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 методів машинного навчання в задачах розподілу обчислювальних ресурсів
(тема)

Виконав:
здобувач другого року навчання,
групи ДСм-24-1

Паєвський І.П.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки

(код і повна назва спеціальності)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Науки про дані (Data Science)

(повна назва спеціалізації)
Керівник проф. Кулішова Н.Є.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Лариса ЧАЛА
(підпис) (прізвище, ініціали)

2025 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-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Науки про дані (Data Science)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Паєвському Івану Петровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів машинного навчання в задачах розподілу обчислювальних ресурсів

затверджена наказом університету від 24 листопада 20 25 р. № 1057Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17 грудня 20 25 р.

3. Вихідні дані до роботи python 3.13, git, бібліотеки os, time, random, psutil, csv, torch, torchvision, pandas, numpy, matplotlib, датасет MNIST.

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі та постановка задачі дослідження
2) Огляд літератури та аналіз аналогів
3) Вибір програмного забезпечення
4) Практична розробка та навчання моделі
5) Експериментальний розділ

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	24.11.2025	виконано
2	Дослідження предметної галузі	25.11.2025	виконано
3	Постановка завдання	28.11.2025	виконано
4	Розробка CNN моделі	01.12.2025	виконано
5	Розробка RL моделі	08.12.2025	виконано
6	Проведення експериментів	11.12.2025	виконано
7	Аналіз результатів	12.12.2025	виконано
8	Оформлення пояснювальної записки	13.12.2025	виконано
9	Захист кваліфікаційної роботи	17.12.2025	

Дата видачі завдання 24 листопада 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Кулішова Н.Є.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 84 с., 9 рис., 4 табл., 1 дод., 29 джерел.

МАШИННЕ НАВЧАННЯ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, НЕЙРОННІ МЕРЕЖІ, ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, ПАРАЛЕЛІЗМ, РОЗПОДІЛ ОБЧИСЛЮВАЛЬНИХ РЕСУРСІВ.

Об’єкт дослідження – процес розподілу обчислювальних ресурсів під час навчання моделей машинного навчання.

Предмет дослідження – методи оптимізації використання процесорних ресурсів у багатопроцесних та багатопотокових середовищах, зокрема під час тренування згорткових нейронних мереж, а також особливості застосування алгоритмів навчання з підкріпленням для керування параметрами обчислювального середовища.

Мета роботи – підвищення ефективності методів машинного навчання, насамперед алгоритмів навчання з підкріпленням, у задачах оптимізації розподілу обчислювальних ресурсів, а також продуктивності навчання згорткових нейронних мереж завдяки впливу динамічного керування параметрами процесів і потоків.

Методи дослідження – моделювання, програмна реалізація, експериментальні дослідження, аналіз продуктивності, формалізація.

У роботі проведено дослідження впливу конфігурації обчислювального середовища на швидкість і стабільність навчання згорткових нейронних мереж. Розроблено агент підкріплювального навчання, який автоматично коригує параметри кількості процесів, потоків, використання фізичних і логічних ядер та інші системні метрики з метою підвищення ефективності тренування моделі. Отримані результати демонструють можливість підвищення продуктивності за рахунок адаптивного керування обчислювальними ресурсами.

ABSTRACT

Master’s thesis contains: 84 pp., 9 fig., 4 tabl., 1 ann., 29 references.

MACHINE LEARNING, NEURAL NETWORKS, OPTIMIZATION, PARALLELISM RESOURCE ALLOCATION, PERFORMANCE, REINFORCEMENT LEARNING.

Object of research – the process of allocating computational resources during the training of machine learning models.

The subject of the study is the methods of optimizing the use of processor resources in multi-process and multi-threaded environments, in particular during the training of convolutional neural networks, as well as the specifics of applying reinforcement learning algorithms for managing parameters of the computational environment.

The purpose of the work is to increase the efficiency of machine learning methods, primarily reinforcement learning algorithms, in the tasks of optimizing the allocation of computing resources, as well as the performance of training convolutional neural networks due to the influence of dynamic control of process and flow parameters.

Research methods – modeling, software implementation, experimental studies, performance analysis, formalization.

The paper presents a study of the influence of computational environment configuration on the speed and stability of convolutional neural network training. A reinforcement learning agent was developed to automatically adjust parameters such as the number of processes, number of threads, utilization of physical and logical cores, and other system metrics to improve training efficiency. The results demonstrate the potential for increasing performance through adaptive management of computational resources.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI – Artificial Intelligence – штучний інтелект;

CNN – Convolutional Neural Network – згорткова нейронна мережа;

CPU – Central Processing Unit – центральний процесор;

GPU – Graphics Processing Unit – графічний процесор;

MDP – Markov Decision Process – марківський процес прийняття рішень;

ML – Machine Learning – машинне навчання;

RAM – Random Access Memory – оперативна пам’ять;

ReLU – Rectified Linear Unit – випрямлена лінійна функція активації;

RL – Reinforcement Learning – навчання з підкріпленням;

SMT – Simultaneous Multithreading – симетрична багатопотоковість.

ВСТУП

Сучасні обчислювальні системи характеризуються високою складністю та потребою в раціональному використанні апаратних ресурсів. Паралельно з цим швидко зростає роль методів машинного навчання, зокрема глибинних нейронних мереж, які активно застосовуються в задачах класифікації, прогнозування, обробки зображень і сигналів. Процес їх навчання є ресурсоємним і значною мірою залежить від того, як саме розподілені та використані процесорні ядра, потоки, оперативна пам’ять і допоміжні підсистеми. Навіть невеликі CNN-моделі можуть демонструвати істотні відмінності у швидкості навчання залежно від конфігурації апаратного середовища.

Фреймворки машинного навчання створюють складну багатопоточну структуру виконання: вони запускають потоки для обробки даних, лінійної алгебри, асинхронних операцій та взаємодії з GPU. Це призводить до того, що поведінка моделі під час навчання стає залежною не лише від самої архітектури нейронної мережі, а й від того, як операційна система розподіляє фізичні й логічні ядра, як взаємодіють між собою процеси-працівники, а також від особливостей багатопотокових бібліотек (OpenMP, MKL). У таких умовах актуальним стає питання оптимального керування параметрами обчислювального середовища [1], [2].

Одним із перспективних напрямів вирішення цієї проблеми є використання методів підкріплювального навчання (Reinforcement Learning). RL-агент здатний аналізувати характеристики системи під час навчання моделі та приймати рішення щодо їх корегування. Такий підхід дозволяє динамічно оптимізувати конфігурацію системи в реальному часі.

У даній роботі запропонована система: агент RL, який отримує дані про стан обчислень і коригує параметри процесів та потоків під час навчання згорткової нейронної мережі. Метою дослідження є аналіз методів машинного навчання, придатних для оптимізації розподілу

обчислювальних ресурсів, а також оцінка впливу автоматичного керування параметрами середовища на продуктивність навчання моделей.

Практична значущість роботи полягає в можливості підвищення ефективності використання апаратних ресурсів та скорочення часу навчання моделей. Отримані результати можуть бути використані для побудови адаптивних систем керування навантаженням, оптимізації локальних і хмарних обчислювальних середовищ та підвищення загальної ефективності ML-платформ.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Опис розв’язуваної проблеми

У сучасних обчислювальних системах спостерігається стійка тенденція до зростання складності програмного забезпечення, збільшення обсягів оброблюваних даних та розширення спектра апаратних платформ. Сучасні моделі машинного навчання, зокрема глибокі нейронні мережі, вимагають значних обчислювальних ресурсів та ефективної організації процесу навчання. На практиці робота таких алгоритмів залежить не лише від архітектури моделі чи якості даних, але й від того, наскільки оптимально виконуються низькорівневі налаштування – кількість задіяних процесорних ядер, структура потоків, розмір батчу, швидкість навчання, механізми паралелізації, конфігурація вводу-виводу та багато інших параметрів [3].

Разом з тим, навіть для відносно простих моделей на кшталт згорткових нейронних мереж (CNN), оптимальні налаштування суттєво залежать від характеристик конкретного апаратного забезпечення: кількості фізичних та логічних ядер, доступної оперативної пам’яті, пропускної здатності системи зберігання даних, наявності підтримки апаратної векторизації тощо. Зміна апаратної платформи, навантаження чи внутрішніх параметрів моделі може призводити до істотного падіння продуктивності, збільшення часу навчання або надмірного споживання ресурсів [4]. Таким чином, налаштування середовища виконання машинного навчання є нелінійною, багатофакторною проблемою, в якій важко передбачити оптимальну конфігурацію вручну.

Традиційні підходи до розподілу ресурсів передбачають статичне або напівавтоматичне налаштування параметрів, засноване на досвіді спеціаліста або загальних рекомендаціях інструментів машинного навчання. У більшості випадків такі налаштування не враховують особливості

конкретної моделі, властивостей датасету або специфіки апаратної системи, що нерідко призводить до часткового простоювання обчислювальних потужностей або, навпаки, до надмірного навантаження на процесор. Ручний підбір оптимальних значень стає дедалі складнішим у зв'язку зі збільшенням кількості логічних потоків та паралельних механізмів, які може використовувати сучасний CPU [5].

Паралельний розвиток методів навчання з підкріпленням(Reinforcement Learning, RL) відкриває можливість автоматичного визначення найбільш ефективних конфігурацій параметрів для конкретних умов виконання. RL-агент може виступати адаптивним контролером, який у режимі реального часу спостерігає за метриками навчання моделі – часом виконання епохи, середнім завантаженням CPU, піковим використанням пам'яті, точністю та втратою – і коригує конфігурацію так, щоб забезпечити найкращу продуктивність згідно з обраною функцією винагороди [6]. Це дозволяє перетворити процес налаштування обчислювальних ресурсів на формальну задачу оптимізації.

Проблема, що розглядається у даній роботі, полягає у дослідженні впливу параметрів обчислювального середовища на ефективність навчання моделі CNN та у створенні механізму, який здатен знаходити оптимальні конфігурації за допомогою навчання з підкріпленням. На відміну від класичного тюнінгу гіперпараметрів, тут оптимізуються саме системні параметри, що визначають поведінку моделі на процесорі: кількість логічних потоків (torch_threads), кількість фізичних процесів для підготовки вибірки (num_workers), розмір батчу, швидкість навчання, а також CPU affinity – спосіб прив'язки процесів до конкретних ядер. Це робить задачу міждисциплінарною, оскільки вона поєднує машинне навчання, системне програмування та моделювання продуктивності.

Додаткову складність становить мультикритеріальний характер задачі. Оптимальна конфігурація повинна враховувати кілька взаємопов'язаних метрик ефективності. Наприклад, зменшення часу

навчання може супроводжуватися зростанням споживання пам’яті, а збільшення кількості потоків – призвести до зниження продуктивності через підвищену частоту перемикань контексту. Тому RL-контролер має здатність не лише шукати оптимум, але й балансувати між суперечливими критеріями.

Таким чином, основна проблема полягає у розробленні підходу до автоматизованого розподілу обчислювальних ресурсів при навчанні моделей машинного навчання з використанням навчання з підкріпленням. Такий підхід дозволяє мінімізувати залежність від ручного налаштування та забезпечити адаптивність системи до умов конкретної апаратної платформи.

1.2 Актуальність дослідження

Стрімкий розвиток інтелектуальних систем та зростання попиту на машинне навчання вимагають більш раціонального підходу до використання обчислювальних ресурсів. Хоча більшість досліджень зосереджені на GPU, значна частина реальних застосувань у промисловості, наукових установах та локальних середовищах продовжує працювати на CPU. Це пов’язано із доступністю, універсальністю та економічною доцільністю використання центральних процесорів для широкого спектра задач – від навчання невеликих моделей до обчислень на периферійних пристроях.

У таких умовах постає критична потреба у підвищенні ефективності CPU-орієнтованих обчислень. Через збільшення кількості ядер та потоків сучасні процесори здатні забезпечувати високу продуктивність, але лише за умови коректного їх використання. У реальних сценаріях більшість систем залишають значну частину обчислювального потенціалу незадіяною через відсутність автоматичних механізмів оптимізації. Як наслідок, моделі навчаються повільніше, енергоспоживання зростає, а обчислювальні

витрати – особливо у хмарних середовищах – стають фінансово невигідними.

Додатковим фактором актуальності є ускладнення архітектур самих ML-фреймворків (TensorFlow, PyTorch). Вони надають можливість керувати потоками, прискорювати лінійну алгебру, розділяти обчислення між процесами та ядрами, однак не пропонують інструментів, які автоматично знаходять оптимальні значення цих параметрів [7]. Розробнику доводиться експериментувати вручну, що є не лише повільним, але й часто неефективним.

Методи навчання з підкріпленням дедалі частіше застосовуються для задач оптимізації в інфраструктурі: керування хмарними ресурсами, балансування навантаження, оптимізація енергоспоживання тощо. Їх здатність адаптуватися до змін навколишнього середовища робить RL природним інструментом для задачі оптимального використання ресурсів під час навчання ML-моделей. Актуальність такого підходу зростає в умовах, коли системи повинні динамічно реагувати на зміну апаратного середовища або навантаження [8], [9].

З наукової точки зору дослідження є важливим, оскільки поєднує два напрями: моделювання продуктивності систем машинного навчання та використання RL для автоматизації керування ресурсами.

Таке поєднання відкриває шлях до створення універсальних систем, здатних самостійно оптимізувати процес навчання моделей без участі інженера.

З прикладної точки зору результат роботи може бути використаний:

– у локальних робочих станціях для прискорення навчання моделей;

– на серверних CPU-системах із великою кількістю потоків;

– у хмарних середовищах, де оптимізація знижує фінансові витрати;

– у наукових обчисленнях, де важлива повторюваність та адаптація експериментів;

– у вбудованих системах та edge-пристроях, які обмежені апаратно.

Тому актуальність дослідження визначається зростанням складності ML-систем, потребою у ефективному використанні CPU-ресурсів та відсутністю універсальних інструментів, які могли б автоматично визначати оптимальну конфігурацію для конкретної апаратної платформи. Реалізація такого підходу сприятиме підвищенню продуктивності і надійності ML-рішень та відкриє нові можливості для автоматизації обчислювальних процесів.

1.3 Розгляд схожих рішень

1.3.1 Застосування Q-навчання для спільного розвантаження обчислень та розподілу ресурсів у периферійній хмарі

Швидке зростання популярності смарт-мобільних пристроїв (SMDs) та поява обчислювально дорогих мобільних застосунків, таких як розпізнавання облич, AR, відеострімінг, різко збільшують потребу в обчислювальних ресурсах мобільних платформ. На тлі обмежень мобільних CPU та повільної еволюції акумуляторів виникає розрив між потребами застосунків та можливостями пристроїв. Класичний Cloud-offloading частково розв’язує проблему, але вносить велику затримку через віддалене розташування дата-центрів. Тому останніми роками сформувалася концепція Mobile Edge Computing (MEC), у якій обчислювальні ресурси розташовані на рівні мережевого краю (edge cloud), у безпосередній близькості до кінцевих користувачів.

Ключова задача MEC – визначити, які частини мобільного застосунку виконувати локально, а які – передавати на edge-сервер, а також скільки ресурсів виділити під кожне завдання, щоб мінімізувати енергоспоживання мобільного пристрою та забезпечити допустиму затримку. У роботі [10] розглядається спільна оптимізація двох аспектів:

– прийняття рішення про offloading (вибір місця виконання задачі);

– розподіл ресурсів (надання певного обсягу CPU edge-серверу).

Через складність графів залежностей між задачами мобільного застосунку та велику просторову динаміку, задача формулюється як MDP, та вирішується методами Reinforcement Learning.

Умови, в яких проводилось дослідження. У роботі розглядається Wi-Fi-базована MEC-архітектура, побудована на ETSI-MEC та стандарті IEEE 802.11ac. На рівні архітектури:

– декілька MEC-серверів розміщені на периметрі стадіону;

– мобільні пристрої підключаються до найближчих точок доступу;

– обчислення можуть виконуватись локально або на edge-сервері;

– передача даних здійснюється Wi-Fi-каналом з багатопотоковою передачею.

Для моделювання використовуються реальні графи задач мобільних застосунків. Залежності між задачами описані орієнтованим графом. Задача обмежена жорсткими вимогами: сумарний час виконання та передача повинні бути меншими за допустиму затримку.

Система працює в статичному сценарії: користувачі не змінюють місце розташування в період виконання offloading.

Дані, що використовувалися в роботі. Дані для оцінки моделі отримані з слідів реальних мобільних застосунків, особливо з галузі обробки обличь.

Мобільний застосунок описується орієнтованим графом G = (V, E):

– V – множина задач (вузли);

– E – залежності між задачами, кожне ребро означає передачу проміжних даних.

