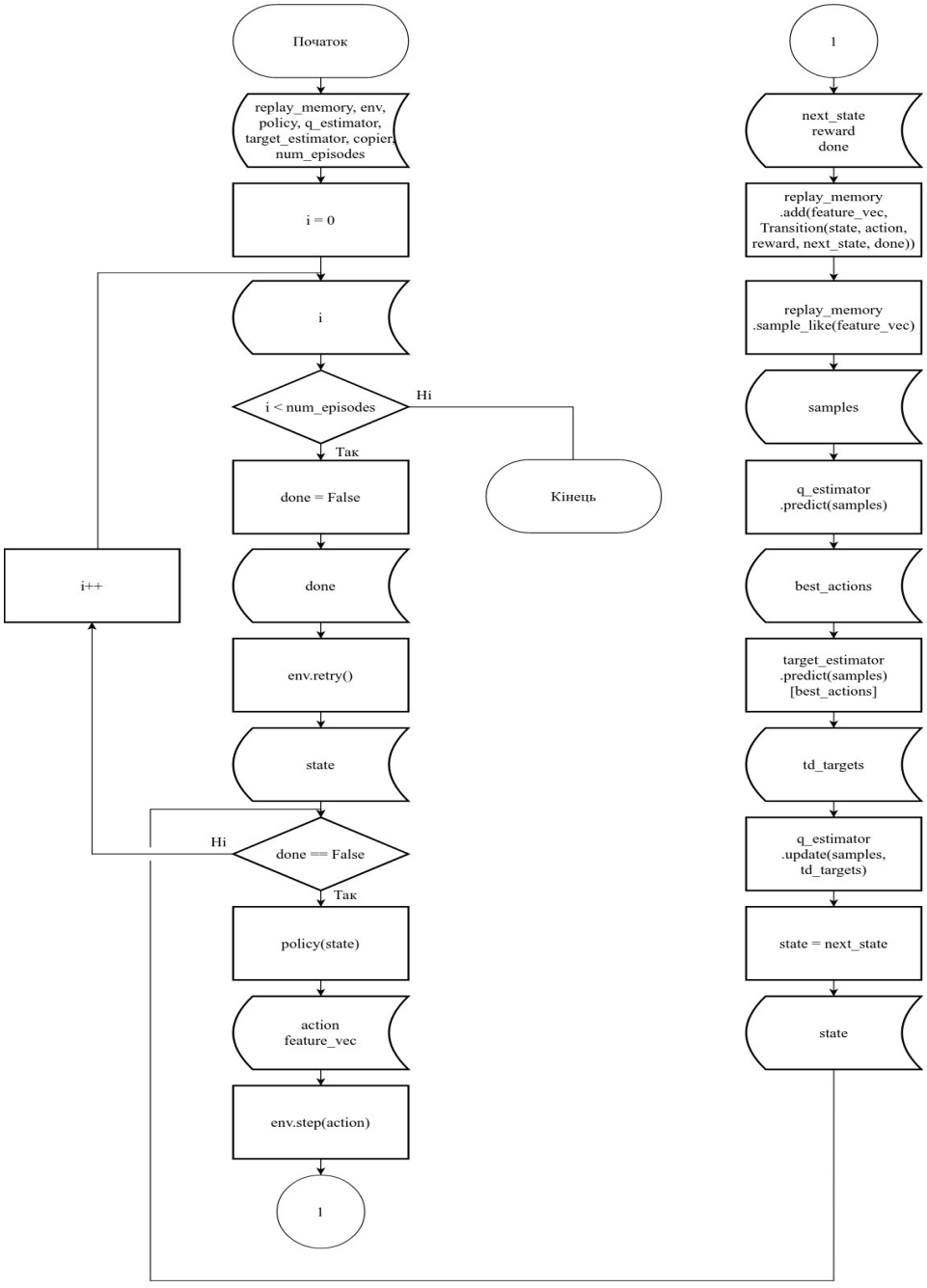


Рис. 4.10. Блок-схема процесу навчання агента

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі розроблено агента навчання з підкріпленням, що вчиться грати у гру “Geometry Dash”. В результаті аналізу, проведеного у розділах 1 і 2, а також проектування у розділі 3 сформовано циклічну модель навчання та взаємодії з заданим середовищем. Задана необхідність у конкретизації трьох елементів системи, а саме: архітектури нейронної мережі моделі-агента, його пам’яті, а також механізму перегравання пам’яті, та самого процесу навчання.

Наведена детальна архітектура нейронної мережі моделі-агента, що складається з наступних елементів: 3 згорткових шари по 32, 64 та 64 фільтри відповідно, для формування певних внутрішніх представлень з поточного стану середовища, 2 прихованих шари нейронної мережі прямого розповсюдження, що доповнюються шарами нормалізації відносно набору даних, по 512 та 128 нейронів, та вихідний шар, що має 2 вихідних нейрони, тобто по 1 на кожну дію. Дана архітектура, а також процеси проходження даних по нейронній мережі та оптимізації середньоквадратичної похибки оцінки функції якості дії, побудовані за допомогою бібліотек Tensorflow та Keras.

Створений модуль пам’яті, що зберігає історію переходів агента у середовищі, що дозволяє підвищити ефективність отримуваної інформації. Для більшого покращення процесу навчання моделі-агента використаний модифікований механізм перегравання пам’яті. З його допомогою агент отримує набори переходів, що мають найбільшу схожість до поточного стану середовища. В якості метрики схожості обрана косинусна схожість, адже вона не залежить від величин векторів, а лише від їх взаємного розташування у просторі.

Окрім перегравання пам’яті реалізований механізм цільової мережі, що імплементується використовуючи нейронну мережу, що має таку ж архітектуру як і модель-агент, та може копіювати значення її вагів. В результаті проектування процес навчання сформований наступним чином:

спочатку відбувається наповнення пам’яті агента певною кількістю тренувальних переходів, після чого відбувається саме тренування. Воно полягає у взаємодії з грою протягом заданої кількості епізодів, під час яких агент за допомогою механізму перегравання пам’яті отримує набір переходів, що максимально стосується поточного стану середовища, та мінімізує середньоквадратичну похибку оцінки функції якості дії, використовуючи методи Стохастичного градієнтного спуску.

РОЗДІЛ 5
ОПИС ТЕСТУВАННЯ СИСТЕМИ

Тестування розробленої системи можна представити у вигляді двох етапів: тестування інтерфейсу взаємодії з середовищем та безпосередньо моделі-агента.

5.1 Тестування інтерфейсу взаємодії

Як вже було зазначено у пункті 3.2.2, для відстеження термінальних станів середовища використана модель SSD MobiliNet V1, що була попередньо натренована на датасеті COCO. Для відстеження персонажа на екрані, за допомогою утиліти labelImg (рис. 3.8) був підготовлений тренувальний датасет, що складається з 344 зображень персонажа у різних ситуаціях у грі:

Рис. 5.1. Приклад розмічених тренувальних зображень

Використовуючи ці тренувальні дані та скрипти *model_main.py* та *generate_tfrecord.py*, модель-детектор була натренована упродовж більше ніж 10 тисяч змін вагів, що у реальному часі складає 6-8 годин (рис. 5.2).

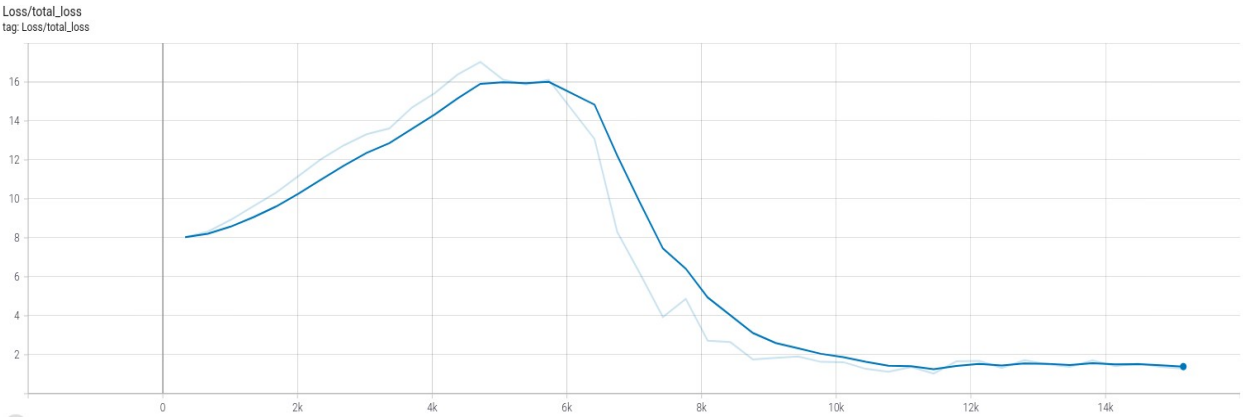


Рис. 5.2. Процес тренування моделі-детектора, відображений у TensorBoard

В результаті було отримано модель, що відстежує персонажа на екрані з непоганою точністю, однак трапляються ситуації коли модель втрачає його на 1-2 кадри, або приймає інші об’єкти в якості персонажа (рис. 5.3).

