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 перший (бакалаврський)

Адаптивний підбір архітектури нейромережі для агентів у симульованих середовищах на основі NAS та RL
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-4
Софія Балаба
(власне ім'я, прізвище)
Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник доц. Марія Головянко
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

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

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Балабі Софії Олександрівні
(прізвище, ім'я, по батькові)

1. Тема роботи Адаптивний підбір архітектури нейромережі для агентів у симульованих середовищах на основі NAS та RL

затверджена наказом університету від 19 травня 20 25 р. № 378Ст

2. Термін подання студентом роботи до екзаменаційної комісії 24 червня 20 25 р.

3. Вихідні дані до роботи аналіз наукових публікацій з теми Neural Architecture Search і підкріплювального навчання, опис структури та реалізації контролера архітектур, специфікація середовищ DM Control Suite, дані експериментів із навчання агентів, результати метрик ефективності, графічні матеріали та інтерпретація результатів

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі та постановка задачі

2) Аналіз середовища

3) Експериментальні дослідження та аналіз результатів

4) Стабільність результатів і узагальнення

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	19.05.2025	виконано
2	Дослідження та аналіз предметної галузі	20.05.2025 – 21.05.2025	виконано
3	Визначення мети та завдань дослідження	22.05.2025 – 24.05.2025	виконано
4	Огляд методологій і програмних рішень	25.05.2025 – 27.05.2025	виконано
5	Збір даних та постановка задачі	28.05.2025 – 30.05.2025	виконано
6	Описання програмної реалізації проекту	02.06.2025 – 03.06.2025	виконано
7	Оформлення пояснювальної записки	04.06.2025 – 05.06.2025	виконано
8	Підготовка до захисту кваліфікаційної роботи	06.06.2025 – 16.06.2025	виконано
9	Захист кваліфікаційної роботи перед ЕК	24.06.2025	виконано

Дата видачі завдання 19 травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Марія Головянко
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 66 с., 9 рис., 1 дод., 20 джерел.

АВТОМАТИЗАЦІЯ ПІДБОРУ ПАРАМЕТРІВ, АДАПТИВНЕ КОНСТРУЮВАННЯ НЕЙРОМЕРЕЖ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, ПОШУК АРХІТЕКТУРИ НЕЙРОННОЇ МЕРЕЖІ, СИМУЛЬОВАНІ СЕРЕДОВИЩА DM CONTROL.

Об'єкт дослідження: процеси навчання агентів у симульованих середовищах з використанням поєднання методів навчання з підкріпленням та автоматичного формування архітектури нейронної мережі.

Предмет дослідження: адаптивне поєднання нейроархітектурного пошуку та навчання з підкріпленням, що дозволяє агенту самостійно оптимізувати власну нейронну мережу з меншими обчислювальними витратами завдяки поетапній процедурі відбору архітектур.

Мета роботи: розробка системи, яка автоматично формує оптимальну архітектуру нейромережі для RL-агента.

Методи: формування початкового набору архітектур через NAS-агента, короткий цикл навчання PPO-агентів у середовищі Cheetah, відбір найкращих архітектур за середньою винагородою, тривале тренування обраних варіантів для остаточного ранжування.

Результати: створено систему, що забезпечує автоматизований вибір конфігурації нейромережі для навчання агента, мінімізуючи участь людини в ручному налаштуванні. Рішення демонструє високу ефективність та може бути розширене на інші типи симульованих середовищ або задач у робототехніці, автономних системах і машинному сприйнятті.

ABSTRACT

Bachelor’s thesis contains: 66 pp., 9 fig., 1 ann., 20 references.

ADAPTIVE NEURAL NETWORK DESIGN, AUTOMATED PARAMETER SELECTION, NEURAL ARCHITECTURE SEARCH, REINFORCEMENT LEARNING, SIMULATED DM CONTROL ENVIRONMENTS.

Object of research: processes of agent learning in simulated environments using a combination of reinforcement learning methods and automatic neural network architecture formation.

Subject of research: adaptive combination of neural architecture search and reinforcement learning, allowing the agent to independently optimize its own neural network with reduced computational costs through a phased architecture selection procedure.

Purpose of the work: development of a system that automatically forms the optimal neural network architecture for an RL agent.