Кожна задача характеризується параметрами:

– кількість CPU-циклів, потрібних для виконання;

– розмір вхідних та вихідних даних;

– розмір даних при переході між задачами.

Ці параметри визначають обчислювальне навантаження, витрати енергії при передачі, затримку обчислення та чергу завдань на MEC-сервері.

Задача розглядається як MDP:

– State S – поточне завдання, ребро залежності, канал зв'язку, MEC-сервер;

– Action A – рішення, де виконувати задачу (локально або на одному з MEC-серверів);

– Reward/Cost ψ – енергоспоживання + обчислювальна затримка;

– Policy π – функція, що обирає дії для станів.

RL-агент мінімізує довгострокову вартість (energy + delay), близько до Bellman оптимальності:

$$\pi^* = \arg\min_{\pi} \sum_{i,s} \psi(s, a_i). \quad (1.1)$$

Властивості запропонованого алгоритму QL-JTAR (Q-Learning Joint Task Assignment & Resource Allocation):

а) online-RL без знання ймовірностей переходів;

б) зберігає Q(s,a) для кожного стану;

в) використовує ε-greedy політику (exploration + exploitation);

г) оновлює Q-функцію на кожному кроці:

– якщо дія покращує показники, зменшується її Q - значення (краще);

– якщо погіршує – збільшується (гірше);

д) використовує вдосконалений метод Relaxed-SMART для швидшої збіжності.

Після завершення навчання для кожного стану вибирається дія з мінімальним Q, що визначає оптимальну політику.

Переваги запропонованого підходу полягають у значному зниженні енергоспоживання на стороні мобільного пристрою та у відчутному скороченні часу обробки завдань. Зокрема, використання алгоритму QL-JTAR дозволяє зменшити витрати енергії приблизно на 89% порівняно з

повністю локальним виконанням застосунку, а також забезпечує до 56% економії у порівнянні з повним перенесенням обчислень у хмару. Подібна тенденція спостерігається щодо затримок: RL-стратегія скорочує час виконання задач майже на 97% відносно локального обчислення, водночас залишаючись дуже близькою до оптимального розв'язку, демонструючи відставання лише приблизно на шість відсотків. Важливо підкреслити й обчислювальну ефективність: QL-JTAR працює приблизно у сто разів швидше, ніж точне розв'язання задачі методом цілочисельного програмування, що особливо критично для прототипів і реальних систем з великими просторами станів. Окрім цього, автори демонструють, що метод здатний масштабуватися до реалістичних сценаріїв – наприклад, експериментальна конфігурація відтворює стан мережі на футбольному стадіоні, де використовуються понад 1200 пристроїв.

Водночас існують певні обмеження, які впливають на універсальність підходу. Модель не враховує динамічну мобільність користувачів, тому її застосування можливе лише в умовах відносно статичних сценаріїв, де положення пристроїв не змінюється під час offloading-процесу. Простір станів також суттєво спрощено: стан описується лише поточним вузлом графа задач та вибраним каналом. Окрім цього, робота фокусується виключно на розподілі CPU-ресурсів і не розглядає перенесення обчислень на GPU, які є ключовими для низки сучасних ML-застосунків. Нарешті, підхід вирішує задачу локально для кожного запиту незалежно, без аналізу глобального контексту, що потенційно може обмежувати ефективність у багатокористувацьких середовищах.

1.3.2 Використання навчання з підкріпленням для розподілу ресурсів завдань у наукових робочих процесах

У великих наукових workflow-системах (біоінформатика, геноміка, хімічне моделювання, обробка сигналів) задачі істотно відрізняються за

споживанням CPU і оперативної пам’яті. Через це користувачі або системи за замовчуванням часто виділяють ресурси з великим запасом, щоб уникнути збоїв. Такий підхід гарантує стабільність, але створює значну перевитрату CPU-годин та пам’яті, зменшує пропускну здатність обчислювального середовища та погіршує ефективність використання кластерів.

Автори [11] досліджують, чи може підкріплювальне навчання (Reinforcement Learning, RL) автоматично навчитися оптимально призначати ресурси для кожної задачі workflow – зменшуючи idle-time CPU, уникнувши OOM-помилок і підвищивши ефективність без ручного налаштування. Проблему формулюють як задачу послідовного прийняття рішень, де агент має підібрати кількість CPU-ядер і обсяг пам’яті перед запуском кожної задачі, а середовище відповідає реальним часом виконання, піковим споживанням пам’яті й поведінкою процесора.

Умови, в яких проводилось дослідження. RL-алгоритми інтегровано у реально працюючу workflow-систему Nextflow, яку було модифіковано засобами моніторингу. При кожному запуску задачі система збирала інформацію про фактичне використання ресурсів і записувала ці дані до бази на час виконання. Це дозволило агентам мати доступ до актуальних характеристик задачі та формувати стани й нагороду на основі реальних метрик.

Експерименти проводились на сервері:

– CPU: AMD EPYC 7282 (16 ядер, 32 потоки, 2.8 GHz);

– RAM: 128 GB DDR4;

– Storage: 2×1 TB SSD RAID 1.

Було обрано п’ять реальних наукових workflow з репозиторію nf-core, кожний перевірено з п’ятьма різними профілями вхідних даних (загалом 25 конфігурацій). Дослідження складалося з запусків кожного workflow під різними режимами: без RL, з RL-bandits, із Q-learning, а також зі спеціальним baseline-заходом (feedback loop).

Дані, що використовувалися в роботі. Для кожної задачі фіксували такі ключові характеристики:

– фактичне використання CPU (у відсотках і перерахунку на кількість ядер);

– пікове використання пам’яті;

– час виконання;

– параметри виділених ресурсів;

– інформацію про помилки, зокрема випадки нестачі пам’яті;

– статистику попередніх запусків (історичні середні runtime).

Ці дані визначали:

– стани RL-агентів (наприклад, поточний рівень CPU/пам’яті);

– нагороди;

– доступні дії, які змінювали кількість виділених ресурсів.

Оскільки RL систематично отримувало зворотний зв’язок у вигляді реального виконання workflow, агенти були здатні адаптуватися до різних типів задач та різних профілів навантаження.

У роботі було реалізовано три RL-підходи, які розглядають проблему з різних сторін.

CPU-bandit використовує градієнтний підхід (gradient bandit). Набір дій – можливі кількості CPU-ядер, які агент може виділити для задачі.

Функція винагороди:

$$\text{reward} = -\text{t} \cdot \left(1 + \text{cpus} - \frac{\text{cpuusage}}{100}\right), \tag{1.2}$$

де t – час виконання задачі;

cpus – виділені ядра;

$\text{cpuusage}/100$ – фактичне використання ядер (2.5 ядер → 250%).

Тобто агент отримує штраф за тривалий час виконання задачі (чим довше працює – тим більше покарання) і додатково карається за простої

процесора, тобто коли йому виділено більше ядер, ніж фактично використовується. Таким чином, bandit одночасно навчається скорочувати час виконання та уникати простоїв CPU.

У роботі також виникає проблема: reward є необмеженим (execution time може бути дуже великим), тому не можна використовувати фіксований крок навчання. Для цього, автори адаптували step-size:

$$\alpha = \frac{1}{\text{avg(time)}}. \tag{1.3}$$

Завдання з більшим runtime отримують менші кроки оновлення, що стабілізує навчання.

Memory Bandit. Дії – дискретні рівні виділення пам'яті. Повний обсяг пам'яті m розбивають на n частин, кожну розміром $\mathrm{c} = \mathrm{m}/\mathrm{n}$. Bandit може виділити $\mathrm{a} \cdot \mathrm{c}$ пам'яті, де $\mathrm{a} \in [1, \mathrm{n}]$.

Функція винагороди наступна:

$$\text{reward} = \begin{cases} -2 \cdot \frac{\text{memasg}}{\text{c}}, \text{якщо задачі не вистачило пам'яті} \\ -1 \cdot \text{chunksunused, інакше} \end{cases}, \tag{1.4}$$

де memasg – виділена пам'ять;

$\mathrm{peakrss}$ – фактичне використання;

$\mathrm{chunksunused} = (\mathrm{memasg} - \mathrm{peakrss})/\mathrm{c}$.

Якщо процесу бракує пам'яті, агент отримує значне покарання, а якщо пам'яті виділено надто багато, він карається пропорційно обсягу невикористаних ресурсів. Нагорода обмежена у діапазоні $[-2\mathrm{n}, 0]$, тому можна застосувати фіксований крок (формула 1.5). Якщо задачі бракувало пам'яті, наступна дія обов'язково збільшує обсяг, інакше ймовірна нова помилка. Такий механізм запобігає повторному вибору завідомо некоректних конфігурацій і прискорює збіжність алгоритму. Крім того,

обмеження винагороди спрощує аналіз поведінки агента та забезпечує стабільність оновлень під час навчання.

$$\alpha = \frac{1}{n}. \tag{1.5}$$

Q-learning Agent. Цей агент комплексно налаштовує одночасно CPU і пам’ять. Стан – поточний розподіл ресурсів (кількість CPU та пам’яті). Дії – збільшити/зменшити CPU, збільшити/зменшити пам’ять, або не змінювати нічого.

Функція винагороди наступна:

$$\text{reward} = -\text{cpuwaste} \cdot \frac{t}{\text{avg}_t} \cdot \text{memwaste}, \tag{1.6}$$

де $\text{cpuwaste} = \max(0.1, \text{cpuunused})$;

$\text{memwaste} = \text{memasg} \cdot (1 - \min(0.75, \text{memuse}))$.

CPU-навантаження обмежують «штучною підлогою» – якщо використано 90% CPU, вважається, що idle-time ≈ 0. Для пам’яті аналогічно: якщо використано 75%, вважається добре.

Агент отримує мінімальний штраф за:

– великі idle CPU;

– надлишкову виділену пам’ять;

– повільні (порівняно з avg time) задачі.

На відміну від bandit-методу, Q-learning не потребує модифікації step-size, оскільки середній runtime вже міститься у reward.

Результати експериментів показують, що підходи на основі підкріплювального навчання здатні суттєво скоротити надлишкове використання обчислювальних ресурсів у наукових workflow. Найкраще себе продемонстрували gradient bandits у частині розподілу CPU: втрати зменшилися до приблизно 10 CPU-годин проти 61 у статичній конфігурації,

тоді як Q-learning досяг близько 34 CPU-годин, а feedback-loop – близько 20. Bandit-підхід залишався стабільним для різних профілів задач, тоді як Q-learning працював гірше саме на складних workflow. Для пам’яті ситуація інша: найкращий результат показав feedback-loop (≈601 GBh), друге місце – bandits (≈2060 GBh), тоді як Q-learning відзначився значно більшими втратами (≈13418 GBh). Статична конфігурація дала найгірший результат(≈20456 GBh). Це вказує на те, що bandit-моделі добре працюють у ситуаціях з відносно простим простором станів, а оптимізація пам’яті потребує більш складної моделі чи розширеного контексту.

Попри очевидні переваги – адаптивність до типу задач, значне зниження перевитрат CPU та відсутність потреби у ручному налаштуванні – підходи мають низку обмежень. Q-learning гірше масштабується при великому просторі станів, а bandits чутливі до параметра кроку оновлення. Крім того, алгоритми не враховують залежності між етапами workflow і оцінюють задачі незалежно, що обмежує їх ефективність у складних сценаріях. Також присутній ефект “холодного старту”, коли на початку система приймає гірші рішення через нестачу досвіду, і відсутня нейронна апроксимація, яка могла б розширити можливості моделі для загальнішого навчання.

1.3.3 Використання навчання з підкріпленням для планування завдань CPU-GPU попередньої обробки, навчання та логічного висновку

У сучасному середовищі сервісів «AI as a Service» (AIaaS) виникає потреба в одночасному виконанні трьох типів задач:

- попередня підготовка даних;
- тренування моделей;
- застосування моделей.

Ці задачі висувають різні вимоги до ресурсів: деякі (наприклад, тренування або застосування) можуть бути легкими й частими, інші –

важкими та тривалими. Система має справлятися з гетерогенним навантаженням, розподіляючи задачі між CPU та GPU так, щоб мінімізувати загальний час обробки (turnaround time), зменшити затримки для користувача, ефективно використовувати ресурси, і при цьому підтримувати сервісну відповідність (наприклад, швидкі відповіді для запитів). Статичні або евристичні планувальники (на кшталт SJF, FCFS) часто не справляються з динамічними, змінними навантаженнями та гетерогенними типами задач.

Отже, поставлена задача в роботі [12]: автоматизувати планування задач у гетерогенній системі CPU–GPU, щоб мінімізувати latency і загальний час виконання, з урахуванням динамічності навантаження.

Умови та контекст:

– середовище – гетерогенна система із CPU і GPU, котра обслуговує AI-сервіси різного типу (обробка, тренування, застосування);

– навантаження – змінне, із чергами задач, які надходять асинхронно; задачі мають різну тривалість, ресурсоємність, вимоги до апаратури;

– архітектура може бути розподіленою (cloud, edge, fog), тобто рішення мають масштабуватись і бути адаптивними до змін продуктивності, наявності ресурсів, heterogeneity;

– вимоги: мінімізація часу очікування і часу виконання задач, баланс між продуктивністю сервісу (особливо inference, latency), і ефективним використанням ресурсів CPU/GPU.

Дані:

– різні типи задач: preprocessing, training, inference;

– черги задач із супровідною інформацією (queue info, task metadata, resource requirements, очікуваний час, пріоритет, тип задачі тощо) – агент бере цю інформацію як стан;

– контекст гетерогенної системи: CPU- та GPU- ресурси, можливість запуску задач або на CPU, або на GPU – залежно від характеру

задачі(наприклад, підготовка даних чи застосування часто краще на CPU, тренування – на GPU);

– метрики ефективності: turnaround time, latency (затримка під час застосування), завантаження ресурсів, баланс між user-experience і продуктивністю.

Алгоритми, які використовувалися:

– запропоновано алгоритм UXP-RL – user-experience-and-performance balanced reinforcement learning scheduler;

– серед вхідних факторів до агента – 11 параметрів, включно з інформацією про чергу задач, ресурси, історію попередніх дій та результатів. Актор приймає рішення, яку задачу і на якому типі ресурсу (CPU або GPU) запускати;

– політика – стохастична: агент обирає задачу на основі оцінки вигоди та очікуваного «вивільнення ресурсів»;

– порівняння з класичними евристиками – на кшталт SJF (shortest-job-first) та FCFS (first-come, first-served) як baseline.

Результати демонструють, що застосування підкріплювального навчання для планування задач у хмарних та edge-середовищах може суттєво прискорити обробку запитів порівняно зі статичними евристиками. Зокрема, середній час виконання задач (turnaround time) скорочується на 27.66 – 57.81% відносно класичних підходів на кшталт SJF та FCFS. У сценаріях із розподіленою обробкою, де ресурси розміщені на cloud/edge/fog рівнях, використання розподілених RL-алгоритмів дає ще відчутніший ефект: час обробки може зменшуватися до 89.07% у порівнянні з централізованими підходами. Модель адаптивно реагує на змінне навантаження, різнорідність обладнання та структуру черги, а замість жорстко заданих правил використовує поведінку, сформовану на основі попереднього досвіду. Це дозволяє враховувати історію виконання, поточні характеристики ресурсів і структуру workload, забезпечуючи гнучкість і наближеність до реальних умов роботи системи.

Водночас підхід має низку обмежень. Для якісного навчання агенту потрібні достатні обсяги даних про завдання, стан системи та споживання ресурсів, що не завжди можливо у виробничих середовищах. Крім того, сам RL-планувальник створює певне додаткове навантаження: процес ухвалення рішень, оцінка стану і оновлення політики потребують обчислювальних ресурсів, які можуть виявитися невиправданими для простих або рідкісних задач. Якщо система працює стабільно, а workload не надто змінюється, традиційні евристики можуть бути достатньо ефективними. У дуже гетерогенних або нестабільних умовах навчання може уповільнюватися або навіть припинятися через часту зміну характеру задач і середовища, що ускладнює пошук оптимальної політики.

1.3.4 Висновки з опрацьованих робіт

Аналіз сучасних робіт демонструє, що методи RL суттєво переважають евристики. Для кластерів та хмарних середовищ показано прискорення обробки задач у 27–57% порівняно з SJF та FCFS, а у багаторівневих інфраструктурах Cloud/Edge/Fog розподілені RL-підходи забезпечують скорочення середнього часу виконання задач на 89%. У наукових workflow градієнтні bandit-моделі суттєво зменшують втрати CPU-ресурсів (≈10 CPUh проти 61 у статичного підходу). У мобільних edge-сценаріях Q-learning дозволяє скоротити енергоспоживання до 89% порівняно з локальним виконанням і наближається до оптимуму за часовими затримками із незначним відставанням (~6%). Таким чином, RL демонструє одночасно адаптивність і здатність працювати у складних стохастичних середовищах, де неможливо визначити оптимальну конфігурацію наперед.