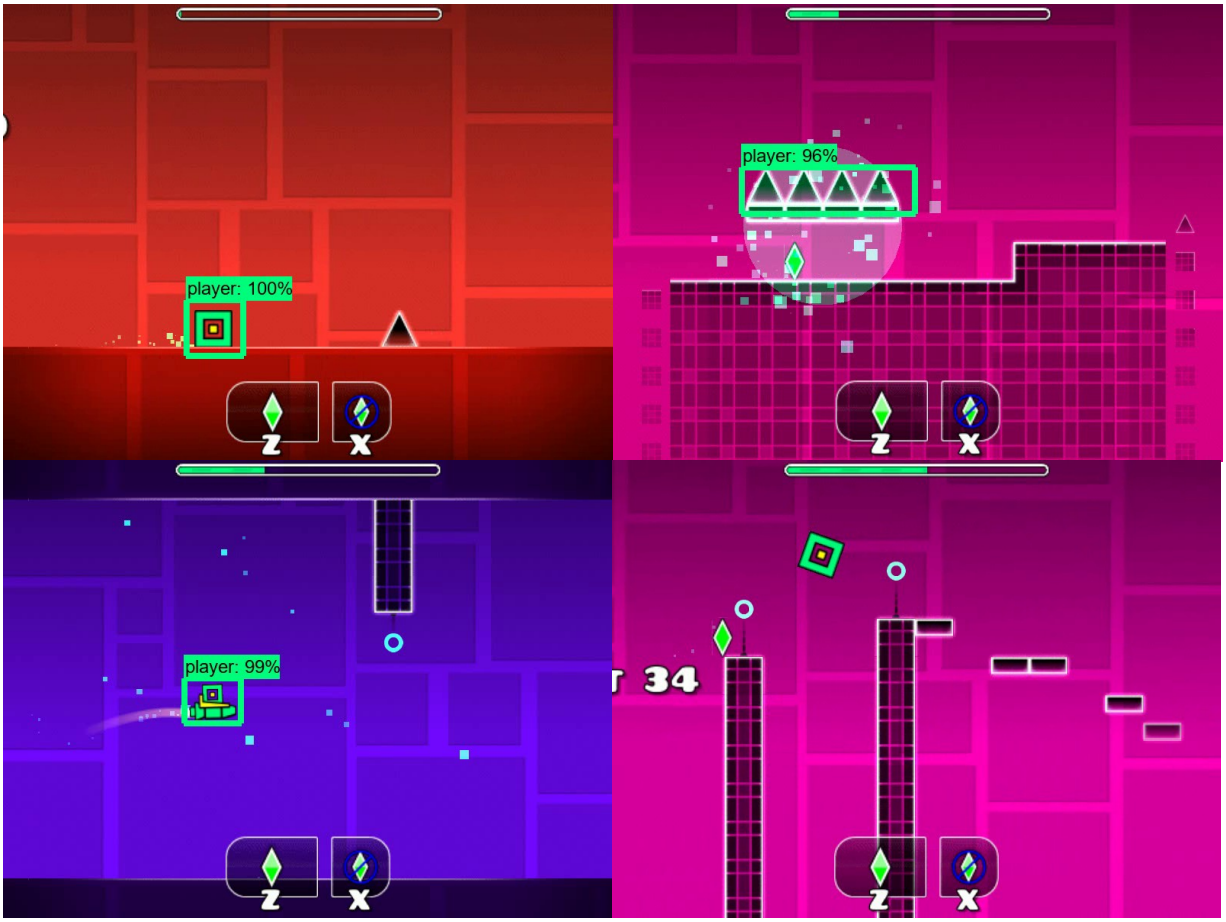


Рис. 5.3. Приклади роботи моделі-детектора

(зліва: очікувана поведінка, справа: нестабільність роботи)

Для уникнення таких ситуацій, необхідна точність для визначення присутності персонажа на екрані встановлена у 98 відсотків, таким чином можливо уникнути похибки детектора, що приймає інші об’єкти за персонажа. Щодо випадків із зникненням його із поля зору на 1 або 2 кадри, для вирішення цієї проблеми у пункті 3.2.2 введена змінна-вікно *agent_absent*, що регулює даний процес.

Використовуючи натреновану модель-детектор протестовано можливості інтерфейсу до передачі інформації про термінальні стани середовища та нагороду за них (рис. 5.4 — 5.6). Як видно на (рис. 5.4) агент перебуває у програшному стані та отримує за нього нагороду -100 та встановлена змінна *done*, що відповідає за відстеження персонажа.

Рис. 5.4. Приклад отримання інформації через інтерфейс взаємодії (Програшний стан)

Рис. 5.5. Приклад отримання інформації через інтерфейс взаємодії (Звичайний стан)

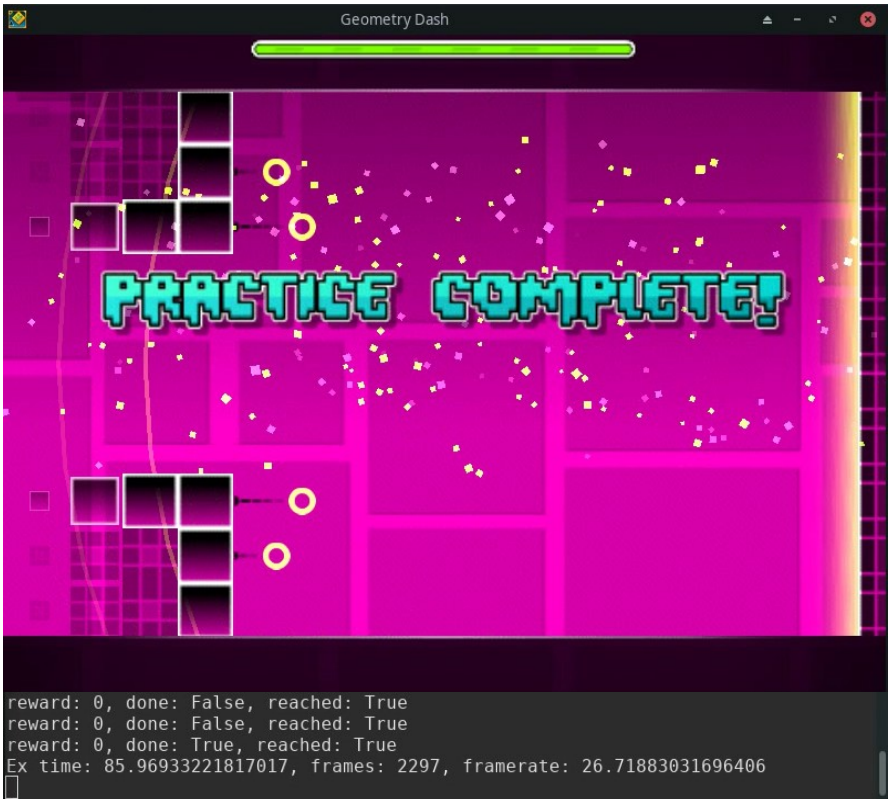


Рис. 5.6. Приклад отримання інформації через інтерфейс взаємодії (Переможний стан)

Щодо ситуацій, зображених на (рис. 5.5), (рис. 5.6), інтерфейс взаємодії передає нагороду 0 у обох ситуаціях, але коли персонаж досягає кінця рівня встановлюється змінна *reached*, що відповідає за відстеження повністю заповненої шкали прогресу рівня.

5.2 Тестування моделі-агента

В результаті проектування у розділі 4 встановлено, що процес навчання агента складається з двох частин: наповнення пам’яті та самого тренування. Процес наповнення пам’яті відбувається наступним чином: агент приймає рішення у середовищі слідуючи політиці, згідно якої він у 10 відсотках випадків обирає випадкову дію, а у всіх інших ту, що дає найбільшу нагороду згідно оцінки функції якості дії. Дана взаємодія відбувається доти, доки в пам’яті агента не зберігатиметься хоча б 5000 переходів, отриманих від середовища. Даний процес зображено на (рис. 5.7), тут у пам’яті було зібрано 5010 переходів виду стан, дія, набір нагород, наступний стан, термінальність.

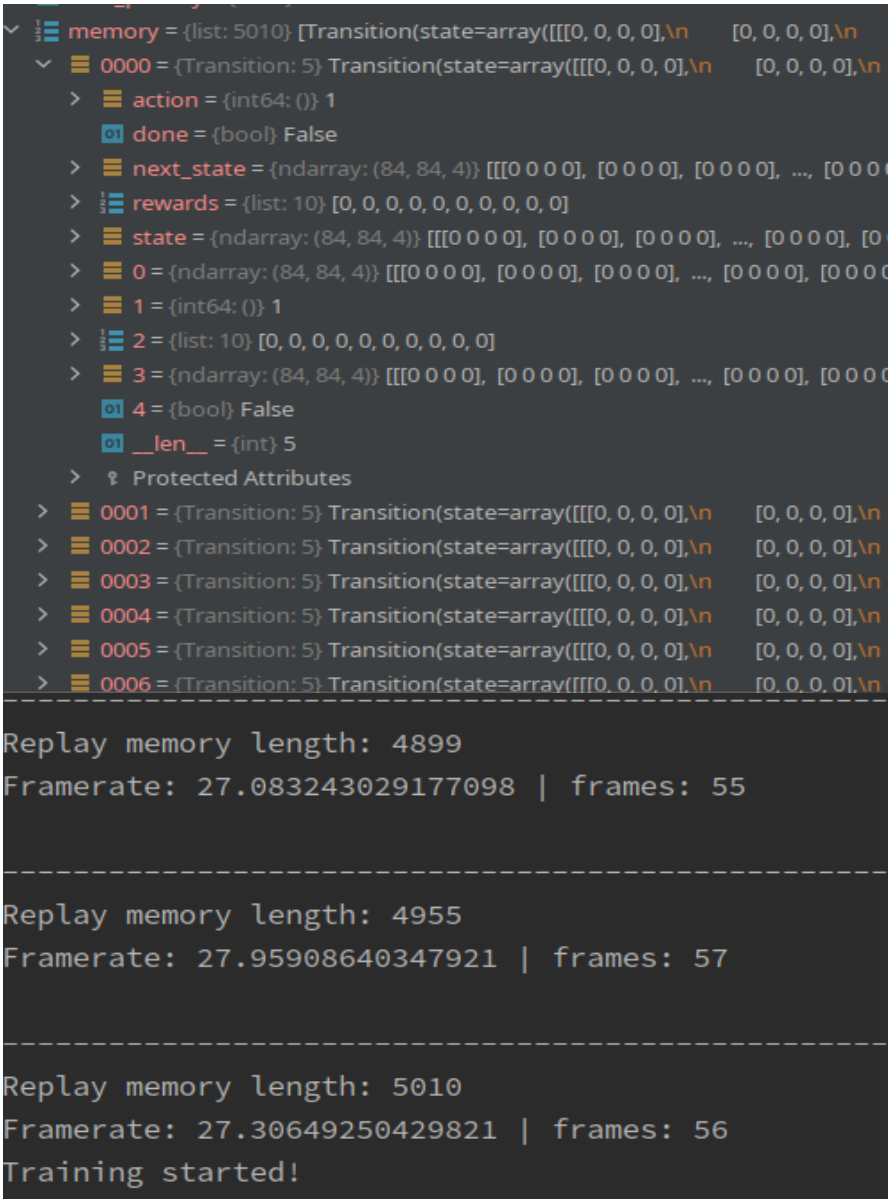


Рис. 5.7. Приклад процесу наповнення пам’яті

Одразу після того як пам’ять була наповнена необхідною кількістю переходів розпочинається процес навчання, що продовжується протягом 20000 епізодів (рис. 5.8). Упродовж усього процесу тренування агент мінімізує середньоквадратичну похибку оцінки функції якості дії, що відстежується за допомогою утиліти TensorBoard (рис. 5.9). Окрім похибки, також важливі параметри max_q_value та attempts, що відповідають за максимальне значення функції якості дії у кожному з пережитих агентом станів та кількість спроб, витрачених на проходження рівня відповідно. Як видно на (рис. 5.9), агент навчився проходити рівень, витрачаючи на нього в середньому 3-5 спроб та мінімізуючи похибку, хоч і з нестабільностями.

```
--------------------------------------------------

Episode's framerate: 17.219385493949652
Replay memory len : 5514
Episode's #14 loss: 200.72099709660122
Action values: [[-41.659676 -41.76991 ]], total_t: 50999
--------------------------------------------------

Episode's framerate: 17.483989331817707
Replay memory len : 5533
Episode's #15 loss: 210.15557880922594
Action values: [[-33.74033  -32.733013]], total_t: 52819
--------------------------------------------------

Episode's framerate: 16.936143574719697
Replay memory len : 5552
Episode's #16 loss: 184.79921963207224
Action values: [[-51.188526 -48.15698 ]], total_t: 54737
--------------------------------------------------

Episode's framerate: 17.334520663625696
Replay memory len : 5571
Episode's #17 loss: 185.5523887543131
Action values: [[ 0.24570681 -0.17203823]], total_t: 549
```