Methods: formation of an initial set of architectures through a NAS agent, short training cycle of PPO agents in the Cheetah environment, selection of the best architectures based on average reward, extended training of selected variants for final ranking.

Results: a system was created that provides automated selection of neural network configuration for agent training, minimizing human involvement in manual tuning. The solution demonstrates high efficiency and can be extended to other types of simulated environments or tasks in robotics, autonomous systems, and machine perception.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 8
Вступ 9
1 Аналіз предметної галузі та постановка задачі 11
1.1 Огляд предметної галузі 11
1.2 Існуючі аналоги 14
1.3 Організація даних 16
1.3.1 Архітектури нейронних мереж 16
1.3.2 Дані навчання та валідації 17
1.3.3 Метадані та результати 19
1.3.4 Організація процесу 20
1.4 Вибір алгоритмів підкріплення для NAS 21
1.5 Постановка задачі 22
2 Аналіз середовища 24
2.1 Методи та підходи до розв’язання задач 24
2.2 Формування простору архітектур 28
2.3 Reward-система та механізми навчання агента 33
2.4 Побудова середовища взаємодії 35
3 Експериментальні дослідження та аналіз результатів 39
3.1 Архітектура контролера LSTM для генерації параметрів агента 39
3.1.1 Опис моделі LSTM 40
3.1.2 Структура контролера 41
3.1.3 Особливості реалізації 44
3.2 Інтеграція контролера з навчальним середовищем 46
3.2.1 Формування архітектури агентів 46
3.2.2 Побудова агента у середовищі 47
3.2.3 Робота з векторним середовищем 48
3.2.4 Інтеграція контролера та середовища як замкнений цикл 48
3.3 Процес навчання контролера 48

3.4 Проблеми та виклики реалізації 50
3.5 Аналіз продуктивності реалізованої системи 52
3.6 Вплив гіперпараметрів контролера на якість вибору 54
4 Стабільність результатів і узагальнення 57
Висновки 61
Перелік джерел посилання 63
Додаток А Відомість кваліфікаційної роботи 66

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

JSON – JavaScript Object Notation – текстовий формат обміну даних між комп’ютерами;

NAS – Neural Architecture Search – автоматизований процес пошуку оптимальної архітектури нейронної мережі для конкретного завдання за допомогою алгоритмів оптимізації;

PPO – Proximal Policy Optimization – алгоритм підкріплювального навчання, що поєднує стабільність та ефективність у процесі оновлення політики агента;

RL – Reinforcement Learning – підхід у машинному навчанні, де агент навчається приймати рішення, взаємодіючи з середовищем та отримуючи винагороду за свої дії.

ВСТУП

Розвиток інтелектуальних агентів, здатних самостійно приймати рішення та адаптуватися до змін середовища, є одним із головних напрямів сучасних досліджень у сфері штучного інтелекту. Особливої актуальності набуває задача автоматичного проєктування архітектур нейронних мереж, які використовуються цими агентами. Поєднання методів навчання з підкріпленням (RL) та нейроархітектурного пошуку (NAS) відкриває нові можливості для створення ефективних моделей без необхідності ручного налаштування.

У рамках цього дослідження розглядається використання симульованих середовищ DM Control Suite для тестування здатності агента самостійно обирати архітектуру власної нейромережі. Зокрема, обрана симуляція Cheetah виступає як базове середовище, в якому агент навчається рухатися з максимальною швидкістю, одночасно експериментуючи з конфігурацією своєї моделі. На кожному етапі агент NAS пропонує набір архітектурних параметрів, таких як кількість шарів чи розмір прихованих нейронів, після чого формується PPO-агент з цією архітектурою, який виконує навчання протягом фіксованої кількості кроків.

Зважаючи на складність та об’ємність простору можливих архітектур, ручний аналіз їх ефективності стає практично неможливим. Саме тому необхідне створення автоматизованого підходу, що дозволяє агенту самостійно вивчати, які архітектури приносять кращі результати. У якості метрики ефективності використовується середній reward, отриманий агентом у симуляції. Ця інформація слугує сигналом зворотного зв’язку, що дозволяє NAS-агенту поступово вдосконалювати процес вибору архітектур на основі накопиченого досвіду.

