Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження та розробка алгоритмів глибокого навчання штучного інтелекту для розв'язання ігрових завдань
(тема)

Виконав:
студент 2 курсу, групи СШІм-21-1
Хамаза В.С.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник проф. Жолудов Ю.Т.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ (підпис) В.О. Філатов (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп’ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
«_____»___________20____р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Хамазі Владиславу Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження та розробка алгоритмів глибокого навчання штучного інтелекту для розв’язання ігрових завдань

затверджена наказом університету від 31 березня 20 23 р. № 306Ст

2. Термін подання студентом роботи до екзаменаційної комісії 23 травня 20 23 р.

3. Вихідні дані до роботи Науково-технічні публікації, дані Інтернет-джерел та відомих наукових проектів щодо розробки та дослідження систем з використанням штучного інтелекту, Microsoft Developer Network (MSDN) documentation

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної області

2) Розробка вимог до розроблювальної системи

3) Опис прийнятних проектних рішень при розробці системи

4) Аналіз та проектування архітектури алгоритму

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)________________
____
____
____
____
____
____
____
____
____

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	05.04.2023	виконано
2	Опис системи	08.04.2023	виконано
3	Моделювання та навчання системи	11.04.2023	виконано
4	Визначення проблем при розробці системи	20.04.2023	виконано
5	Знаходження та вирішення проблем	25.04.2023	виконано
6	Розробка інтелектуальної системи	03.05.2023	виконано
7	Написання пояснювальної записки	15.05.2023	виконано
8	Попередній захист	19.05.2023	виконано
9	Захист перед ЕК	23.05.2023	виконано

Дата видачі завдання 3 квітня 20 23 р.

Студент ________________________________
(підпис)

Керівник роботи ____________________________ проф. Жолудов Ю.Т.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 66 с., 11 рис., 1 дод., 20 джерел.

ІГРОВИЙ ДВИГАТЕЛЬ, МАШИННЕ НАВЧАННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, C++, PYTHON, Q-LEARNING.

Об'єктом досліджень магістерської кваліфікаційної роботи є алгоритми глибокого навчання штучному інтелекту для вирішення ігровихзавдань.

Предметом досліджень кваліфікаційної роботи є процес розробка алгоритму для глибокого навчання штучному інтелекту для вирішення ігрових завдань.

Мета досліджень: розробка алгоритму глибокого навчання штучному інтелекту для вирішення ігрових завдань.

Результати роботи: розроблені компоненти програмного забезпечення підтримки навчання штучного інтелекту.

Область застосування: забезпечення та автоматизація створення ігор, кіно та поліпшення розробки продукту на етапі проектування та побудови.

ABSTRACT

Explanatory note: 66 p., 11 fig., 1 ann., 20 sources;

ARTIFICIAL INTELLIGENCE, C++, GAME ENGINE, MACHING LEARNING, PYTHON, Q-LEARNING.

The object of research of the master's qualification thesis is algorithms of deep learning of artificial intelligence for solving game tasks.

The subject of the research of the qualification work is the process of developing an algorithm for deep learning of artificial intelligence for solving game tasks.

The purpose of research: development of a deep learning algorithm for artificialintelligence for solving game tasks.

Results: developed components of the software for supporting the learning ofhumorous intelligence.

Scope: enabling and automating the creation of games, movies and improvingproduct development at the design and build stage.

ЗМІСТ

Вступ 7
1 Аналіз предметної галузі 8
1.1 Аналіз предметної галузі 8
1.2 Огляд та аналіз засобів розробки ШІ 9
1.3 Визначення сфери застосування розроблювальної системи 13
1.4 Постановка завдання 22
1.4.1 Рух та стрільба 26
1.4.2 Прийняття рішень 28
1.4.3 Сприйняття 30
1.4.4 Пошук шляху та тактичний ШІ 31
1.5 Огляд структури задач обробки природної мови 33
1.5.1 Хаки 33
1.5.2 Евристика 34
1.5.3 Алгоритми 35
2 Розробка вимог до розроблювальної системи 36
2.1 Розробка системних вимог до системи ШІ 36
2.2 Розробка функціональних вимог до системи ШІ 37
2.3 Розробка моделей потоків даних систем ШІ 39
3 Опис прийнятих проектних рішень при розробці системи 41
3.1 Обґрунтування вибору мови програмування 41
3.2 Опис архітектури (структури) розробленої системи 44
4 Аналіз та проектування архітектури алгоритму 46
4.1 Обрання алгоритму навчання 46
4.2 Основні концепції й таксономія високого рівня 60
4.3 Q-Learning 61
Висновки 63
Перелік джерел посилання 64
Додаток А Відомість кваліфікаційної роботи 66

ВСТУП

Найперша відеогра була побудована повністю на апаратному рівні, але швидкий розвиток мікропроцесорів все змінив. В наші дні відеоігри можна грати на універсальних ПК і спеціалізованих ігрових консолях, які використовують програмне забезпечення, щоб дати можливість пропонувати величезну різноманітність ігрових можливостей. Минуло 50 років з тих пір, як з'явились перші примітивні ігри. Вона може бути молода, та якщо придивитись уважніше, виявиться, що все швидко розвивається.

Відеоігри – це зараз багатомільярдні індустрія, яка охоплює широкий спектр демографія.

Відеоігри бувають всіх форм і розмірів, потрапляючи в категорії або «Жанри», що охоплюють всі, від пасьянсів до багатокористувацьких мережевих ігор, рольові ігри, і в ці ігри грають практично на чому завгодно. У наші дні ви можете отримати ігри для свого ПК, мобільного телефону, а також ряд різних спеціалізованих ігрових консолей.

Платформи, що випускаються циклічно, стали називати консольними «поколіннями». Лідером цього останнього покоління є PlayStation 5 mf Xbox One, але ніколи не слід випускати з уваги ПК.

З глибоким розвитком штучного інтелекту алгоритми глибокого навчання стали надзвичайно популярними у багатьох галузях, включаючи галузь ігор. Ігрові завдання можуть вимагати складних стратегій та рішень, які потребують глибокого аналізу та навчання. Дослідження та розробка алгоритмів глибокого навчання для реалізації ігрових завдань можуть допомогти створити ефективні та інноваційні ігрові системи, які можуть забезпечити найкращі результати та досвід для гравців. У цій роботі буде проаналізовано інші підходи та техніку глибокого навчання для виконання ігрових завдань, включаючи використання нейронних мереж та інших методів машинного навчання.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Штучний інтелект полягає в тому, щоб зробити комп'ютери здатними виконувати завдання мислення, на які здатні люди та тварини.

Ми вже можемо запрограмувати комп'ютери на надлюдські здібності у вирішенні багатьох завдань: арифметики, сортування, пошуку тощо. Ми навіть можемо змусити комп'ютери грати в деякі настільні ігри краще, ніж будь-яка інша людина. Багато з проблем спочатку вважалися проблемами штучного інтелекту, але в міру того, як їх вирішували все більш комплексними способами, вони вийшли зі сфери розробників штучного інтелекту. Але є багато речей, у яких комп'ютери погані, але які ми вважаємо тривіальними: розпізнавати знайомі обличчя, розмовляти рідною мовою, вирішувати, що робити далі, і бути творчим. Це сфера діяльності штучного інтелекту: ми намагаємося визначити, які типи алгоритмів потрібні для їх відображення властивості [3].

В академічному середовищі деяких дослідників ШІ мотивує філософія: розуміння природи мислення таприроди інтелекту та створення програмного забезпечення для моделювання того, як мислення може працювати. Деякі мотивовані психологією: розумінням механіки людського мозку та розумових процесів. Інших мотивує інженерія: створення алгоритмів для виконання людських завдань. Це потрійне розрізнення лежить в основі академічного ШІ, і різні уявлення відповідають за різні підполе предмета. Як розробникиігор, в першу чергу зацікавлені лише в інженерній стороні: створенні алгоритмів, які роблять ігрових персонажів схожими на людей чи тварин. Розробники завжди малювали з академічних досліджень, де це дослідження допомагає їм виконувати роботу.

1.2 Огляд та аналіз засобів розробки ШІ

Існує безліч засобів розробки програмного забезпечення для створення ШІ. Нижче наведено огляд та аналіз найбільш популярних з них:

Python – це одна з найпопулярніших мов програмування для розробки ШІ. Вона має простий та зрозумілий синтаксис, велику кількість бібліотек та модулів для роботи з даними, машинним навчанням та іншими задачами ШІ. Python також підтримує багато платформ, включаючи Windows, Linux та MacOS.

Java – це інша популярна мова програмування для розробки ШІ. Вона має широке застосування в багатьох сферах, таких як фінанси, телекомунікації, виробництво та багато інших. Java має велику кількість бібліотек та фреймворків, що дозволяє ефективно розробляти ШІ.

C/C++ – ці мови програмування використовують для розробки ШІ з високою рівнем продуктивності та низьким рівнем системних вимог. Вони знаходять у багатьох галузях, таких як вбудовані системи, ігри, операційні системи та інші сфери, де потрібна швидкість та ефективність.

MATLAB – це середовище для чисельних обчислень та розробки ШІ, яке забезпечує доступ до великої кількості завдань для аналізу даних, обробки сигналів та зображень, машинного навчання та ін. MATLAB підтримується на багатьох платформах, включаючи Windows, Linux та MacOS.

R – це мова програмування та середовище для статистичного аналізу даних та машинного навчання. R має велику кількість бібліотек та пакетів.

Це все що стосується мови программування, також є безліч фреймворків які значно прискорюють процес розробки. Деякі з найважливіших інструментів і фреймворків розглянуто далі.

Scikit Learn.

Scikit-learn – одна з найвідоміших бібліотек ML. Він лежить в основі

багатьох адміністративних і неконтрольованих навчальних обчислень. Прецеденти включають прямі та обчислені рецидиви, дерева вибору, групування, k-імпліки тощо:

– він розширює дві основні бібліотеки Python, NumPy і SciPy;

– він містить багато обчислень для звичайних завдань штучного інтелекту та інтелектуального аналізу даних, включаючи групування, повторення та порядок. Дійсно, навіть такі заходи, як зміна інформації, визначення характеристик і методи ансамблю, можуть бути виконані за пару рядків;

– для новачка в ML Scikit-learn є більш ніж адекватним інструментом для роботи, поки ви не почнете актуалізувати все складніші обчислення. Tensorflow.

Якщо ви перебуваєте у царстві штучного інтелекту, ви, швидше за все, дізналися, спробували або виконали певний тип обчислення глибокого навчання. Чи правильно сказати, що вони важливі? Не постійно. Чи правильно сказати, що вони круті, коли зроблено правильно? Воістину!

Захоплююча річ у Tensorflow полягає в тому, що коли ви створюєте програму на Python, ви можете впорядкувати її та продовжувати працювати на центральному чи графічному процесорі. Тож вам не потрібно створювати на рівні C++ або CUDA, щоб продовжувати працювати на графічних процесорах. Він використовує багаторівневі концентратори, які дають змогу швидко налаштовувати, навчати та надсилати підроблені нейронні системи з величезними наборами даних. Це те, що дозволяє Google розпізнавати запитання на фотографіях або розуміти словесно виражені слова в програмі голосового підтвердження.

Theano.

Theano чудово складено над Keras, бібліотекою нейронних систем аномального стану, яка працює майже паралельно з бібліотекою Theano. Фундаментальна сприятлива позиція Keras полягає в тому, що це помірна бібліотека Python для глибоких відкриттів, яка може працювати над Theano

або TensorFlow.

Він був створений, щоб зробити актуалізацію моделей глибокого навчання настільки швидкою та простою, наскільки це можливо для інноваційної роботи.

Він продовжує працювати на Python 2.7 або 3.5 і може постійно виконуватися на GPU та CPU.

Theano відрізняє те, що він використовує графічний процесор ПК. Це дає змогу підраховувати кількість інформації, яка надсилається, у кілька разів швидше, ніж коли він працює лише на ЦП. Швидкість Theano робить його особливо вигідним для поглибленого навчання та інших складних обчислювальних завдань.

Caffe.

Caffe – це глибока навчальна структура, створена з головним пріоритетом артикуляції, швидкості та виміряної якості. Він створений Центром бачення та навчання Берклі (BVLC) та донорами мережі. DeepDream від Google залежить від Caffe Framework. Ця структура є авторизованою BSD бібліотекою C++ з інтерфейсом Python.

MxNet.

Це дозволяє обмінювати час обчислень для пам’яті через «забутню резервну пропозицію», яка може бути дуже корисною для рекурентних мереж на дуже довгих послідовностях:

– створено з урахуванням масштабованості (досить проста у використанні підтримка навчання з декількома GPU та кількома машинами);

– багато цікавих функцій, наприклад легке написання власних шарів мовами високого рівня;

– на відміну від майже всіх інших основних фреймворків, він безпосередньо не керується великою корпорацією, що є здоровою ситуацією для фреймворку з відкритим кодом, розробленого спільнотою;

– підтримка TVM, яка ще більше покращить підтримку розгортання

та дозволить працювати на багатьох нових типах пристроїв.

Keras.

Якщо вам подобається спосіб роботи на Python, Keras для вас. Це бібліотека високого рівня для нейронних мереж, яка використовує TensorFlow або Theano як серверну частину:

– більшість практичних задач більше схожі на: вибір архітектури, яка підходить для проблеми, для проблем з розпізнаванням зображень – використання ваг;

– навчених на ImageNet;

– налаштування мережі для оптимізації результатів (тривалий ітеративний процес).

У всьому цьому Керас є перлиною. Крім того, він пропонує абстрактну структуру, яку можна легко конвертувати в інші фреймворки, якщо це необхідно (для сумісності, продуктивності тощо).

Auto ML.

З усіх інструментів і бібліотек, перерахованих вище, Auto ML, мабуть, є одним із найпотужніших і відносно нещодавніх доповнень до арсеналу інструментів, доступних інженеру з машинного навчання.

Як описано у вступі, оптимізація має суттєве значення для завдань машинного навчання. Хоча переваги, отримані від них, є прибутковими, успіх у визначенні оптимальних гіперпараметрів не є легким завданням. Особливо це стосується «чорних скриньок», таких як нейронні мережі, де визначення важливих речей стає все складнішим із збільшенням глибини мережі.

Таким чином ми входимо в нову сферу мета, де програмне забезпечення допомагає створювати програмне забезпечення. AutoML – це бібліотека, яка використовується багатьма інженерами машинного навчання для оптимізації своїх моделей.

Окрім очевидної економії часу, це також може бути надзвичайно корисним для тих, хто не має великого досвіду у сфері машинного

навчання, а тому не має інтуїції чи минулого досвіду, щоб самостійно вносити певні зміни гіперпараметрів.