Водночас аналіз відповідних підходів виявив і важливі обмеження, які визначають напрям розвитку запропонованого рішення. Деякі моделі RL погано масштабуються при великому просторі станів або складній структурі

залежностей (наприклад, складні графи workflow). Q-learning втрачає ефективність при обмеженому обсязі даних, а gradient bandit чутливий до параметрів оновлення. Крім того, багато підходів оцінюють задачі незалежно, без врахування зв’язків у межах більшої системи, та не використовують нейронну апроксимацію функції значення. У контексті оптимізації параметрів навчання моделей виникає ще одна фундаментальна особливість – відсутність наперед розмічених даних або еталонних рішень: немає «правильної» конфігурації batch size, кількості потоків чи CPU-affinity, яку можна було б передати у вигляді навчальної вибірки для supervised-методу. Система може оцінювати якість рішення лише через побічні метрики (тривалість епохи, завантаження CPU, точність моделі), тобто через delayed reward. Саме така постановка є типовою для задач із частковою або відкладеною винагородою, що робить підкріплювальне навчання природним вибором.

З урахуванням цих факторів у даній роботі обрано фокус на простій, але адаптивній політиці, яка використовує ключові метрики системи як стан середовища та реалізує пошук політики засобами policy-gradient (REINFORCE з baseline), що дозволяє працювати без наперед підготовленої розмітки і поступово «навчатися» на власному досвіді.

1.4 Постановка задачі дослідження

Сучасні системи машинного навчання демонструють високу залежність як від обчислювальних ресурсів, так і від їх раціонального використання під час навчання моделей, що робить задачу автоматизованого керування обчислювальним середовищем важливою складовою забезпечення стабільного й ефективного навчання ML-моделей.

Метою роботи є підвищення ефективності методів машинного навчання, насамперед алгоритмів навчання з підкріпленням, у задачах оптимізації розподілу обчислювальних ресурсів, а також продуктивності

навчання згорткових нейронних мереж завдяки впливу динамічного керування параметрами процесів і потоків. Для цього пропонується розробити систему автоматичного розподілу обчислювальних ресурсів під час навчання моделей машинного навчання на основі підкріплювального навчання. Необхідно побудувати RL-агента, здатного адаптивно обирати параметри навчання (batch size, кількість DataLoader workers, кількість потоків PyTorch, CPU-affinity та learning rate) у відповідь на стан системи та характеристики попередніх епох.

Об’єкт дослідження – система автоматичного розподілу обчислювальних ресурсів під час виконання задач машинного навчання.

Предмет дослідження – алгоритми підкріплювального навчання та методи машинного навчання, що використовуються для оптимізації параметрів виконання моделі, а також принципи побудови політик у середовищах з невизначеністю та стохастичністю.

Під час роботи потрібно розв’язати такі часткові задачі дослідження:

– дослідити сучасні підходи до оптимізації обчислювальних ресурсів у задачах ML, які використовують RL, bandit-моделі, Q-learning та policy-based методи для управління CPU, пам’яттю та GPU;

– проаналізувати моделі та алгоритми підкріплювального навчання, що можуть бути застосовані для розв’язання задачі динамічного вибору параметрів тренування моделей;

– розробити архітектуру рішення, що пов’язує RL-агента та процес навчання моделі CNN, включаючи формалізацію стану, множини дій та функції винагороди;

– обґрунтувати вибір алгоритму RL, що відповідає вимогам до операційної системи та сценарію навантаження: мінімізація idle-time, прискорення епохи та збереження точності;

– реалізувати прототип системи, що виконує навчання CNN-моделі на MNIST з можливістю керування конфігурацію обчислень (batch size, num_workers, torch threads, CPU affinity, learning rate);

– провести експерименти, спрямовані на перевірку впливу конфігурацій на час навчання, завантаження CPU, пікове використання пам’яті та точність моделі;

– порівняти результати із статичними конфігураціями, оцінити стабільність, збіжність політики, поведінку на різних етапах навчання та ефект «холодного старту»;

– сформувати висновки щодо ефективності підходу, включаючи визначення його переваг, недоліків та потенційних напрямів розвитку.

2 КОМПОНЕНТИ ТА ПАРАМЕТРИ ОБЧИСЛЮВАЛЬНОГО СЕРЕДОВИЩА

Обчислювальне середовище, у якому виконується навчання моделей машинного навчання, складається з низки апаратних та програмних компонентів, кожен з яких визначає продуктивність і ефективність обробки даних. Основними елементами виступають процесорні ядра, логічні потоки, окремі процеси та механізми паралелізму, що забезпечують виконання декількох обчислювальних задач одночасно. Їх взаємодія визначає, наскільки раціонально використовуються апаратні можливості системи, як відбувається розподіл навантаження та яка конфігурація буде оптимальною для конкретного алгоритму.

У сучасних CPU існує кілька типів структурних та програмних одиниць, що беруть участь у виконанні обчислень: фізичні ядра, логічні ядра (потоки), користувацькі та системні процеси, а також програмні потоки, створювані фреймворками машинного навчання. Кожен із цих типів виконує свою роль у забезпеченні паралельності і визначає певні обмеження та можливості для оптимізації.

2.1 Фізичні ядра

Фізичне ядро – компонент центрального процесора [13]. Це базовий структурний елемент сучасних обчислювальних систем, який виконує всі ключові операції, пов’язані з опрацюванням інструкцій програм. Фізичне ядро є незалежною апаратною одиницею, що має власні функціональні блоки, конвеєри, кеші та виконує інструкції повністю автономно. Саме тому кількість фізичних ядер визначає верхню межу реального паралелізму в комп’ютерній системі.

Структура фізичного ядра сформована таким чином, щоб забезпечити максимально швидке виконання команд та ефективне використання

доступної пам’яті. Основу складає арифметико-логічний пристрій, який виконує математичні та логічні операції [14]. Разом із ним працює пристрій керування, що відповідає за інтерпретацію інструкцій, формування керуючих сигналів та організацію взаємодії між усіма внутрішніми блоками. Регістрів у ядрі небагато, проте вони є найшвидшим видом пам’яті та слугують для тимчасового розміщення даних, що беруть участь у виконанні інструкцій. Через регістри дані потрапляють до арифметико-логічного пристрою, а результати повертаються назад у процесорну підсистему.

Ще однією складовою ядра є кеш-пам’ять. Вона мінімізує затримки при доступі до часто використовуваних даних і команд чим зменшує звернення до оперативної пам’яті, яка працює повільніше. Є 3 рівня кешу. Найшвидший кеш першого рівня зазвичай розділений на окремі галузі для інструкцій і даних. Кеш другого рівня має більший обсяг і слугує проміжною ланкою між ядром та загальною кеш-пам’яттю третього рівня, яка може бути спільною для всіх ядер процесора [15] (рисунок 2.1).

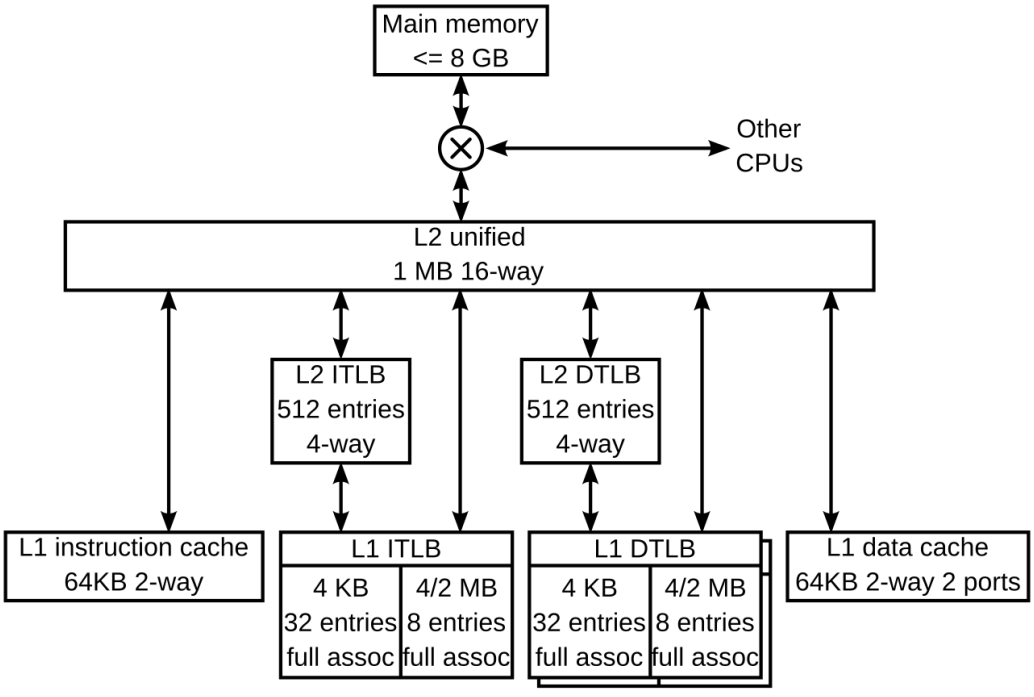

Рисунок 2.1 – Приклад ієрархії кеш-пам’яті процесора [16]

Окрему роль відіграють блоки, відповідальні за роботу з пам'яттю на рівні адресації. Пристрій генерації адрес формує коректні фізичні або віртуальні адреси під час звернень до пам’яті, а модуль керування пам’яттю забезпечує підтримку віртуальної пам’яті, трансляцію адрес та роботу зі сторінками.

Робота фізичного ядра ґрунтується на циклі «вибір-декодування-виконання». На першому етапі з пам’яті отримується чергова інструкція, адреса якої зберігається у спеціальному лічильнику. Далі інструкція декодується, перетворюючись на внутрішні операції, які розуміють функціональні блоки процесора. На завершальному етапі інструкція виконується, а результат записується назад у регістри або кеш. У сучасних процесорах цей цикл реалізований у вигляді конвеєра, що дозволяє одночасно опрацьовувати різні стадії кількох інструкцій і суттєво підвищує продуктивність [17], [18].

2.2 Логічні ядра

Логічне ядро є віртуалізованою одиницею обчислювальних ресурсів, яку операційна система розглядає як окремий процесор для виконання команд. Логічне ядро не є самостійним апаратним елементом. Воно виникає внаслідок застосування технологій апаратної багатопотоковості, таких як SMT або Hyper-Threading, які дозволяють одному фізичному ядру підтримувати кілька потоків виконання. У результаті операційна система бачить кожен із цих потоків як окреме логічне ядро, навіть якщо апаратно обчислення здійснюються в межах одного фізичного блоку [19].

Функціонування логічних ядер нерозривно пов’язане з поняттям потоку. Потік містить власний лічильник команд, набір регістрів і стек, але розділяє пам’ять і ресурси того процесу, до якого належить. Сучасні програми, на відміну від старших моделей, де один процес мав лише один потік, складаються з багатьох потоків, що дозволяє одночасно виконувати

кілька операцій. Створення й перемикання потоків значно легше та швидше, ніж робота з окремими процесами, тому багатопотоковість стала основою логічних ядер як механізму розширеної паралельності [20].

Завдяки логічним ядрам операційна система рівномірно розподіляє навантаження між кількома потоками в межах одного фізичного ядра, забезпечуючи більш повне використання його апаратних можливостей. Наприклад, якщо фізичне ядро простоює через очікування даних із пам’яті, інше логічне ядро може отримати процесорний час і продовжувати виконання свого потоку, що дозволяє підвищити загальну пропускну здатність системи без збільшення кількості фізичних блоків.

2.3 Процеси

Процес – базова одиниця організації виконання програм. Історично для позначення виконуваних завдань використовували термін job, який виник у часи пакетних обчислень, коли комп’ютери отримували набори завдань у вигляді черги. З появою інтерактивних та багатозадачних систем термін process поступово витіснив стару термінологію, однак обидва поняття в окремих контекстах залишаються взаємопов’язаними. Процес – це програма, що перебуває у стані виконання. Вона має власний лічильник команд, набір регістрів, стек, виділені ресурси та певні атрибути у структурі операційної системи.

Програма, яка зберігається у вигляді файлу на диску, сама по собі є пасивним набором інструкцій. Процес виникає тоді, коли операційна система завантажує програму до пам’яті та створює для неї середовище виконання. Ініціювати запуск може, як користувач, так і інша програма. Після завантаження початковий вміст регістрів ініціалізується, лічильник команд отримує адресу першої інструкції, а структурі процесу призначаються необхідні ресурси. Слід зазначити, що два процеси можуть

використовувати один і той самий виконуваний файл, але кожен матиме власні дані, стек і динамічну пам’ять [21].

Пам’ять процесу організована у декілька галузей. Виконуваний код програми міститься в текстовій секції, що зазвичай має фіксований розмір і не змінюється протягом роботи. Дані глобальних змінних і статичних об’єктів розміщуються у, незмінній після запуску, секції даних. Динамічні структури та об’єкти, що створюються під час роботи програми, розміщуються в галузі heap, обсяг якої може збільшуватися або зменшуватися відповідно до викликів виділення й звільнення пам’яті. Стекова галузь використовується для зберігання тимчасових даних під час виклику функцій, включаючи локальні змінні, параметри та адреси повернення. Розміри heap і stack змінюються динамічно, а операційна система запобігає взаємному перекриванню цих галузей.

Процес – це не лише набір даних, а й об’єкт, який перебуває у певному стані: новий, готовий до виконання, таким, що виконується, чекає на певну подію. Стани різняться залежно від конкретної операційної системи, однак загальна модель однакова: один процес може активно виконуватися на одному ядрі, тоді як інші перебувають у черзі очікування. Перехід між станами зумовлений як машинними подіями (наприклад, завершенням операції введення–виведення), так і рішеннями планувальника процесів операційної системи, який визначає, який процес отримує процесорний час [22].

Для підтримки інформації про процес операційна система створює структуру – блок керування процесом (Process Control Block, PCB), де зберігається поточний стан процесу, значення лічильника команд, вміст регістрів, відомості про використання пам’яті, параметри планування, інформація про відкриті файли та інше. Момент перемикання між процесами називається контекстне переключення. Система зберігає поточний стан процесу в його PCB, а потім відновлює стан нового процесу

з його власного PCB, що забезпечує продовження виконання процесу з того моменту, на якому він був зупинений.

У багатоядерних системах можливе одночасне виконання кількох процесів, причому кожен із них може бути призначений на окреме фізичне чи логічне ядро. Якщо кількість процесів перевищує кількість доступних ядер, надлишкові процеси чекають у стані готовності.

Окремою формою організації виконання є процеси, які виступають середовищем виконання для інших програм. Розглянемо приклад Java-програм. Вони зазвичай виконуються у віртуальній машині JVM, яка сама працює як звичайний процес операційної системи. Усередині цього процесу JVM інтерпретує байт-код Java-програми, забезпечуючи платформну незалежність і додатковий рівень абстракції [23].

3 ФУНДАМЕНТАЛЬНІ ОСНОВИ НАВЧАННЯ З ПІДКРІПЛЕННЯМ

3.1 Загальна характеристика підкріплювального навчання

3.1.1 RL як напрямок машинного навчання

Підкріплювальне навчання (Reinforcement Learning, RL) – один з методів машинного навчання, під час якого система (агент) навчається, взаємодіючі з певним середовищем. RL поєднує елементи теорії керування, стохастичних процесів, оптимізації та статистичного навчання, утворюючи формальний підхід до моделювання послідовних рішень в умовах невизначеності (рисунок 3.1).

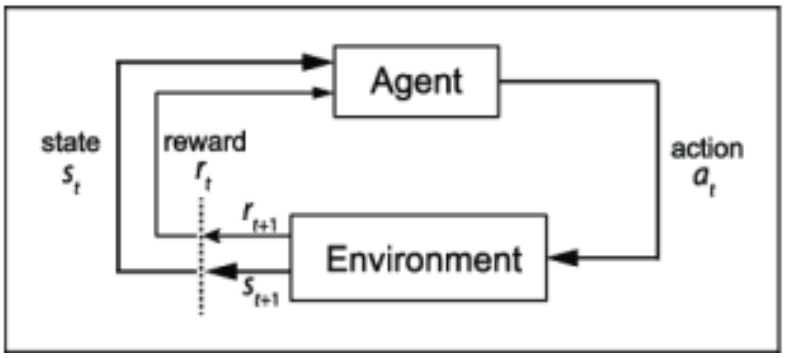

Рисунок 3.1 – Схема взаємодії агента з середовищем

У теоретичній основі RL лежить уявлення про середовище у вигляді марківського процесу прийняття рішень (Markov Decision Process, MDP), який формалізує поведінку системи через множину станів, множину можливих дій, функцію переходів та функцію винагород. Завдання агента полягає у знаходженні політики, тобто відображення зі станів у дії, яка максимізує сумарну очікувану винагороду з урахуванням фактору дисконтування. Такий підхід дозволяє описати широкий спектр задач, де

результати дій проявляються не миттєво, а з певною затримкою, і де дія впливає на подальший перебіг процесу (рисунок 3.1).