Рис. 5.8. Процес навчання моделі-агента

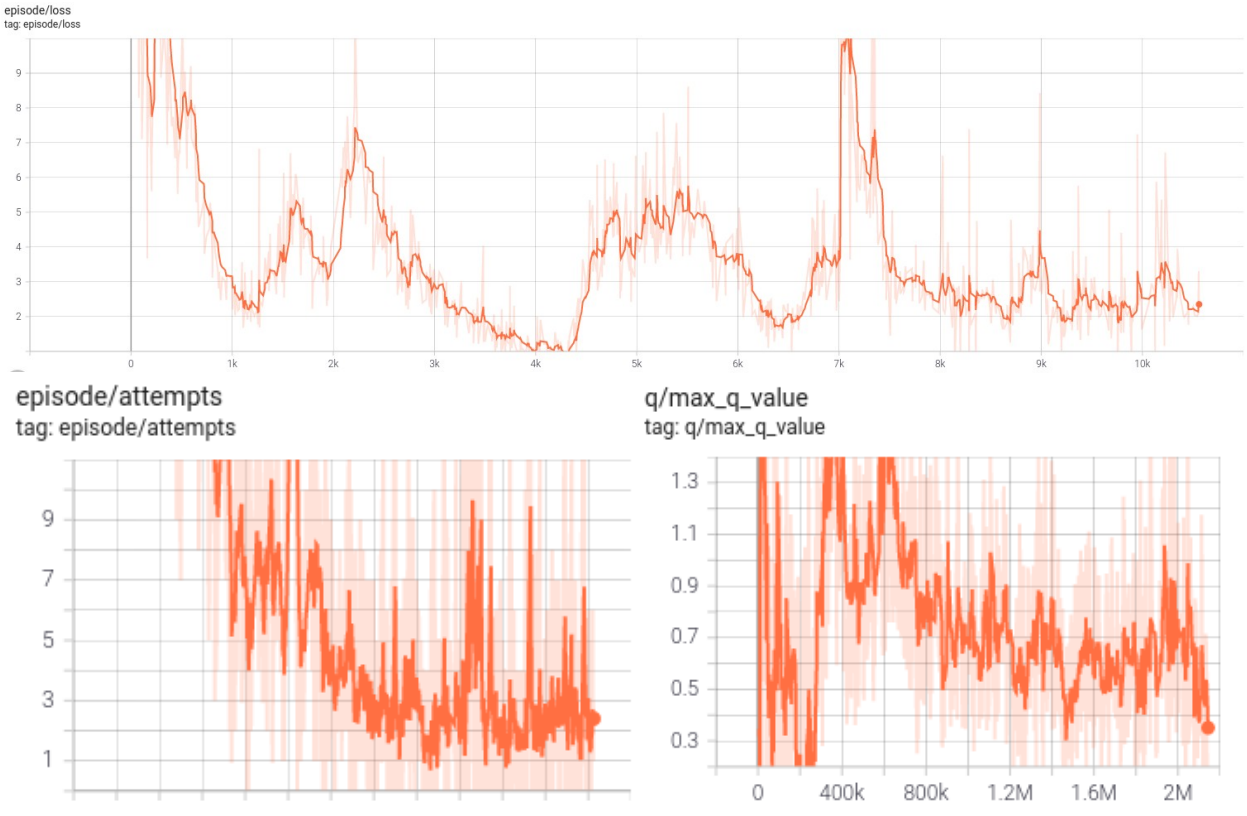


Рис. 5.9. Графіки результату процесу навчання

Показники, що продемонстровані на (рис. 5.9), отримані в результаті процесу тренування агента на другому рівні гри протягом 30 годин реального часу.

Протягом процесу тестування з'ясовано, що саме використання шарів нормалізації відносно набору даних та допоміжного детектора граней (Canny edge detector) призводить до найбільш сприятливих результатів. Використання даного детектору спрощує кількість станів середовища, що пришвидшує процес навчання в цілому, а використання шарів нормалізації допомагає його стабілізувати. Відсутність цих елементів у процесі навчання часто призводить до розбіжності середньоквадратичної похибки, як показано на (рис. 5.10).

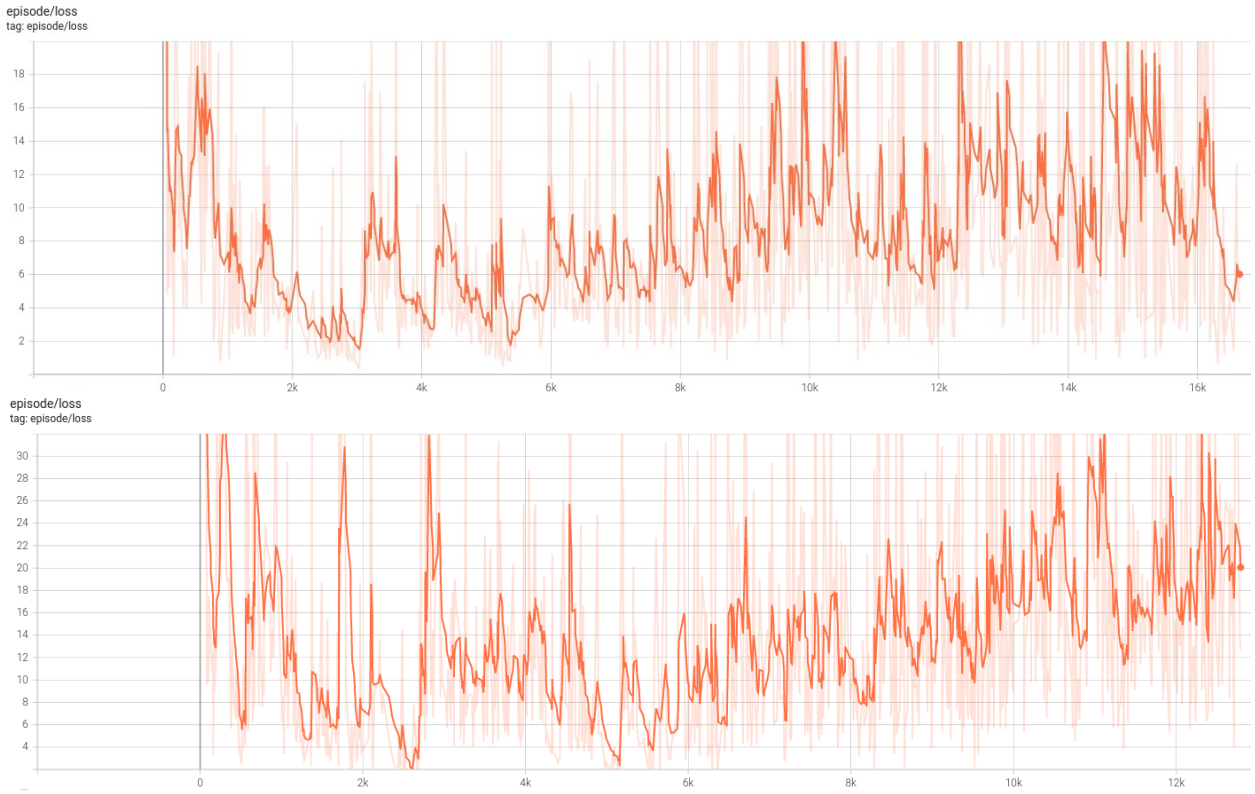


Рис. 5.10. Графіки середньоквадратичної похибки невдалих тренувань

Більш того, відмова від використання допоміжного детектора граней та шарів нормалізації призводить до мінімізації кількості спроб, що витрачаються на проходження рівня, лише до 30-50, що є значною втратою у порівнянні з кінцевим варіантом.

На додачу до архітектури, визначеної у пункті 4.2, а також інших особливостей навчання, описаних у розділі 4, в результаті тестування

виявлено набір параметрів, що виявився найбільш вдалим у процесі тренування:

- Епізодів навчання: 10000
- Розмір пам’яті агента (переходів): 90000
- Оновлення цільової мережі кожні (кількість зміни вагів): 10000
- Параметри дисконтування: 0.9
- Розмір набору даних (переходів): 32
- Параметр, що задає швидкість навчання: спадає з 0.00025 до 0.000025 протягом 500000 змін вагів
- n-крокове Q-Навчання з параметром n: 10
- Кількість останніх кадрів для представлення поточного стану: 4
- Межі для допоміжного детектору граней (Canny edge detector): 275, 300
- Параметр альфа (Коефіцієнт перегравання пам’яті): 0.6

ВИСНОВКИ ДО РОЗДІЛУ 5

В даному розділі протестовано роботу інтерфейсу взаємодії з середовищем, а також самого агента та його процесу навчання.

Для тестування інтерфейсу взаємодії підготовлений тренувальний набір даних кадрів зі гри “Geometry Dash”, що відображає знаходження персонажа у різних станах середовища. Ці дані використані для тренування моделі-детектора, що може відстежувати наявність персонажа на екрані з непоганою точністю. Використовуючи дану модель протестовано можливість інтерфейсу передавати інформацію про поточний стан середовища, а саме представлення поточного стану, нагороду за перебування у даному стані, а також можливість інтерфейсу до відокремлення різних термінальних станів.

В процесі тестування моделі-агента виконана перевірка наповнення пам’яті агента відповідною кількістю тренувальних переходів, отриманих від інтерфейсу взаємодії. Далі, протестовано сам процес навчання, що було проведено протягом 2 мільйонів змін вагів нейронної мережі. В результаті тестування отримано графіки мінімізації середньоквадратичної похибки оцінки функції якості дії, що хоч і з невеликими нестабільностями, поступово сходяться до 0. Також отримано показники кількості спроб, що агент витрачає на проходження рівня, та максимальне значення функції якості дії. З огляду на значення даних показників отримано модель-агента, що може проходити другий рівень у грі, витрачаючи на нього в середньому 3-5 спроб. Додатково протестовано вплив різних елементів системи на процес навчання та з’ясовано, що використання додаткового детектору граней для представлення поточного стану середовища та шарів нормалізації відносно набору даних у архітектурі нейронної мережі моделі-агента мають найбільший вплив на процес навчання в цілому. Шари нормалізації даних прискорюють та стабілізують процес, а використання детектору граней дає можливість спростити представлення поточного стану середовища. В якості результатів тестування представлено оптимальні параметри процесу навчання.

ВИСНОВКИ

В результаті даної роботи спроектовано та розроблено програмну систему глибокого навчання з підкріпленням для гри “Geometry Dash”.

В першому розділі проведено аналіз алгоритмів навчання з підкріпленням для знаходження оптимальної поведінки у заданому середовищі. Проведено детальний розбір математичних та алгоритмічних основ даного підходу та сформовано оптимальний алгоритм навчання, що об’єднує переваги усіх розглянутих методів: n-крокове Q-Навчання.