1.3 Визначення сфери застосування розроблювальної системи

Оскільки розроблювальна система, яку ми досліджуємо, є системою глибокого навчання штучного інтелекту для вирішення ігрових завдань на двигуні Unreal Engine з використанням алгоритму Q–Learning, то її сферою застосування є геймдев (game development) або розробка ігор.

Така система може бути використана для розробки різних видів ігор, де потрібно створювати інтелектуальних супротивників, які здатні навчатися та адаптуватися до геймплею. Наприклад, це можуть бути ігри жанру стратегій, де комп'ютерні супротивники повинні здатися на різні стратегії та тактики гравця, або ігри жанру шутерів, де ігровий процес потрібно робити більш реалістичним та цікавим для гравця.

Рольові ігри містять ряд елементів, якими зазвичай керує ШІ. До них належать як антагоністичні персонажі (вороги, боси та неігрові персонажі), так і хороші чи нейтральні персонажі (крамарі та інші члени групи). Оскільки основний ігровий процес рольових ігор багато в чому обертається навколо взаємодії персонажів, бойових чи інших, кожен із цих елементів може бути досить складним.

Enemes (Вороги).

Більшість населення більшості RPG світів є ворогами. Потрібна майже нескінченна кількість ворогів, щоб надати гравцеві щось для відправлення та отримання очок досвіду, грошей і потужних нових предметів. Рольові ігри в минулому використовували майже виключно те, що можна описати як статистичний ШІ, оскільки атрибути (сила, розмір, очки життя тощо) монстрів визначали все про них: атаки, які вони використовують, спосіб боротьби, наскільки міцний вони взагалі, який скарб вони кидають, коли помирають, і так далі. Сучасні ігри йдуть трохи

далі і мають ворогів, які більше підібрані вручну. Ці сучасні вороги також використовують складніші моделі поведінки, включаючи втечу, самолікування, бої в групах, оточуючи гравця та використовуючи додаткові методи атаки тощо.

Оскільки ворогів у рольових іграх зазвичай трапляється у такій кількості під час гри, штучний інтелект спеціально налаштований так, щоб бути більше A, а не стільки I. Покрокові рольові ігри минулого (Bard's Tale, Phantasy Star, Chrono Trigger), так називається бойова рольова гра в реальному часі (The Legend of Zelda, пізніші ігри Ultima™, Diablo, Terranigma), а також варіанти злиття, які нещодавно виникли (Baldur's Gate або Icewind Dale, які є іграми в реальному часі, які можна призупинити і, таким чином, , створені для покрокових дій) усі майже зводять ворогів до комбінованих контейнерів (багатства та очок досвіду) та перешкод (будучи «стінами» певної кількості очок здоров'я, які герой має знищити, щоб пройти). Дуже небагато ігор виходять за межі такого простого стилю ворога, щоб створити будь–що з індивідуальністю, винахідливістю чи змінною стратегією. Звичайно, це зроблено задумом. Коли гравець, який провів 60 або більше годин, граючи у вашу гру, заходить до кімнати й бачить наближення монстра, схожого на ворожого персонажа, якого він бачив раніше, він повинен відчути одне з трьох способів:

– я можу перемогти цього хлопця. Я знаю, які атаки він використовує, скільки у нього приблизно здорових очок, і що у мене є зброя, яка впливає на цього ворога;

– я думаю, що зможу перемогти цього хлопця. Він дуже схожий на ворога, з яким я вже бився, але має інший колір або особливе ім'я, що робить його незвичайним і, можливо, більш просунутим. По суті, я вважаю, що він належить до «типу» ворога, але я не впевнений щодо його твердості;

– я не можу перемогти цього хлопця. Він занадто сильний, або я не маю необхідної зброї, щоб пробити його броню. Я знаю, тому що я

намагався раніше, але не вдалося, або хтось у грі попередив мене. Це ще один спосіб занурити гравця в гру і змусити його відчути себе частиною світу, оскільки він «знає» ворогів на досвіді. Якщо скромний орк раптово дістає гранату (після марного підбігу та використання іржавого кинджала в останніх п’ятдесяти зіткненнях) і вдарить гравця ядерною зброєю, гравець відчує себе дещо ошуканим. Однак цю основну вказівку іноді можна обійти, якщо гравцеві дозволено зберігати гру, коли він забажає, або якщо гра насправді автоматично зберігається досить часто. Таким чином, дуже незвичайна зустріч із особливим ворогом може вбити гравця, але він не втратить багато ігрового часу, якщо в нього буде сейв. Так, це веде до більшої поведінки гравця «збережи, потім заверни, убий одного монстра, а потім врятуй», але це також дає вам більше свободи додавати елементи несподіванки до ваших випадкових зустрічей.

Bosses (Боси).

Боси – це більші та складніші ігрові персонажі, гуманоїди чи істоти, яких можна знайти в кінці кожного рівня (чи ігрового світу, чи підрозділу) після перемоги над ордою менших ворогів. Зазвичай вони еквівалентні лідерам монстрів, королям монстрів. Це конкретні, зазвичай унікальні вороги, які можуть порушувати всі попередні правила. Гравці очікують бути здивованими силою, навичками, зброєю тощо, які використовують ці персонажі. У світі рольових ігор босів навіть вважають ласощами, і хороший бос може компенсувати багато недоліків гри, або в області середнього ігрового процесу, або просто в період виснажливого підвищення рівня, необхідного для продовження в грі. ігровий світ. Таким чином, монстри-боси зазвичай сильно прописані зі спеціальними атаками та поведінкою, яку виконують лише вони. Монстри-боси також зазвичай спілкуються з гравцем у формі інформації про розвиток сюжету або чистих образ. Отже, ШІ для цих істот має включати використання системи діалогів для гри. Монстри-боси серії Final Fantasy – це чудо спеціалізованого кодування, зіткнення з різними етапами битв і розмов

можуть тривати години реального часу. Ці зіткнення строго контролюються розробниками, із запланованими залпами переваги гравця, за якими слідує перевага ворога, сценарієм переривання з іншими ворогами або спеціальними ігровими подіями та будь-яким іншим, що придумають дизайнери. Ще одна випробувана тактика Боса полягає в тому, що «не можна вбити». . . але бос». Це пов'язано з босом, якого гравці можуть довести майже до смерті, щоб лише дивом втекти, кричачи «Я повернуся!» і обіцяю бути більшим і поганішим наступного разу. Хоча це дещо банально, це ігровий еквівалент простого розвитку персонажа, коли Поганий хлопець розвивається протягом гри так само, як і ви.

У деяких іграх використовується позначення «суб-боса», щоб ще більше розшарувати монстрів у грі, хоча зазвичай це просто дуже жорсткі версії звичайних істот, як-от «унікальні» істоти, які значною мірою заповнюють серію Diablo. Але навіть у Diablo, яку багато хто вважав «спрощеною RPG-фестиваллю», також використовуються набагато більш спеціалізовані істоти-боси, які використовують додаткові діалоги, анімацію, ефекти заклинань і зброї, а також спеціальні сили. Позначення «Бос» також включає останню істоту (чарівник/бог/злочинець), яку гравцеві потрібно буде перемогти, щоб виграти гру, також звану Кінцевим босом. Цей персонаж справді дуже важливий, і багато хороших ігор отримали погані оцінки за те, що вони розчаровують або антикліматичного кінцевого боса.

Гравець повинен виконати всі трюки, які він або вона навчився під час гри, і максимально розширити набуті навички, щоб знищити цього персонажа, а сам Кінцевий бос повинен мати можливість робити те, чого гравець ніколи раніше не бачив у гра. Кінцевий бос, звичайно, має бути сильним із точки зору статистики (з великою кількістю очок життя та імунітетом до зброї чи заклинань), але Кінцевий бос також має бути здатним до поведінки, що перевищує типову. Ось чому цей персонаж є кінцевим босом.

Nonplayer Characters (NPCs).

NPC – це будь–яка особа в грі, яка не є людиною. Однак зазвичай термін NPC відноситься до персонажів гри, з якими гравець може взаємодіяти іншими способами, окрім бою. NPC – це персонажі, які населяють міста, напівмертвих солдатів на стежці, які дають гравцеві цінні підказки про небезпеку, що загрожує попереду, і випадкового старого, який пропонує персонажу гравця гроші, щоб врятувати доньку старого. Як правило, NPC можна згрупувати в один із двох типів:

– одноразові персонажі (тобто вони мають щось для гравця один раз протягом гри, але потім лише вітатимуть гравця з вдячністю), як люди, які беруть участь у побічному квесті;

– персонажі, що викидають інформацію, з якими гравець може продовжувати спілкуватися в різні моменти гри. Ці персонажі можуть знати щось додаткове про те, що зараз є «новим» у потоці гри.

NPC, як правило, не дуже розумні; вони зазвичай не повинні бути. Усе, що вони додають, окрім інформації чи розвитку історії, – це лише смак для гри. Однак вони також представляють собою одне з найбільших джерел інформації, яку має гравець про перебіг сюжетної лінії. NPC також можуть слугувати підтримкою в грі, яка може повернути гравця, який застряг або втратив, у відповідність до цілей гри. Таким чином, багато ігор експериментували з різними способами ведення розмови NPC. Деякі ігри дають гравцеві ключові слова, які представляють питання, які гравець ставить перед NPC (як в іграх Ultima), інші дають гравцеві вибір між кількома повними реченнями, які представляють різні погляди, які гравець може прийняти з NPC. Еволюція цих систем триватиме, оскільки граматичні системи стануть кращими, швидшими та загальновизнаними. Одного разу гравці можуть поспілкуватися безпосередньо із загальним NPC зі штучним інтелектом, який може давати широкі відповіді, індексуючи базу знань персонажа та формуючи речення на льоту. До того часу ми робимо все, що можемо.

Shopkeepers.

Крамарі – це спеціальні NPC, які ведуть бізнес з гравцем; купівля і продаж спорядження, навчання гравця новим навичкам і так далі. Власники магазинів зазвичай не набагато розумніші за звичайних NPC, але їх особливо відзначають, оскільки вони зазвичай мають розширені інтерфейси, які, у свою чергу, вимагають спеціального коду, щоб вони здавалися розумними та зручними. Іноді крамарі можуть бути частиною сценарію квесту чи ігрової послідовності, тобто вони стають крамарями лише пізніше в грі або після виконання завдання. Таким чином, власник магазину може мати уявлення про те, чи подобається йому гравець, що потім вплине на його ставлення та ціни під час роботи з цим гравцем. Деякі ігри мають загальний атрибут харизми для персонажів у грі (або якийсь похідний; значення: «Наскільки добре інші люди сприймають вас природно», враховуючи перше враження, зовнішність гравця та його розмовні здібності), а також певну форму системи репутації, яка представляє свого роду «рейтинг», що відображає кількість добрих і злих вчинків гравця, а також прапори, що представляють конкретні дії гравця, які NPC можуть помітити та відреагувати на них. Існує природна людська схильність надавати неживим речам людські якості, і ця тенденція безпосередньо пов'язана з кількістю часу, який ми витрачаємо на справу. Існує також кореляція з тим, скільки нам коштував цей об'єкт. Дуже небагато людей приписували б своєму взуттю людські якості, але багато людей називають свою машину, знають її стать, знають, як визначити, чи у неї поганий день, і навіть благатимуть, якщо вона працює погано. Обидва об'єкти (взуття та автомобіль) виконують приблизно однакову функцію: допомагають захистити наше тіло від тягарів подорожі, тож чому ця невідповідність? Відповідь очевидна. Без рухомих частин і простої процедури, яку ми навчилися, коли нам було три роки, ми взуваємося вранці й забуваємо про них. Купівля нової пари не потребує перевірки кредитоспроможності.

Наші машини – це точнісінько навпаки. Те саме стосується AI Shopkeeper. Якщо у вашій грі є одноразовий NPC, ви можете робити все, що завгодно, з його поведінкою, діалогами та взаємодією з гравцем. Гравець не очікує багато чого і сприйме більшість речей за чисту монету. Але з власником магазину, особливо таким, до якого гравець змушений буде постійно повертатися протягом більшої частини гри, кожен нюанс, відповідь і анімаційний кадр будуть уважно спостерігатися, запам'ятовуватися та олюднюватися. Чи є у вас система обміну (яка насправді бере бал харизми гравця, додає випадковий коефіцієнт і визначає невелику знижку, на яку гравець може торгуватися) у вашій грі? З часом гравець-людина почне уявляти складні правила, пов'язані з порядком предметів, з якими він веде бізнес, часом доби, настроєм власника крамниці та багатьма іншими факторами, які насправді можуть не існувати. Саме ця тенденція до гуманізації дозволяє розробникам ігор обійтися так мало деталей у своїх іграх, тому що гравець-людина заповнить всю складність там, де їх немає.

Party Member.

Члени пригодницької групи гравця також є особливими NPC, за винятком того, що вони подорожують з гравцем і або повністю контролюються гравцем (у покрокових рольових іграх або в пізніших іграх, які дозволяють гравцям призупиняти дію, щоб мати час дати детальні команди) або пов'язати з ними код ШІ. Ці члени групи на основі штучного інтелекту потребують ретельного кодування, оскільки дурні члени групи швидко відганяють потенційних гравців. У багатьох бойових іграх у реальному часі використовується простий груповий штучний інтелект, щоб гравець міг передбачити (і покладатися на це), що кожен член групи збирається робити під час бою.

Важливим фактором, про який слід пам'ятати в бойових RPG у реальному часі, є пошук шляху. У покрокових бойових системах члени партії гравця просто приєднані до гравця або безпосередньо слідують за

гравцем (як ігри Final Fantasy або навіть ранні Bard's Tale), але в іграх у реальному часі вони фактично мають шукати шлях до стежити за гравцем. У напівзакритому просторі (наприклад, підземному підземеллі), де немає місця для маневру, один або кілька членів групи можуть втекти, щоб пройти якийсь наддовгий мальовничий маршрут, який вдалося знайти досліднику. Сліпий шлях може бути надзвичайно розчаровуючим для гравця, оскільки він може змусити цих збентежених членів групи пробігати крізь зграї монстрів в інших частинах карти, навіть призводячи недружніх вбігти в кімнату позаду «корисних» друзів, щоб приєднатися до бою. Ось місце, де розумний член партії може сказати: «Хм, я не можу обійти цього хлопця безпосередньо, щоб використати свій меч. Але у мене в рюкзаку є лук і стріли, і я непогано володію стрільбою з лука, можливо, я спробую атакувати з дистанції». Простішим рішенням може бути «Не можу обійти напряму, тому я не можу атакувати». Може, мені варто поплескати по плечу свого слабшого товариша, якого розтерзає якась істота, і замінити його на передовій». Така «розумність» (а не необізнане пошук шляху та слідування сценарію) є різницею між корисними членами партії та неефективними спільниками, яких гравець повинен няньчити. Якщо персонажі, з якими гравець стикається з пригодами, часто зіпсовують, роблять правильні речі неправильно або постійно вбивають самих себе (або, що ще гірше, гравця), гравець не захоче продовжувати з ними грати. Baldur’s Gate (і його нащадки) навіть дозволяють користувачам редагувати сценарії, які керують штучним інтелектом членів групи, щоб користувачі мали ще більше контролю над цим ключовим елементом гри. Деякі користувачі спільноти створили дуже просунуті сценарії штучного інтелекту та розмістили їх на веб-сайтах шанувальників для всіх. Дивиться наступний розділ «Сценарії». Додавання системи сценаріїв для редагування стороннього штучного інтелекту – це ретельний баланс. Якщо ви зробите його занадто простим у використанні та не надасте достатньої складності та функціональності, він нічого не вартий.