У результаті реалізується система, здатна адаптувати структуру нейромережі агента до особливостей середовища без людського втручання. Такий підхід дає змогу значно скоротити витрати часу на експерименти,

підвищити якість навчання та забезпечити гнучкість моделі до нових задач і симуляцій. Розроблений підхід може бути адаптований для:

– робототехнічних платформ, де агенти повинні оперативно реагувати на зміну середовища (наприклад, автономна навігація, маніпулятори, дрони);

– систем автономного управління, які потребують навчання в умовах часткової спостережуваності або з високим ступенем невизначеності;

– інженерних та промислових застосувань, де оптимальні архітектури дозволяють зменшити затрати на обчислення та підвищити надійність систем управління;

– наукових досліджень у галузі штучного інтелекту, де необхідна масштабована та відтворювана система тестування ефективності NAS у поєднанні з RL.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд предметної галузі

Автоматизація проєктування архітектур нейронних мереж є одним з ключових напрямів розвитку штучного інтелекту. У сучасних умовах стрімкого зростання складності моделей та обсягів даних ручне проєктування архітектур стає все менш ефективним. У відповідь на це виникла концепція NAS – підходу, що дозволяє автоматично шукати оптимальні архітектури для конкретних задач (рисунок 1.1).

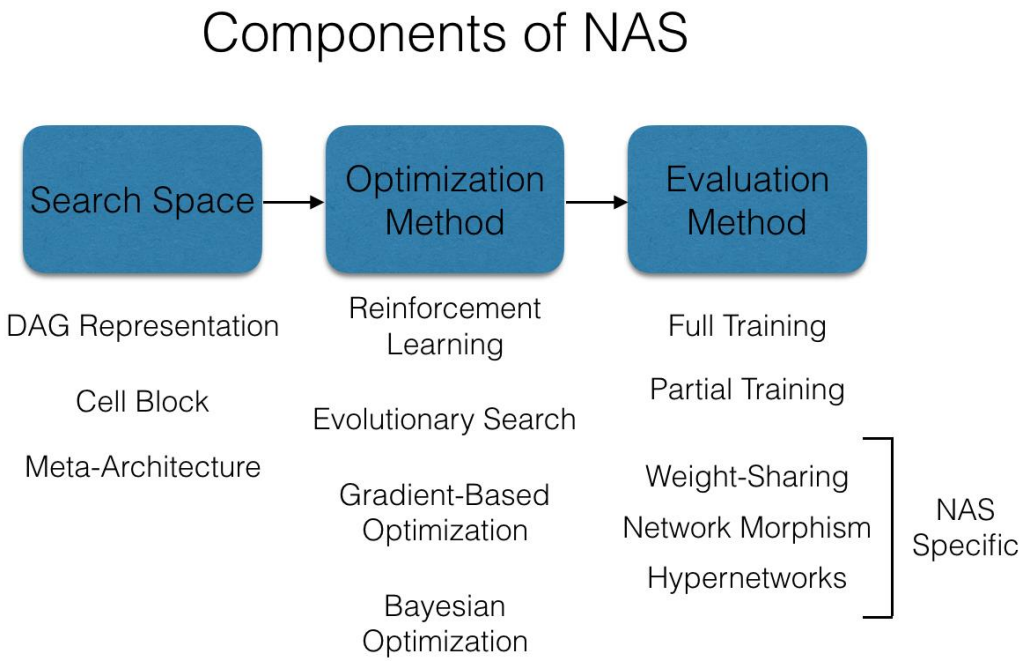

Рисунок 1.1 – Компоненти NAS

Метою NAS є формування моделі, яка заздалегідь не визначена вручну, а побудована алгоритмічно шляхом дослідження простору архітектур. Такий підхід дозволяє знайти нетривіальні рішення, які іноді перевершують за якістю навіть проєкти, створені експертами. Проте одночасно NAS створює нові виклики: вартість пошуку часто є дуже

високою, особливо якщо кожна архітектура вимагає повного навчання для оцінки її якості. Існує кілька класів NAS-підходів:

– еволюційні алгоритми: використовують механізми природного відбору для поступового поліпшення архітектур;

– методи з підкріпленням (RL): агент RL навчається генерувати архітектури, отримуючи зворотний зв’язок у вигляді метрики якості;

– градієнтні методи (Differentiable NAS): формують безперервний простір архітектур, що дозволяє застосовувати градієнтний спуск;

– методи на основі мета-навчання та байєсівської оптимізації.