У другому розділі висвітлена проблема задач, з кількістю станів, що значно перевищує норму. Для вирішення цієї проблеми введений механізм апроксимації функцій, продемонстрований на прикладі нейронних мереж. Проведено аналіз використання нейронних мереж у обробці зображень та обґрунтовані переваги використання згорткових нейронних мереж. В результаті з’ясовано що для формування оцінки функції якості дії необхідно використовувати комбінацію із згорткових нейронних мереж та нейронних мереж прямого розповсюдження.

У третьому розділі спроектовано інтерфейс взаємодії з середовищем, що використовується агентом для навчання. Даний інтерфейс дозволяє отримувати представлення поточного стану середовища, використовуючи кадри зі гри “Geometry Dash”, а також відстежує можливі термінальні стани середовища та формує відповідну нагороду за перебування у кожному з них. Окрім цього, інтерфейс дає можливість за допомогою емуляції вводу не тільки управляти діями персонажа, а й керувати грою у цілому.

У четвертому розділі спроектовано модель-агента навчання з підкріпленням, а також модуль його пам’яті та процес тренування. Архітектура агента побудована відповідно до аналізу у розділі 2, тобто використовуючи комбінацію із 3 згорткових шарів, а також 3 шарів нейронних мереж прямого розповсюдження. Модуль пам’яті агента розроблено таким чином, щоб отримувати під час тренування ту інформацію, що найбільше стосується поточного стану середовища. Процес навчання

агента складається з 2 послідовних етапів: етапу наповнення пам’яті та процесу тренування. Під час тренування агент взаємодіє з середовищем та формує новий перехід та передає його на модуль пам’яті, після чого отримує новий набір переходів з якого формує відповідні оцінки функції якості дії та оптимізує середньоквадратичну похибку.

У п’ятому розділі проведено тестування розробленої системи. Виконано перевірку можливостей моделі-детектора по відстеженню персонажа на екрані та зроблено відповідні корегування. Протестовано механізм наповнення пам’яті агента та процес його тренування. В результаті тренування отримано модель, що проходить другий рівень у грі “Geometry Dash”, витрачаючи на нього в середньому 3-5 спроб. Проведено додаткове тестування та з’ясовано, що додатковий детектор граней на зображенні та шари нормалізації відносно набору даних є важливими елементами моделі-агента.

Отже, протестовано сучасні алгоритми навчання штучного інтелекту та побудована програмна система, що може проходити рівні у грі “Geometry Dash” на рівні, порівняному з рівнем людини.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Zoabi Y. Machine learning-based prediction of COVID-19 diagnosis based on symptoms / Y. Zoabi, S. Deri-Rozov, N. Shomron // Nature Partner Journals Digital Medicine, 2021. – Vol. 4, No 3. DOI:10.1038/s41746-020-00372-6.
2. Bang C.H. Automated severity scoring of atopic dermatitis patients by a deep neural network / C.H. Bang, J.W. Yoon, J.Y. Ryu *et al.* // Nature Scientific Reports, 2021. – Vol. 11, No. 6049. DOI:10.1038/s41598-021-85489-8.
3. Ramesh A. Zero-Shot Text-to-Image Generation / A. Ramesh, M. Pavlov, G. Goh *et al*. ArXiv:2102.12092.
4. Hinz T. Semantic Object Accuracy for Generative Text-to-Image Synthesis / T. Hinz, S. Heinrich, S. Wermter. ArXiv:1910.13321.
5. Kim T. Short Research on Voice Control System Based on Artificial Intelligence Assistant / T. Kim // International Conference on Electronics, Information, and Communication (ICEIC), 2020. – Barcelona, Spain. – P. 1-2. DOI:10.1109/ICEIC49074.2020.9051160.
6. Santana E. Learning a Driving Simulator / E. Santana, G. Hotz. ArXiv:1608.01230.
7. Fridman L. MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of Driver Behavior and Interaction with Automation / L. Fridman, D.E. Brown, M. Glazer *et al*. ArXiv:1711.06976.
8. Hua G. Introduction to the special section on real-world face recognition / G. Hua *et al.* // IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011. – Vol. 33, No. 10. – P. 1921-1924. DOI:10.1109/TPAMI.2011.182.
9. David E. DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess / E. David, N.S. Netanyahu, L. Wolf. ArXiv:1711.09667.

10. Huang G.B. Learning hierarchical representations for face verification with convolutional deep belief networks / G.B. Huang, H. Lee, E. Learned-Miller // IEEE Conference on Computer Vision and Pattern Recognition, 2012. – USA, Rhode Island, Providence. – P. 2518-2525. DOI:10.1109/CVPR.2012.6247968.

11. Brahma P. P. Why Deep Learning Works: A Manifold Disentanglement Perspective / P.P. Brahma, D. Wu, Y. She // IEEE Transactions on Neural Networks and Learning Systems, 2016. – Vol. 27, No. 10. – P. 1997-2008. DOI:10.1109/TNNLS.2015.2496947.

12. O'Mahony N. Deep Learning vs. Traditional Computer Vision / N. O'Mahony, S. Campbell, A. Carvalho *et al.* ArXiv:1910.13796.

13. Sutton R.S. Reinforcement Learning: An Introduction. Second edition / R.S. Sutton, A.G. Barto. – Cambridge, Massachusetts. London, England. 2018. – P. 58.

14. Bellman R. The theory of dynamic programming / R. Bellman // Bulletin of the American Mathematical Society, 1954. – Vol. 60, No. 6. – P. 503-515. DOI:10.1090/S0002-9904-1954-09848-8.

15. Liu D. Generalized Policy Iteration Adaptive Dynamic Programming for Discrete-Time Nonlinear Systems / D. Liu, Q. Wei, P. Yan // IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2015. – Vol. 45, No. 12. – P. 1577-1591. DOI:10.1109/TSMC.2015.2417510.

16. Metropolis N. The Monte Carlo Method / N. Metropolis, S. Ulam // Journal of the American Statistical Association, 1949. – Vol. 44, No. 247. – P. 335-341. DOI:10.2307/2280232

17. Sutton R.S. Learning to predict by the methods of temporal differences / R.S. Sutton // Machine Learning, 1988. – Vol. 3, No. 1. – P. 9-44. DOI:10.1007/BF00115009.

18. Mete A. Reward Biased Maximum Likelihood Estimation for Reinforcement Learning / A. Mete, R. Singh, P.R. Kumar. ArXiv:2011.07738.

19. Watkins C. *Q*-learning / C. Watkins, P. Dayan // Machine Learning, 1992. – Vol. 8, No. 3-4. – P. 279-292. DOI:10.1007/BF00992698.

20. Francisco S. M. Convergence of Q-learning: A simple proof.

URL: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

21. Watkins C. Learning From Delayed Rewards. PhD Thesis / Cambridge University. Cambridge, England, 1989. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

22. Tesauro G. TD-Gammon: A Self-Teaching Backgammon Program / G. Tesauro // F. A. Murray. Application of Neural Networks, 1995. – Boston, Massachusetts - P. 267-285. DOI:10.1007/978-1-4757-2379-3_11.

23. Silver D. Mastering the game of Go without human knowledge / D. Silver, J. Schrittwieser, K. Simonyan *et al.* // Nature, 2017. – Vol. 550, No. 7676. – P. 354-359. DOI:10.1038/nature24270.

24. Ruder S. An overview of gradient descent optimization algorithms / S. Ruder. ArXiv:1609.04747.

25. Mohamed S. Monte Carlo Gradient Estimation in Machine Learning / S. Mohamed, M. Rosca, M. Figurnov, A. Mnih. ArXiv:1906.10652.

26. Etienne B. Temporal-Difference Methods and Markov Models / B. Etienne // IEEE Transactions on Systems, Man, and Cybernetics, 1993. – Vol. 23, No. 2. – P. 357-365. DOI:10.1109/21.229449.

27. Brandfonbrener D. Geometric Insights into the Convergence of Nonlinear TD Learning / D. Brandfonbrener, J. Bruna. ArXiv:1905.12185.

28. Freund Y. Large Margin Classification Using the Perceptron Algorithm / Y. Freund, R.E. Schapire // Machine Learning, 1999. – Vol. 37, No. 3. – P. 277-296. DOI:10.1023/A:1007662407062.

29. Agarap A. F. Deep Learning using Rectified Linear Units (ReLU) / A. F. Agarap. ArXiv:1803.08375.

30. Bebis G. Feed-forward neural networks / G. Bebis, M. Georgiopoulos // IEEE Potentials, 1994. – Vol. 13, No. 4. – P. 27-31. DOI:10.1109/45.329294.

31. Steinbach B. Neural Networks - A Model of Boolean Functions / B. Steinbach, R. Kohut. URL: https://www.researchgate.net/profile/Bernd-Steinbach/publication/246931125_Neural_Networks_-_A_Model_of_Boolean_Functions/links/53ecdac30cf2981ada10f470/Neural-Networks-A-Model-of-Boolean-Functions.pdf.

32. Cybenko G. Approximation by superpositions of a sigmoidal function / G. Cybenko // Mathematics of Control, Signals and Systems, 1989. – Vol. 2, No. 4. – P. 303-314. DOI:10.1007/BF02551274.

33. Ilin R. Abstraction hierarchy in deep learning neural networks / R. Ilin, T. Watson, R. Kozma // International Joint Conference on Neural Networks (IJCNN), 2017. – USA, Alaska, Anchorage. – P. 768-774. DOI:10.1109/IJCNN.2017.7965929.

34. Lecun Y. A Theoretical Framework for Back-Propagation / Y. Lecun. URL: https://www.researchgate.net/profile/Yann-Lecun/publication/2360531_A_Theoretical_Framework_for_Back-Propagation/links/0deec519dfa297eac1000000/A-Theoretical-Framework-for-Back-Propagation.pdf.

35. Fan J. A Theoretical Analysis of Deep Q-Learning / J. Fan, Z. Wang, Y. Xie, Z. Yang. ArXiv:1901.00137

36. Daday M. Enhancing Feed-Forward Neural Network in Image Classification / M. Daday, A. Fajardo, R. Medina // Proceedings of the 2nd International Conference on Computing and Big Data, 2019. – Republic od China, Taiwan, Taichung. – P. 86-90. DOI:10.1145/3366650.3366651.

37. Ying X. An Overview of Overfitting and its Solutions / X. Ying // Journal of Physics: Conference Series, 2019. – Vol. 1168, No. 2. DOI:10.1088/1742-6596/1168/2/022022.

38. Krizhevsky A. ImageNet Classification with Deep Convolutional Neural Networks / A. Krizhevsky, I. Sutskever, G. Hinton // Communications of the ACM, 2017. – Vol. 60, No. 6. – P. 84-90. DOI:10.1145/3065386.

39. Chang J. Learning representations of emotional speech with deep convolutional generative adversarial networks / J. Chang, S. Scherer // IEEE International Conference on Acoustics, Speech and Signal Processing, 2017. – USA, LA, New Orleans. – P. 2746-2750. DOI:10.1109/ICASSP.2017.7952656.

40. Bashivan P. Learning Representations from EEG with Deep Recurrent-Convolutional Neural Networks / P. Bashivan, I. Rish, M. Yeasin, N. Codella. ArXiv:1511.06448.