Але якщо система занадто потужна, вона може перевантажити звичайних гравців і знову стати марною для значної частини вашої аудиторії. Техніка, яка використовується в багатьох спортивних іграх, щоб дозволити гравцям налаштовувати ШІ у своїх іграх, полягає в тому, щоб виявляти конкретні тенденції поведінки у вигляді «повзунків» (смуг прокрутки, пов'язаних із змінною), які гравець може встановити. Для спортивних ігор це означає, що гравці можуть організувати баскетбольну гру, де штучний інтелектніколи не намагатиметься вкрасти м'яч, також не буде захисником і кращим буде виконувати триочкові кидки, установивши повзунки на певні точки. Подібну систему можна використовувати, щоб надати більш звичайним гравцям доступ до редагування ШІ без необхідності писати код сценарію. Навіть деякі з більш складних застосувань системи сценаріїв, як- от налаштування того, коли певні заклинання будуть накладені персонажем AI-магу, можуть бути представлені у вигляді повзунків, які є специфічними для цього заклинання. Це перекладається на багато потенційних повзунків, але знову ж таки, це безумовно більш доступне для більшої аудиторії, ніж файли сценаріїв.

Крім того, розроблювальна система може бути використана для створення навчальних ігор, де гравці можуть навчатися та розвивати свої навички у різних сферах, таких як навчання мовам, математиці, інформатиці тощо.

Якщо розглянути її більш ширше розроблення систем глибокого навчання штучного інтелекту для ігрових застосувань має великий потенціал і може бути застосоване в різних галузях, наприклад:

– медицина: системи глибокого навчання можуть допомогти у виявленні хвороб та діагностиці на ранніх стадіях, а також у прогнозуванні підходів до лікування;

– промисловість: системи глибокого навчання можуть допомогти у прогнозуванні відмов промислового обладнання, а також у підвищенні

його ефективності та зниженні витрат на обслуговування;

– транспорт: системи глибокого навчання можуть бути застосовані для підвищення безпеки дорожнього руху, оптимізації маршрутів транспортних засобів та прогнозування ризиків аварій;

– фінанси: системи глибокого навчання можуть бути використані для аналізу фінансових ринків, прогнозування ризиків та виявлення шахраїв.

Отже, системи глибокого навчання можуть бути застосовані у багатьох галузях, де потрібно обробляти великі обсяги даних та приймати рішення на основі аналізу цих даних. Розробка систем глибокого навчання для ігрових застосувань може допомогти розширити можливості застосування цієї технології та знайти нові способи застосування її у різних галузях.

1.4 Постановка завдання

Дослідити та розробити алгоритм глибокого навчання штучного інтелекту для вирішення ігрових завдань до якого входить:

– встановлення Unreal Engine та налаштування оточення для розробки ШІ: перший етап полягає у встановлення Unreal Engine та налаштування оточення для розробки штучного інтелекту. Для цього можна використовувати спеціальні плагіні та бібліотеки, такі як Tensorflow, для підтримки глибокого навчання;

– збір та підготовка даних для тренування: для розробки алгоритму потрібно зібрати відповідний набір даних для тренування та перевірки алгоритму. Це можуть бути ігрові дані, такі як карти, рівні, діалоги між персонажами, аудіо та відео записи геймплею тощо;

– вибір та налаштування алгоритму: для вирішення ігрових завдань може бути використаний алгоритм Q-Learning, який є одним із найпоширеніших алгоритмів підсиленого навчання. Для роботи з глибоким навчанням може бути використана нейронна мережа, яка буде навчатися за

допомогою алгоритму Q-Learning;

– тренування нейронної мережі: після вибору алгоритму можнарозпочати процес тренування нейронної мережі. Для цього використовуються раніше підготовлені дані, які передаються у вхідні кулі мережі. Процес тренування полягає у зміні ваг нейронної мережі таким чином, щоб вона могла зробити правильний висновок на основі вхідних даних та вирішувати.

Ігрові завдання:

– оцінка та підтвердження ефективності алгоритму: після тренування нейронної мережі необхідно оцінити її ефективність на нових даних, які не використовувалися під час тренування. Для цього можна використовувати різні метрики, такі як точність, час виконання тощо. Якщо ефективність алгоритму не задовільна, можна провести додаткове налаштування параметрів або додати нові дані для тренування;

– інтеграція з ігровим двигуном та тестування: після підтвердження ефективності алгоритму можна інтегрувати його з ігровим двигуном Unreal Engine та провести тестування в реальному часі. Під час тестування можна використовувати різні сценарії геймплею та перевірити роботу штучного інтелекту в різних умовах.

В результаті дослідження та розробки алгоритму глибокого навчання штучного інтелекту з використанням алгоритму Q-Learning для вирішення ігрових завдань на двигуні Unreal Engine можна досягти покращення в якості геймплею та зробити гри більш цікавими та складними для гравців.

Автономний агент – це система, яка розташована всередині середовища та є його частиною, яка відчуває це середовище та діє на нього з часом, переслідуючи його власний порядок денний і щоб впливати на те, що він відчуває в майбутньому. У цьому розділі я буду використовувати термін «автономний агент» стосовно агентів, які мають певний ступінь автономного руху. Якщо автономний агент натрапляє на несподівану ситуацію, як-от знаходячи стіну на своєму шляху, він матиме можливість

відреагувати та скорегувати свій рух відповідно.

Динамічні перешкоди – це ті об’єкти у вашому ігровому світі, якірухаються або змінюють положення, наприклад інші агенти, розсувні двері тощо. Враховуючи відповідне середовище, включення правильної поведінки керма в ігровий персонаж унеможливить написання спеціального коду пошуку шляху для обробки динамічних перешкод – автономний агент матиме можливість впоратися з ними, якщо і коли це буде потрібно. Рух автономного агента можна розбити на три рівні: Вибір дії: це частина поведінки агента, відповідальна за вибір його цілей і рішення, якого плану слідувати. Це частина, яка говорить «іди сюди» і «зроби А, В, а потім С».

Керування: цей рівень відповідає за обчислення бажаних траєкторій, необхідних для досягнення цілей і планів, встановлених рівнем вибору дій. Поведінка керування є реалізацією цього рівня. Вони створюють керуючу силу, яка описує, куди повинен рухатися агент і як швидко він повинен рухатися, щоб туди потрапити. Локомоція: нижній рівень, локомоція, представляє більш механічні аспекти руху агента. Це те, як подорожувати від А до Б. Наприклад, якби ви застосували механіку верблюда, акваріума та золотої рибки, а потім дали команду їм подорожувати на північ, усі вони використовували б різні механічні процеси для створення руху, навіть якщо їхній намір (рухатися на північ) ідентичний. Відокремивши цей рівень від рівня керування, можна використовувати, з невеликими змінами, однакову поведінку керування для абсолютно різних типів пересування. Рейнольдс використовує чудову аналогію, щоб описати ролі кожного з цих рівнів у своїй статті «Поведінка керування для автономних персонажів».

Розробка алгоритмів глибокого навчання для вирішення ігрових завдань може включати в себе створення різних моделей інтелектуальних агентів, таких як боти для гри, які можуть взаємодіяти з гравцями, автоматизовані роботи для ігрового середовища тощо.

Основні задачі, які потрібно при дослідженні та розробці алгоритмів

глибокого навчання для вирішення ігрових завдань, пов'язаних із складністю ігрових середовищ та забезпеченістю вирішення проблем «прокладання шляху» (pathfinding), розпізнавання об'єктів, аналізу стратегій гравців та відповіді на завдання з високою ступенем незначності. Загальна мета досліджень у цій області перебуває в розробці інтелектуальних агентів, які можуть вирішувати складні завдання в ігрових середовищах, збільшуючи тим самим ефективність використання ігрових середовищ для розробки інтелекту [5].

З появою Wolfenstein 3D і Doom, жанр шутера став синонімом персонажів, які пересуваються пішки із камерою, прив'язані до характеру гравця. Вороги зазвичай складаються з відносно невеликої кількості персонажів на екрані. У багатьох шутерах ворожі персонажі представлені у вигляді «ботів»: керованих комп'ютером персонажів із фізичними можливостями, подібними до гравців. Інші ігри пропонують гарматне м'ясо, більшу кількість менш досвідчених ворогів. Найважливіші потреби ШІ для цього жанру:

– рух – контроль над ворогами;

– стрільба – точне керування вогнем;

– прийняття рішень – зазвичай прості автомати стану;

– сприйняття – визначення, в кого стріляти та де вони знаходяться;

– пошук шляху – часто (але не завжди) використовується, щоб дозволити персонажам планувати свій маршрут до рівня;

– тактичний ШІ – знову ж таки, часто використовується, щоб дозволити персонажам визначати безпечні позиції для переміщення або для більш просунутих тактик, таких як влаштування засідки [6]. З них перші два це ключові проблеми, які спостерігаються в усіх іграх цього жанру.

1.4.1 Рух та стрільба

Рух є найпомітнішою частиною поведінки персонажа, і, поступаючись лише симуляторам людей, шутери мають найскладніші

набори анімації. Незвично, коли персонажі поєднують десятки чи сотні послідовностей анімації разом з іншими контролерами, такими як інверсна кінематика або фізика ragdoll. Персонаж у F.E.A.R. 2 може працювати, запускати та переглядати все одночасно. Перші два – це анімаційні канали, а третій – це процедурна анімація, керована напрямком, у якому дивиться персонаж (як і напрямок, але не загальний рух стріляючої руки), як зазначено на рисунку 1.1 [6].

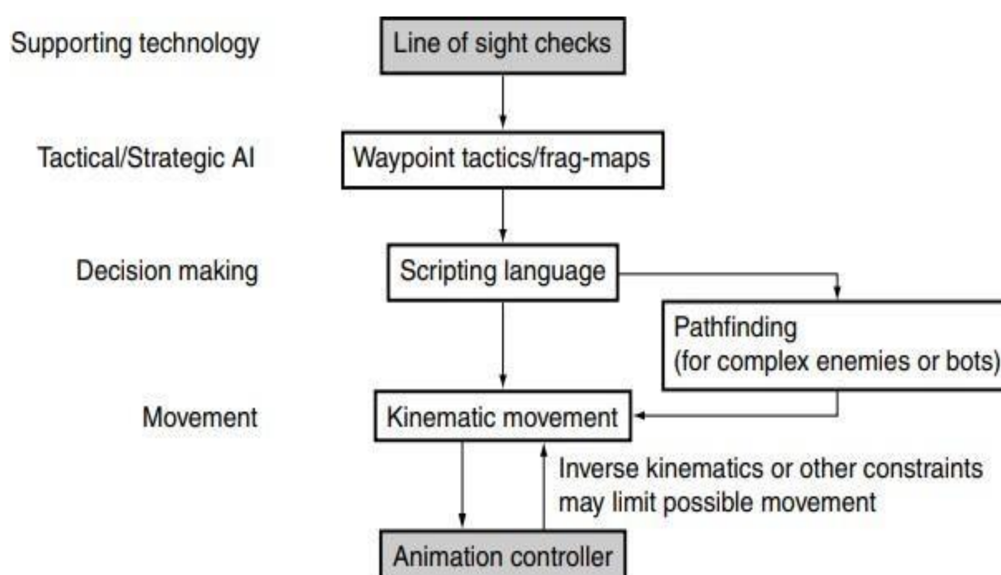

Рисунок 1.1 – Система ігрової механіки

У No One Lives Forever 2 [Monolith Productions, Inc., 2002] ніндзя-персонажі мають складні рухові здібності, що ускладнює синхронізацію руху та анімації. Вони можуть крутити колеса, стрибати через перешкоди та стрибати між будівлями. Просте пересування по рівню стає проблемою. Штучний інтелект повинен не тільки розробити маршрут, але також повинен мати можливість розбити цей рух на анімацію. Більшість ігор

розділяють дві частини: AI вирішує, куди рухатися, а інша частина коду перетворює це на анімацію. Це дає штучному інтелекту повну свободу руху, але має недолік – допускає дивні комбінації анімації та руху, щоможе виглядати приголомшливо для гравця. Цю проблему на сьогоднішнійдень вирішено шляхом додавання багатшої палітри анімацій, що робить більш імовірним пошук розумної комбінації. Кілька ігор, які використовують мови сценаріїв для керування своїми персонажами, надають ті самі елементи керування ШІ, які використовує гравець. Замість того, щоб виводити бажаний рух або цільові локації, штучний інтелект має вказати, наскільки швидко він рухається вперед чи назад, повертається, змінює зброю тощо [6]. Це дозволяє дуже легко під час розробки видалити персонажа ШІ та замінити його людиною (наприклад, граючи по мережі). Більшість ігор, у тому числі ті, що базуються на ліцензуванні найвідоміших ігрових движків, мають макрокоманди.

```
1 sleep 3
2 gotoactor PathNodeLoc1
3 gotoactor PathNodeLoc2
4 agentcall Event_U_Wave 1
5 sleep 2
6 gotoactor PathNodeLoc3
7 gotoactor PathNodeLoc0
```

Рисунок 1.2 – Приклад команд заданого маршруту

Через обмежений внутрішній характер рівнів у багатьох шутерах персонажі майже напевно потребують певного маршруту. Це може бути так само просто, як інструкції gotoactor у сценарії Unreal вище, як на рисунку 1.2 або це можебути повна система пошуку шляху. Яку б форму це не прийняло, маршрутів потрібно дотримуватися. Персонаж повинен правильнореагувати на рух інших персонажів. Найчастіше це робиться за допомогою простої сили відштовхування між усіма персонажами. Якщо персонажі

підійдуть занадто близько, то вони розсунуться. У Mace Griffin: Bounty Hunter [Warthog Games, 2003] така ж техніка використовується для уникнення зіткнень між персонажами на землі та між бойовими космічними кораблями під час глибокого космосу. Для створення маршрутів у приміщенні використовується пошук шляху. У космосі замість цього використовується система руху пласта. Потоп у Halo [Bungie Software, 2001] та прибульці у Alien vs. Predator рухаються по стінах і стелі, а також по підлозі [7].