Серед них RL-NAS виділяється гнучкістю і можливістю адаптуватися до нових завдань. RL-агент виконує роль контролера, який генерує послідовність архітектурних рішень (наприклад, тип шару, кількість нейронів, розмір ядра тощо), отримує винагороду (наприклад, точність моделі) і поступово вдосконалює свої дії (рисунок 1.2).

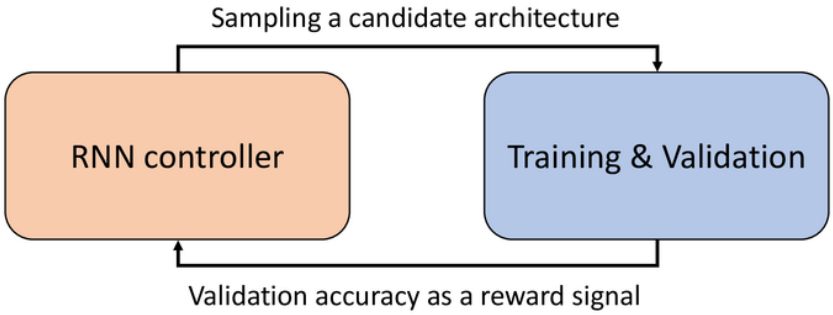

Рисунок 1.2 – Алгоритм RL-NAS

Такий підхід демонструє гарні результати, але потребує значних обчислювальних ресурсів, адже кожна нова архітектура повинна пройти етап тренування хоча б на частині даних. Це породжує ряд викликів:

– необхідність багаторазового тренування моделей для оцінки: для того, щоб оцінити якість кожної згенерованої архітектури, її потрібно навчити – іноді повністю, іноді частково. Оскільки NAS генерує багато варіантів архітектур, це означає, що тренування доводиться повторювати

багато разів. Такий підхід потребує значного часу і обчислювальних ресурсів, особливо коли кількість кандидатів зростає. Це ускладнює експерименти і робить процес пошуку тривалим і ресурсомістким;

– обмеженість у використанні NAS у практичних умовах: через високі вимоги до обчислювальної потужності та тривалий час пошуку NAS часто є недоступним для застосування в реальних проєктах із обмеженими ресурсами. Малопотужне обладнання, бюджетні обмеження або потреба у швидких рішеннях роблять повноцінний NAS неприйнятним для багатьох задач, що обмежує його широке впровадження поза лабораторіями та дослідницькими центрами;

– висока вартість експериментів при великих просторах архітектур: зі збільшенням розміру та складності простору архітектур, кількість потенційних кандидатів зростає експоненційно. Це призводить до значного збільшення часу і ресурсів, необхідних для проведення пошуку. В результаті повний перебір або навіть адекватний випадковий пошук стає надто дорогим або неможливим без застосування спеціальних оптимізацій, що ускладнює ефективне використання NAS у великих масштабах.

З метою подолання основних проблем, пов’язаних із високою обчислювальною вартістю та тривалістю процесу пошуку архітектур, дослідники пропонують низку оптимізованих стратегій, які значно підвищують ефективність NAS-процесів. Одним із ключових напрямків є скорочення кількості повноцінних тренувань моделей. Це досягається завдяки застосуванню методів попереднього відбору архітектур, що базуються на частковому або скороченому навчанні (short training). Такі методи дозволяють оцінювати потенціал кандидатів без необхідності довготривалого повного навчання, тим самим суттєво економлячи час та обчислювальні ресурси.

Додатково, для підвищення стабільності та прискорення процесу генерації архітектур, використовують фіксовану або попередньо натреновану політику агента RL. Замість того, щоб починати навчання з

нуля, агент отримує базове уявлення про простір архітектур, що дозволяє йому більш цілеспрямовано генерувати перспективні варіанти з самого початку. Це не лише скорочує час пошуку, а й зменшує ризик застрягання у локальних оптимумах.

Ще одним ефективним підходом є впровадження багаторівневої системи відбору, яка послідовно відсіює неефективні або низькоякісні архітектури вже на ранніх етапах оцінки. Така ієрархічна схема дозволяє зосередити обчислювальні ресурси на найбільш перспективних кандидатах, знижуючи загальне навантаження на систему. Наприклад, на першому рівні можуть оцінюватися дуже швидкі евристики або короткі тренування, а на наступних – більш глибокі та точні процедури навчання і перевірки.