41. Mnih V. Human-level control through deep reinforcement learning / V. Mnih, K. Kavukcuoglu, D. Silver *et al.* // Nature, 2015. – Vol. 518, No. 7540. – P. 529-533. DOI:10.1038/nature14236.

Додатки

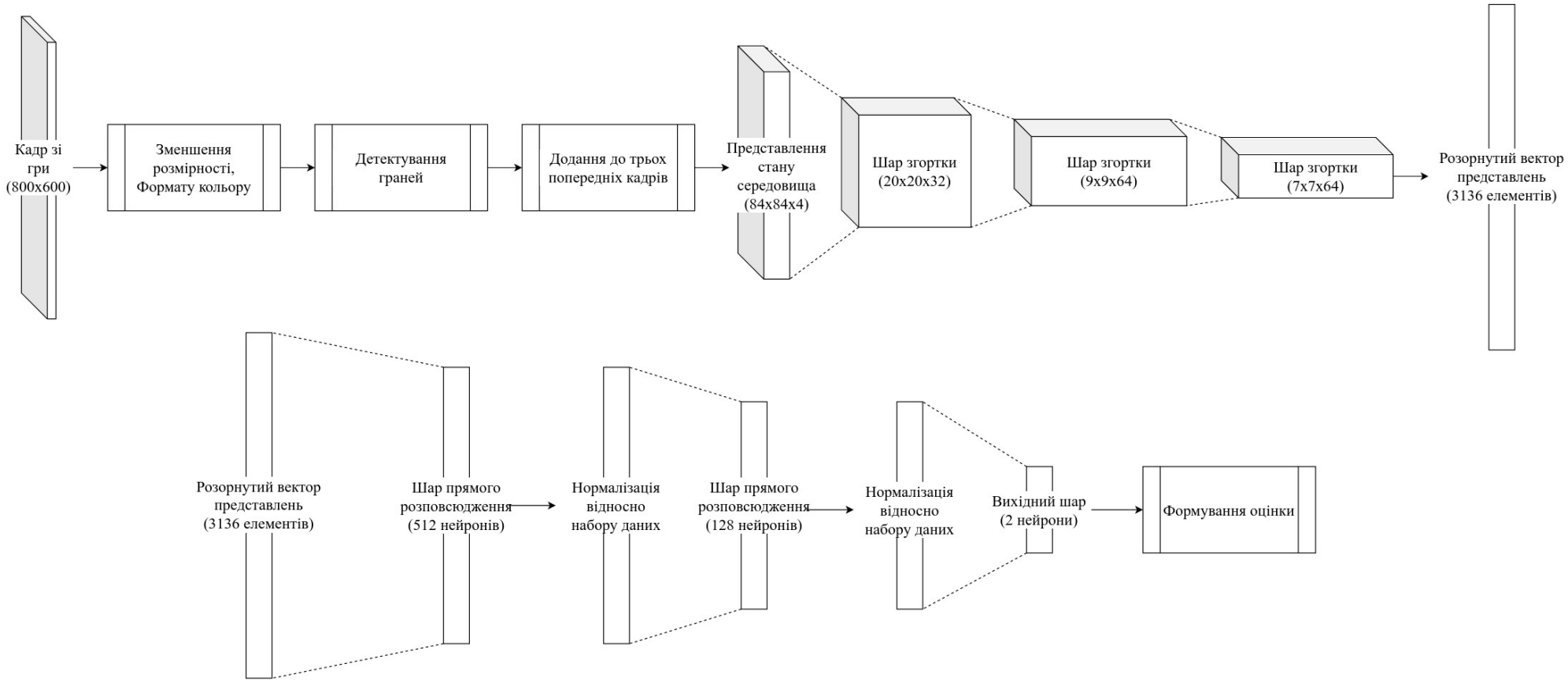
ДОДАТОК 1

Програмна система глибокого навчання з підкріпленням для гри “Geometry Dash”

Схема структурна: схема формування оцінки функії якості дії

Аркушів 1

2021



					ІАЛЦ.467100.003 Д1	
Змн.	Арк.	№ докум.	Підпис	Дата	Схема структурна Схема формування оцінки функії якості дії	Літ. / Маса / Масш.
Розроб.		Тутевич В.Є.				
Перевір.		Новотарський М.А.				Аркуш 1 Аркушів 1
Т. Контр.						
					Дипломна робота	НТУУ “КПІ” ФІОТ ІО-71
Н. Контр.		Сімоненко В.П.				
Затверд.		Стіренко С.Г.				

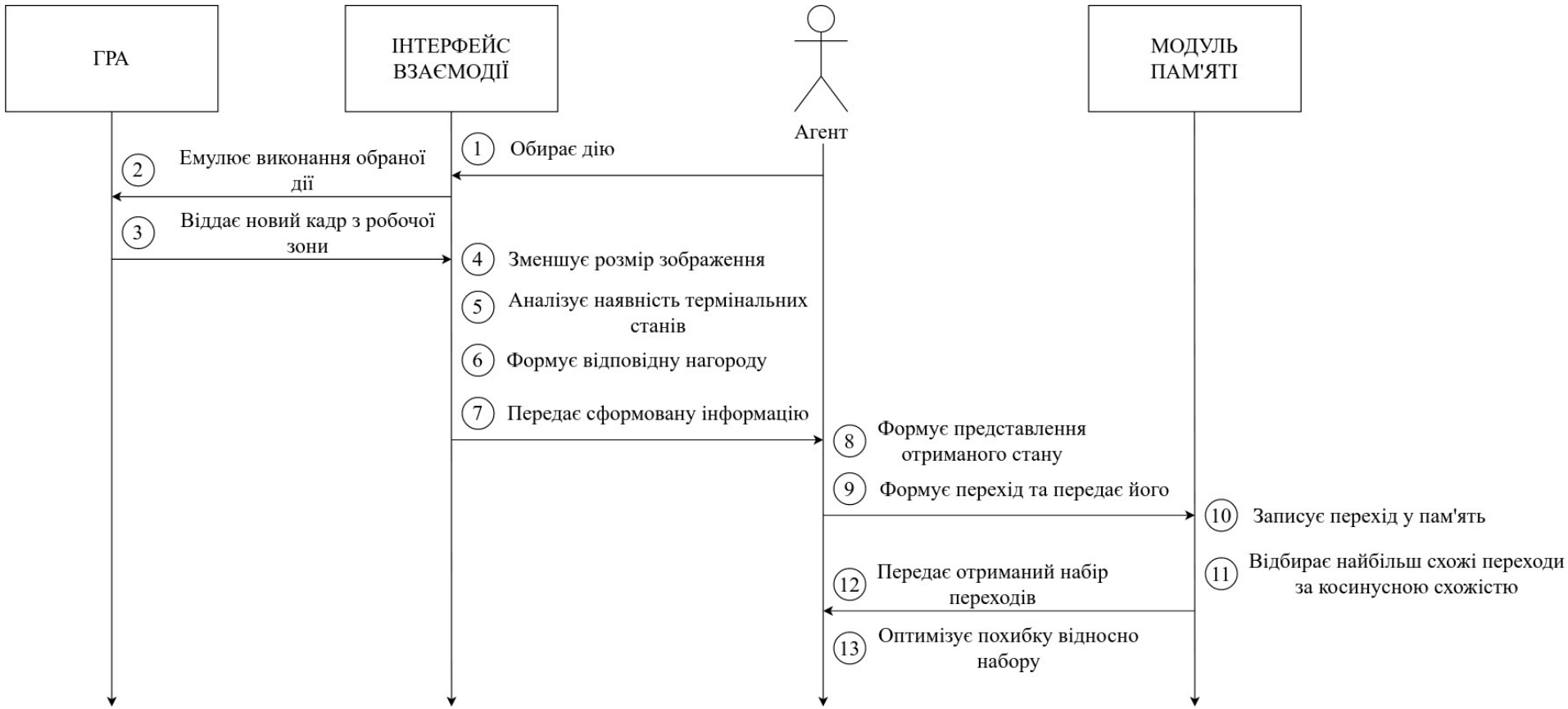
ДОДАТОК 2

Програмна система глибокого навчання з підкріпленням для гри “Geometry Dash”

Схема функціональна: діаграма послідовності кроку тренування

Аркушів 1

2021



					ІАЛЦ.467100.004 Д2		
					Схема функціональна Діаграма послідовності кроку тренування	Літ.	Маса / Масш.
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Тутевич В.Є.				Аркуш 1	Аркушів 1
Перевір.		Новотарський М.А.					
Т. Контр.							
Н. Контр.		Сімоненко В.П.			Дипломна робота	НТУУ “КПІ” ФІОТ ІО-71	
Затверд.		Стіренко С.Г.					

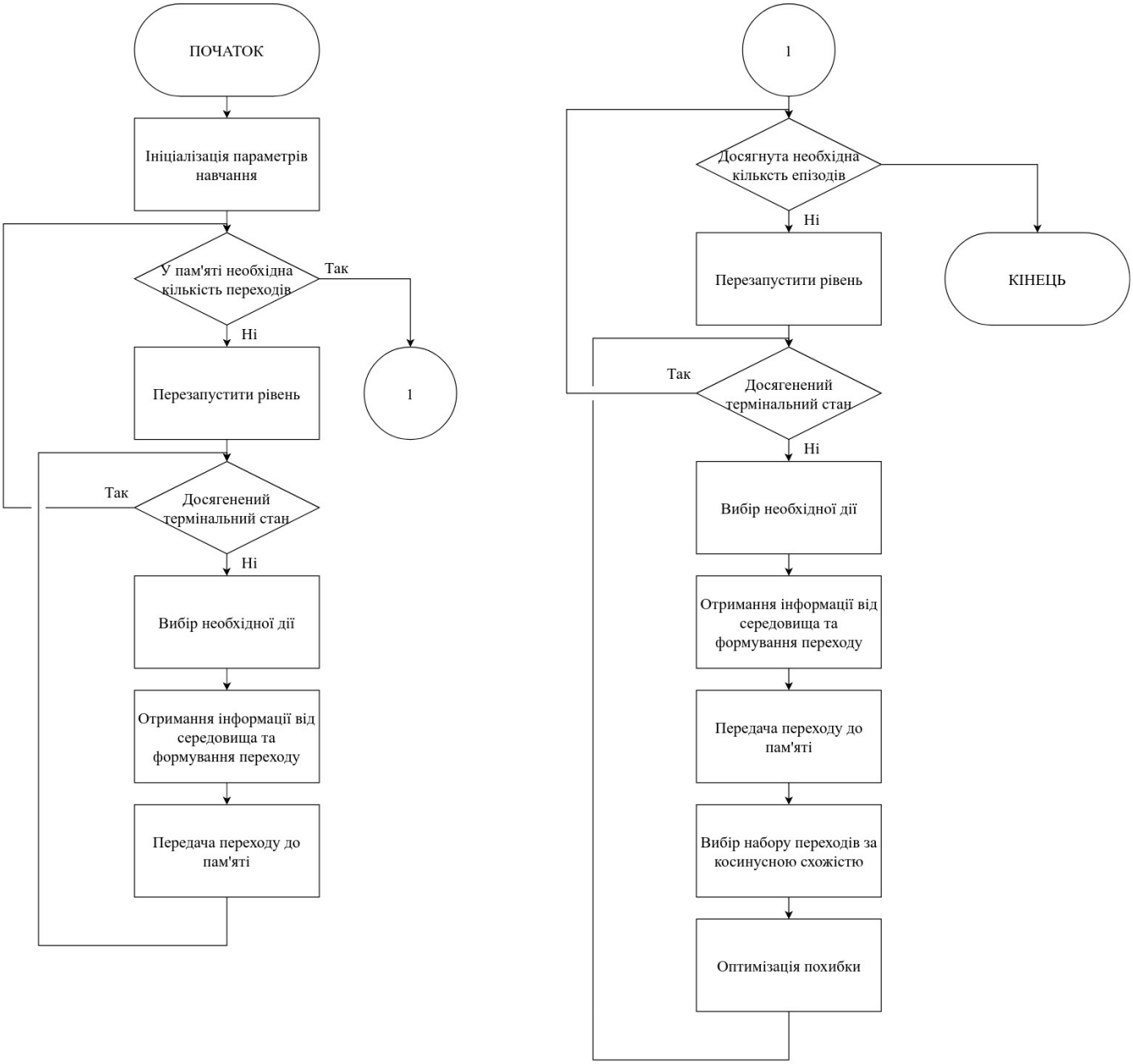
ДОДАТОК З

Програмна система глибокого навчання з підкріпленням для гри “Geometry Dash”