ШІ для стрільби є вирішальним у шутерах. Перші два втілення Doom сильно критикували за неймовірну точність стрільби, розробники сповільнили вхідні снаряди, щоб дозволити гравцеві відійти від шляху, інакше точність була б надзвичайною. Більш реалістичні ігри, такі як Medal of Honor: Airborne і Far Cry 2 використовують моделі стрільби, які дозволяють персонажам промахуватися захоплюючими способами, тобто вони намагаються промахнутися там, де гравець може бачити кулю.

1.4.2 Прийняття рішень

Прийняття рішень зазвичай досягається за допомогою кінцевих автоматів і все частіше за допомогою дерев поведінки. Вони можуть бути дуже простими за допомогою лише поведінки «видимий гравець» і «невидимий гравець». Дуже поширеним підходом до прийняття рішень у шутерах є розробка системи сценаріїв для ботів. Викликається сценарій, написаний на специфічній для гри мові сценаріїв (яка в деяких випадках скомпільована JIT для прискорення). Сценарій має цілий набір функцій, за допомогою яких він може визначити, що може сприйняти персонаж. Зазвичай вони реалізуються шляхом безпосереднього опитування поточного стану гри [7]. Потім сценарій може вимагати виконання дій, включаючи відтворення анімації, рух і в деяких випадках запити пошуку шляху. Потім ця мова сценаріїв стає доступною для користувачів гри, щоб

вони могли змінювати ШІ або створювати власних автономних персонажів. Це підхід, який використовується в Unreal [Epic Games, 1998] і наступних іграх, і він починає використовуватися в нешутерах, таких як Neverwinter Nights [Bioware, 2002] (як інструмент виключно для дизайнерів рівнів, коли це не доступний для кінцевих користувачів, це набагато частіше). Що стосується Sniper Elite [Rebellion, 2005], Rebellion хотів бачити невідповідну поведінку, яка була б різною в кожній грі. Щоб досягти цього, вони застосували ряд автоматів стану, що працюють на шляхових точках на рівні гри. Багато поведінки залежали від дій інших персонажів або зміни тактичної ситуації в найближчих точках. Невелика випадковість у процесі прийняття рішень дозволяла персонажам щоразу поводитися по-різному та діяти у очевидній співпраці, не потребуючи ШІ на основі загону.

Дещо інший підхід до автономного штучного інтелекту був створений у No One Lives Forever 2 [Monolith Productions, Inc., 2002]. Monolith поєднав автомати стану з цілеспрямованою поведінкою. Кожен персонаж мав би заздалегідь визначений набір цілей, які могли б вплинути на його поведінку. Персонажі періодично оцінювали свої цілі та вибирали ту, яка була для них найбільш актуальною на той момент. Тоді ця мета контролювала б поведінку персонажа. Всередині кожної цілі був кінцевий автомат, який використовувався для керування персонажем, доки не було обрано іншу ціль. У грі використовуються шляхові точки (які вони називаються вузлами), щоб переконатися, що персонажі знаходяться в правильному положенні для таких дій, як перегляд шаф для документів, використання комп'ютерів і ввімкнення світла. Наявність цих шляхових точок поблизу персонажа дозволяє персонажу зрозуміти, які дії доступні. Механізм ШІ Monolith продовжує вдосконалюватися. У F.E.A.R. [Monolith Productions, Inc., 2005], використовується така сама цільова орієнтована поведінка, але попередньо створені автомати замінюються механізмом планування, який намагається поєднати доступні дії таким чином, щоб

досягти мети. F.E.A.R. мав одну з перших повноцінних цільових систем планування дій. У Halo 2 і Halo 3 [Bungie Software, 2004, Bungie Software, 2007] дерева рішень використовувалися, щоб дозволити ШІ-персонажам здійснювати елементарне планування під час своїх дій. Коли вузли в селекторі в дереві поведінки виходять з ладу, ШІ повертається до інших вузлів, що представляють інші плани, надаючи ШІ широкий спектр тактичних можливостей, які було б важко визначити за допомогою кінцевого автомата [8].

1.4.3 Сприйняття

Сприйняття іноді фальсифікується, розміщуючи радіус навколо кожного персонажа ворога, і цей ворог «оживає», коли гравець знаходиться всередині нього. Це підхід, використаний в оригінальному Doom. Однак після успіху Goldeneye 007 [Rare Ltd., 1997] очікувалося більш складне моделювання сприйняття. Це не обов’язково означає систему управління відчуттями, але принаймні персонажі повинні бути поінформовані про те, що відбувається навколо них через певні повідомлення. В іграх Ghost Recon [Red Storm Entertainment, Inc., 2001] симуляція сприйняття є значно складнішою. Система управління відчуттями, яка надає інформацію персонажам зі штучним інтелектом, враховує кількість розбитого покриття, створеного кущами, і перевіряє фон за персонажами, щоб визначити, чи збігається їхній камуфляж. Це досягається шляхом збереження набору ідентифікаторів шаблонів для кожного матеріалу в грі. Перевірка прямої видимості проходить через будь-який частково прозорий об’єкт, доки не досягне персонажа, який тестується. Потім він продовжує діяти за межами персонажа та визначає наступне, з чим стикається. Потім ідентифікатор камуфляжу та ідентифікатор фонового матеріалу перевіряються на сумісність. Ігри Splinter Cell [Ubisoft Montreal Studios, 2002] використовують іншу тактику.

Оскільки є лише один персонаж гравця (у Ghost Recon їх багато), кожен ШІ просто перевіряє, чи його видно. Кожен рівень може містити динамічні тіні, туман та інші ефекти приховування. Персонаж гравця перевіряється закожним із них, щоб визначити рівень приховування. Якщо це значення нижче певного порогу, це означає, що ворожий ШІ помітив персонаж гравця. Рівень приховування не враховує фон, як це роблять ігри Ghost Recon; якщо персонаж стоїть у темній тіні посеред світлого коридору, йогоне буде видно, навіть якщо охоронцям він здасться великою чорною фігурою на світлому фоні. Рівні розроблено таким чином, щоб мінімізувати кількість випадків, коли це обмеження є очевидним [8].

Персонажі зі штучним інтелектом у Splinter Cell також використовують конус зору для перевірки зору, і існує проста звукова модель, у якій звук поширюється в поточній кімнаті до певного радіусу залежно від гучності звуку. Дуже схожі методи використовуються в серії ігор Metal Gear Solid [Konami Corporation, 1998].

1.4.4 Пошук шляху та тактичний ШІ

У Soldier of Fortune 2: Double Helix [Raven Software, 2002] посилання в графі пошуку шляху були позначені типом дії, необхідної для їх проходження. Коли персонаж досягав відповідної ланки на шляху, він міг змінити поведінку, щоб виглядати так, ніби він знає місцевість. Посилання може являти собою перешкоду для склепіння, двері для відкриття, бар’єр для прориву або стіну для спуску. Відповідальна команда ШІ Крістофер Рід і Бен Гайслер називають цей підхід «вбудованою навігацією». Стає майже універсальним використання певної тактики маршрутних точок у шутерах. В оригінальній Half-Life [Valve, 1998] штучний інтелект використовує шляхові точки, щоб визначити, як оточити гравця. Групу персонажів ШІ буде скоординовано таким чином, щоб вони займали набір хороших оборонних позицій, які оточують поточне місце розташування

гравця, якщо це можливо. У грі персонаж ШІ часто відчайдушно пробігає повз гравця, щоб зайняти флангову позицію. Якщо ваші ворожі персонажі не завжди кидають гравця, як в оригінальному Doom, вам, ймовірно, доведеться реалізувати рівень пошуку шляху. Внутрішні рівні більшості шутерів можна представити за допомогою відносно невеликих графіків пошуку шляху, які швидко шукають. Rebellith використовували ту саму систему шляхових точок для пошуку шляху та тактичного штучного інтелекту в Sniper Elite, тоді як Monolith створили зовсім інше представлення для No One Lives Forever 2 [8]. У рішенні Monolith область, куди міг рухатися персонаж, була представлена перекриваючимися «об'ємами штучного інтелекту», який потім сформував граф пошуку шляху. Шляхові точки його системи дій не брали безпосередньої участі у пошуку шляху (за винятком мети для планування слідом). Під час першого видання цієї книги розробники використовували низку представлень для пошуку шляху. Відтоді використання навігаційних сіток для представлення внутрішніх просторів стало майже (але не зовсім) повсюдним. Потрібно більше зусиль, щоб об'єднати підхід навігаційної сітки з надійним тактичним аналізом, і нерідко можна побачити, що тактичний аналіз на основі сітки виконується поруч із навігаційними сітками для пошуку шляху [10]. Однак існують інші життєздатні підходи. Обсяги пошуку шляхів у Monolith є ще одним підходом, і багато ігор, що розгортаються на відкритому повітрі, все ще покладаються на графіки пошуку шляху на основі сітки. Ігри, які відбуваються переважно в приміщенні, природно розбивають свої рівні на сектори, часто розділені порталами (технологія оптимізації візуалізації). Ці сектори можуть природно діяти як граф пошуку шляху вищого рівня для планування маршруту на великі відстані. Це робить ієрархічні алгоритми пошуку шляхів природним підходом для реалізацій, здатних мати справу з великими рівнями.

1.5 Огляд структури задач обробки природної мови

Ігри завжди піддавалися критиці за те, що вони погано запрограмовані: вони використовують трюки, таємничу оптимізацію та неперевірені технології, щоб отримати додаткову швидкість або чіткі ефекти. Ігровий штучний інтелект нічим не відрізняється. Однією з найбільших бар'єрів між людьми, які займаються ШІ в іграх, і науковцями ШІ є те, що кваліфікується як ШІ. Штучний інтелект для гри – це рівнозначні хакерство (спеціальні рішення та чіткі ефекти), евристика (емпіричні правила, які працюють лише в більшості, але не у всіх випадках) і алгоритми («правильні» речі) [4].

1.5.1 Хаки

Психологічним корелятом є біхевіоризм. Ми вивчаємо поведінку, і, розуміючи, як побудована поведінка, ми розуміємо все, що можемо про те, що поводиться. Як психологічний підхід він має своїх прихильників, але був значною мірою витіснений (особливо з появою нейропсихології). Це падіння моди також вплинуло на штучний інтелект. Хоча колись було цілком прийнято дізнаватися про людський інтелект, створивши машину для його копіювання, зараз це вважається поганою наукою. І не без підстав, зрештою, створення машини для гри в шахи включає алгоритми, які дивляться на десятки ходів вперед. Людина просто не здатна на це. З іншого боку, для внутрішньо ігрового штучного інтелекту біхевіоризм часто є правильним шляхом. Нас не цікавить природа реальності чи розуму, ми хочемо, щоб персонажі виглядали правильно. У більшості випадків це означає починати з людської поведінки та намагатися розробити найпростіший спосіб реалізувати її в програмному забезпеченні. GoodAI в іграх зазвичай працює в цьому напрямку. Розробники рідко створюють чудовий новий алгоритм, а потім запитують себе: «То що я

можу з цим зробити?» Натомість починається з дизайну персонажа та застосовується найелегантніший інструмент, щоб отримати результат [4]. Це означає, що те, що кваліфікується як ігровий ШІ, може бути нерозпізнаним як техніка ШІ. Генерування випадкового числа не є технікою ШІ як такою. У більшості мов є вбудовані функції для отримання випадкового числа, тому немає сенсу давати для цього алгоритм! Але це може спрацювати в неймовірній кількості ситуацій. Хорошим прикладом креативної розробки ШІ є The Sims. Хоча під поверхнею відбуваються досить складні речі, багато поведінки персонажів передається за допомогою анімації. У «Зоряних війнах: Епізод 1 Гонщик» персонажі, які роздратовані, злегка відштовхнуть інших персонажів. У Quake II є команда «жест», за допомогою якої персонажі (і гравці) можуть відкинути свого ворога. Усе це не потребує значної інфраструктури ШІ. Їм не потрібні складні когнітивні моделі, навчання чи генетичні алгоритми. Їм просто потрібен простий фрагмент коду, який виконує анімацію в потрібній точці. Запустити його у потрібному місці набагато легше, ніж намагатися відобразити емоційний стан персонажа через його дії.

1.5.2 Евристика

Евристика – це емпіричне правило, приблизне рішення, яке може спрацювати в багатьох ситуаціях, але навряд чи спрацює в усіх. Люди постійно використовують евристику. Ми не намагаємося прорахувати всі наслідки наших дій. Замість цього ми покладаємося на загальні принципи, які, як виявилося, працювали в минулому. Воно може варіюватися від чогось такого простого, як «якщо ви щось втратите, то повторіть свій шлях» до евристик, які керують нашим життєвим вибором.

Існує компроміс між швидкістю та точністю в таких сферах, як прийняття рішень, рух і тактичне мислення (включно зі штучним інтелектом у настільних іграх). Коли точність приноситься в жертву, це

зазвичай відбувається шляхом заміни пошуку правильної відповіді евристикою. Широкий діапазон евристик можна застосувати до загальних проблем штучного інтелекту, які не потребують певного алгоритму [4].

У Pac-Man привиди захоплюються гравцем, вибираючи маршрут на перехресті, що веде до його поточного положення. Шлях до гравця може бути досить складним; це може включати повернення до себе, і це може бути остаточно марним, якщо гравець продовжує рухатися. Але емпіричне правило (рухатися в поточному напрямку гравця) працює і дає гравцеві достатню компетентність, щоб зрозуміти, що привиди не є абсолютно випадковими у своєму русі. У Warcraft [Blizzard Entertainment, 1994] (та в багатьох інших іграх у реальному часі) існує евристика, яка переміщує персонажа трохи вперед у діапазон зброї дальнього бою, якщо ворог знаходиться за межами досяжності персонажа. Хоча це працювало в більшості випадків, це не завжди був найкращий варіант. Багато гравців були розчаровані, оскільки комплексні захисні споруди йшли з місця, коли вороги підходили близько. Пізніше ігри RTS дозволяли гравцеві вибирати, увімкнути цю поведінку чи ні. У багатьох стратегічних іграх, включно з настільними, різним одиницям або фігурам надається одне числове значення, яке відображає їх «хорошість» [5].

У RTS він може знайти найкращу наступальну одиницю для створення, порівнюючи кількість із вартістю. Чимало корисних ефектів можна досягти, просто маніпулюючи числом. Для цього немає ні алгоритму, ні техніки.