У межах RL виділяються два основних класи методів: методи, що ґрунтуються на оцінці значень станів або станів-дій (value-based methods), та методи, що безпосередньо оптимізують політику (policy-based methods). Перші використовують апроксимацію функцій корисності для визначення найкращих дій, другі – обчислюють градієнт очікуваної винагороди щодо параметрів політики. Окремим напрямом є актор-критикові методи, які поєднують властивості обох підходів та дозволяють стабілізувати процес навчання.

Важливою рисою RL є відсутність наперед підготовленого набору правильних відповідей. Агент отримує інформацію не у формі пар «вхід – правильний вихід», а у вигляді винагороди, яка може бути не прямою інструкцією, а узагальненою характеристикою якості виконаної дії в певних умовах. Це зумовлює необхідність балансування між дослідженням середовища, яке дозволяє отримати нову інформацію про поведінку системи, та використанням накопичених знань, що спрямоване на досягнення максимальної винагороди на основі здобутого.

Підкріплювальне навчання набуло поширення завдяки своїй здатності працювати з динамічними та складними середовищами, через що активно застосовується в задачах оптимального керування, робототехніці, автономних системах, оптимізації мережевих сервісів, рекомендаційних системах, ігрових середовищах.

3.1.2 Відмінність від класичного ML

Підкріплювальне навчання відрізняється від традиційних підходів машинного навчання низкою фундаментальних ознак, пов’язаних із формою доступних даних, структурою взаємодії з середовищем та постановкою задачі. У класичних методах машинного навчання дані

подаються у статичному вигляді: модель отримує фіксований набір прикладів і навчається знаходити закономірності, що дозволяють узагальнювати інформацію на нові випадки. У supervised learning ці приклади мають мітки, які визначають правильний результат. У unsupervised learning міток немає, але сама структура даних визначає закономірності, які необхідно виявити. RL, натомість, працює у динамічному середовищі, де дані формуються як наслідок дій самого агента.

У класичних підходах навчання відбувається за наявності повного набору даних, що дозволяє використовувати статистичні методи для мінімізації похибки між прогнозом та цільовим значенням. У RL основним джерелом інформації є зворотний зв’язок у вигляді винагороди, який може бути відкладеним у часі. Одна дія може не мати безпосереднього ефекту, але може впливати на результат у майбутньому. Це ускладнює оцінювання якості рішень та потребує використання механізмів, здатних враховувати довгострокові наслідки, наприклад функції цінності та дисконтування.

Іншою принциповою відмінністю є той факт, що у підкріплювальному навчанні агент отримує дані послідовно, у процесі взаємодії з середовищем. Набір даних не є завершеним; він формується поступово, а дії агента впливають на те, які дані він отримає надалі. Це створює залежності між спостереженнями, що відрізняє RL від класичних підходів, які здебільшого передбачають незалежність і однаковий розподіл вибірок.

У supervised learning критерій оптимізації є визначеним і повністю відомим під час навчання, оскільки цільові мітки є частиною набору даних. У RL критерій оптимальності задається функцією винагороди, яка може бути недетермінованою або неповною. Додатковою складністю є те, що агент має оптимізувати не значення винагороди за окрему дію, а сумарну винагороду за довготривалий період взаємодії зі середовищем.

Важливою відмінністю є також наявність циклічної структури взаємодії «агент – середовище». Класичне машинне навчання працює з даними, які не змінюються внаслідок роботи моделі. У підкріплювальному

навчанні середовище описується функцією переходів, що пов’язує кожну дію агента з новим станом системи. Тому RL поєднує навчання з процесом керування, тоді як класичне ML фокусується переважно на моделюванні залежностей у даних.

Підсумовуючи, підкріплювальне навчання відрізняється від класичних методів машинного навчання динамічністю даних, відсутністю фіксованого набору прикладів, послідовним характером взаємодії, наявністю відкладеної винагороди та вимогою оптимізувати поведінку у довгостроковій перспективі.

Вище наведені особливості визначають специфіку алгоритмів RL і обґрунтовують необхідність використання окремих математичних формалізацій і методів оптимізації, відмінних від традиційних підходів машинного навчання.

3.1.3 Марковський процес прийняття рішень

Марковська властивість є ключовою умовою, на якій базується формалізація марковських процесів прийняття рішень. Вона стверджує, що еволюція системи не залежить від усієї попередньої історії, а визначається виключно її поточним станом. Це записують як

$$p(s_t|s_0, s_1, \dots . s_{t-1}) = p(s_t|s_{t-1}). \quad (3.1)$$

Таким чином, система не «пам’ятає» шляху, яким вона прийшла до поточної конфігурації, і вся інформація, необхідна для прийняття рішення, міститься у стані s_{t-1}.

Марковський процесом прийняття рішень називають набір з п’яти елементів (чотирьох множин та одного числа):

$$M = (S, A, T, R, \gamma), \quad (3.2)$$

де S – простір станів;

A – простір дій;

T – оператор переходів множини станів T, який формують функції переходів;

$R(s, a)$ – середня винагорода, яку отримує агент;

γ – коефіцієнт дисконтування.

Простір станів S містить усі можливі конфігурації, у яких може перебувати система під час взаємодії агента із середовищем. У кожний момент часу, коли необхідно прийняти рішення, агент спостерігає поточний стан та обирає дію. Простір станів може бути скінченним, нескінченним або неперервним, залежно від складності моделі. У разі скінченного простору кількість станів позначають |S| = N. Марківська властивість припускає, що майбутній розвиток системи залежить тільки від поточного стану та дії, а не від попередньої історії.

У практичних задачах стан часто має складну структуру: наприклад, у задачах робототехніки він може описувати координати, швидкість і орієнтацію робота; в обчислювальних системах – завантаження CPU, кількість потоків, затримку виконання чи доступні ресурси.

Простір дій A може мати різні властивості: бути скінченним або неперервним, залежати від поточного стану або бути однаковим для всіх станів. У скінченних MDP дії мають дискретний характер, тоді як у більш складних системах використовуються неперервні дії, що вимагає альтернативних методів оптимізації політики.

Властивості множини дій впливають на вибір RL-алгоритму: скінченні множини підходять для класичних методів типу табличного Q-learning, тоді як неперервні дії потребують методів оптимізації політики або актор-критикових підходів.

Функція переходів $P(s' \mid s, a)$ задає ймовірність того, що система перейде зі стану s до стану s' за умови виконання дії a. Функція переходів

володіє марковською властивістю: ймовірність майбутнього стану залежить тільки від поточного стану й дії.

У стаціонарних процесах функція переходів не змінюється з часом. У нестаціонарних моделях переходи можуть залежати від моменту часу або зовнішніх умов. Формально:

$$P(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a). \quad (3.3)$$

Функція переходів визначає стохастичний характер системи та впливає на вибір RL-алгоритму.

Функція винагороди $R(s, a)$ визначає значення, яке агент отримує після виконання дії в певному стані. Вона може залежати від попереднього стану, вибраної дії та нового стану. У загальному випадку винагорода позначається як $R(s, a, s')$, що відображає залежність від фактичного переходу.

Коефіцієнт дисконтування $\gamma \in [0,1]$ визначає відносну важливість майбутніх винагород. Сумарна очікувана винагорода для політики π на нескінченному горизонті визначається як:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \quad (3.4)$$

Коефіцієнт дисконтування γ визначає, наскільки агент цінує майбутні винагороди:

– якщо γ прямує до 0, агент діє короткозоро та орієнтується переважно на негайний результат, фактично ігноруючи довготривалі наслідки;

– коли γ наближається до 1, агент стає далекоглядним і враховує винагороди, що будуть отримані в майбутньому, майже без їх знецінення.

Вибір γ визначає математичну коректність моделі у довготривалих процесах. У моделях зі скінченним горизонтом N роль γ є менш критичною, оскільки загальна кількість кроків обмежена, і майбутні винагороди

природно не можуть накопичуватися нескінченно. У задачах з нескінченним горизонтом використання $\gamma < 1$ є обов'язковим: це забезпечує збіжність сумарної винагороди та запобігає її необмеженому зростанню.

3.1.4 Послідовність взаємодії в MDP

Марківський процес прийняття рішень описує динаміку послідовної взаємодії між агентом і середовищем. Взаємодія відбувається дискретними кроками часу й утворює ланцюг вигляду:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots . \quad (3.5)$$

На кожному кроці агент спостерігає поточний стан S_t, який відображає ситуацію в середовищі в момент часу t. На основі цього стану агент обирає дію A_t, після чого середовище реагує: формує числове значення винагороди R_{t+1}та переводить систему до нового стану S_{t+1}. Цей цикл повторюється, утворюючи траєкторію – послідовність станів, дій і винагород, що описує весь епізод взаємодії.

Таке представлення дає можливість формально моделювати процес прийняття рішень з урахуванням стохастичності середовища, невизначеності переходів та різних стратегій поведінки агента. У скінченних MDP множини станів, дій і можливих значень винагород обмежені, що робить систему керованою та дозволяє застосовувати класичні алгоритми оптимізації.

3.1.5 Політики

У підкріплювальному навчанні центральним елементом формулювання задачі є політика (policy), яка визначає спосіб поведінки агента в середовищі. Політика задає правило вибору дій у кожному

можливому стані системи та визначає характер траєкторій, що формуються в результаті взаємодії агента з середовищем.

Політику позначають як π і визначають як відображення

$$\pi(a \mid s) = P(A_t = a \mid S_t = s), \tag{3.6}$$

тобто ймовірність вибрати дію a у стані s у момент часу t.

У загальному випадку розрізняють два типи політик. Детермінована політика однозначно визначає дію: для кожного стану існує лише одне допустиме рішення, тобто

$$\pi(a \mid s) \in \{0,1\}. \tag{3.7}$$

Стохастична політика задає розподіл ймовірностей над множиною дій, що є важливим у задачах, де потрібне активне дослідження середовища або де оптимальна поведінка містить елемент випадковості.

Політика може бути представлена у різних формах: від табличних відображень для простих середовищ до параметризованих моделей, наприклад, нейронних мереж, які повертають ймовірнісний розподіл над діями. В обох випадках політика визначає спосіб, у який агент формує рішення, а тому вона є основним об’єктом оптимізації.

3.1.6 Функції цінності

Більшість алгоритмів RL прямо або опосередковано оптимізують політику, щоб максимізувати очікувану сумарну винагороду. Для кількісної оцінки якості політики використовують value-функції, що відображають очікувану сумарну винагороду агента при певній поведінці. Найпоширенішими є функція цінності стану та функція цінності пари «стан–дія».

Функція цінності стану під конкретною політикою π позначається $v_\pi(s)$ і визначає математичне сподівання сумарної винагороди, яку агент отримає, починаючи зі стану s і надалі дотримуючись політики π. Це можна записати як:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s], \tag{3.8}$$

де G_t – очікувана винагорода, або наскільки «вигідно» опинитися в певному стані за умови фіксованої поведінки агента.

Функція цінності дій (action-value function) або Q-функція описує очікуване повернення при виконанні конкретної дії у певному стані:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]. \tag{3.9}$$

Таким чином, Q-функція оцінює не лише стан, а й внесок конкретної дії в майбутню винагороду. Це робить її ключовою для алгоритмів оптимізації, що намагаються визначити найкращу дію для кожного стану.

Ці функції пов’язані між собою. Функція цінності стану може бути представлена через функцію цінності дій:

$$v_\pi(s) = \sum_{a \in A} \pi(a \mid s)\, q_\pi(s, a). \tag{3.10}$$

3.1.7 Рівняння Беллмана

Рівняння Беллмана посідають центральне місце у формалізмі марковських процесів прийняття рішень і задають рекурсивну залежність між значенням поточного стану та очікуваною цінністю його наступників. Вони забезпечують математичний апарат, який дозволяє оцінювати довгострокову корисність дій агента, ґрунтуючись на локальних взаємодіях зі середовищем.

Для довільної політики π значення стану визначається як математичне сподівання повернення, отриманого агентом, починаючи зі стану s і дотримуючись політики надалі. Це формально визначається через рівняння Беллмана для $v_\pi(s)$:

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s)[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) v^\pi(s')]. \quad (3.11)$$

У цьому виразі політика визначає ймовірність вибору дії у стані s, тоді як функція переходів відображає стохастичну природу середовища. Рівняння показує, що цінність стану дорівнює сумі очікуваної негайної винагороди та дисконтованої вартості наступного стану. Будь-який MDP зі стаціонарними переходами може бути описаний саме через таку рекурсивну структуру: значення кожного стану залежить лише від його безпосередніх продовжень, а не від усієї історії взаємодій.

Аналогічно визначається і дія-цінність $q_\pi(s,a)$, яка враховує не лише стан, а й конкретну дію:

$$q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma\, v^\pi(s')]. \quad (3.12)$$

Ця формула дозволяє оцінювати якість окремих рішень, що відіграє ключову роль у методах, які будують політику через порівняння дій за їх очікуваними результатами.

3.1.8 Рівняння оптимальності

Коли метою є знаходження оптимальної політики π*, необхідно сформулювати оптимальні варіанти рівняння Беллмана. У цьому випадку значення стану визначається максимальним можливим поверненням, що

агент може отримати, почавши з цього стану та далі завжди обираючи найкращі дії. Це відображає оптимальне рівняння Беллмана для $v^*(s)$:

$$v^*(s) = \max_{a \in A}[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) v^*(s')]. \quad (3.13)$$

Рекурсія демонструє принцип оптимальності: оптимальна цінність стану отримується шляхом вибору дії, яка забезпечує найвищу суму негайної винагороди та очікуваної цінності наступних станів. Цей підхід лежить в основі алгоритмів «ітерації стратегій» і «ітерації вартостей», що застосовуються для точного розв'язання MDP у випадку скінченних просторів станів.

Оптимальне значення для пари стан–дія задається рівнянням:

$$q^*(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} \mathrm{q}^*(s', a')]. \quad (3.14)$$

Тут максимізація за діями у наступному стані відображає передбачення подальшої оптимальної поведінки агента. Фактично, це рівняння лежить в основі Q-learning, де цінність $q^*(s,a)$ аппроксимується шляхом повторного застосування цієї рекурсії на елементарних кроках взаємодії.

3.1.9 Інтерпретація та значення рівнянь Беллмана

Рівняння Беллмана задають принципово важливу структуру: значення (value) визначається через очікування майбутнього значення. У термінах інженерної інтерпретації, це процес поширення корисності назад у часі, коли інформація про винагороди, що очікуються в майбутньому, «повідомляє» нинішнім станам, наскільки вони вигідні. Така властивість

дозволяє будувати методи динамічного програмування та алгоритми RL, які поступово уточнюють оцінки функцій вартості.

Завдяки рекурсивному характеру рівнянь можливим стає розрахунок оптимальної політики без повного перебору всіх можливих траєкторій агента. Натомість алгоритми використовують локальні переходи та оцінюють довгострокові вигоди за принципом наближення.

Значущість оптимальних рівнянь полягає в тому, що вони задають умову, якої повинні досягти функції $v^*(s)$ та $q^*(s,a)$ в точці збіжності. У практичних реалізаціях RL це дозволяє використовувати методи градієнтного спуску, ручного оновлення, стохастичного наближення або навчання нейронних мереж для виконання ролі функцій-оцінювачів.

3.2 Алгоритми навчання з підкріпленням

Методи навчання з підкріпленням можна класифікувати на три основні групи: методи, засновані на функції цінності (value-based), методи прямої оптимізації політики (policy-based) та гібридні підходи actor-critic, які поєднують сильні сторони двох попередніх класів. Кожен з цих напрямків відрізняється способом представлення політики агента, характером оптимізації та галуззю застосування. Вибір конкретної групи методів залежить від властивостей середовища, структури простору дій, необхідності стохастичності політики та вимог до стабільності навчання.

3.2.1 Value-based

Value-based, або навчання з підкріпленням на основі цінностей, зосереджене на пошуку оптимальної функції цінності, яка вимірює, наскільки добре для агента перебувати в заданому стані (або виконувати задану дію).