Схема принципова: схема алгоритму тренування

Аркушів 1

2021



					ІАЛЦ.467100.005 Д3		
					Схема принципова Схема алгоритму тренування	*Літ.*	*Маса* / *Масш.*
Змн.	Арк.	№ докум.	*Підпис*	*Дата*			
Розроб.		Тутевич В.Є.				*Аркуш* 1	*Аркушів* 1
Перевір.		Новотарський М.А.					
Т. Контр.							
Н. Контр.		Сімоненко В.П.			Дипломна робота	НТУУ “КПІ” ФІОТ ІО-71	
Затверд.		Стіренко С.Г.					

ДОДАТОК 4

Програмна система глибокого навчання з підкріпленням для гри “Geometry Dash”

Лістинг програми

Аркушів 15

2021

agent.py

```
from itertools import count
import numpy as np
import os
import tensorflow.compat.v1 as tf
import gd_utils
from random import sample as external_sample
import time
import cv2
from collections import namedtuple
import shelve
import keyboard
tf.disable_v2_behavior()
os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'
Transition = namedtuple("Transition", ["state", "action", "rewards", "next_state",
"done"])


class RelatedMemoryReplay:
    def __init__(self, feature_shape, memory_len=75_000, file_name='replay_memory'):
        self.feature_vectors = np.zeros(shape=(memory_len, feature_shape),
dtype=np.float32)
        self.priorities = np.empty(shape=(memory_len), dtype=np.float32)
        self.memory = []
        self.max_priority = 1.0
        if os.path.exists(os.path.join(os.path.curdir, f"{file_name}.dat")):
            self.load(file_name)
            self.BASE_MEMORY = len(self.memory)
            self.priorities[:self.BASE_MEMORY] = np.array([self.max_priority for _ in
range(self.BASE_MEMORY)])
        else:
            self.BASE_MEMORY = 0
        self.index = 0
        self.MAX_ELEMENTS_VOLATILE = memory_len - self.BASE_MEMORY
        self.v = tf.placeholder(tf.float32, shape=feature_shape)
        self.vs = tf.placeholder(tf.float32, shape=self.feature_vectors.shape)
        normalize_v = tf.nn.l2_normalize(self.v, 0)
        normalize_vs = tf.nn.l2_normalize(self.vs, 1)
        self.cos_similarity = tf.math.divide(tf.math.add(1.0,
tf.reduce_sum(tf.multiply(normalize_v, normalize_vs), axis=1)), 2.0)

    def __len__(self):
        return len(self.memory)

    def add(self, feature_vec, transition):
        self.index %= self.MAX_ELEMENTS_VOLATILE
        index = self.index + self.BASE_MEMORY
        if len(self.memory) < self.MAX_ELEMENTS_VOLATILE + self.BASE_MEMORY:
            self.memory.append(transition)
        else:
            self.memory[index] = transition
        self.feature_vectors[index] = feature_vec
        self.priorities[index] = self.max_priority
        self.index += 1

    def sample_like(self, feature_vec, batch_size: int, sess, alpha=0.6):
        similarities = sess.run(self.cos_similarity,
                                feed_dict={self.v: feature_vec, self.vs:
self.feature_vectors})
        ixs = np.arange(self.MAX_ELEMENTS_VOLATILE + self.BASE_MEMORY)
[:len(self.memory)]
        similarities = similarities[:len(self.memory)]
```

```
        similarities = similarities ** alpha
        similarities /= np.sum(similarities)
        ixs = np.random.choice(ixs, batch_size, p=similarities)
        return [self.memory[i] for i in ixs], ixs

    def rebase(self, estimator, sess):
        for i, entry in enumerate(self.memory):
            self.feature_vectors[i] = estimator.get_feature_vector(sess, entry.state)

    def sample_random(self, batch_size: int):
        ixs_random = external_sample(range(len(self.memory)), batch_size)
        return [self.memory[i] for i in ixs_random]

    def save(self, file_name="replay_memory"):
        with shelve.open(file_name) as sh:
            sh["memory"] = self.memory
            sh["feature_vectors"] = self.feature_vectors

    def load(self, file_name="replay_memory"):
        with shelve.open(file_name) as sh:
            self.memory = sh["memory"]
            loaded_vectors = sh["feature_vectors"]
        self.feature_vectors[:len(self.memory)] = loaded_vectors[:len(self.memory)]


class Estimator:
    """Q-Value Estimator neural network.

    This network is used for both the Q-Network and the Target Network.
    """

    def __init__(self, scope="estimator", summaries_dir=None):
        self.scope = scope
        # Writes Tensorboard summaries to disk
        self.summary_writer = None
        with tf.variable_scope(scope):
            # Build the graph
            self._build_model()
            if summaries_dir:
                summary_dir = os.path.join(summaries_dir,
"summaries_{}".format(scope))
                if not os.path.exists(summary_dir):
                    os.makedirs(summary_dir)
                self.summary_writer = tf.summary.FileWriter(summary_dir)

    def _build_model(self):
        """
        Builds the Tensorflow graph.
        """

        # Placeholders for our input
        # Our input are 4 grayscale frames of shape 84, 84 each
        self.X_pl = tf.placeholder(shape=[None, 84, 84, 4], dtype=tf.uint8, name="X")
        # The TD target value
        self.y_pl = tf.placeholder(shape=[None], dtype=tf.float32, name="y")
        # Integer id of which action was selected
        self.actions_pl = tf.placeholder(shape=[None], dtype=tf.int32, name="actions")
        self.lr = tf.placeholder(tf.float32, shape=[])
        self.training = tf.compat.v1.placeholder_with_default(True, shape=[])

        X = tf.to_float(self.X_pl) / 255.0
        batch_size = tf.shape(self.X_pl)[0]

        # Three convolutional layers
```

```
        conv1 = tf.keras.layers.Conv2D(32, 8, 4, activation=tf.nn.relu,
kernel_initializer='glorot_normal')(X)
        conv2 = tf.keras.layers.Conv2D(64, 4, 2, activation=tf.nn.relu,
kernel_initializer='glorot_normal')(conv1)
        conv3 = tf.keras.layers.Conv2D(64, 3, 1, activation=tf.nn.relu,
kernel_initializer='glorot_normal')(conv2)

        # Fully connected layers
        flattened = tf.keras.layers.Flatten()(conv3)
        self.fc1_l = tf.keras.layers.Dense(512, activation=tf.nn.relu,
                                           kernel_initializer='glorot_normal',
                                           )
        self.fc1_o  = self.fc1_l(flattened)
        self.bn1 = tf.keras.layers.BatchNormalization()(self.fc1_o,
training=self.training)
        self.fc2_l = tf.keras.layers.Dense(128, activation=tf.nn.relu,
                                           kernel_initializer='glorot_normal',
                                           )
        self.fc2_o  = self.fc2_l(self.bn1)
        self.bn2 = tf.keras.layers.BatchNormalization()(self.fc2_o,
training=self.training)
        self.predictions_l = tf.keras.layers.Dense(len(VALID_ACTIONS),
activation=None,
                                           kernel_initializer='glorot_normal',
                                           )
        self.predictions = self.predictions_l(self.bn2)

        # Get the predictions for the chosen actions only
        gather_indices = tf.range(batch_size) * tf.shape(self.predictions)[1] +
self.actions_pl
        self.action_predictions = tf.gather(tf.reshape(self.predictions, [-1]),
gather_indices)

        # Calculate the loss
        self.td_errors = tf.subtract(self.y_pl, self.action_predictions)
        self.losses = tf.squared_difference(self.y_pl, self.action_predictions)
        self.loss = tf.reduce_mean(self.losses)

        # Optimizer Parameters from original paper
        self.optimizer = tf.train.RMSPropOptimizer(self.lr, 0.99, 0.0, 1e-6)
        self.train_op = self.optimizer.minimize(self.loss,
global_step=tf.train.get_global_step())

        # Summaries for Tensorboard

        self.summaries = tf.summary.merge([
            tf.summary.scalar("loss", self.loss),
            tf.summary.histogram("loss_hist", self.losses),
            tf.summary.histogram("q_values_hist", self.predictions),
            tf.summary.scalar("max_q_value", tf.reduce_max(self.predictions))
        ])

    def predict(self, sess, s, training=True):
        """
        Predicts action values.

        Args:
          sess: Tensorflow session
          s: State input of shape [batch_size, 4, 120, 120, 1]

        Returns:
          Tensor of shape [batch_size, NUM_VALID_ACTIONS] containing the estimated
          action values.
        """
```

```
        return sess.run(self.predictions, {self.X_pl: s, self.training: training})

    def get_feature_vector(self, sess, s):
        feature_vec = sess.run(self.bn2, {self.X_pl: np.expand_dims(s, 0)})
        return np.squeeze(feature_vec)

    def calc_loss(self, sess, s, a, y):
        feed_dict = {self.X_pl: s, self.y_pl: y, self.actions_pl: a}
        loss = sess.run(
            [self.loss],
            feed_dict)
        return loss[0]

    def update(self, sess, s, a, y, lr):
        """
        Updates the estimator towards the given targets.

        Args:
          sess: Tensorflow session object
          s: State input of shape [batch_size, 4, 120, 120, 1]
          a: Chosen actions of shape [batch_size]
          y: Targets of shape [batch_size]

        Returns:
          The calculated loss on the batch.
        """

        feed_dict = {self.X_pl: s, self.y_pl: y, self.actions_pl: a, self.lr: lr}
        summaries, global_step,  _, loss, td_errors = sess.run(
            [self.summaries, tf.train.get_global_step(), self.train_op, self.loss,
self.td_errors],
            feed_dict)
        if self.summary_writer:
            self.summary_writer.add_summary(summaries, global_step)
        return loss, td_errors


class ModelParametersCopier:
    """
    Copy model parameters of one estimator to another.
    """

    def __init__(self, estimator1, estimator2):
        """
        Defines copy-work operation graph.
        Args:
          estimator1: Estimator to copy the paramters from
          estimator2: Estimator to copy the parameters to
        """
        e1_params = [t for t in tf.trainable_variables() if
t.name.startswith(estimator1.scope)]
        e1_params = sorted(e1_params, key=lambda v: v.name)
        e2_params = [t for t in tf.trainable_variables() if
t.name.startswith(estimator2.scope)]
        e2_params = sorted(e2_params, key=lambda v: v.name)

        self.update_ops = []
        for e1_v, e2_v in zip(e1_params, e2_params):
            op = e2_v.assign(e1_v)
            self.update_ops.append(op)

    def make(self, sess):
        """
        Makes copy.
```

```
        Args:
            sess: Tensorflow session instance
        """
        sess.run(self.update_ops)


def make_epsilon_greedy_policy(estimator, nA):
    """
    Creates an epsilon-greedy policy based on a given Q-function approximator and
epsilon.

    Args:
        estimator: An estimator that returns q values for a given state
        nA: Number of actions in the environment.

    Returns:
        A function that takes the (sess, observation, epsilon) as an argument and
returns
        the probabilities for each action in the form of a numpy array of length nA.

    """

    def policy_fn(sess, observation, epsilon, batch_size, training=True):
        if batch_size:
            A = np.ones((batch_size, nA), dtype=float) * epsilon / nA
            q_values = estimator.predict(sess, observation, training)
            best_actions = np.argmax(q_values, axis=1)
            A[np.arange(batch_size), best_actions] += (1.0 - epsilon)
        else:
            A = np.ones(nA, dtype=float) * epsilon / nA
            q_values = estimator.predict(sess, np.expand_dims(observation, 0),
training)[0]
            best_action = np.argmax(q_values)
            A[best_action] += (1.0 - epsilon)
        return A, q_values

    return policy_fn


def deep_q_learning(sess,
                    env,
                    q_estimator,
                    target_estimator,
                    num_episodes,
                    experiment_dir,
                    replay_memory_size=50_000,
                    replay_memory_init_size=10_000,
                    update_target_estimator_every=10000,
                    discount_factor=0.99,
                    epsilon_start=0.1,
                    epsilon_end=0.0001,
                    epsilon_decay_steps=300_000,
                    batch_size=32):
    """
    N(10)-step Q-Learning algorithm for off-policy TD control using Function
Approximation.