1.5.3 Алгоритми

Створення алгоритмів для підтримки цікавої поведінки персонажів. Хакі та евристики допоможуть багато пройти, але покладатися виключно на них означає, що доведеться постійно винаходити велосипед.

2 РОЗРОБКА ВИМОГ ДО РОЗРОБЛЮВАЛЬНОЇ СИСТЕМИ

2.1 Розробка системних вимог до системи ШІ

Розробка системи штучного інтелекту передбачає визначення системних вимог, які має задовольняти система. Системні вимоги – це характеристики системи, які визначають, як вона повинна працювати та взаємодіяти з користувачем та зовнішніми системами.

Основні системні вимоги до системи штучного інтелекту можуть включати такі пункти:

– функціональність: система повинна мати здатність аналізувати та обробляти дані, навчатися на основі даних та приймати рішення на основі цього аналізу;

– ефективність: система повинна бути ефективною та швидко працювати навіть з великими обсягами даних.

– надійність: система повинна бути надійною та стійкою до відмов;

– масштабованість: система повинна бути здатна масштабуватисядля обробки великого об'єму даних та для виконання завдань у режимі реальної години;

– безпека: система повинна бути захищеною від несанкціонованого доступу та зберігати конфіденційні дані у безпеці;

– зручність використання: система повинна мати зручний та легкий інтерфейс користувача, що дозволяє користувачам легко взаємодіяти з системою;

– сумісність: система повинна бути сумісною з існуючими технологіями та системами;

– документація та підтримка: система повинна мати докладну документацію та підтримку, що дозволяє користувачам легко знайти потрібну інформацію та отримати допомогу у разі проблем із системою.

Розробка системи штучного інтелекту може бути складним

завданням, але правильне визначення системних вимог може допомогти.

2.2 Розробка функціональних вимог до системи ШІ

IDEF0 – широко застосовувана техніка для структурованого аналізу та проектування систем. Його використання в покращенні продуктивності та комунікацій в інтегрованих комп'ютерних виробничих системах, а останнім часом і як інструменту для реінжинірингу бізнес-процесів широко задокументовано.

На рисунку 2.1 відображений основний бізнес-процес системи, а саме навчання штучного інтелекту за допомогою методології IDEF0.

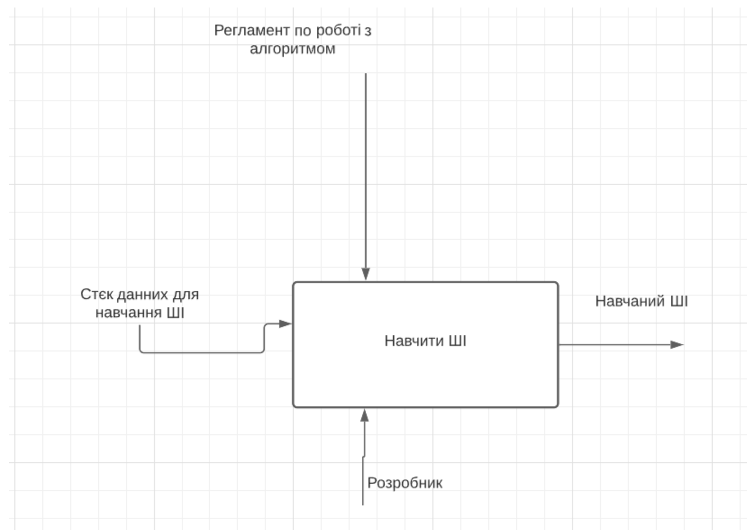

Рисунок 2.1 – Контексна діаграма IDEF0

Ліва стрілка означає «вхідні дані», які находять у бізнес-процес. На вхід до головного процесу надходять наступні дані: стек даних для навчання ШІ – це всі данні на яких буде навчатись ШІ.

Нижня стрілка «механізм» визначає яким саме чином буде виконуватися бізнес-процес, тобто за допомогою якого програмного продукту або за допомогою яких ресурсів (техніка, співробітники) вхід

буде переходити у вихід. До механізмів, які надходять до бізнес-процесу, належить робітник.

Верхня стрілка «управління» це установчий документ, який визначає правильність та послідовність виконання процесу. До документів даної інформаційної системи відносять регламент по роботі з алгоритмом.

Права стрілка означає «вихід», в якості виходу може виступати готова продукція чи програмний продукт (документи). Кожна дія повинна мати хоча б один вихід, інакше такий процес не має сенсу. Отже, на виході отримуємо навчаний ШІ.

На діаграмі декомпозиції (рисунок 2.2.) демонструються функції процесувиконання програми.

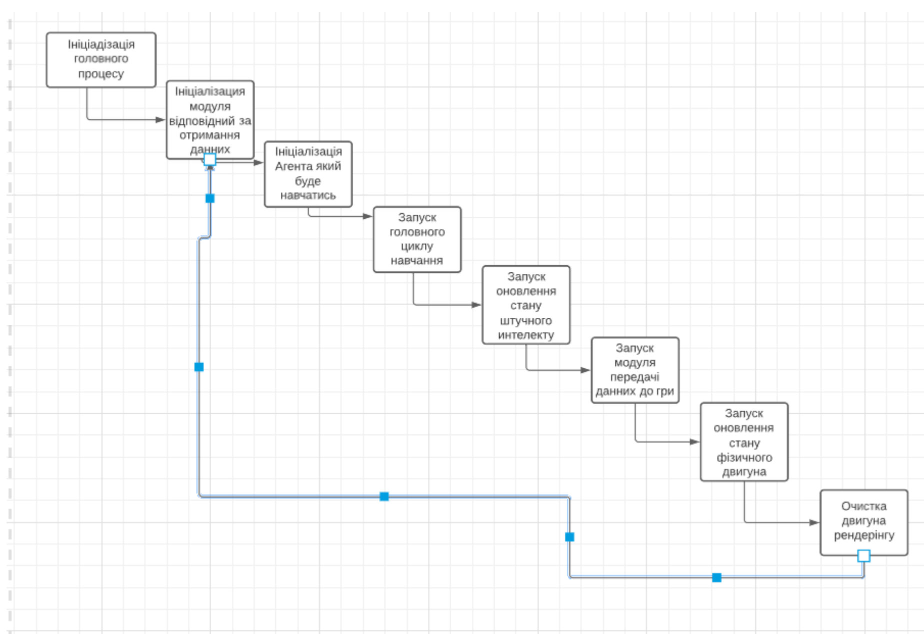

Рисунок 2.2 – Декомпозована діаграма IDEF0

Спочатку ініціюєтся головний процес запуску програми в ньому ініціалізуется всі бібліотеки які потрибні будуть у роботі.

Далі на другому блоці проходить ініціалізація модуля який буде працювати з данними, а саме займатись їх сбором.

Третій блок виконує ініціалізацію Agent, компоненті якого

відповідають за навчання штучного інтелекту.

Після цього починається головний нескінченний цикл програми, доки результат навчання не буде нас задовільняти (рисунок 2.3).

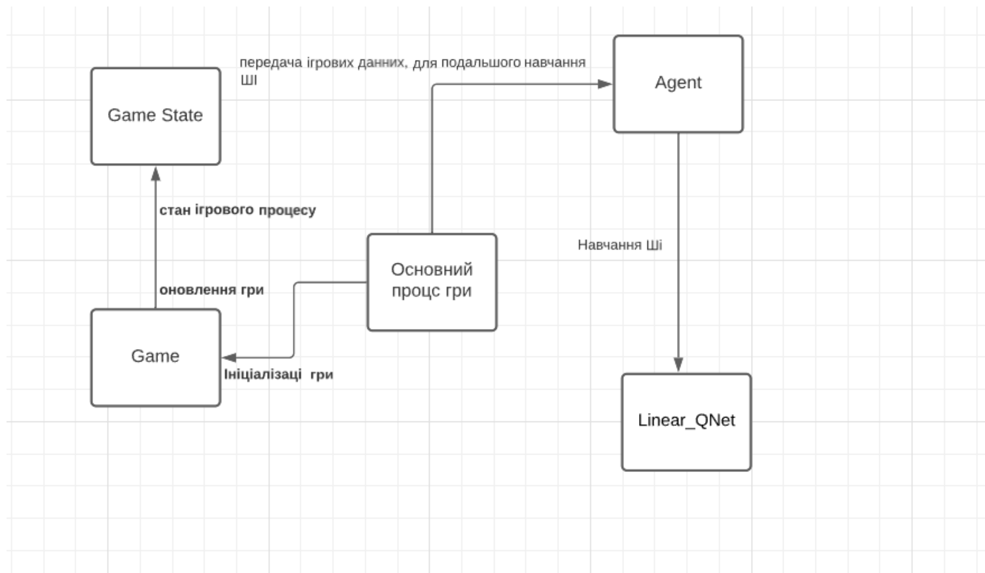

Рисунок 2.3 – Контекстна діаграма потоків даних(DataFlowDiagrams)

2.3 Розробка моделей потоків даних систем ШІ

Розробка моделей потоків даних є важливим етапом при проектуванні систем ШІ. Модель потоків даних допомагає визначити, як дані будуть рухатися в системі, як вони будуть оброблятися та передаватися між компонентами системи.

DFD графічно представляє функції або процеси, які фіксують, маніпулюють, зберігають та розподіляють дані між системою та її середовищем, а також між компонентами системи. Візуальне представлення робить його хорошим інструментом спілкування між користувачем та конструктором системи.

Основні складові моделі потоків даних:

– джерело даних: компонент або система, що постачає дані до

системи ШІ;

– обробка даних: компоненти, що відповідають за обробку та аналіз даних, включаючи алгоритми машинного навчання та інші методи ШІ;

– зберігання даних: компоненти, що відповідають за збереження даних, такі як бази даних, файлові системи та інші;

– відправлення даних: компоненти, що відповідають за передачу даних між різними компонентами системи;

– прийом даних: компоненти, що відповідають за прийом та обробку даних, що надходять до системи.

– відображення результатів: компоненти, що відповідають за відображення результатів роботи системи ШІ, наприклад, візуалізація даних.

3 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ ПРИ РОЗРОБЦІ СИСТЕМИ

3.1 Обґрунтування вибору мови програмування

Оскільки все більше організацій розробляють додатки на основі штучного інтелекту, важливо використовувати мову програмування, яка зменшує складність коду та забезпечує легке його впровадження. Сьогодні велика увага приділяється штучному інтелекту (AI), науці про дані та машинному навчанню (ML) за допомогою Python. чому Оскільки він пропонує прості у використанні та гнучкі інструменти, є розширюваним, має велику кількість бібліотек і велику спільноту розробників Python.

У цій статті ми розглянемо, чому Python є найкращим вибором для штучного інтелекту та машинного навчання та як він порівнюється з іншими популярними мовами.

Популярність Python у AI/ML.

Величезна кількість бібліотек і фреймворків. Створення додатків AI/ML є складним і трудомістким. Однак існує багато бібліотек, сумісних з Python. Це головна причина, чому розробники віддають перевагу цьому над іншими мовами.

Наприклад, бібліотека Scikit-learn надає низку реалізацій алгоритмів ML, таких як лінійна регресія, логістична регресія, опорні векторні машини тощо. Інші бібліотеки включають spaCy, Natural Language Toolkit (NLTK) тощо. Тим часом TensorFlow, PyTorch та Бібліотеки Keras популярні серед спільноти ШІ. Кілька інших, як-от NumPy, Pandas і Seaborn, дозволяють легко маніпулювати даними. Ці бібліотеки, доступні лише в Python, скорочують час і ускладнюють написання коду.

Легкий синтаксис. Синтаксис Python простий і нагадує повсякденну англійську мову. Це скорочує годину, потрібну розробникам для вивчення та розуміння синтаксису та його впровадження. Крім того,

Python не передбачає використання дужок, оскільки використовує відступи, що знову ж таки зменшує складність.

Немає необхідності перекомпілювати вихідний код. Розробникам не потрібно кожного разу перекомпілювати вихідний код. Навпаки, вони можуть швидко внести зміни та побачити результати. Ця гнучкість є однією з найбільших переваг використання Python.

Незалежний від платформи. Код Python може працювати на різних платформах, таких як Windows, Mac, UNIX і Linux.

Велика підтримка спільноти. Будучи популярною мовою програмування з відкритим кодом, Python має велику глобальну спільноту. База користувачів готова допомогти та вітає участь усіх рівнів. Це також корисно під часналагодження коду.

Філософія Python. Основна філософія Python узагальнена в документі «Дзен Python» Тіма Пітерса, як зазначено нижче:

– красиве краще, ніж потворне;

– явне краще, ніж неявне;

– просте краще складного;

– комплекс краще, ніж складний;

– плоский краще, ніж вкладений;

– рідкий краще, ніж щільний;

– читабельність має значення;

– особливі випадки недостатньо особливі, щоб порушувати правила;

– хоча практичність перемагає чистоту;

– помилки ніколи не повинні проходити мовчки;

– якщо це явно не замовчено;

– зіткнувшись з двозначністю, відмовтеся від спокуси вгадати;

– має бути один очевидний спосіб зробити це;

– хоча це може бути неочевидним спочатку, якщо ви не голландець;

– зараз краще, ніж ніколи.;

– хоча ніколи не буває краще, ніж зараз;

– якщо реалізацію важко пояснити, це погана ідея;

– якщо впровадження легко пояснити, це може бути хорошою ідеєю.

Як зазначено на рисунку 3.1, немає сумнівів у популярності Python для розробки програм штучного інтелекту та машинного навчання. Четверте видання щорічного опитування розробників Python, проведеного JetBrains і Python Software Foundation, показало, що 85% розробників Python використовували Python як свою основну мову, тоді як решта 15% використовували її як свою додаткову мову. В опитуванні взяли участь понад 28 000 розробників Python.

Parameter	Python	Java	JavaScript	C++
Runtime	Slower	Faster	Faster than Python	Slower
Length of code	Very short	5-10 times longer than Python	Short but longer than Python	5-10 times longer than Python
Community support for AI/ML	Large community	Small community	Small community	Small community
Syntax	Easy to write and similar to the English language	Difficult to master and involves brackets, unlike Python	Difficult to master compared to Python	Difficult to master and involves brackets, unlike Python

Рисунок 3.1 – Порівняння язиків програмування

Встановлено, що JavaScript є найпопулярнішою мовою, яку розробники поєднують із Python. Опитування також показало, що більше 50% розробників віддають перевагу Python для обробки даних. Цікаво відзначити, що лише 3% веб-розробників використовували виключно Python, тоді як 73% використовували Python і JavaScript.

Java.

Хоча під час виконання Java вважається швидшою за Python, розробка програми Python займає менше часу через скорочення рядків коду. Коди Python зазвичай у 3-5 разів коротші порівняно з Java. Ця різниця є значною, оскільки програмісти Python не витрачають час на

оголошення типів аргументів або змінних.