Всі ці стратегії спрямовані на досягнення балансу між якістю пошуку архітектур та ефективністю використання ресурсів, що є критично важливим для практичного застосування NAS. Запропоновані підходи дозволяють адаптувати процес пошуку до обмежень конкретного обладнання і задачі, зберігаючи при цьому здатність знаходити високоякісні рішення. Саме цей аспект став ключовою мотивацією для розробки та реалізації системи, що розглядається у межах цього дослідження, і він відображає сучасний напрямок розвитку NAS-технологій, орієнтований на практичність та масштабованість.

1.2 Існуючі аналоги

Пошук архітектур нейронних мереж з використанням методів навчання з підкріпленням уже отримав чималу увагу в науковому середовищі. Було запропоновано низку підходів, у яких агент RL навчається будувати архітектури нейронних мереж, орієнтуючись на метрику якості – зазвичай точність класифікації після навчання моделі.

Одним із перших значних прикладів була система NASNet, у якій контролер на основі рекурентної нейронної мережі генерував опис

архітектури, а винагорода формувалася за результатами навчання моделі на наборі даних. Проте реалізація такого підходу потребувала величезних обчислювальних ресурсів: пошук найкращої моделі займав тисячі GPU-годин.

Згодом з'явилися модифікації, орієнтовані на зменшення вартості:

– ENAS (Efficient NAS): запропоновано спільне використання ваг для різних архітектур, що значно скорочує час на тренування;

– ProxylessNAS: використовує апроксимації та часткове навчання для пришвидшення оцінки;

– DARTS: реалізує диференційовний підхід, що дозволяє уникнути дискретного пошуку;

– AlphaX: застосовує MCTS (Monte Carlo Tree Search) у поєднанні з RL для більш ефективної навігації у просторі архітектур.

У сфері RL-NAS існує також лінія експериментів, де RL використовується для керування багаторівневим процесом пошуку. Однак більшість таких підходів мають спільну обмежувальну рису: агент RL навчається «з нуля» для кожної нової задачі або нового набору даних, що унеможливлює повторне використання політики, а отже – призводить до значних витрат часу й ресурсів.

Ще однією критичною перешкодою в RL-NAS є повільність зворотного зв'язку, що виникає через потребу хоча б частково навчати кожну згенеровану архітектуру перед оцінюванням її якості. Це істотно сповільнює весь цикл пошуку і вимагає значних обчислювальних ресурсів.

Для пом'якшення цієї проблеми дослідники пропонують різноманітні компромісні стратегії:

– використання попередньо натренованих агентів, що дозволяє скоротити кількість навчальних епізодів, але обмежує гнучкість агента;

– впровадження багаторівневого фільтрування, при якому спочатку проводиться коротке попереднє навчання великої кількості архітектур, а згодом більш глибоке навчання обраних фіналістів [8];

– застосування мета-ознак або моделей-замінників (surrogate models) для попереднього прогнозування ефективності архітектур без їх прямого тренування.

У межах даного дослідження було реалізовано саме багаторівневу стратегію оцінки, яка поєднує NAS на основі RL із поетапним відбором архітектур. Система виконує генерацію великої кількості варіантів, які проходять коротке тренування з використанням методу PPO, після чого кращі з них відбираються для подальшого, більш глибокого навчання. Такий підхід дозволяє уникнути повторного навчання агента, зберегти гнучкість пошуку та суттєво скоротити час до отримання якісної фінальної моделі.

1.3 Організація даних

У рамках задачі автоматизованого пошуку нейронних архітектур із використанням методів підкріплювального навчання (RL-NAS) особливо важливо ретельно продумати структуру і організацію даних, що використовуються на різних етапах процесу. Від правильного розподілу і типів даних залежить ефективність генерації архітектур, оцінка їх якості та навчання контролера, а також загальна продуктивність системи.

Дані у цій системі умовно поділяються на чотири основні категорії, кожна з яких виконує свою специфічну функцію в загальному циклі NAS.