    Finds the optimal greedy policy while following an epsilon-greedy policy.

    Args:
        sess: Tensorflow Session object
        env: Geometry Dash environment emulator
        q_estimator: Estimator object used for the q values
        target_estimator: Estimator object used for the targets
        num_episodes: Number of episodes to run for
```

```
        experiment_dir: Directory to save Tensorflow summaries in
        replay_memory_size: Size of the replay memory
        replay_memory_init_size: Number of random experiences to sample when
initializing
          the replay memory.
        update_target_estimator_every: Copy parameters from the Q estimator to the
          target estimator every N steps
        discount_factor: Gamma discount factor
        epsilon_start: Chance to sample a random action when taking an action.
          Epsilon is decayed over time and this is the start value (Not used)
        epsilon_end: The final minimum value of epsilon after decaying is done (Not
used)
        epsilon_decay_steps: Number of steps to decay epsilon over (Not used)
        batch_size: Size of batches to sample from the replay memory

    Returns:
        Yields tuple of (Episode loss, Number of parameter updates) at the end of
every episode.
    """

    replay_memory = RelatedMemoryReplay(128, replay_memory_size)
    copier = ModelParametersCopier(q_estimator, target_estimator)

    # Create directories for checkpoints and summaries
    checkpoint_dir = os.path.join(experiment_dir, "checkpoints")
    checkpoint_path = os.path.join(checkpoint_dir, "model")

    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)

    if not os.path.exists(videos_dir):
        os.makedirs(videos_dir)

    saver = tf.train.Saver()
    # Load a previous checkpoint if we find one
    latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
    if latest_checkpoint:
        print("Loading model checkpoint {}...\n".format(latest_checkpoint))
        saver.restore(sess, latest_checkpoint)

    # Get the current time step
    total_t = sess.run(tf.train.get_global_step())

    # The learning rate decay schedule
    lrs = np.linspace(0.00025, 0.000025, 500_000)

    epsilon = 0.1
    # The policy we're following
    policy = make_epsilon_greedy_policy(
        q_estimator,
        len(VALID_ACTIONS))

    print("Populating replay memory...")
    n_step = 10
    while len(replay_memory) < replay_memory_init_size:
        env.retry()
        done = False
        elapsed_time = time.perf_counter()
        state, _, _, fr, _, _ = env.step(0)
        state = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)
        state = cv2.Canny(state, threshold1=275, threshold2=300)
        state = np.stack([state] * 4, axis=2)

        rewards_window = []
```

```
        states_window = []
        actions_window = []
        feature_vecs = []
        for t in count(start=1):
            # Populate replay memory!
            action_probs, _ = policy(sess, state, epsilon, None, training=False)
            action = np.random.choice(VALID_ACTIONS, p=action_probs)
            next_frame, reward, done, fr, _, _ = env.step(action)
            feature_vec = target_estimator.get_feature_vector(sess, state)

            rewards_window.append(reward)
            states_window.append(state)
            actions_window.append(action)
            feature_vecs.append(feature_vec)

            next_frame_g = cv2.cvtColor(next_frame, cv2.COLOR_RGB2GRAY)
            image = cv2.Canny(next_frame_g, threshold1=275, threshold2=300)
            next_state = np.append(state[:, :, 1:], np.expand_dims(image, axis=2),
axis=2)

            if len(rewards_window) == n_step:
                replay_memory.add(feature_vecs[0],
                                  Transition(states_window[0], actions_window[0],
rewards_window[:], next_state, done),
                                  )
                states_window.pop(0)
                rewards_window.pop(0)
                actions_window.pop(0)
                feature_vecs.pop(0)

            if done:
                while len(rewards_window) != 0:
                    replay_memory.add(feature_vecs[0],
                                      Transition(states_window[0], actions_window[0],
                                                 rewards_window[:] + [0 for _ in
range(n_step - len(rewards_window))],
                                                 next_state, done),
                                      )
                    states_window.pop(0)
                    rewards_window.pop(0)
                    actions_window.pop(0)
                    feature_vecs.pop(0)
                print("\n" + "-" * 50)
                break
            else:
                state = next_state
                time.sleep(1.0 / 70)

        elapsed_time = time.perf_counter() - elapsed_time
        fps =  fr / elapsed_time
        print(f"Replay memory length: {len(replay_memory)}")
        print(f"Framerate: {fps} | frames: {fr}")

    print("Training started!")
    n_step_discount = np.array([discount_factor ** i for i in range(n_step)])
    for i_episode in range(num_episodes):
        # Reset the environment
        env.retry()
        ep_reward = 0
        elapsed_time = time.perf_counter()
        state, _, _, fr, reached, attempts = env.step(0)
        state = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)
        state = cv2.Canny(state, threshold1=275, threshold2=300)
        state = np.stack([state] * 4, axis=2)
```

```
        frames_to_write = [env.record_frame]

        loss = 0
        update_num = 0.001
        done = False
        rewards_window = []
        states_window = []
        actions_window = []
        feature_vecs = []

        # One step in the environment
        for t in count(start=1):
            # Maybe update the target estimator
            if (total_t + 1) % update_target_estimator_every == 0:
                env.pause()
                copier.make(sess)
                env.unpause()

            # Take a step in the environment
            action_probs, action_values = policy(sess, state, 0, None)
            action = np.random.choice(VALID_ACTIONS, p=action_probs)
            next_frame, reward, done, fr, reached, attempts = env.step(action)
            feature_vec = target_estimator.get_feature_vector(sess, state)

            rewards_window.append(reward)
            states_window.append(state)
            actions_window.append(action)
            feature_vecs.append(feature_vec)

            if (i_episode + 1) % 100 == 0:
                frames_to_write.append(env.record_frame)
            print(f"\rAction values: {action_values}, total_t: {total_t}", end="")

            # Save transition to replay memory
            next_frame_g = cv2.cvtColor(next_frame, cv2.COLOR_RGB2GRAY)
            image = cv2.Canny(next_frame_g, threshold1=275, threshold2=300)
            next_state = np.append(state[:, :, 1:], np.expand_dims(image, axis=2),
axis=2)

            if len(rewards_window) == n_step:
                replay_memory.add(feature_vecs[0],
                                  Transition(states_window[0], actions_window[0],
rewards_window[:], next_state,
                                             done),
                                  )
                states_window.pop(0)
                rewards_window.pop(0)
                actions_window.pop(0)
                feature_vecs.pop(0)
            ep_reward += reward

            # Sample related transitions from memory replay
            samples_rlt, ixs = replay_memory.sample_like(feature_vec, batch_size,
sess, alpha=0.6)
            states_batch, action_batch, reward_batch, next_states_batch, done_batch =
map(np.array, zip(*samples_rlt))

            # This is where Double Q-Learning comes in!
            q_values_next = q_estimator.predict(sess, next_states_batch)
            best_actions = np.argmax(q_values_next, axis=1)
            q_values_next_target = target_estimator.predict(sess, next_states_batch)
            td_targets = np.dot(reward_batch, n_step_discount) + (1 - done_batch) * \
                         discount_factor ** n_step *
q_values_next_target[np.arange(len(best_actions)), best_actions]
```

```
            # Learning rate decays until 500k parameter updates
            lr = lrs[min(499_999, total_t)]
            batch_loss, _ = q_estimator.update(sess, states_batch, action_batch,
td_targets, lr)
            loss += batch_loss
            update_num += 1
            total_t += 1

            if done:
                while len(rewards_window) != 0:
                    replay_memory.add(feature_vecs[0],
                                      Transition(states_window[0], actions_window[0],
                                                 rewards_window[:] + [0 for _ in
range(n_step - len(rewards_window))],
                                                 next_state, done),
                                      )
                    states_window.pop(0)
                    rewards_window.pop(0)
                    actions_window.pop(0)
                    feature_vecs.pop(0)
                print("\n" + "-" * 50)
                break
            else:
                state = next_state

        loss /= update_num
        elapsed_time = time.perf_counter() - elapsed_time
        fps = fr / elapsed_time
        print(f"\nEpisode's framerate: {fps}")
        print(f"Replay memory len : {len(replay_memory)}")

        if (i_episode + 1) % 25 == 0:
            env.pause()
            # Save the current checkpoint
            saver.save(tf.get_default_session(), checkpoint_path)
            # Write episode to video every 100 episodes:
            if (i_episode + 1) % 100 == 0:
                height, width, _ = frames_to_write[0].shape
                output = cv2.VideoWriter(os.path.join(videos_dir, f'episode{i_episode
+ 1}.avi'),
                                         cv2.VideoWriter_fourcc(*'DIVX'), 20, (width,
height))
                for frame in frames_to_write:
                    output.write(frame.astype('uint8'))
                output.release()
            env.unpause()

        # Add summaries to tensorboard
        episode_summary = tf.Summary()
        if done and reached:
            episode_summary.value.add(simple_value=attempts, tag="episode/attempts")
        episode_summary.value.add(simple_value=ep_reward, tag="episode/reward")
        episode_summary.value.add(simple_value=fr, tag="episode/length")
        episode_summary.value.add(simple_value=loss, tag="episode/loss")
        q_estimator.summary_writer.add_summary(episode_summary, i_episode)
        q_estimator.summary_writer.flush()
        yield total_t, loss

    return 0


env = gd_utils.GDenv((800, 600), mode="practice")
VALID_ACTIONS = [0, 1]
```

```
tf.reset_default_graph()
config = tf.ConfigProto()
config.gpu_options.allow_growth = True

# Where we save our checkpoints and graphs
experiment_dir = os.path.abspath("./experiments/Geometry Dash")
videos_dir = os.path.abspath("./videos")

# Create a glboal step variable
global_step = tf.Variable(0, name='global_step', trainable=False)

# Create estimators
q_estimator = Estimator(scope="q", summaries_dir=experiment_dir)
target_estimator = Estimator(scope="target_q")

# Run it!
with tf.Session(config=config) as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    for i, (t, ep_reward) in enumerate(deep_q_learning(sess,
                                                       env,
                                                       q_estimator=q_estimator,

target_estimator=target_estimator,
                                                       experiment_dir=experiment_dir,
                                                       num_episodes=10_000,
                                                       replay_memory_size=90_000,
                                                       replay_memory_init_size=1_000,

update_target_estimator_every=10_000,
                                                       discount_factor=0.9,
                                                       batch_size=32)
                                            ):
        print(f"Episode's #{i + 1} loss: {ep_reward}")
print("Training done!")
```