Наприклад, під час обчислення a+b у Python інтерпретатор спочатку визначає тип змінних, що не потрібно в Java. Натомість у Java компілятор уже знає тип змінної і, отже, компілюється швидше, ніж Python. Тим часом Python дозволяє перевантажувати оператор у визначених користувачем випадках, що неможливо в Java. З цих причин Python краще характеризувати як «склеючу» мову.

JavaScript.

Як і JavaScript, Python реалізує стиль програмування, який використовує прості функції та змінні. Однак використання JavaScript має недоліки, коли йдеться про підтримку спільноти, оскільки користувачі можуть не знайти достатньо матеріалу для створення програм штучного інтелекту та машинного навчання. Python також підтримує об’єктно-орієнтоване програмування, яке дозволяє багаторазове використання класів коду та успадкування. Це не стосується JavaScript.

C++**.**

Те, що написано для Java, також можна застосувати до C++. Однак код Python у 5-10 разів коротший за код C++. Будучи з’єднуючою мовою, він поєднує компоненти в C++.

Очевидно, що величезна кількість переваг, які пропонує Python, зробили його однією з найпопулярніших мов програмування для створення великомасштабних додатків на основі ШІ.

3.2 Опис архітектури (структури) розробленої системи

Архітектура навчання штучного інтелекту з алгоритмом Q-learn відноситься до класу навчання з підкріпленням (reinforcement learning). Вона заснована на використанні функції Q, яка оцінює корисність певної дії конкретної ситуації.

Алгоритм Q-learn працює за таким принципом: агент приймає

рішення, засновані на його поточному стані та очікуваній нагороді за кожну доступну дію. Ця очікувана нагорода обчислюється за допомогою функції Q, яка оцінює сумарну нагороду, яку агент може отримати в майбутньому, вибравши цю дію.

На початку процесу навчання функція Q заповнюється випадковими значеннями. Агент починає дослідження навколишнього середовища, вибираючи дії випадковим чином і оновлюючи значення функції Q на основі отриманих нагород. У міру того, як навчання просувається, агент стає більш досвідченим і оновлює функцію Q на основі більш точних оцінок.

Одна з найпоширеніших архітектур для навчання з алгоритмом Q-learn є нейронна мережа з функцією активації ReLU та останнім шаром, вихід якого відповідає значенням функції Q для кожної доступної дії. В процесі навчання нейронна мережа отримує на вхід поточний стан середовища та вибирає найбільш вигідну дію на основі значень функції Q. Нейронна мережа навчається шляхом мінімізації різниці між очікуваною нагородою (обчислюваною за допомогою функції Q) та фактичною нагородою, отриманою агентом за обрану дію.

4 АНАЛІЗ ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ АЛГОРИТМУ

4.1 Обрання алгоритму навчання

Розглянемо, наприклад, проблему навчання грі в шахи. Контрольованому навчальному агенту потрібно повідомити правильний хід для кожної позиції, з якою він стикається, але такий зворотний зв'язок доступний рідко. За відсутності зворотного зв'язку від вчителя агент може вивчити модель переходу для власних ходів і, можливо, навчитися передбачати рухи суперника, але без зворотного зв'язку про те, що добре, а що погано, агент не матиме підстав для цього. вирішити, який хід зробити. Агент повинен знати, що сталося щось хороше, коли він (випадково) поставив опоненту мат, і що сталося щось погане, коли йому поставили мат – або навпаки, якщо гра – це шахи-самогубці. Цей вид зворотного зв'язку називається винагородою підкріпленням. У таких іграх, як шахи, підкріплення отримується лише в кінці партії. В інших середовищах нагороди приходять частіше.

У пінг-понгу кожне набране очко можна вважати нагородою; при навчанні повзання будь-який рух вперед є досягненням. Таким чином, здається, тварини налаштовані сприймати біль і голод як негативну винагороду, а задоволення та споживання їжі як позитивну винагороду. Підкріплення ретельно вивчалося зоопсихологами вже понад 60 років.

Оптимальна політика – це політика, яка максимізує очікувану загальну винагороду. Завдання навчання з підкріпленням полягає в тому, щоб використовувати спостереження винагороди для вивчення оптимальної (або майже оптимальної) політики щодо навколишнього середовища. Уявіть, що ви граєте в нову гру, правила якої ви не знаєте; після ста чи близько того ходів ваш опонент оголошує: «Ви програли». У двох словах це навчання з підкріпленням. У багатьох складних областях навчання з підкріпленням є єдиним можливим способом навчити програму

працювати на високих рівнях. Наприклад, під час гри людині дуже важко надати точні та послідовні оцінки великої кількості позицій, які потрібні для навчання функції оцінки безпосередньо на прикладах. Натомість програмі можна повідомити, коли вона виграла чи програла, і вона може використовувати цю інформацію, щоб дізнатися функцію оцінки, яка дає досить точні оцінки ймовірності виграшу з будь-якої позиції. Так само надзвичайно важко запрограмувати агента керувати вертольотом; але отримав відповідну негативну винагороду за падіння, хитання або відхилення від заданого курсу, агент може навчитися літати сам. Навчання з підкріпленням може розглядатися як таке, що охоплює весь ШІ: агент поміщається в середовище і повинен навчитися успішно поводитися в ньому. Щоб розділ був керованим, ми зосередимося на простих середовищах і простих дизайнах агентів. Здебільшого ми будемо припускати, що середовище повністю доступне для спостереження, так що поточний стан забезпечується кожним сприйняттям. З іншого боку, агент не знає, як працює середовище або що роблять його дії, і ми припустимо ймовірні результати дії. Таким чином, агент стикається з невідомим марковським процесом прийняття рішень.

Розглянемо три конструкції агента:

– агент, заснований на корисності, вивчає функцію корисності на станах і використовує її для вибору дій, які максимізують корисність очікуваного результату.

– агент Q-learning вивчає функцію корисності дії або Q-функцію, надаючи exQ-FUNCTION очікувану корисність виконання заданої дії в заданому стані.

– рефлексивний агент вивчає політику, яка безпосередньо відображає стани на дії. Агент, заснований на корисності, також повинен мати модель середовища, щоб приймати рішення, оскільки він повинен знати стани, до яких приведуть його дії. Наприклад, щоб скористатися функцією оцінки гри в нарди, програма гри в нарди повинна знати, які її допустимі ходи та

як вони впливають на позицію дошки. Тільки таким чином він може застосувати функцію корисності до кінцевих станів. Агент Q-навчання, з іншого боку, може порівнювати очікувані корисності для своїх доступних варіантів без необхідності знати їхні результати, тому йому не потрібна модель середовища. З іншого боку, оскільки вони не знають, куди ведуть їхні дії, агенти Q-навчання не можуть дивитися вперед; це може серйозно обмежити їх здатність до навчання, як ми побачимо.

Пасивний Reinforcement Learning.

Щоб спростити речі, ми починаємо з випадку пасивного агента навчання, який використовує представлення на основі стану в повністю доступному для спостереження середовищі. У пасивному навчанні політика агента π є фіксованою: у стані s він завжди виконує дію π(s). Його мета – просто дізнатися, наскільки якісною є політика, тобто дізнатися функцію корисності U π (s). Очевидно, що завдання пасивного навчання подібне до завдання оцінки політики, що є частиною алгоритму ітерації політики. Основна відмінність полягає в тому, що агент пасивного навчання не знає моделі переходу P(s ′ | s, a), яка визначає ймовірність досягнення стану s ′ зі стану s після виконання дії a; також не знає функції винагороди R(s), яка визначає винагороду для кожного стану.

Агент виконує набір випробувань у середовищі, використовуючи свою політику π. У кожному випробуванні агент починає зі стану (1,1) і переживає послідовність переходів між станами, поки не досягне одного з кінцевих станів (4,2) або (4,3). Його сприйняття забезпечують як поточний стан, так і винагороду, отриману в цьому стані. Типові випробування можуть виглядати так: (1, 1)–0,04 (1, 2)–0,04 (1, 3)–0,04 (1, 2)–0,04 (1, 3)–0,04 (2, 3)–0,04 (3, 3)–0,04 (4, 3)+1 (1, 1) –0,04 (1, 2) –0,04 (1, 3)–0,04 (2, 3)–0,04 (3, 3) –0,04 (3, 2)–0,04 (3, 3)–0,04 (4, 3)+1 (1, 1)–0,04 (2, 1)–0,04 (3, 1)–0,04 (3, 2) –0,04 (4, 2)–1.

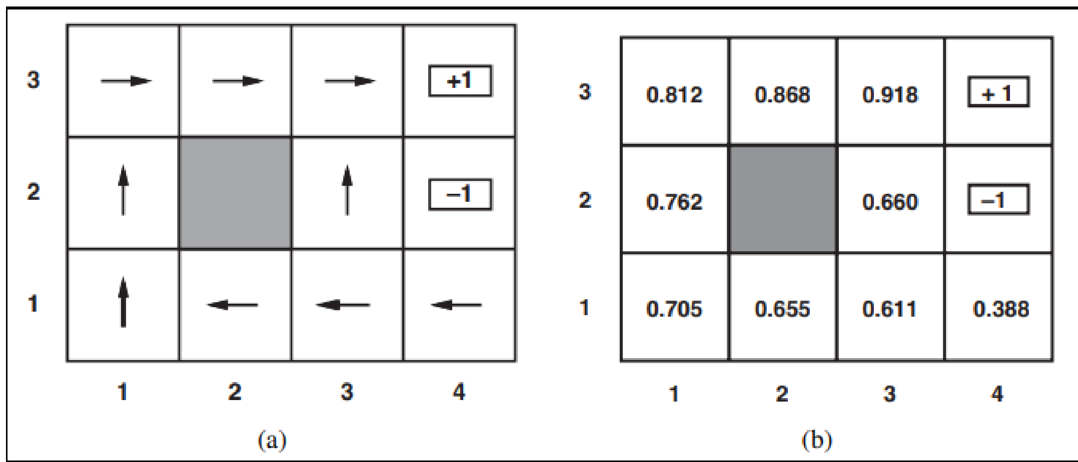

Рисунок 4.1 – Винагорода за певну дію

Зауважте, що кожен перцепт стану підписується отриманою винагородою. Мета полягає в тому, щоб використати інформацію про винагороди, щоб дізнатися очікувану корисність U π (s), пов’язану з кожним нетермінальним станом s. Корисність визначається як очікувана сума винагород, отриманих якщо дотримується політики π. Ми записуємо U π (s) = E "X∞ t = 0 γ tR(St) # (21.1), де R(s) є винагородою за стан, є станом, досягнутим у момент часу t під час виконання політики π, iS0 = s. Ми включимо коефіцієнт дисконтування γ в усі наші рівняння, але для світу 4 × 3 ми встановимо γ = 1.

Активний Reinforcement Learning.

Пасивний навчальний агент має фіксовану політику, яка визначає його поведінку. Активний агент повинен вирішити, які дії вжити. По-перше, агенту потрібно буде вивчити повну модель із ймовірностями результатів для всіх дій, а не лише модель для фіксованої політики. Для цього чудово підійде простий механізм навчання, який використовує PASSIVE-ADP-AGENT. Далі потрібно враховувати те, що агент має можливість вибору дій. Утиліти, які йому потрібно вивчити, визначені оптимальною політикою; вони підкоряються рівнянням Беллмана, які ми повторюємо тут для зручності: U(s) = R(s) + γ max a X s ′ P(s ′ | s, a)U(s ′).

Ці рівняння можна розв’язати для отримання функції корисності U за допомогою алгоритмів ітерації значень або ітерації політики. Останнє питання полягає в тому, що робити на кожному кроці. Отримавши функцію корисності U, яка є оптимальною для вивченої моделі, агент може отримати оптимальну дію за допомогою однокрокового перегляду, щоб максимізувати очікувану корисність; альтернативно, якщо він використовує ітерацію політики, оптимальна політика вже доступна, тому він повинен просто виконати дію, рекомендовану оптимальною політикою.

Розвідка.

На рисунку 4.2 показано результати однієї послідовності випробувань для агента ADP, який дотримується рекомендацій щодо оптимальної політики для вивченої моделі на кожному кроці. Агент не вивчає справжні утиліти чи справжню оптимальну політику! Натомість відбувається те, що під час 39-го випробування він знаходить політику, яка досягає винагороди +1 за нижнім маршрутом через (2,1), (3,1), (3,2) і (3,3). Після експериментів із незначними варіаціями, починаючи з 276-го випробування і далі, він дотримується цієї політики, ніколи не вивчаючи утиліти інших станів, а GREEDY AGENT ніколи не знаходить оптимального маршруту через (1,2), (1,3) та (2,3). Ми називаємо цього агента жадібним агентом. Багаторазові експерименти показують, що жадібний агент дуже рідко наближається до оптимальної політики для цього середовища, а іноді наближається до дійсно жахливої політики. Відповідь полягає в тому, що вивчена модель не збігається зі справжнім середовищем; те, що є оптимальним у вивченій моделі, тому може бути неоптимальним у справжньому середовищі. На жаль, агент не знає, що таке справжнє середовище, тому він не може обчислити оптимальну дію для справжнього середовища. Таким чином, агент повинен зробити компроміс між експлуатацією для дослідження, щоб максимізувати свою винагороду, як це відображено в його поточних оцінках корисності, та розвідкою для максимального

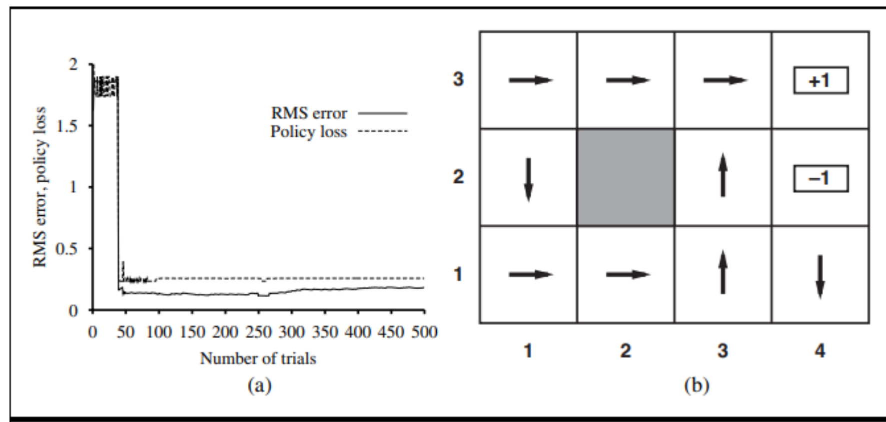

Рисунок 4.2 – Game Specifics Subsystem

збільшити його довгострокове благополуччя. Чиста експлуатація ризикує застрягти в колії. Чисте дослідження для вдосконалення своїх знань не принесе користі, якщо ви ніколи не застосовуєте ці знання на практиці. У реальному світі людині постійно доводиться вирішувати між тим, щоб продовжувати комфортне існування чи вирушати в невідоме в надії відкрити нове та краще життя. Незважаючи на те, що проблеми бандитів надзвичайно важко розв’язати, щоб отримати оптимальний метод дослідження, тим не менш, можна придумати розумну схему, яка зрештою призведе до оптимальної поведінки агента. Схема GLIE повинна спробувати кожну дію в кожному стані необмежену кількість разів, щоб уникнути кінцевої ймовірності того, що оптимальна дія буде пропущена через надзвичайно погану серію результатів. Агент ADP, який використовує таку схему, зрештою дізнається справжню модель середовища. Схема GLIE також повинна з часом стати жадібною, щоб дії агента стали оптимальними щодо вивченої моделі. Існує декілька схем GLIE; один із найпростіших полягає в тому, щоб агент вибрав випадкову дію у частці 1/t часу та дотримувався політики жадібності в іншому випадку. Хоча це врешті-решт наближається до оптимальної політики, воно може бути надзвичайно повільним. Більш розумний підхід надавав би певної ваги діям, які агент робив не дуже часто, водночас намагаючись