1.3.1 Архітектури нейронних мереж

Центральним об’єктом аналізу у задачі автоматизованого пошуку нейронних архітектур є конфігурації мереж, згенеровані агентом підкріплювального навчання. Кожна така конфігурація формується як послідовність рішень, які агент приймає на основі свого політики – вибираючи, наприклад, тип шару, кількість нейронів або каналів, параметри

згортки, а також функції активації. Ці дії поступово формують повну структуру мережі, яку можна інтерпретувати як опис архітектури.

Для подальшої обробки та тренування згенеровані конфігурації зберігаються у вигляді спеціалізованих структурованих даних. Найчастіше це або графові представлення, де кожен вузол відповідає певному шару мережі, а ребра відображають зв'язки між ними, або послідовності токенів, де кожен токен кодує конкретний елемент архітектури. Графова форма дозволяє відображати складні топології, включаючи паралельні шляхи або резидуальні з'єднання, тоді як послідовності краще підходять для алгоритмів, що працюють з послідовними даними, наприклад, рекурентних моделей.

Після генерації конфігурації декодуються у конкретні моделі, які реалізуються у фреймворках глибокого навчання, таких як PyTorch або TensorFlow. Це дає змогу почати їх тренування і оцінку продуктивності. Важливою особливістю є те, що збереження архітектур у структурованому та стандартизованому форматі сприяє швидкому та надійному відтворенню експериментів, а також полегшує аналіз результатів і подальший розвиток системи.

1.3.2 Дані навчання та валідації

Для оцінки ефективності згенерованих агентом конфігурацій нейронних мереж необхідно використовувати відповідні набори даних або симуляційні середовища, які відповідають конкретній задачі. У більшості випадків це стандартизовані датасети або відомі середовища підкріплювального навчання, що дозволяють отримати об'єктивну і порівнянну оцінку продуктивності кожної моделі.

У випадку задачі класифікації зображень найбільш поширеними є набори типу MNIST, CIFAR-10 чи ImageNet. Вони широко використовуються не лише через свою доступність, але й завдяки добре

вивченій структурі, що дає змогу чітко оцінити, наскільки згенерована архітектура підходить для конкретного класу завдань. Ці датасети забезпечують різноманітність та складність, необхідні для тестування здатності моделей до узагальнення та виявлення корисних особливостей візуальних даних.

Для задач, що належать до області підкріплювального навчання, застосовуються симуляційні середовища, такі як CartPole, LunarLander, MuJoCo чи BipedalWalker. Вони надають контролеру або агенту платформу для взаємодії з динамічним середовищем, де ефективність архітектури вимірюється в термінах сумарної винагороди за виконання певного завдання. Використання таких середовищ, часто реалізованих у OpenAI Gym чи DM Control Suite, дає змогу стандартизувати умови тестування і порівнювати результати різних підходів у єдиному контексті.

Оскільки повне навчання кожної архітектури є обчислювально дорогим, у практиці пошуку нейронних архітектур часто застосовується поетапний підхід. На першому етапі проводиться скорочене або часткове навчання моделей – це дозволяє швидко отримати первинну оцінку якості архітектури без значних витрат часу та ресурсів. Такий попередній відбір дає змогу відсіяти явно слабкі конфігурації і сфокусувати обчислювальні зусилля на більш перспективних варіантах.

Лише після цього відбувається повне навчання найкращих архітектур із розширеним числом ітерацій або епох. Це необхідно для того, щоб об’єктивно оцінити їхню кінцеву продуктивність та узагальнювальну здатність на тестових даних або в робочому середовищі. Такий підхід забезпечує баланс між якістю результату і ресурсною ефективністю, що є критичним у задачах NAS, де кількість потенційних моделей надзвичайно велика. Такий фінальний етап дозволяє переконатися, що обрана архітектура не лише ефективна під час відбору, але й зберігає стабільну продуктивність при масштабованому використанні.

1.3.3 Метадані та результати

Під час кожного циклу навчання згенерованих нейронних архітектур у системі автоматичного пошуку особлива увага приділяється збору і збереженню метаданих, що характеризують продуктивність та ресурсоємність моделей. Ці метадані є критично важливими для подальшого аналізу, ранжування та відбору найбільш перспективних архітектур.