Функція q дає змогу напряму визначати оптимальну дію через правило:

$$a^* = \arg\max_a q(s, a). \tag{3.15}$$

Тобто агент вибирає дію, що веде до найбільш перспективного наступного стану. Це правило називається жадібною політикою.

У value-based методах оптимальна політика не задається безпосередньо. Вона формується опосередковано шляхом пошуку такої функції $q(s, a)$, яка відповідає оптимальній оцінці майбутніх винагород. Ці методи є фундаментальними для задач з дискретними діями, де кількість можливих варіантів управління є обмеженою [24], [25].

Основні алгоритми value-based методу.

Q-learning. Це метод, який наближає оптимальну дію незалежно від політики, за якою агент досліджує середовище [26]. Оновлення функції Q виконується за формулою Беллмана:

$$q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma \max_{a'} q(s', a') - q(s, a)]. \tag{3.16}$$

SARSA**.** Його назва походить від структури оновлення:

$$(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} \dots). \tag{3.17}$$

Це алгоритм навчання з підкріпленням, який наближає функцію дій $q(s, a)$ на основі дій, що фактично виконує агент. Алгоритм використовує саме послідовність дій агента, включно з тим, яку дію агент вибрав у наступному стані. Це означає, що SARSA навчає політику, яку агент реально використовує під час взаємодії із середовищем [27].

Оновлення Q-функції в SARSA зображено на формулі 3.18.

$$q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma q(s',a') - q(s,a)]. \quad (3.18)$$

З цієї формули видно, що SARSA враховує реальну дію a', яка може бути не оптимальною.

DQN (Deep Q-Network). Для великих простору станів Q-функція апроксимується нейронною мережею. DQN включає механізми стабілізації – буфер досвіду та фіксовану цільову мережу.

Value-based підходи мають низку суттєвих переваг, які визначають їхню популярність у задачах з дискретними наборами дій. Вони добре працюють у середовищах, де простір дій є обмеженим і має чітку структуру, а функція $q(s,a)$ може бути точно обчислена або ефективно апроксимована. Завдяки цьому такі методи забезпечують достатню ефективність у моделях із помірною складністю та дозволяють отримати коректні оцінки цінності станів і дій. Ще однією властивістю є відносна простота реалізації: алгоритми цього класу не вимагають складних параметризацій політики, а процес налагодження є менш трудомістким у порівнянні з альтернативними підходами.

Водночас value-based методи мають і важливі обмеження. Вони погано адаптуються до неперервних або високорозмірних просторів дій, оскільки вибір дії через операцію argmax стає обчислювально складним або навіть неможливим. Політика в таких алгоритмах не представлена явно, а виводиться опосередковано з функції цінності, що ускладнює інтеграцію додаткових вимог до поведінки агента та унеможливлює формування стохастичних політик без спеціальних модифікацій. У великих просторах станів можуть спостерігатися коливання та нестабільність під час навчання, особливо у випадках, коли апроксимація Q-функції виконується нейронною мережею без додаткових механізмів стабілізації. Додатково слід зазначити, що value-based підходи чутливі до вибору функції винагороди та параметрів навчання, зокрема коефіцієнта дисконту та швидкості оновлення, що безпосередньо впливає на швидкість збіжності.

3.2.2 Policy-based

Методи прямої оптимізації політики формують окремий клас алгоритмів навчання з підкріпленням, у яких основним об’єктом оптимізації є сама політика, а не функція цінності. На відміну від value-based підходів, де політика виводиться неявно через функцію $q(s, a)$, у policy-based методах політика задається параметризованою функцією виду

$$\pi(a \mid s; \theta), \tag{3.19}$$

де θ – вектор параметрів моделі (зазвичай нейронної мережі).

Підхід дозволяє безпосередньо моделювати поведінку агента і оптимізувати її відповідно до критерія очікуваної сумарної винагороди.

Мета policy-based методів полягає у знаходженні таких параметрів θ, які максимізують функцію очікуваної винагороди:

$$J(\theta) = \mathbb{E}[G_t]. \tag{3.20}$$

Оскільки $J(\theta)$ рідко має аналітичний градієнт, у навчанні застосовуються стохастичні оцінки градієнта, отримані на основі траєкторій взаємодії агента із середовищем.

Одним із базових алгоритмів цього класу є REINFORCE, який належить до монте-карлівських методів. Його оновлення здійснюється за формулою:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi(a_t \mid s_t; \theta), \tag{3.21}$$

де ∇_θ – похідна за параметрами політики.

Це правило змінює параметри політики у напрямку, що збільшує ймовірність вибору дій, які привели до високих сумарних винагород. Такий

підхід не потребує знання моделі середовища, оскільки всі необхідні величини обчислюються з емпіричних даних.

Однак градієнтні оцінки в REINFORCE характеризуються підвищеною дисперсією, оскільки використовуються повні траєкторії епізоду. Висока варіативність у значеннях G_t може спричиняти коливання градієнта та уповільнення збіжності політики, особливо в складних або стохастичних середовищах.

Для зменшення нестабільності застосовують базову оцінку $b(\mathrm{s})$:

$$\theta \leftarrow \theta + \alpha(G_t - b(s_t))\nabla_\theta \log \pi(a_t \mid s_t; \theta). \tag{3.22}$$

Використання бази не впливає на математичне сподівання градієнта, але суттєво зменшує його дисперсію. Найчастіше як $b(\mathrm{s})$ застосовують функцію вартості $V(\mathrm{s})$, що дає змогу формувати величину переваги

$$A_t = G_t - V(s_t). \tag{3.23}$$

Це дозволяє оцінювати не абсолютний результат дії, а відхилення від очікуваної поведінки, що покращує стабільність оновлень і пришвидшує навчання.

Ефективність policy-based підходу зросла після появи алгоритмів, що вводять контроль над величиною оновлень політики.

Trust Region Policy Optimization (TRPO). Цей алгоритм запроваджує обмеження у вигляді «довірчого регіону», в межах якого може змінюватися політика під час одного оновлення [28]. Такий підхід запобігає надто різким змінам параметрів, що могло б призвести до деградації поведінки агента.

Proximal Policy Optimization (PPO). PPO пропонує спрощену версію обмеження на зміну політики, використовуючи механізм обмеження величини оновлення. Це дозволяє досягати продуктивності, близької до TRPO, але з набагато меншими обчислювальними витратами [29].

Ключовою перевагою прямих методів оптимізації політики є здатність працювати з неперервними просторами дій, де застосування value-based алгоритмів значно ускладнюється або стає неможливим. Крім того, policy-based методи природно підтримують стохастичні політики, що дозволяє моделювати невизначеність та покращує якість дослідження середовища. Параметризована політика може реалізовувати складні поведінкові стратегії, які важко отримати через неявне виведення політики з Q-функції.

Попри свої переваги, policy-based методи мають низку недоліків. Вони є вибірково неефективними, тобто потребують великої кількості епізодів для надійної оцінки градієнта. Крім того, алгоритми цього класу чутливі до налаштування параметрів – таких як швидкість навчання, форма базової функції чи структура нейронної мережі. Через високу дисперсію градієнтів можливі значні коливання під час навчання, що ускладнює процес оптимізації та вимагає стабілізуючих технік на кшталт використання переваги, нормалізації градієнтів або ентропійної регуляризації.

3.2.3 Actor-Critic

Actor-critic методи становлять окремий клас алгоритмів навчання з підкріпленням, які поєднують властивості методів прямої оптимізації політики та методів, заснованих на функції цінності. Ідея цих алгоритмів полягає в розділенні процесів вибору дії та оцінювання якості прийнятого рішення між двома взаємодоповнюючими компонентами. Компонент actor, відповідає за формування політики агента; це параметризована функція, яка визначає, яку дію слід обрати в певному стані. Другий компонент, critic, виконує роль оцінювача дій та станів, наближаючи функцію вартості або функцію переваги, що дозволяє оцінити, наскільки вдалою була дія у конкретному контексті. Завдяки наявності критика методи actor-critic

можуть зменшувати дисперсію градієнтних оцінок, що є серйозною проблемою для класичних policy-based алгоритмів, таких як REINFORCE.

Ключова властивість actor-critic підходу полягає в тому, що actor визначає напрямок оновлення політики, а critic коригує цей напрямок на основі оцінки цінності станів або дій. У стандартному градієнтному методі політики агент оновлює параметри виключно на основі сумарної винагороди епізоду, що призводить до значних коливань градієнтів. Натомість critic забезпечує локальну оцінку якості, що дозволяє швидше й стабільніше коригувати політику. Таким чином, методи actor-critic використовують повноцінну структуру функції цінності як базову функцію, що істотно знижує нестабільність та забезпечує надійніші оновлення параметрів політики.

Різні реалізації actor-critic методів відрізняються способом організації взаємодії між цими компонентами, механізмом збору досвіду та формулою оновлення політики. Одним із найпростіших і найвідоміших представників цього підходу є Advantage Actor-Critic (A2C). У цьому алгоритмі використовується оцінка переваги, що відображає різницю між реальною цінністю дії та оцінкою стану, що дозволяє actor приймати рішення, ґрунтуючись не на абсолютних значеннях винагороди, а на оцінці того, наскільки конкретна дія була кращою або гіршою за очікуване середнє. Близьким до A2C є алгоритм A3C, який використовує асинхронне навчання з залученням декількох паралельних агентів. Це дозволяє підвищити різноманітність досвіду, зменшити кореляції між переходами та пришвидшити навчання у складних середовищах.

До переваг actor-critic методів належить зменшена дисперсія градієнтів у порівнянні з чистими policy-based алгоритмами, здатність працювати у просторих або неперервних просторах дій та хороша збіжність у задачах складного управління. Водночас цей підхід має і свої обмеження. Однією з головних проблем є складність архітектури, яка потребує одночасного навчання політики та критика. Це створює ризик взаємної

нестабільності, коли помилки критика можуть негативно впливати на оновлення політики та навпаки. Крім того, навчання двох нейронних мереж підвищує обчислювальні вимоги та потребує ретельно підібраних гіперпараметрів.

3.3 Вибір алгоритму до конкретної задачі

У роботі досліджується задача оптимізації параметрів обчислювального середовища під час навчання згорткової нейронної мережі. Дії агента включають:

- зміну кількості потоків (torch threads);
- регулювання кількості DataLoader workers;
- зміну batch size;
- налаштування CPU-affinity;
- варіації learning rate.

Це середовище має низку властивостей, що прямо впливають на вибір RL-підходу:

– дискретний, але високорозмірний простір дій. Окремі параметри є дискретними, але їхня комбінація породжує велику кількість можливих дій. Табличні методи (Q-learning) не є придатними. DQN теоретично можливий, але вимагав би великих ресурсів і значної стабілізації;

– відсутність гладкості переходів. Зміна однієї дії може призвести до різких змін у часі епохи або завантаженні CPU. Таке стохастичне середовище важко апроксимувати класичними value-based методами;

– наявність стохастики та нестабільної винагороди. Час навчання, використання пам’яті, точність моделі – значення можуть змінюватися від запуску до запуску через випадковість у навчанні CNN. Стратегія агента повинна враховувати невизначеність і не бути повністю детермінованою;

– потреба в роботі з параметризованою політикою. Агент має формувати «м’який» режим вибору дій, у якому менш вигідні дії мають

невелику, але ненульову ймовірність. Це неможливо реалізувати через greedy-політики value-based підходів, де дія вибирається як argmax.

У таких умовах застосування value-based методів є менш доцільним, оскільки вони зазвичай передбачають детерміновану або майже детерміновану динаміку середовища і вимагають чіткої оцінки функції цінності для кожної комбінації стану та дії. Крім того, комбінація можливих дій у задачі оптимізації ресурсів утворює великий дискретний простір, що створює значні труднощі для класичних методів типу Q-learning чи SARSA, особливо в частині апроксимації Q-функції та забезпечення стабільної збіжності.

У таких умовах більш придатними є policy-based методи, оскільки вони дозволяють оптимізувати політику без необхідності формувати точну модель функції цінності. Параметризована політика задає безпосередньо ймовірності вибору дій, що є важливою властивістю у стохастичних контекстах. Стохастична політика, яку легко реалізувати в межах policy gradient, природно відповідає характеру задачі: навіть для однакових дій середовище може реагувати по-різному, тому детермінована політика не завжди є оптимальною. До того ж policy-based методи дають змогу безпосередньо оптимізувати очікувану сумарну винагороду, яка визначає якість обраної конфігурації параметрів. Наприклад, винагорода може відображати одночасно зменшення часу навчання та збереження точності, що важко врахувати при побудові Q-функції, але природно інтегрується у об’єктивну функцію політики.

Незважаючи на ці переваги, чисті policy-based методи, зокрема REINFORCE, відомі високою дисперсією оцінок градієнта, що може значно уповільнювати або навіть дестабілізувати процес навчання. Цей ефект особливо помітний у середовищах зі складною та шумною функцією винагороди, якою є оптимізація параметрів навантаження на CPU та інших обчислювальних ресурсів. Саме тому використання actor-critic підходів є ще більш доцільним. Додавання критика, що оцінює функцію вартості або

переваги, дозволяє суттєво зменшити дисперсію стохастичного градієнта та забезпечити більш стабільне оновлення параметрів політики. Це означає, що критик виступає у ролі коригуючого елемента, який допомагає формувати напрямок градієнта не на основі повної накопиченої винагороди, а з урахуванням відхилення отриманої винагороди від очікуваної. У результаті політика змінюється плавніше, що покращує збіжність у середовищах зі значною випадковістю.

Actor-critic методи також демонструють переваги у великих просторах дій, оскільки вони не потребують зберігання або апроксимації повної Q-функції. Для задачі оптимізації ресурсів, де комбінація можливих налаштувань утворює простір високої розмірності, цей підхід є особливо ефективним. Крім того, actor-critic методи краще справляються зі складними, нерівномірними та шумними функціями винагороди. Наприклад, час епохи навчання може змінюватися залежно від взаємодії між кількістю потоків, CPU-affinity та обсягом доступної пам’яті. У таких ситуаціях наявність критика допомагає стабілізувати градієнт політики та уникнути різких змін поведінки агента.

У практичній частині роботи застосовано підхід, який концептуально близький до actor-critic методів. Політика оновлюється за градієнтним принципом, а для зменшення дисперсії використовується базова функція у вигляді оцінки переваги. Це узгоджується з сучасними підходами параметризації політик, де оцінка переваги A(s,a) дозволяє визначати відносну корисність дії, що значно поліпшує стабільність та ефективність навчання. Завдяки цьому агент може швидше адаптуватися до змін середовища, формувати більш узгоджену поведінку та досягати кращих результатів у задачах оптимізації обчислювальних ресурсів.

4 РОЗРОБКА ТА НАВЧАННЯ СИСТЕМИ

4.1 Набір використаних програмних інструментів

Розробка та навчання системи оптимізації обчислювальних ресурсів на основі підкріплювального навчання та згорткової нейронної мережі потребує використання низки програмних засобів, які забезпечують реалізацію моделей, проведення експериментів, збір метрик та управління обчисленнями. У даній роботі застосовано сучасні інструменти машинного навчання та засоби аналізу продуктивності, що дозволило реалізувати як базову модель CNN, так і RL-агент, який модифікує параметри навчання.

Уся система реалізована мовою Python версії Python 3.13.9. Розробка проводилася в середовищі:

– Visual Studio Code – основне IDE з підтримкою інтегрованого інтерпретатора Python, дебагера та менеджера пакетів;

– Jupyter Notebook – інструмент для попередніх експериментів, графічної візуалізації та аналізу поведінки алгоритму.

Основним програмним середовищем для побудови нейронних мереж є фреймворк PyTorch (література), який забезпечує модульну архітектуру, автоматичне диференціювання та підтримку апаратного прискорення за допомогою CUDA. PyTorch використано для побудови та навчання згорткової нейронної мережі, параметризації політики RL-агента та реалізації механізмів зворотного поширення градієнта.

Для роботи з даними використано бібліотеку torchvision, яка містить навчальні набори, засоби попередньої обробки зображень та стандартні перетворення. У якості базового датасету застосовано MNIST, що містить 60 000 зображень для навчання та 10 000 для тестування.

Для збору системних метрик було використано бібліотеку psutil. Метрики наступні:

– завантаження процесора;

– кількість активних потоків;

– використання оперативної пам’яті;

– CPU affinity поточного процесу.

Дана бібліотека дає змогу отримати доступ до низькорівневих характеристик операційної системи і необхідна для формування станів середовища у моделі Reinforcement Learning.