уникати дій, які вважаються малокорисними. Це можна реалізувати, змінивши рівняння обмежень так, щоб воно призначало вищу оцінку корисності відносно невивченим парам стан-дія. По суті, це дорівнює оптимістичному пріоритету над можливими середовищами та спонукає агента спочатку поводитися так, ніби повсюди розкидані чудові нагороди. Використовуємо U +(s), щоб позначити оптимістичну оцінку корисності стану s, і нехай N(s, a) буде кількістю спроб дії a в штаті c. Припустімо, що ми використовуємо ітерацію значення в агенті навчання ADP; тоді потрібно переписати рівняння оновлення, щоб включити оптимістичну оцінку. Це робить наступне рівняння: U +(s) ← R(s) + γ max a f P s ′ P(s ′ | s, a)U +(s ′), N(s, a). Тут f(u, n) називається функцією дослідження. Він визначає, як жадібність (надання переваги високим значенням u) поєднується з цікавістю (надання переваги діям, які не часто застосовувалися та мають низький n). Функція f(u, n) повинна бути зростаючою по u і спадною по n. Очевидно, існує багато можливих функцій, які відповідають цим умовам. Одним особливо простим визначенням є f(u, n) = R+, якщо n < Ne u, інакше, де R+ є оптимістичною оцінкою найкращої можливої винагороди, яку можна отримати в будь-якому стані, а Ne є фіксованим параметром.

Це призведе до того, що агент спробує кожну пару дія-стан щонайменше N разів. Той факт, що U +, а не U, з’являється в правій частині рівняння, є дуже важливим. Під час дослідження стани та дії поблизу початкового стану можуть бути спробовані велику кількість разів. Якби ми використали U, більш песимістичну оцінку корисності, агент незабаром би не захотівдосліджувати далі. Використання U + означає, що переваги дослідження поширюються назад від країв недосліджених регіонів, тому дії, які ведуть до недосліджених регіонів, набувають більшої ваги, а не просто дії, які самі по собі незнайомі. Майже оптимальна політика знайдена лише після 18 випробувань. Це пояснюється тим, що агент досить швидко припиняє досліджувати невигідні частини простору

станів, відвідуючи їх лише «випадково». Однак для агента має сенс не піклуватися про точні корисності станів, які, як він знає, є небажаними та яких можна уникнути.

Learning an action-utility function.

Найбільш очевидною зміною порівняно з пасивним випадком є те, що агент більше не оснащений фіксованою політикою, тому, якщо він вивчає функцію корисності U, йому потрібно буде вивчати модель, щоб мати можливість вибрати дію на основі U за допомогою поетапного перегляду. Проблема отримання моделі для агента TD ідентична проблемі для агента ADP. Можливо, дивно, що правило оновлення залишається незмінним. Це може здатися дивним з наступної причини: припустімо, що агент робить крок, який зазвичай веде до гарного пункту призначення, але через недетермінізм у середовищі агент потрапляє в катастрофічний стан.

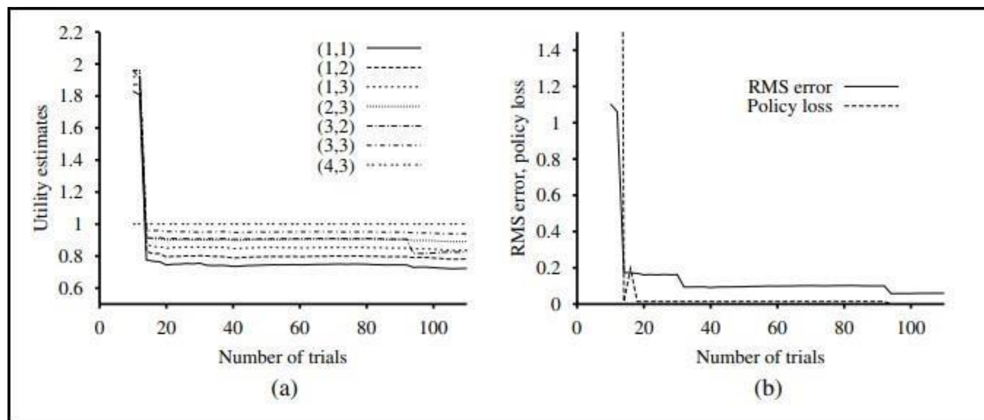

Рисунок 4.3 – Learning an action-utilize function

Правило оновлення TD сприйме це так само серйозно, як якби результат був звичайним результатом дії, тоді як можна припустити, що, оскільки результат був випадковим, агенту не варто надто турбуватися про це. Насправді, звичайно, малоймовірний результат траплятиметься лише рідко у великому наборі тренувальних послідовностей; отже, у довгостроковій перспективі його наслідки будуть зважені пропорційно

його ймовірності, як ми сподіваємося. Ще раз можна показати, що алгоритм TD буде сходитися до тих самих значень, що й ADP, оскільки кількість тренувальних послідовностей прагне до нескінченності. Існує альтернативний метод TD, який називається Q-навчанням, який вивчає представлення корисності дії замість вивчення корисностей. Ми будемо використовувати позначення Q(s, a) для позначення значення виконання дії a у стані s. Значення Q безпосередньо пов'язані зі значеннями корисності таким чином: U(s) = max a Q(s, a). Q-функції можуть здаватися просто ще одним способом зберігання інформації про корисність, але вони мають дуже важливу властивість: агент TD, який вивчає Q-функцію, не потребує моделі у формі P(s ′ | s, a), або для навчання, або для вибору дії. З цієї причини Q-навчання називають безмодельним методом. Як і у випадку з корисностями, ми можемо написати рівняння обмежень, яке має БЕЗМОДЕЛЬНО зберігати рівновагу, коли Q-значення правильні: Q(s, a) = R(s) + γ X s ′ P(s ′ | s, a) max a ′ Q(s ′ , a′). Як і в агенті навчання ADP, ми можемо використовувати це рівняння безпосередньо як рівняння оновлення для ітераційного процесу, який обчислює точні значення Q, враховуючи оцінену модель. Однак для цього потрібно також вивчати модель, оскільки рівняння використовує P(s ′ | s, a). Підхід часових різниць, з іншого боку, не вимагає моделі переходів станів – усіхйому потрібні значення Q. Рівняння оновлення для TD Q-навчання таке: Q(s, a) ← Q(s, a) + α(R(s) + γ max a ′ Q(s ′, a′) − Q(s, a)) , (21.8), який обчислюється кожного разу, коли дія a виконується в стані s, що призводить до стану s ′. Повний дизайн агента дослідницького агента Q- навчання з використанням TD показано на малюнку. Зауважте, що вінвикористовує ту саму функцію дослідження f, що й дослідницький агент ADP, отже, необхідно зберігати статистику вжитих дій (таблиця N). Якщо використовується простіша політика дослідження – скажімо, випадкова дія на деякій частині кроків, де частка зменшується з часом – тоді ми можемо обійтися без статистики. SARSA Q-learning має близького родича під назвою SARSA (для State-

Action-Reward-State-Action). Правило оновлення для SARSA дуже схоже на рівняння (21.8): $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma\, Q(s', a') - Q(s, a))$, (21.9) де a' – фактично виконана дія в стані s'. Правило застосовується в кінці кожної п'ятірки s, a, r, s', a' – звідси й назва. Відмінність від Q-навчання досить непомітна: у той час як Q- навчання резервує найкраще значення Q від стану, досягнутого під час спостережуваного переходу, SARSA чекає, доки фактично не буде виконано дію, і резервує значення Q для цієї дії. Тепер для жадібного агента, який завжди виконує дію з найкращим значенням Q, два алгоритми ідентичні. Однак коли розвідка відбувається, вони суттєво відрізняються. Оскільки Q-навчання використовує найкраще значення Q, воно не звертає уваги на фактичну політику, яка дотримується – це OFF-POLICY – це алгоритм навчання, який не відповідає політиці, тоді як SARSA – цеалгоритм, який відповідає політиці. Q-навчання є ON-POLICY більш гнучким, ніж SARSA, у тому сенсі, що агент Q-навчання може навчитися правильно поводитися, навіть керуючись політикою випадкового чи змагального дослідження. З іншого боку, SARSA є більш реалістичним: наприклад, якщо загальна політика навіть частково контролюється іншими агентами, краще вивчити Q-функцію для того, що насправді станеться, а не для того, що хоче статися агенту. І Q-learning, і SARSA вивчають оптимальну політику для світу 4 × 3, але роблять це набагато повільніше, ніж агент ADP. Це пояснюється тим, що локальні оновлення не забезпечують узгодженість між усіма значеннями Q через модель. Порівняння викликає загальне запитання: що краще вивчати модель і функцію корисності чи вивчати функцію дії- корисності без моделі? Це питання в основах штучного інтелекту. Як ми зазначали в розділі 1, однією з ключових історичних характеристик більшості досліджень штучного інтелекту є їх прихильність до підходу, заснованого на знаннях. Це означає припущення, що найкращий спосіб представити функцію агента – це побудувати представлення деяких аспектів середовища, в якому знаходиться агент. Деякі дослідники, як всередині,

так і поза ШІ, стверджували, що доступність методів без моделі, таких як Q-навчання, означає, що підхід, заснований на знаннях, непотрібний. Однак тут мало чого, крім інтуїції. Наша інтуїція, чого це варте, полягає в тому, що в міру того, як середовище стає складнішим, переваги підходу, заснованого на знаннях, стають більш очевидними. Це підтверджується навіть у таких іграх, як шахи, шашки (шашки) і нарди (див. наступний розділ), де спроби вивчити функцію оцінки за допомогою моделі досягли більшого успіху, ніж методи Q-навчання.

Узагальнення в РІ.

Поки що ми припускали, що функції корисності та Q-функції, отримані агентами, представлені в табличній формі з одним вихідним значенням для кожного вхідного кортежу. Такий підхід досить добре працює для невеликих просторів станів, але час збіжності та (для ADP) час на ітерацію швидко зростають із збільшенням простору. За допомогою ретельно контрольованих, наближених методів ADP можна обробляти 10 000 станів або більше. Цього достатньо для двовимірного середовища, схожого на лабіринт, але про більш реалістичні світи не може бути й мови. Нарди та шахи є крихітними підмножинами реального світу, але їхні простори станів містять порядку 1020 та 1040 станів відповідно. Одним із способів вирішення таких проблем є використання апроксимації функції, яка просто апроксимація функції означає використання будь-якого виду представлення Q-функції, окрім таблиці пошуку. Представлення розглядається як наближене, тому що справжня функція корисності або Q-функція може бути не представлена у вибраній формі. Алгоритм навчання з підкріпленням може вивчати значення для параметрів θ = θ1, . . . , θn так, що функція оцінки Uˆ θ апроксимує справжню функцію корисності. Замість, скажімо, 1040 значень у таблиці, цей апроксиматор функції характеризується, скажімо, n = 20 параметрами – величезне стиснення. Хоча ніхто не знає справжньої функції корисності для шахів, ніхто не вірить, що її можна точно представити 20 числами. Проте, якщо

наближення достатньо хороше, агент все ще може грати в шахи.3 Апроксимація функцій робить практичним представлення функцій корисності для дуже великих просторів станів, але це не є її основною перевагою. Стиснення, досягнуте апроксиматором функції, дозволяє агенту навчання узагальнювати стани, які він відвідав, до станів, які він не відвідував. Тобто найважливішим аспектом апроксимації функції є не те, що вона вимагає менше місця, а те, що вона допускає індуктивне узагальнення над вхідними станами. Щоб дати вам деяке уявлення про потужність цього ефекту: досліджуючи лише один з кожних 1012 можливих станів гри в нарди, можна вивчити корисну функцію, яка дозволяє програмі грати так само добре, як і будь-якій людині. З іншого боку, звісно, існує проблема, що у вибраному просторі гіпотез може не бути жодної функції, яка достатньо добре апроксимує справжню функцію корисності. Як і в будь-якому індуктивному навчанні, існує компроміс між розміром простору гіпотез і часом, необхідним для вивчення функції. Більший простір гіпотез збільшує ймовірність того, що можна знайти хороше наближення, але також означає, що конвергенція, ймовірно, буде відкладена. З апроксимацією функції це екземпляр навчання під наглядом. Для навчання з підкріпленням має сенс використовувати онлайн-алгоритм навчання, який оновлює параметри після кожного випробування.

Припустімо, що ми проводимо випробування, і загальна отримана винагорода, починаючи з (1,1), становить 0,4. Це свідчить про те, що Uˆ θ(1, 1), яке на даний момент становить 0,8, є занадто великим і його потрібно зменшити. Як потрібно налаштувати параметри, щоб цього досягти? Як і під час навчання нейронної мережі, ми пишемо функцію помилок і обчислюємо її градієнт відносно параметрів.

Навчання з підкріпленням (RL) – це підхід до машинного навчання, натхненний біхевіористською психологією та, зокрема, спосіб, у якому люди та тварини вчаться приймати рішення за допомогою (позитивних чи негативних) винагород, отриманих їх середовищем. У навчанні з

підкріпленням зразків гарної поведінки недоступні (як у навчанні під наглядом); натомість, подібно до еволюційного (підкріплення) навчання, сигнальний алгоритм навчання надається середовищу на основі того, як агент взаємодіє з ним, як показано на рисунку 4.5. У певний момент часу агент перебуває в певному стані з і вирішує виконувати дію з усіх доступних дій у поточному стані. У відповідь оточення забезпечує негайну винагороду.