gd_utils.py

```
import pyautogui as pygui
from os.path import join, normpath
from collections import namedtuple
from random import shuffle
import cv2
import time
import numpy as np
from PIL import ImageGrab
from xdo import Xdo
import keyboard
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
Region = namedtuple("Region", ['x', 'y', 'width', 'height'])
res_dir = normpath(r"/home/scream/GD_RL/res")


class GDenv:
    def __init__(self, resolution, mode="usual"):
        self.region_corner = Region(*pygui.locateOnScreen(join(res_dir,
"gd_upper_new_mnjr.png")))
        self.region_main = Region(self.region_corner.x, self.region_corner.y +
self.region_corner.height, *resolution)
        self.region_main = dict(zip(('left', 'top', 'width', 'height'),
[*self.region_main]))
        pygui.click(self.region_corner)
```

```
        self.retry_img = cv2.imread(join(res_dir, "retry.png"))
        self.progress_bar_img = cv2.cvtColor(cv2.imread(join(res_dir,
"progress_bar.png")), cv2.COLOR_RGB2GRAY)
        self.retry_pos = (self.region_main['left'] + 230, self.region_main['top'] +
500)
        self.fr = 0
        self.record_frame = None
        self.pressed = False
        self.mode = mode
        self.reached = True
        self.xdo = Xdo()
        self.current_level = 0
        self.levels = [1, 2, 4, 5, 6, 7]
        self.window = self.xdo.get_focused_window()
        if mode == "practice":
            self.practice_pos = (self.region_main['left'] + 230,
self.region_main['top'] + 350)
            self.attempt_counter = 0
            MODEL_NAME = '/home/scream/GD_RL/src/detection_model/result_model'
            PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'
            detection_graph = tf.Graph()
            with detection_graph.as_default():
                od_graph_def = tf.compat.v1.GraphDef()
                with tf.compat.v1.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
                    serialized_graph = fid.read()
                    od_graph_def.ParseFromString(serialized_graph)
                    tf.import_graph_def(od_graph_def, name='')
            config = tf.compat.v1.ConfigProto()
            config.gpu_options.allow_growth = True
            self.detection_sess = tf.compat.v1.Session(graph=detection_graph,
config=config)
            self.agent_absent = []
            self.image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
            self.scores = detection_graph.get_tensor_by_name('detection_scores:0')

    def start_practice(self):
        self.xdo.send_keysequence_window_down(self.window, b"space")
        self.xdo.send_keysequence_window_up(self.window, b"space")
        time.sleep(0.5)
        self.xdo.send_keysequence_window_down(self.window, b"Escape")
        self.xdo.send_keysequence_window_up(self.window, b"Escape")
        time.sleep(1)
        pygui.click(self.practice_pos)
        time.sleep(0.25)

    def random_level(self):
        self.xdo.send_keysequence_window_down(self.window, b"Escape")
        self.xdo.send_keysequence_window_up(self.window, b"Escape")
        time.sleep(0.5)
        self.xdo.send_keysequence_window_down(self.window, b"Escape")
        self.xdo.send_keysequence_window_up(self.window, b"Escape")
        time.sleep(0.5)
        previous_level = self.levels[self.current_level]
        self.current_level += 1
        self.current_level %= len(self.levels)
        if self.current_level == 0:
            shuffle(self.levels)
        moves_num = previous_level - self.levels[self.current_level]
        direction = b"Right" if moves_num < 0 else b"Left"
        for i in range(abs(moves_num)):
            self.xdo.send_keysequence_window_down(self.window, direction)
            self.xdo.send_keysequence_window_up(self.window, direction)
            time.sleep(0.5)
        self.xdo.send_keysequence_window_down(self.window, b"space")
```

```
            self.xdo.send_keysequence_window_up(self.window, b"space")
            self.reached = True
            print(f"Level chosen: {self.levels[self.current_level]}")

        def retry(self):
            self.fr = 0
            if self.mode == "practice":
                self.agent_absent = [False for _ in range(10)]
                if self.reached:
                    self.attempt_counter = 0
                    restart_visible = False
                    while not restart_visible:
                        time.sleep(0.5)
                        frame = np.array(
                            ImageGrab.grab(
                                bbox=(
                                    self.region_main['left'], self.region_main['top'],
                                    self.region_main['left'] + self.region_main["width"],
                                    self.region_main['top'] + self.region_main["height"]
                                )
                            ),
                            dtype=np.uint8
                        )
                        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
                        restart_visible = self.level_failed(frame)
                    self.start_practice()
                else:
                    self.attempt_counter += 1
                    appeared = False
                    t = time.perf_counter()
                    while not appeared and (time.perf_counter() - t) < 7:
                        frame = np.array(
                            ImageGrab.grab(
                                bbox=(
                                    self.region_main['left'], self.region_main['top'],
                                    self.region_main['left'] + self.region_main["width"],
                                    self.region_main['top'] + self.region_main["height"]
                                )
                            ),
                            dtype=np.uint8
                        )
                        image_np = np.expand_dims(frame, axis=0)
                        scores = self.detection_sess.run(
                            self.scores,
                            feed_dict={self.image_tensor: image_np})
                        if scores.max() >= 0.98:
                            appeared = True
                    if not appeared:
                        self.uncheckpoint()
            else:
                self.xdo.send_keysequence_window_down(self.window, b"space")
                self.xdo.send_keysequence_window_up(self.window, b"space")
            self.reached = False

        def level_failed(self, frame):
            return pygui.locate(self.retry_img, frame[453:547, 188: 278], confidence=0.72)
    is not None

        def reached_heaven(self, frame):
            return pygui.locate(self.progress_bar_img,
                                cv2.cvtColor(frame[4:22, 228:576], cv2.COLOR_BGR2GRAY),
    confidence=0.99) is not None

        def step(self, action):
```

```
        if not self.pressed and action == 1:
            self.xdo.send_keysequence_window_down(self.window, b"Up")
            self.pressed = True
        elif self.pressed and action == 0:
            self.xdo.send_keysequence_window_up(self.window, b"Up")
            self.pressed = False
        frame = np.array(
            ImageGrab.grab(
                bbox=(
                    self.region_main['left'], self.region_main['top'],
                    self.region_main['left'] + self.region_main["width"],
                    self.region_main['top'] + self.region_main["height"]
                )
            ),
            dtype=np.uint8
        )
        self.record_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
        if self.mode == "practice":
            image_np = np.array(frame[22:, :])
            image_np = np.expand_dims(image_np, axis=0)
            scores = self.detection_sess.run(
                self.scores,
                feed_dict={self.image_tensor: image_np})
        new_state = cv2.resize(frame[22:, :], (84, 84))
        self.reached |= self.reached_heaven(self.record_frame)
        if self.mode == "practice":
            if scores.max() < 0.98:
                self.agent_absent.append(True)
            else:
                self.agent_absent.append(False)
            self.agent_absent.pop(0)
            done = True if all(self.agent_absent) else False
        else:
            done = self.level_failed(self.record_frame)
        reward = 0
        if done:
            if self.reached:
                reward = 0
            else:
                reward = -100
                self.uncheckpoint()
                self.uncheckpoint()
            if self.pressed:
                self.xdo.send_keysequence_window_up(self.window, b"Up")
        self.fr += 1
        if self.mode == "practice":
            return new_state, reward, done, self.fr, self.reached,
self.attempt_counter
        else:
            return new_state, reward, done, self.fr, self.reached, 1

    def pause(self):
        self.xdo.send_keysequence_window_down(self.window, b"Escape")
        self.xdo.send_keysequence_window_up(self.window, b"Escape")
        time.sleep(1)

    def unpause(self):
        self.xdo.send_keysequence_window_down(self.window, b"space")
        self.xdo.send_keysequence_window_up(self.window, b"space")
        time.sleep(1)

    def uncheckpoint(self):
        self.xdo.send_keysequence_window_down(self.window, b"x")
        self.xdo.send_keysequence_window_up(self.window, b"x")
```

```
    def test(self):
        for _ in range(10):
            frames = []
            self.retry()
            last_time = time.time()
            done = False
            fr = 0
            reached = False
            self.step(0)
            state = self.record_frame
            state = cv2.cvtColor(state, cv2.COLOR_BGR2GRAY)
            while not done:
                # 800x600 windowed mode
                res, reward, done, fr, _, _ = self.step(0)
                frames.append(self.record_frame)
                reached |= self.reached_heaven(self.record_frame)
                fr += 1
                next_frame_g = cv2.cvtColor(self.record_frame, cv2.COLOR_BGR2GRAY)
                next_state = (state * 0.6 + 0.4 * next_frame_g).astype(np.uint8)
                print(f'reward: {reward}, done: {done}, reached: {reached}')
                time.sleep(1 / 60)
                state = next_state
            ex_time = time.time() - last_time
            print(f'Ex time: {ex_time}, frames: {fr}, framerate: {float(fr /
ex_time)}')


if __name__ == "__main__":
    e = GDenv((800, 600), mode="practice")
    e.test()
```