4.2 Базова модель

Базова модель, що використовується у даній роботі, представляє собою згорткову нейронну мережу, призначену для класифікації зображень рукописних цифр з датасету MNIST. Ця модель є центральним компонентом побудови середовища для навчання з підкріпленням: вона виступає як об’єкт оптимізації, для якого RL-агент підбирає параметри обчислювального середовища, її поведінка у процесі навчання формує стани та винагороди, на основі яких агент навчає політику.

4.2.1 Архітектура CNN

Структура моделі побудована за класичним принципом: кілька згорткових шарів для виділення ознак та один або кілька повнозв’язних шарів для класифікації. Мережа складається з двох основних блоків: блоку згорткових операцій та блоку повнозв’язних шарів.

Згортковий блок складається з трьох послідовних шарів виду:

– Conv2d(1 → 32, kernel_size=3);

– Conv2d(32 → 64, kernel_size=3);

– Conv2d(64 → 64, kernel_size=3).

Ці шари поетапно збільшують кількість фільтрів, дозволяючи мережі формувати ознаки. Для задачі MNIST така архітектура є оптимальною для

досягнення високої точності, оскільки зображення мають низьку складність та малий розмір (28×28 пікселів).

Кожен згортковий шар супроводжується функцією активації ReLU, що сприяє отриманню нелінійності та запобігає проблемі зникання градієнта.

Після згорткових операцій використовується шар:

– Flatten → Linear(64 × 22 × 22 → 10).

Результат згорткових перетворень перетворюється у вектор, який слугує входом до лінійного шару, що прогнозує 10 класів (цифри 0–9).

Як функцію втрат застосовано CrossEntropyLoss. Оптимізація параметрів здійснюється за допомогою Adam.

4.2.2 Процес навчання та оцінювання моделі

У процесі навчання модель проходить через повний цикл:

– завантаження пакета даних;

– виконання прямого проходу через CNN;

– обчислення функції втрат;

– обчислення градієнтів;

– оновлення ваг оптимізатором.

Тренувальні дані завантажуються за допомогою DataLoader, який забезпечує пакетну обробку та випадкове перемішування вибірки.

Кожна епоха складається з послідовного проходження всіх пакетів даних, після чого вимірюються метрики продуктивності. Ці метрики (час епохи, точність, завантаження CPU, використання пам’яті, кількість потоків) формують стани та винагороди у RL-середовищі. Це забезпечує формування зворотного зв’язку, за якого зміни параметрів обчислювального середовища безпосередньо відображаються на станах та винагородах RL-агента.

4.3 План впровадження RL алгоритму

Алгоритм навчання з підкріпленням у даній роботі побудований таким чином, щоб агент міг впливати на параметри обчислювального середовища під час навчання згорткової нейронної мережі та, на основі спостережуваних метрик, поступово покращувати політику вибору цих параметрів. Система реалізує цикл «агент – середовище», де середовищем виступає процес навчання CNN у фреймворку PyTorch, а агентом є політика, яка обирає конфігурацію ресурсів на кожну епоху.

4.3.1 Загальна схема роботи RL-модуля

Роботу системи можна описати наступним чином (блок-схема послідовності дій показана на рисунку 4.1). Спочатку виконується ініціалізація даних, побудова простору дій (усіх можливих конфігурацій параметрів), задання політики у вигляді лінійної моделі з softmax над діями, а також підготовка структур для логування результатів. Далі запускається основний цикл по епохах, у якому одна ітерація циклу відповідає одному кроку взаємодії агента з середовищем.

На початку кожної епохи формується вектор стану, що містить нормалізовану інформацію про номер епохи, попередню точність, нормовані значення часу, завантаження процесора, використання пам’яті та останню винагороду. На основі цього стану політика породжує розподіл ймовірностей над попередньо визначеним набором дій (конфігурацій ресурсів) та випадковим чином вибирає одну дію згідно з цим розподілом. Вибрана дія застосовується до середовища: змінюється кількість потоків PyTorch, CPU affinity процесу, параметри DataLoader та коефіцієнт навчання моделі.

Після цього створюється новий екземпляр згорткової нейронної мережі та оптимізатора, формується DataLoader із параметрами, заданими

дією, і запускається навчання однієї епохи. Навчання обгорнуте в спеціальну функцію моніторингу, яка в процесі виконання вимірює середнє завантаження процесора, пікове використання оперативної пам’яті та загальну тривалість епохи.

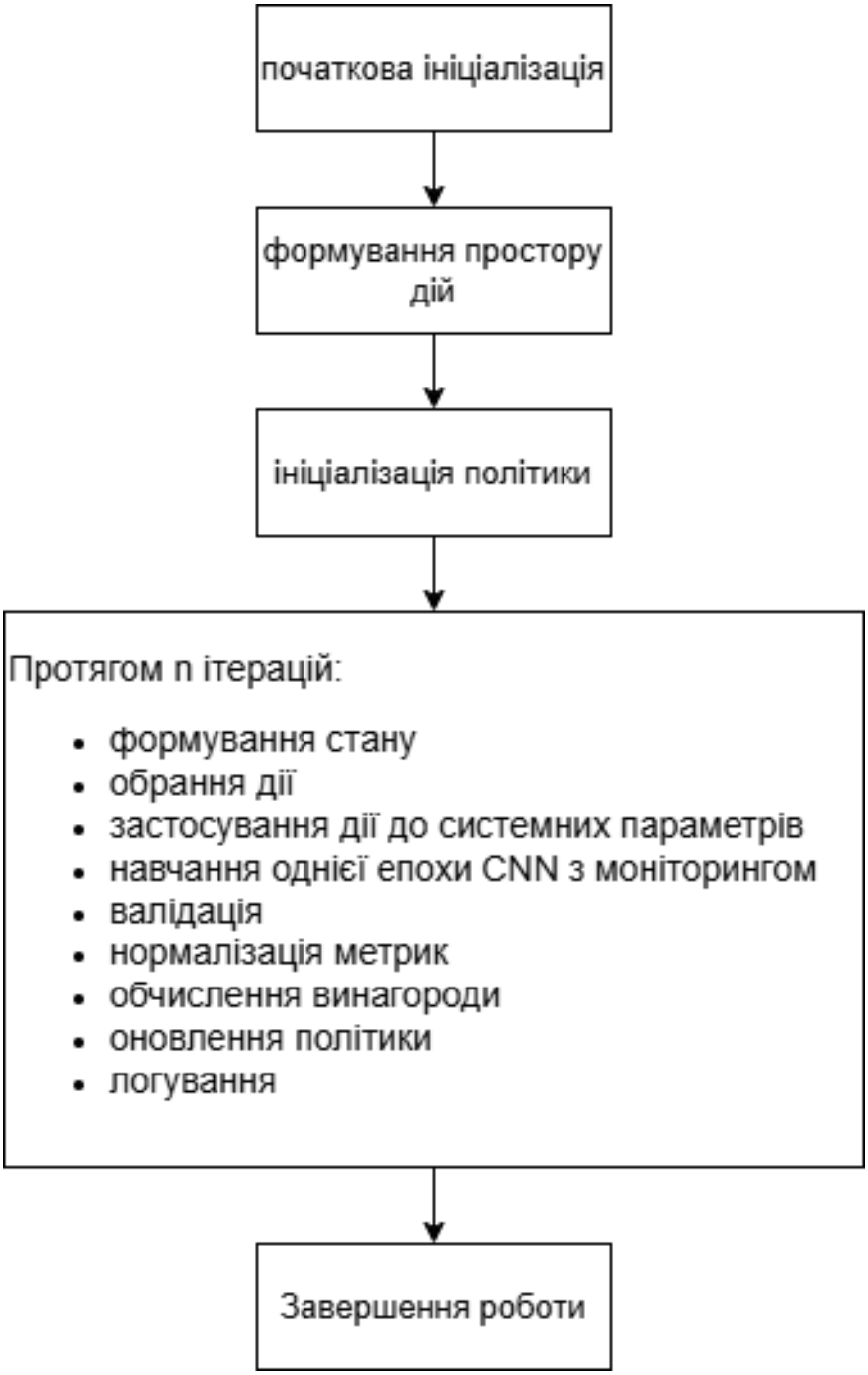

Рисунок 4.1 – Блок-схема послідовності дій системи розподілу ресурсів на основі алгоритму навчання з підкріпленням

Після завершення навчання проводиться валідація моделі на відкладеній вибірці, обчислюється точність, а усі метрики нормалізуються відносно максимальних спостережених значень.

На наступному кроці обчислюється значення функції винагороди, яка поєднує нормовану точність, час, завантаження CPU та використання пам’яті з різними вагами залежно від обраного режиму (наприклад, «balanced» чи «speed»). Отримана винагорода використовується для оновлення параметрів політики за градієнтним правилом з використанням базової оцінки (baseline), що зменшує дисперсію градієнта. Також усі дані епізоду зберігаються у CSV-файл, що дозволяє виконувати подальший аналіз.

4.3.2 Простір дій

Першим ключовим кроком у побудові RL-системи є задання простору можливих дій агента. У коді це реалізовано через dataclass Action, який описує одну конфігурацію параметрів:

– batch_size – розмір пакета даних для навчання;

– num_workers – кількість процесів-завантажувачів, які готують батчі у DataLoader;

– torch_threads – кількість потоків, що використовуються PyTorch для обчислень;

– cpu_affinity – список логічних ядер, на яких дозволено виконання процесу;

– lr – коефіцієнт навчання оптимізатора Adam.

Функція make_affinity_presets() будує набір допустимих пресетів для прив’язки процесу до логічних ядер CPU. На основі кількості доступних логічних ядер формуються різні варіанти: від одного ядра до половини та всіх ядер системи. Це дозволяє експериментально досліджувати вплив різних конфігурацій прив’язки на час навчання та завантаження процесора.

У функції start_rl формується повний список дій actions: для кожного batch_size у множині {32, 64, 128}, кожного числа workers {0, 2, 4}, числа потоків {1, 2, 4, 6}, значень коефіцієнта навчання {1e-2, 3e-4} та для кожного шаблону CPU affinity створюється окремий екземпляр Action. Потім, щоб обмежити розмір простору дій, при надмірно великій кількості можливих конфігурацій виконується вибір 80 з них. Таким чином, агент працює з дискретним, але достатньо різноманітним простором дій, що дозволяє йому вивчити різні режими використання ресурсів.

4.3.3 Опис станів та ініціалізація політики

Стан системи на кожному кроці описується невеликим вектором фіксованої розмірності 6. Він включає такі компоненти:

– номер епохи;

– значення досягнутої точності у попередній епосі;

– час виконання попередньої епохи;

– середнє завантаження CPU у попередній епосі;

– пікове використання оперативної пам’яті у попередній епосі;

– остання винагорода.

Таким чином, агент на кожному кроці отримує узагальнену інформацію про те, як працювала попередня конфігурація, і може приймати рішення, спираючись не лише на номер епохи, а й на якість роботи системи.

Політика реалізована класом SoftmaxPolicy. Це лінійна модель, параметризована матрицею ваг policy_weights розміру [n_actions, state_dim]. Для заданого стану обчислюються логіти дій logits = W · state, після чого до них застосовується функція softmax, що дає ймовірнісний розподіл над діями. Дія вибирається методом torch.multinomial, згідно з отриманим розподілом. Такий підхід відповідає стохастичній політиці, де навіть дії з меншою ймовірністю мають шанс бути обраними, що сприяє дослідженню простору дій.

Політика має власну швидкість навчання lr та скалярну змінну baseline, яка використовується для зменшення дисперсії оцінки градієнта. Параметр remembering_old визначає, наскільки «інерційно» оновлюється baseline через експоненціальне згладжування. Додатково це дозволяє зробити процес оновлення політики більш стабільним і зменшити вплив випадкових коливань винагороди між окремими епохами.

4.3.4 Моніторинг системних ресурсів під час навчання

Однією з ключових особливостей системи є те, що винагорода базується не лише на точності моделі, але й на характеристиках використання обчислювальних ресурсів. Для цього в коді передбачено спеціальну функцію-обгортку measure_cpu_ram_time. Вона отримує на вхід функцію, яка виконує епоху навчання, і запускає її в окремому потоці. У головному потоці з певним інтервалом (0.2 секунди) зчитуються:

– поточне завантаження CPU через p.cpu_percent(interval=0.2);

– фактичне використання оперативної пам’яті через p.memory_info().rss.

Ці значення накопичуються у список вибірок CPU та змінну пікового значення RAM. Після завершення потоку з навчанням обчислюється середнє завантаження процесора, пікове використання пам’яті та загальна тривалість епохи за різницею часових міток. Саме ці три величини є основою для простору станів, разом із середнім значенням функції втрат. Додатково вони використовуються для нормалізації метрик і коректного порівняння результатів між різними конфігураціями виконання, що дозволяє оцінювати не лише швидкість навчання, але й ефективність використання ресурсів у кожній епосі.

Такий підхід дозволяє не втручатися у внутрішню логіку навчання моделі, але при цьому отримувати детальну інформацію про поведінку процесу з погляду системних ресурсів.

4.3.5 Функція винагороди та нормалізація метрик

Винагорода у задачі визначається як функція від нормованих значень точності, часу, завантаження CPU та використання пам’яті. Нормалізація виконується відносно поточних максимумів, що зберігаються у змінних max_time, max_cpu та max_ram. Для кожної епохи:

– time_norm = dur / max_time;

– cpu_norm = cpu_mean / max_cpu;

– ram_norm = ram_peak / max_ram;

– acc_norm = acc (точність уже лежить у [0,1]).

Функція build_reward_fn створює конкретну функцію винагороди залежно від обраного режиму mode. Наприклад, у режимі «speed» більша вага надається часу, тоді як у «balanced» час, CPU та RAM мають більш збалансовані вагові коефіцієнти. Усередині сформованої функції винагорода обчислюється як лінійна комбінація:

$$reward = w_{acc} \cdot acc_{norm} - w_{time} \cdot time_{norm} - w_{cpu} \cdot cpu_{norm} - \\ -w_{ram} \cdot ram_{norm}. \quad (4.1)$$

Таким чином, агент завжди зацікавлений у збільшенні точності, але водночас штрафується за надто тривалий час виконання та надмірне завантаження ресурсів. Це безпосередньо відображає постановку задачі: не лише навчити модель добре класифікувати, але й зробити це з меншими витратами часу й ресурсів.

4.3.6 Оновлення політики та цикл навчання

Коли стан, дія, ймовірнісний розподіл та винагорода відомі, викликається метод оновлення політики. У ньому реалізовано варіант REINFORCE з базовою оцінкою. Спершу обчислюється перевага:

$$advantage = reward - baseline, \tag{4.2}$$

де $baseline$ оновлюється через експоненціальне згладжування:

$$baseline \leftarrow \rho \cdot baseline + (1 - \rho) \cdot reward, \tag{4.3}$$

де ρ – коефіцієнт забування старої інформації.

Далі обчислюється градієнт логарифму політики у вигляді вектора, який дорівнює мінус вектору ймовірностей, а для обраної дії до відповідної компоненти додається 1. Це відповідає класичному виразу $\nabla \log \pi(a \mid s)$ для softmax-політики.

Оновлення ваг політики виконується за правилом:

$$W \leftarrow W + \alpha \cdot advantage \cdot (\nabla_\theta \log \pi(\, a_t \mid s_t\,)) \cdot s_t^\top. \tag{4.4}$$

Таким чином, якщо винагорода перевищує $baseline$, то ймовірність обраної дії збільшується, а якщо нижча – зменшується.

5 ЕКСПЕРИМЕНТАЛЬНИЙ РОЗДІЛ

5.1 Опис датасету та вихідних даних

У рамках експериментальної частини дослідження було використано датасет MNIST, що є класичним набором рукописних зображень у задачах класифікації. Датасет містить:

– 60 000 зображень у тренувальній вибірці;

– 10 000 зображень у тестовій вибірці.

Кожен об’єкт представлений у вигляді монохромного зображення 28×28 пікселів та належить до одного з 10 класів (цифри від 0 до 9). Кожен клас має приблизно рівномірний розподіл прикладів, що формує збалансовану вибірку. Відсутність кольорових каналів і невеликий розмір зображень роблять MNIST зручним для швидких експериментів, пов’язаних із дослідженням впливу системних ресурсів на навчання згорткових нейронних мереж.

У роботі використовувалися як повні вибірки, так і їхні підмножини (55 000 елементів для тренування і 5 000 – для валідації), що дозволяло відтворити реалістичний процес оптимізації продуктивності на різних обсягах даних.

5.2 Конфігурація апаратного забезпечення

Усі експерименти проводилися на персональному ноутбуці з такою апаратною конфігурацією:

а) процесор: Intel® Core™ i7-13620H, 13-те покоління;

– 10 фізичних ядер (6 продуктивних + 4 енергоефективних);

– базова частота: 2.40 GHz;

б) оперативна пам’ять: 16 ГБ (15.7 ГБ доступно користувачу).