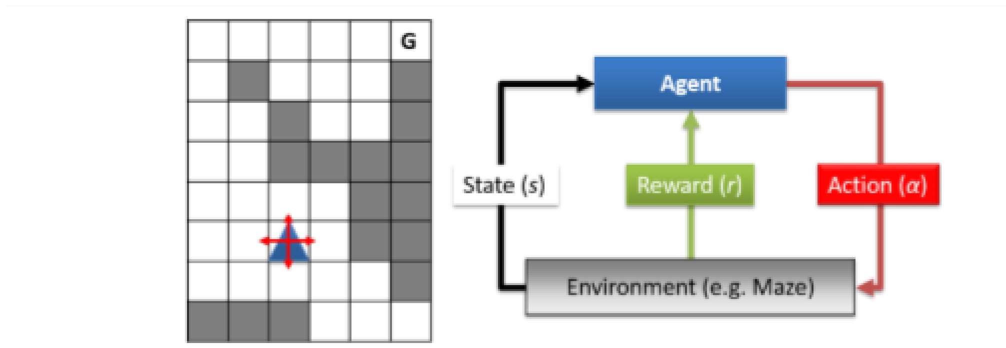

Рисунок 4.5 – Game Specifics Subsystem

Завдяки безперервній взаємодії між агентом і його оточенням, у агента збільшується вибір дій, які максимізують суму винагород. RL вивчався з різних дисциплінарних точок зору, включаючи дослідження операцій, теорії ігор, теорії інформації та генетичних алгоритмів, і успішно застосовувався в задачах, які включають баланс між довгостроковими та короткостроковими винагородами, такими як керування роботами та іграми. Більш формально метаагента виникає в тому, щоб визначити політику (π) для вибору дій, які максимізують розмір довгострокової винагороди, такої як очікувана накопичувальна винагорода. Політика – це стратегія, якою займається агент у виборі дій, враховуючи стан, у який він перебуває. Якщо функція, яка характеризує значення кожної дії, існує або засвоєна, оптимальну політику (π∗) можна отримати, вибравши дію з найвищим значенням. Взаємодія з навколишнім середовищем відбувається

в дискретних кроках у часі ($t = \{0,1,2,...\}$) і моделюється як марківський процес прийняття рішень (MDP).

MDP додатково:

– S: набір станів $\{s1,...,sn\} \in S$. Стан середовища є функцією інформаційного агента про середовище (тобто вхідних даних агента);

– A: Набір дій $\{a1,...,am\} \in A$, можливих у кожному стані s. Дії представляють різні способи дії агента в середовищі;

– P(s,s,a): ймовірність переходу від s до s за умови a. П дає ймовірність завершення у стані після вибору дій у стані, і це слідує властивості Маркова, яка означає, що майбутні стани процесу залежать лише від поточного стану, а не від конкуренції подій, які йому передували. Як внаслідок, властивість Маркова П робить можливим передбачення однокрокової динаміки;

– R(s,s ,a): функція винагороди при переході від s до s при заданому a. Коли агент у стані s вибирає дію a і переходить до стану s, він отримує негайну винагороду r від середовища.

P і R починає модель світу і представляє, відповідно, динаміку навколишнього середовища (P) і довгострокового винагороду (R) для кожної політики. Якщо модель світу відомо, немає потреби вчитися оцінювати ймовірність переходу та функції винагороди, і ми, таким чином, одночасно розраховуємо оптимальну стратегію (політику), використовуючи підходи на основі моделі, так само як динамічне програмування. Якщо натомість моделі світу невідома, ми наближаємо перехід і функції винагороди, вивчаючи оцінки майбутніх винагород, отриманих шляхом вибору дій a в стані с. Потім ми розраховуємо нашу політику на основі цих оцінок. Навчання відбувається за допомогою безмодельних методів, таких як пошук Монте-Карло та години різницеве навчання [10].

4.2 Основні концепції й таксономія високого рівня

Центральним питанням у проблемах RL є правильний баланс між використанням поточних отриманих знань і дослідженням нових невидимих територій у просторі пошуку. Як випадковий вибір дій (без використання), так і завжди жадібний вибір найкращої дії відповідно до міри ефективності чи винагороди (без дослідження) є стратегіями, які зазвичай дають погані результати в стохастичних середовищах. Популярний і досить ефективний механізм вибору дії RL називається ε-жадібним, який визначається параметром $\varepsilon \in [0,1]$. Відповідно до ε-greedy агент RL обирає дію, яка, на його думку, поверне найбільшу майбутню винагороду з імовірністю $1 - \varepsilon$; інакше він вибирає дію рівномірно випадковим чином. Проблеми RL можна класифікувати на епізодичні та наростаючі. У першому класі навчання алгоритму відбувається офлайн і в межах кінцевого горизонту кількох екземплярів навчання. Кінцева послідовність станів, дій і сигналів винагороди, отриманих у межах цього горизонту, називається епізодом. Методи Монте-Карло, які покладаються на повторну випадкову вибірку, наприклад, є типовим прикладом епізодичного RL. Натомість в останньому класі алгоритмів навчання відбувається онлайн і не обмежене горизонтом [1].

Бутстрапінг є центральним поняттям у RL, яке класифікує алгоритми на основі того, як вони оптимізують значення стану. Бутстреп оцінює, наскільки хороший стан, на основі того, наскільки хорошим, на нашу думку, є наступний стан. Іншими словами, за допомогою початкового завантаження ми оновлюємо оцінку на основі іншої оцінки. Як у навчанні TD, так і в динамічному програмуванні використовується завантаження, щоб вивчати досвід відвідування станів та оновлення їхніх значень. Натомість методи пошуку за методом Монте-Карло не використовують початкове завантаження й таким чином вивчають значення кожного стану окремо. Нарешті, поняття резервного копіювання є центральним у RL і діє

як відмінна риса серед алгоритмів RL. За допомогою резервного копіювання ми повертаємося від стану в майбутньому, st+h, до (поточного) стану, який ми хочемо оцінити, st, і враховуємо значення проміжних станів у наших оцінках. Операція резервного копіювання має дві основні властивості: її глибину, яка варіюється від одного кроку назад до повного резервного копіювання, і її ширину, яка змінюється від (випадково) обраної кількості станів вибірки в межах кожного кроку часу до повної резервної копії. На підставі наведених вище критеріїв ми можемо виділити три основні типи алгоритму RL:

– динамічне програмування. У динамічному програмуванні потрібне знання моделі світу (P і R), а оптимальна політика розраховується за допомогою початкового завантаження;

– методи Монте-Карло. Для методів Монте-Карло знання моделі світу не вимагається. Алгоритми цього класу (наприклад, MCTS) ідеально підходять для офлайнового (епізодичного) навчання, і вони вивчаються за допомогою резервного копіювання на всю вибірку та повної глибини. Однак методи Монте-Карло не використовують завантаження;

– навчання ТД. Як і для методів Монте-Карло, знання моделі світу не вимагається, тому її оцінюють.

Алгоритми цього типу (наприклад, Q-навчання) навчаються на досвіді через завантаження та варіанти резервного копіювання.

4.3 Q-Learning

Q-learning – це алгоритм TD-навчання без моделі, поза політикою, який спирається на табличне представлення значень Q(s,a) (звідси його назва). Неформально Q(s,a) представляє, наскільки добре вибрати дію a у стані s. Формально Q(s,a) є очікуваним дисконтованим підкріпленням виконання дії a у стані s. Агент Q-learning вчиться на досвіді, вибираючи дії та отримуючи винагороди через завантаження [9]. Метою агента Q-

навчання є максимізація очікуваної винагороди шляхом вибору правильних дій у кожному стані. Винагорода, зокрема, є зваженою сумою очікуваних значень дисконтованих майбутніх винагород. Алгоритм Q-навчання – це просто оновлення значення Q ітеративним способом. Первісно таблиця Q має вільне значення, встановлені дизайнером. Тоді кожен раз, коли агент вибирає дію a зі стану s, він відвідує стан s, отримує негативну винагороду r і оновлює своє значення Q(s,a) наступним чином: $Q(s,a) \leftarrow Q(s,a) + \alpha\{r + \gamma \max a\, Q(s\ ,a\ ) - Q(s,a)\}$.

Швидкість навчання оцінює ступінь, до якої нова оцінка для Q замінює стару оцінку. Коефіцієнт знижки зумовлює значимість ранніх винагород із пізнішими; čim ближче γ до 1, тим більша вага надається майбутнім посиленням. Як видно з рівняння (2.10), алгоритм використовує початкове завантаження, оскільки він підтримує оцінки того, наскільки добре є пара стан-дія (тобто Q(s,a)) на основі того, наскільки добрим він вважає наступний стан (тобто Q(s ,a)). Він також використовує повну резервну копію в один крок для оцінки Q, беручи до уваги всі значення Q усіх можливих дій a нещодавно відвіданого стану s . Приведено, що за допомогою правил навчання рівня (2.10) значення Q(s,a) збігаються до очікуваної майбутньої дисконтованої винагороди. цю оптимальну політику можна розрахувати на основі значення Q; агент у стані з вибирає дію a з найбільшим значенням Q(s,a). Підсумовуючи, основні кроки алгоритму такі: дано негайну функцію винагороди r і таблицю значення Q(s,a) для всіх можливих дій у кожній стані: 1. Ініціалізуйте таблицю вільними значеннями Q; наприклад, Q(s,a) = 0. 2. s ← Початковий стан. 3. Поки не закінчено*, виконайте: (a) Виберіть дію a на основі політики, отриманої з Q (наприклад, ε-жадібний). (b) Застосувати дію, перейти до стану s і отримати негайну винагороду r. (c) Оновіть значення Q(s,a) відповідно до (2.10). (г) s ← s . Найчастіше використовуваними умовами завершення є швидкісний алгоритм, тобто зупинка протягом кількох ітерацій, або якість конвергенції, зупинка, якщо ви задоволені отриманою політикою [2].

ВИСНОВКИ

Дослідження та розробка алгоритмів глибокого навчання штучному інтелекту для вирішення ігрових завдань є важливою темою, яка привертає увагу дослідників та розробників у галузі інформаційних технологій. Глибоке навчання використовується для створення штучного інтелекту, який може навчатися на прикладах та вирішувати складні завдання, які раніше були виключно у межах людських здібностей.

Дослідження у сфері глибокого навчання вирішує завдання, пов'язані з поліпшенням ефективності та точності різних видів систем штучного інтелекту. Ці системи можуть використовуватись у різних галузях, таких як медицина, фінанси, виробництво та інші.

Ігри є відмінним засобом для тестування алгоритмів глибокого навчання, оскільки вони можуть мати складні завдання, які потребують швидкого вирішення. Розробка алгоритмів глибокого навчання ігрових завдань може допомогти у розвитку інноваційних продуктів, які використовують штучний інтелект.

Отже, дослідження та розробка алгоритмів глибокого навчання штучному інтелекту для вирішення ігрових завдань є важливим напрямом роботи у галузі інформаційних технологій. Це може сприяти створенню нових продуктів та технологій, які покращують якість життя людей та забезпечують розвиток різних галузей економіки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bulitko, V. et al. 2008. Modeling culturally and emotionally affected behavior. Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 10–15. AAAI Press.

2. Lee, G., V. Bulitko, and E. Ludvig. 2013. Automated story selectionfor color commentary in sports. IEEE Transactions on Computational Intelligence and AI in Games 6(2):144–155.

3. Riedl, M. et al. 2008. Dynamic experience management in virtual worlds for entertainment, education, and training. International Transactions on Systems Science and Applications 4(2):23–42.

4. Mark O. Riedl and Alexander Zook. AI for game production. In IEEE Conference on Computational Intelligence in Games (CIG). IEEE, 2013.

5. William L. Raffe, Fabio Zambetta, and Xiaodong Li. A survey of procedural terrain generation techniques using evolutionary algorithms. In IEEE Congress on Evolutionary Computation (CEC). IEEE, 2012.

6. Ierusalimschy, R. (2006). Programming in Lua (2nd ed.). Published by lua.org.

7. Kourkolis,M. (Ed.), (1986). APP-6 Military symbols for land-based systems. NATO Military Agency for Standardization (MAS).

8. McCulloch,W. S., & Pitts,W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5:115–133.

9. Millington, I. (2007). Game physics engine development. San Francisco: Morgan Kaufmann.

10. Wimmer, H. & Perner, J. (1983). Beliefs about beliefs: Representation and constraining function of wrong beliefs in young children’s understanding of deception. Cognition, 12, 103–128.

11. C. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. Computer Graphics, 21(4), 1987, pp. 25–34.

12. Newell, A. & Simon, H. (1961), GPS, a program that simulates

thought, in H. Billing, ed., Lernende Automaten, R. Oldenbourg, Munich, Germany, pp. 109–124

13. Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. Communications of the Association for Computing Machinery, 19:111–126.

14. Russell, S., & Norvig, P. (2002). Artificial intelligence: A modern approach (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

15. Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(1):67–82.

16. Waltz, F. M., & Miller, J. W. V. (1998). An efficient algorithm for Gaussian blur using finite-state machines. In: Proceedings of the SPIE Conference on Machine Vision Systems for Inspection and Metrology VII

17. U.S. Army Infantry School. (Ed.), (2002). FM 3-06.11 Combined arms operation in urban terrain. Washington DC: Department of the Army.

18. U.S. Army Infantry School. (Ed.), (1992). FM 7-8 Infantry rifle platoon and squad. Washington DC: Department of the Army.

19. Abelson, H., & Sussman, G. J. (1996). Structure and interpretation of computer programs (2nd ed.). Cambridge, MA: MIT Press.

20. Ierusalimschy, R. (2006). Programming in Lua (2nd ed.). Published by lua.org.

ДОДАТОК А

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	66 с.
	Інші документи	
2.	Презентаційні матеріали	___плакатів

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td rowspan="3" colspan="3"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>.</td><td>.</td><td>Прізвище та ініціали.</td><td>Підп.</td><td>Дата</td></tr>
<tr><td colspan="2">Розробив</td><td>Хамаза В.С.</td><td></td><td></td><td rowspan="4">Дослідження та розробка алгоритмів глибокого навчання штучного інтелекту для розв’язання ігрових завдань</td><td>Шифр групи</td><td>Код напр./спец.</td></tr>
<tr><td colspan="2">Перевірив</td><td>Жолудов Ю.Т.</td><td></td><td></td><td>СШІм-21-1</td><td>122</td></tr>
<tr><td colspan="2">Н.контр.</td><td>Малєєва І.А.</td><td></td><td></td><td rowspan="2" colspan="2">ХНУРЕ
кафедра ШІ</td></tr>
<tr><td colspan="2">Затв.</td><td>Філатов В.О.</td><td></td><td></td></tr>
</table>