Графічний прискорювач у цьому експерименті не використовувався, тому всі результати відображають поведінку системи виключно в CPU-обчисленнях, що дозволило оцінити вплив потоків, кількості робочих процесів DataLoader та прив’язки до ядер.

5.3 Структура конфігурацій експерименту

Оптимізаційні рішення визначалися через дискретний простір можливих дій, який формувався класом Action. Кожна дія – це унікальна комбінація таких параметрів:

а) batch_size ∈ {32, 64, 128};

б) num_workers ∈ {0, 2, 4};

в) torch_threads ∈ {1, 2, 4, 6};

г) cpu_affinity – набір ядер CPU, який може включати:

– одне ядро;

– два ядра;

– перші п’ять ядер;

– половину доступних ядер;

– усі ядра системи;

д) learning_rate ∈ {0.01, 0.0003}.

У результаті утворюється кілька сотень можливих конфігурацій, проте в рамках одного запуску алгоритм випадково відбирав 80 дій з усього простору, що зберігало обчислювальну ефективність.

Усі результати зберігалися у структуру EpisodeLog, яка фіксує повну статистику навчання на кожній епосі, як показано в таблиці 5.1:

– accuracy, loss, duration_s;

– середнє завантаження CPU;

– пікове споживання RAM;

– reward;

– параметри обраної дії.

Таблиця 5.1 – Приклад запису EpisodeLog

Показник статистики навчання	Значення показника
epoch	1
action_id	73
acc	0.97
loss	0.1581666858361526
duration_s	607.4495787620544
cpu_mean	153.7043509272468
ram_peak_mb	361.28125
reward	-0.6300000000000001
batch_size	32
num_workers	0
torch_threads	6
lr	0.0003
cpu_affinity	[0, 1]

CSV-файли, згенеровані системою, містили повну історію роботи агента. На їх основі були побудовані графіки динаміки, порівняння конфігурацій і павучкові діаграми.

5.4 Дизайн експерименту

Експериментальна частина складалася з декількох послідовних етапів, кожен з яких відповідав на окреме дослідницьке питання: як змінюється продуктивність моделі за зміни системних ресурсів і як агент навчання з підкріпленням адаптується до цих умов.

Для оцінки нижньої межі продуктивності система спочатку запускалася в умовах, близьких до найменш ефективних:

– batch_size = 32;

– num_workers = 0 (один процес обробки даних);

– torch_threads = 1;

– cpu_affinity = одне ядро CPU;

– lr = 0.01.

Мета цього етапу – отримати орієнтири, які показують поведінку моделі при максимальному дефіциті ресурсів. Очікувано, час епох був одним з найбільших, а стабільність навчання – найнижчою. Ці результати наочно ілюструють вплив жорстких обмежень характеристики навчання та поведінку метрик моделі.

Параметри статистики навчання наведені в таблиці 5.2, графіки тривалості та точності навчання – на рисунках 5.2, 5.3.

Таблиця 5.2 – Метрики експерименту при максимальному дефіциті ресурсів

Параметр статистики навчання	Значення параметрів навчання залежно від епохи							
epoch	1	2	3	4	5	6	11	12
action_id	0	0	0	0	0	0	0	0
acc	0.95	0.96	0.96	0.97	0.96	0.96	0.96	0.96
loss	0.20	0.09	0.08	0.07	0.06	0.06	0.05	0.05
duration_s	96.05	94.83	94.62	94.15	93.71	94.88	96.40	96.68
cpu_mean	98.27	98.82	98.16	98.57	98.57	98.53	98.72	98.83
ram_peak_mb	366.5	483.7	459.7	456.2	479.8	456.4	483.5	456.3
reward	-0.6	-0.6	-0.6	-0.6	-0.6	-0.6	0.6	0.6
batch_size	32	32	32	32	32	32	32	32
num_workers	0	0	0	0	0	0	0	0
torch_threads	1	1	1	1	1	1	1	1
lr	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
cpu_affinity	1	1	1	1	1	1	1	1

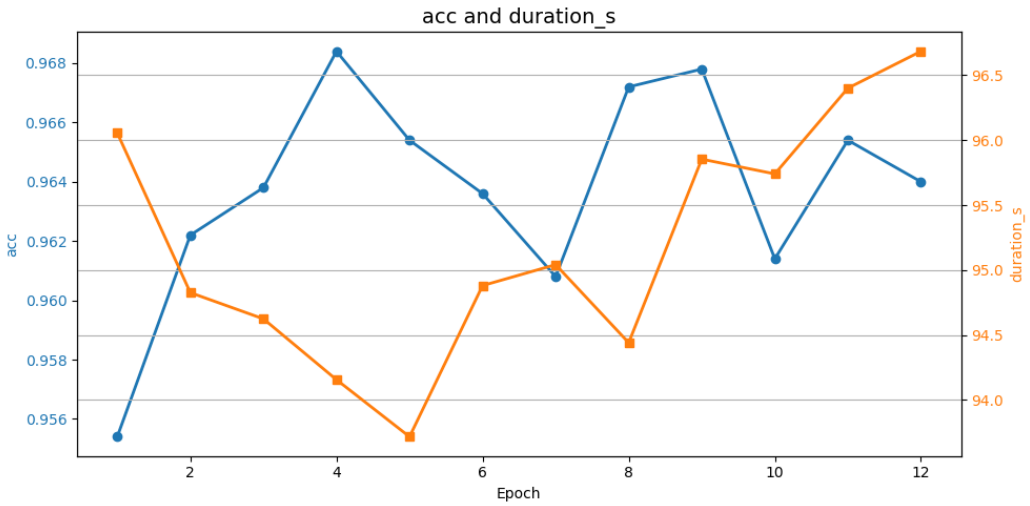

Рисунок 5.2 – Графіки часу та точності моделі при максимальному дефіциті ресурсів

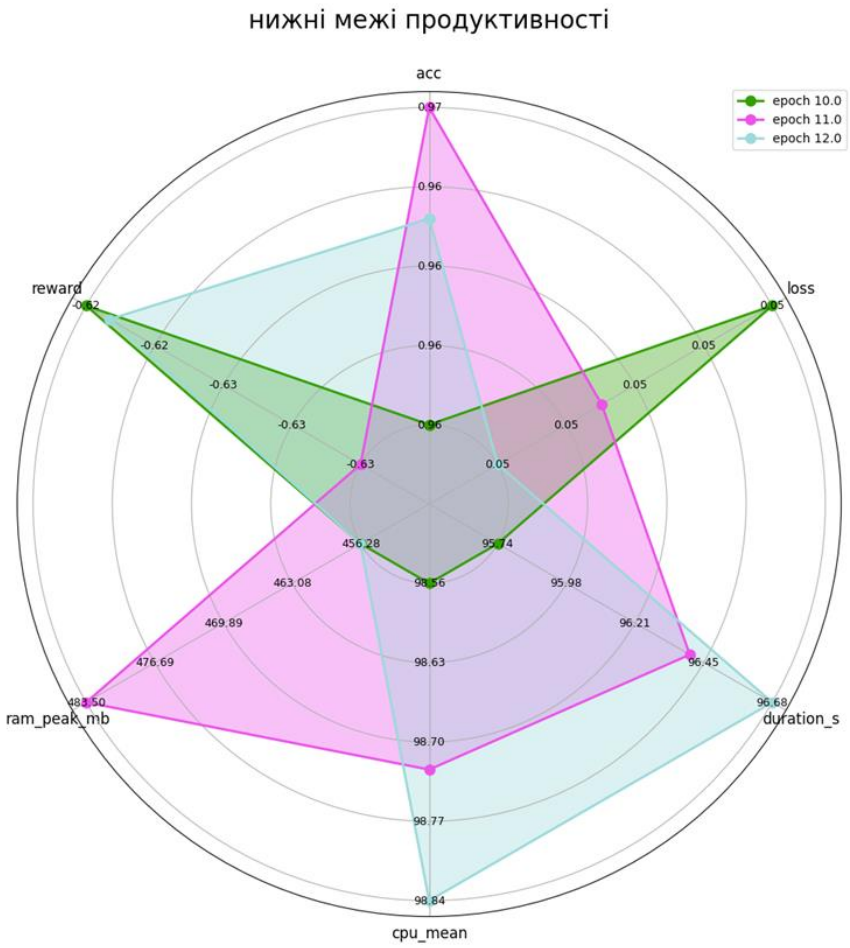

Рисунок 5.3 – Метрики та показники точності навчання моделі при максимальному дефіциті ресурсів

Другий етап дослідження – перевірка конфігурацій, що максимально завантажують систему:

– batch_size = 128;

– num_workers = 4;

– torch_threads = 6;

– cpu_affinity = усі ядра CPU;

– lr = 0.0003.

Це дозволило оцінити граничні можливості ноутбука. Час епох суттєво зменшився, проте у деяких випадках спостерігалися пікові навантаження CPU до 350–380%, що впливало на температуру. Параметри статистики навчання наведені в таблиці 5.3, графіки тривалості та точності навчання – на рисунках 5.4, 5.5.

Таблиця 5.3 – Метрики експерименту при максимальному навантаженні

Параметр статистики навчання	Значення параметрів навчання залежно від епохи							
epoch	1	2	3	4	5	6	11	12
action_id	0	0	0	0	0	0	0	0
acc	0.96	0.97	0.98	0.98	0.98	0.98	0.98	0.98
loss	0.25	0.06	0.04	0.03	0.02	0.01	0.006	0.005
duration_s	48.7	40.9	39.1	40.3	41.6	41.7	41.4	42.2
cpu_mean	356.4	367.6	372.9	364.4	373.5	374.6	376.6	375.7
ram_peak_mb	463.3	504.5	529.9	504.5	529.1	529.0	526.6	526.6
reward	-0.63	-0.49	-0.46	-0.45	-0.49	-0.50	-0.49	-0.50
batch_size	128	128	128	128	128	128	128	128
num_workers	4	4	4	4	4	4	4	4
torch_threads	6	6	6	6	6	6	6	6
lr	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
cpu_affinity	16	16	16	16	16	16	16	16

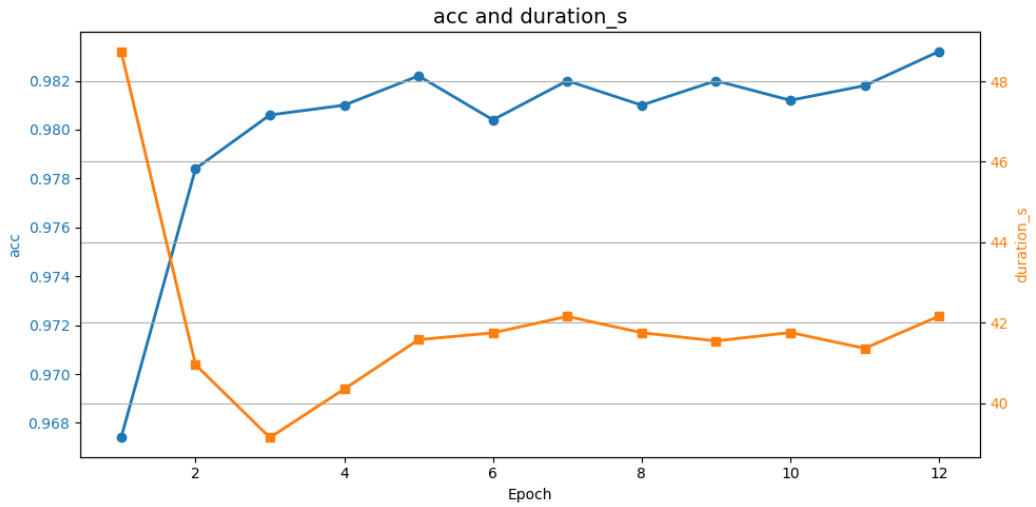

Рисунок 5.4 – Графіки часу та точності моделі при максимальному навантаженні

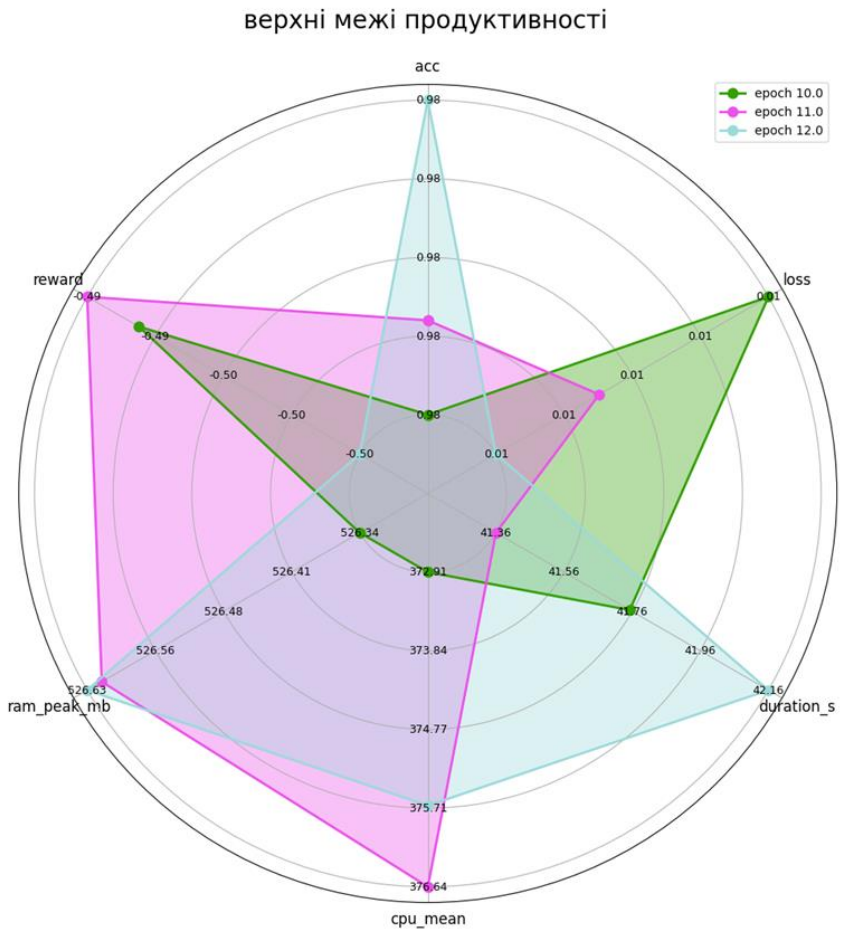

Рисунок 5.5 – Метрики та показники точності навчання моделі при максимальному навантаженні

5.4.1 Дослідження параметрів у критичних точках

На цьому етапі проводився аналіз, коли всі параметри встановлені на максимум, окрім одного, який варіюється. Для кожного параметра було виконано повний набір тестів.

Спершу, досліджено batch_size ∈ {32,64,128}. Фіксовані параметри: num_workers=4, torch_threads=6, affinity=всі ядра. Спостереження наступні:

– збільшення batch_size зменшує час епохи до певної межі;

– при надмірно великому batch_size можливі втрати точності на ранніх етапах.

Потім, розглянуто параметр num_workers ∈ {0,2,4}. Спостереження наступні:

– перехід з 0 до 2 дає найбільший приріст;

– 4 workers іноді перевантажують CPU при дрібних batch.

Наступним параметром став torch_threads ∈ {1,2,4,6}. Помічено наступні залежності:

– збільшення потоків пришвидшує обчислення;

– проте у деяких випадках виникає контеншен ресурсів.

Фінальним параметром під час дослідження CPU affinity. Варіюють набори ядер: 1 → 2 → 5 → 8 → 16.

– обмеження на 1–2 ядра значно погіршує час виконання;

– використання всіх ядер є оптимальним, але призводить до найвищого пікового CPU.

5.4.2 Аналіз взаємодії двох параметрів одночасно

Щоб знайти комбінації, які дають найкращий баланс між продуктивністю та ресурсами, було протестовано пари параметрів:

– (batch_size, num_workers);

– (num_workers, torch_threads);

– (threads, affinity);

– (learning_rate, batch_size).

Ці експерименти показали, що параметри не є незалежними. Наприклад:

– великий batch_size має сенс лише при достатній кількості потоків;

– CPU affinity сильно впливає саме у комбінації з torch_threads.

5.4.3 Навчання агента RL

У фінальній частині експериментів запускався повний RL-алгоритм:

– політика SoftmaxPolicy вибирала дії на основі поточного стану;

– reward враховував точність, час, CPU та RAM;

– baseline зменшував дисперсію;

– за 12 епох політика адаптувалася і починала вибирати стабільні конфігурації.

Спостерігалося, що після 8 – 11 епох агент майже перестає вибирати погані конфігурації й тяжіє до параметрів, які мінімізують час та максимізують точність. Параметри статистики навчання наведені в таблиці 5.4, графіки тривалості та точності навчання – на рисунках 5.6, 5.7.

Таблиця 5.4 – Метрики експерименту при повному RL-алгоритмі

Параметр статистики навчання	Значення параметрів навчання залежно від епохи							
epoch	1	2	3	4	5	6	11	12
action_id	33	13	42	51	26	71	39	69
acc	0.93	0.97	0.97	0.96	0.96	0.97	0.96	0.96
loss	0.25	0.06	0.05	0.08	0.09	0.04	0.04	0.03
duration_s	57.55	93.21	107.16	104.33	218.04	44.13	39.08	45.38
cpu_mean	167.77	98.75	98.84	86.92	275.15	269.09	288.12	271.26
ram_peak_mb	399.51	499.66	540.74	547.99	559.57	564.00	568.40	560.14

Продовження таблиці 5.4

Параметр статистики навчання	Значення параметрів навчання залежно від епохи							
reward	-0.63	-0.41	-0.41	-0.36	-0.63	0.02	0.22	0.25
batch_size	64	32	64	128	32	128	64	128
num_workers	2	0	0	4	4	4	2	4
torch_threads	2	2	2	1	6	4	4	4
lr	0.01	0.0003	0.0003	0.01	0.01	0.0003	0.01	0.01
cpu_affinity	16	2	1	2	5	16	8	8

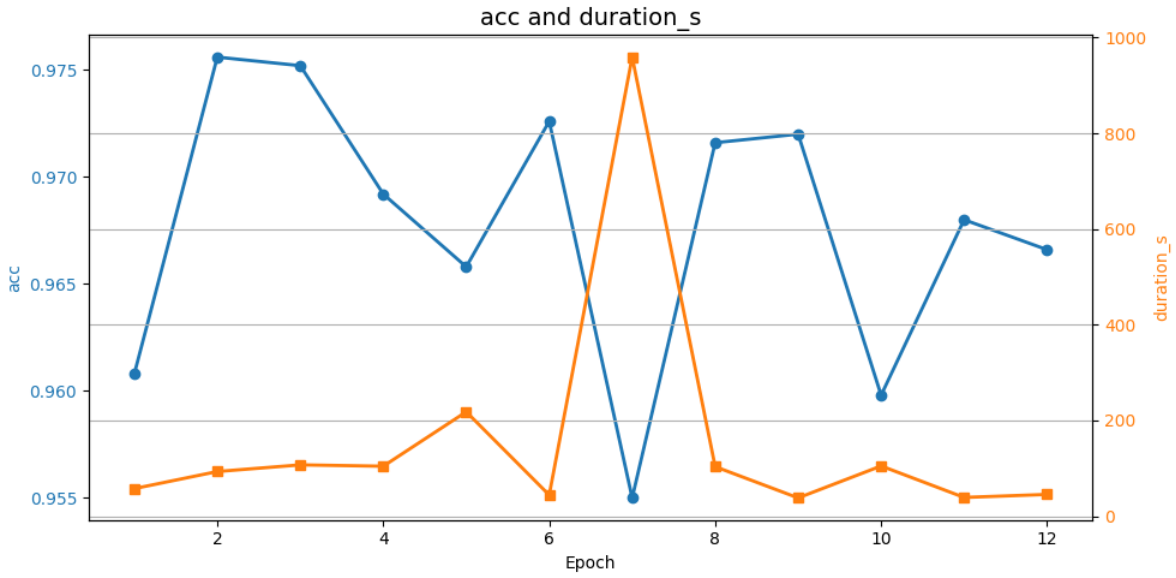

Рисунок 5.6 – Графіки часу та точності моделі при повному RL-алгоритмі

Проведені експерименти показали, що продуктивність навчання CNN істотно залежить від узгодженого вибору параметрів обчислювального середовища, причому як надмірний дефіцит, так і максимальне навантаження ресурсів не завжди є оптимальними. Аналіз критичних точок і пар параметрів підтвердив наявність сильних взаємозв'язків між batch size, кількістю потоків, workers та CPU affinity, що унеможливлює ефективне ручне налаштування.

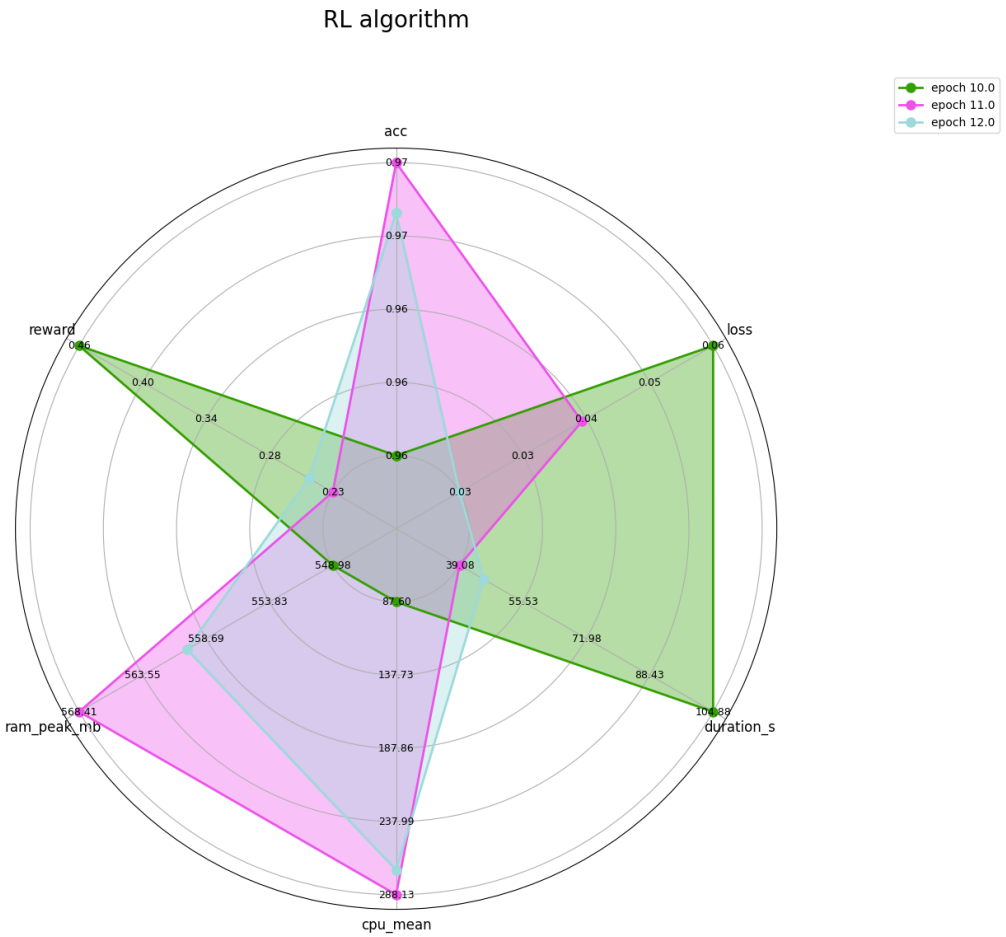

Рисунок 5.7 – Метрики та показники точності навчання моделі при повному RL-алгоритмі

RL-агент, використовуючи багатокритеріальну функцію винагороди, навчився автоматично знаходити компромісні конфігурації, які забезпечують високу точність при помірному часі виконання та контрольованому навантаженні CPU і RAM.

ВИСНОВКИ

У ході виконання роботи досліджено можливості застосування навчання з підкріпленням для оптимізації розподілу обчислювальних ресурсів під час тренування згорткових нейронних мереж. Проведений теоретичний аналіз дозволив визначити ключові особливості функціонування фізичних і логічних ядер, процесів і потоків, а також механізмів багатопотоковості, характерних для сучасних ML-фреймворків.

У практичній частині реалізовано агент підкріплювального навчання, здатний динамічно змінювати параметри обчислювального середовища – кількість процесів, потоків та розподіл навантаження між ядрами – на основі метрик навчання моделі. Для цього було створено експериментальну платформу, яка дозволила оцінити поведінку системи у різних конфігураціях та визначити, як зміна параметрів впливає на швидкість і стабільність тренування CNN. Особливу увагу приділено взаємозв’язку між системними параметрами та характеристиками навчального процесу, зокрема тривалістю епох і коливаннями точності.

Результати експериментів показали, що запропонований підхід не забезпечує різкого прискорення навчання на всіх етапах, однак у фінальних епохах зафіксовано стабільне й помірне покращення продуктивності порівняно з фіксованими конфігураціями. Це свідчить про здатність RL-агента адаптуватися до стану системи та поступово знаходити більш ефективні варіанти розподілу ресурсів.

Дослідження також виявило низку особливостей застосування RL у подібних задачах, зокрема залежність швидкості збіжності від складності середовища, взаємодії потоків та роботи паралельних бібліотек. Було встановлено, що за умов динамічної зміни конфігурацій обчислювального середовища агент потребує певного періоду адаптації, протягом якого можливі коливання продуктивності. Незважаючи на це, результати

демонструють практичну доцільність використання підкріплювального навчання для адаптивного керування ресурсами під час навчання моделей.

Загалом виконана робота підтверджує перспективність підходу та демонструє можливість подальшого розвитку систем автоматичного керування обчислювальними ресурсами. Отримані результати можуть бути використані як основа для розширення досліджень у напрямі оптимізації продуктивності ML-систем, їх інтеграції у хмарні середовища та побудови інтелектуальних платформ для високопродуктивних обчислень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Exploring the use of Hyper-Threading technology for multimedia applications with Intel/spl reg/ OpenMP compiler / Xinmin Tian et al. *International parallel and distributed processing symposium (IPDPS 2003)*, Nice, France. URL: https://doi.org/10.1109/ipdps.2003.1213118 (date of access: 05.12.2025).

2. Bober E., Bylina B. Teaching parallel programming on the CPU based on matrix multiplication using MKL, openmp and SYCL libraries. *17th international conference on computer supported education*, Porto, Portugal, 1–3 April 2025. P. 713–720. URL: https://doi.org/10.5220/0013279100003932 (date of access: 05.12.2025).

3. A survey of advancements in scheduling techniques for efficient deep learning computations on gpus / R. Kaur et al. *Electronics*. 2025. Vol. 14, no. 5. P. 1048. URL: https://doi.org/10.3390/electronics14051048 (date of access: 05.12.2025).

4. Kaur R., Mohammadi F. Power estimation and comparison of heterogenous CPU-GPU processors. *2023 IEEE 25th electronics packaging technology conference (EPTC)*, Singapore, 5–8 December 2023. 2023. URL: https://doi.org/10.1109/eptc59621.2023.10457590 (date of access: 05.12.2025).

5. Preuveneers D., Tsingenopoulos I., Joosen W. Resource usage and performance trade-offs for machine learning models in smart environments. *Sensors*. 2020. Vol. 20, no. 4. P. 1176. URL: https://doi.org/10.3390/s20041176 (date of access: 05.12.2025).

6. A resource utilization prediction model for cloud data centers using evolutionary algorithms and machine learning techniques / S. Malik et al. *Applied sciences*. 2022. Vol. 12, no. 4. P. 2160. URL: https://doi.org/10.3390/app12042160 (date of access: 04.12.2025).

7. Analysis of the application efficiency of tensorflow and pytorch in convolutional neural network / O.-C. Novac et al. *Sensors*. 2022. Vol. 22, no. 22. P. 8872. URL: https://doi.org/10.3390/s22228872 (date of access: 03.12.2025).

8. Reinforcement learning approach for optimizing cloud resource utilization with load balancing / P. V. Lahande et al. *IEEE access*. 2023. P. 1. URL: https://doi.org/10.1109/access.2023.3329557 (date of access: 04.12.2025).

9. Shaw R., Howley E., Barrett E. Applying Reinforcement Learning towards automating energy efficient virtual machine consolidation in cloud data centers. *Information systems*. 2021. P. 101722. URL: https://doi.org/10.1016/j.is.2021.101722 (date of access: 04.12.2025).

10. Boutheina D., Nadjib A., Rami L. Q-Learning algorithm for joint computation offloading and resource allocation in edge cloud. *Proceedings of the 2019 IFIP/IEEE symposium on integrated network and service management (IM)*, Arlington, VA, 20 May 2019 , pp. 45–52.

11. Leveraging reinforcement learning for task resource allocation in scientific workflows / J. Bader et al. *2022 IEEE International Conference on Big Data (Big Data)*, Osaka, Japan, 2022, pp. 3714–3719. URL: https://doi.org/10.1109/bigdata55660.2022.10020688 (date of access: 05.12.2025).

12. Reinforcement learning for AI as a service: CPU-GPU task scheduling for preprocessing, training, and inference tasks / Y.-D. Lin et al. *IEEE transactions on network and service management*. vol. 22, no. 4, pp. 3433–3448, Aug. 2025. URL: https://doi.org/10.1109/tnsm.2025.3564480 (date of access: 05.12.2025).

13. Osida H. Research about CPU | PDF | central processing unit | microprocessor. *Scribd*. URL: https://www.scribd.com/document/699470005/Research-about-CPU (date of access: 22.11.2025).

14. Structural analysis of multi-core processor and reliability evaluation model / S. Tsiramua et al. *Mathematics*. 2025. Vol. 13, no. 3. P. 515. URL: https://doi.org/10.3390/math13030515 (date of access: 22.11.2025).

15. Oluwatosin Abayomi A., Abayomi Olukayode A., Oluwole Olakunle G. An overview of cache memory in memory management. *Automation, control and intelligent systems*. 2020. Vol. 8, no. 3. P. 24. URL: https://doi.org/10.11648/j.acis.20200803.11 (date of access: 05.12.2025).

16. Contributors to Wikimedia projects. CPU cache – Wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/CPU_cache (date of access: 7.12.2025).

17. SPEC CPU2000: measuring CPU performance in the New Millennium. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/869367/ (date of access: 20.11.2025).

18. CPU design simplified. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/8566475/ (date of access: 20.11.2025).

19. Yuan Tian, Chuang Lin, Kangqiao Hu. The performance model of Hyper-Threading Technology in Intel Nehalem microarchitecture. *2010 3rd international conference on advanced computer theory and engineering (ICACTE 2010)*, Chengdu, China, 20–22 August 2010. 2010. URL: https://doi.org/10.1109/icacte.2010.5579564 (date of access: 04.12.2025).

20. Lee E. A. The problem with threads. *Computer*. 2006. Vol. 39, no. 5. P. 33–42. URL: https://doi.org/10.1109/mc.2006.180 (date of access: 20.11.2025).

21. Processes and threads - win32 apps. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads (date of access: 20.11.2025).

22. Performance measurement of processes and threads controlling, tracking and monitoring based on shared-memory parallel processing approach / K. H. Sharif et al. *2020 3rd international conference on engineering technology and its applications (IICETA)*, Najaf, Iraq, 6–7 September 2020. 2020. URL: https://doi.org/10.1109/iiceta50496.2020.9318800 (date of access: 20.11.2025).

23. How to read java: understanding, debugging, and optimizing JVM applications. Manning Publications Co. LLC, 2022.

24. Byeon H. Advances in value-based, policy-based, and deep learning-based reinforcement learning. *International journal of advanced computer science and applications*. 2023. Vol. 14, no. 8. URL: https://doi.org/10.14569/ijacsa.2023.0140838 (date of access: 05.12.2025).

25. McKenzie M. C., McDonnell M. D. Modern value based reinforcement learning: a chronological review. *IEEE access*. 2022. P. 1. URL: https://doi.org/10.1109/access.2022.3228647 (date of access: 04.12.2025).

26. Alavizadeh H., Alavizadeh H., Jang-Jaccard J. Deep q-learning based reinforcement learning approach for network intrusion detection. *Computers*. 2022. Vol. 11, no. 3. P. 41. URL: https://doi.org/10.3390/computers11030041 (date of access: 05.12.2025).

27. Sewak M. Temporal Difference Learning, SARSA, and Q-Learning. *Deep reinforcement learning*. Singapore, 2019. P. 51–63. URL: https://doi.org/10.1007/978-981-13-8285-7_4 (date of access: 05.12.2025).

28. Twin trust region policy optimization / H. Xu et al. *IEEE transactions on systems, man, and cybernetics: systems*. 2025. P. 1–15. URL: https://doi.org/10.1109/tsmc.2025.3573513 (date of access: 05.12.2025).

29. Del Rio A., Jimenez D., Serrano J. Comparative analysis of A3C and PPO algorithms in reinforcement learning: a survey on general environments. *IEEE access*. 2024. P. 1. URL: https://doi.org/10.1109/access.2024.3472473 (date of access: 05.12.2025).