Основними метриками, які фіксуються під час тренування, є середня винагорода (reward) – вона відображає загальну якість роботи агента або продуктивність моделі у задачах підкріплення або класифікації. Для більш глибокого розуміння стабільності навчання відслідковується також відхилення нагород по епізодах, що дозволяє виявити моделі, які демонструють непередбачувану поведінку або значні флуктуації у результатах. Така інформація допомагає не лише оцінити якість архітектури, а й її надійність у різних умовах.

Ще одним важливим параметром є кількість параметрів моделі, що визначає її складність і потенційний розмір. Ця метрика має значення з точки зору ефективності – менш складні моделі часто краще підходять для розгортання на обмежених апаратних ресурсах, навіть якщо вони трохи поступаються у максимальній продуктивності. Тому баланс між розміром моделі і якістю роботи є одним із ключових критеріїв у багаторівневій системі відбору.

Також фіксується час тренування, що дає уявлення про обчислювальні витрати на отримання певного рівня якості. Цей параметр особливо важливий для практичних застосувань NAS, оскільки високі часові витрати можуть суттєво обмежити можливість пошуку в великих просторах архітектур.

Крім того, в системі збираються і додаткові обчислювальні характеристики, такі як кількість операцій з плаваючою точкою (FLOPs),

затримка (latency), споживання пам’яті і енергетична ефективність. Ці показники особливо актуальні при розгортанні моделей у мобільних або вбудованих пристроях, де ресурси обмежені і оптимізація обчислень є пріоритетом.

Всі ці метадані формують основу для багаторівневої системи відбору архітектур, яка дозволяє не лише сортувати моделі за абсолютною продуктивністю, а й враховувати їхню ефективність і відповідність вимогам конкретного застосування. Такий підхід забезпечує більш збалансований вибір оптимальних конфігурацій, що відповідають і технічним, і прикладним критеріям, що вкрай важливо в контексті реальних проєктів.

Збереження та систематизація метаданих також полегшує повторний аналіз, порівняння результатів між різними експериментами, а також підвищує прозорість і відтворюваність досліджень, що є однією з ключових вимог сучасної наукової практики.

1.3.4 Організація процесу

Для ефективного управління потоками даних у процесі автоматизованого пошуку нейронних архітектур створена централізована система зберігання та обробки інформації. Вона забезпечує цілісність і структурованість усіх даних, що виникають на різних етапах – від генерації архітектур до їх оцінки та відбору. Як формати зберігання можуть використовуватися легкі та гнучкі текстові формати, такі як JSON або CSV, а для більш складних випадків – реляційні чи нереляційні бази даних, що дозволяють масштабувати систему і забезпечувати швидкий доступ до необхідної інформації.

Централізоване сховище є сполучною ланкою між ключовими модулями системи. Модуль генерації архітектур, який реалізує RL-агент, передає сформовані послідовності рішень у сховище, фіксуючи при цьому

всі параметри та контекст їх створення. Це дозволяє відслідковувати, як формувалась кожна конфігурація, та забезпечує прозорість процесу пошуку.

Модуль тренування, який зазвичай реалізований на основі популярних фреймворків, таких як PPO або бібліотеки Stable-Baselines, отримує архітектури зі сховища для запуску навчання у стандартизованому середовищі. Результати тренування, включно з усіма метриками, повертаються назад у систему зберігання, що дозволяє акумулювати повний облік прогресу кожної моделі.

Додатково, спеціальний модуль збирання метрик агрегує інформацію про якість, стабільність, ресурсоємність і інші важливі характеристики, формуючи структуровані дані, які будуть використовуватись для подальшого аналізу. Така централізація дає змогу легко порівнювати різні архітектури за різними параметрами в одному місці.

На основі зібраних даних працює модуль фільтрації та вибору найкращих варіантів. Він реалізує багаторівневу систему ранжування, враховуючи не тільки абсолютні показники продуктивності, а й ефективність використання ресурсів, стабільність роботи та інші критерії. Завдяки цьому до подальшої фази тренування та тестування потрапляють лише перспективні архітектури, що значно оптимізує загальний процес пошуку.

1.4 Вибір алгоритмів підкріплення для NAS

Підхід нейроеволюції з використанням підкріплення (RL-NAS) добре масштабується до задач генерації архітектур, оскільки дозволяє агенту (контролеру) поетапно вивчати простір можливих рішень, отримуючи відгук у вигляді нагороди за якість згенерованої моделі. З огляду на дискретну природу пошуку та відкладену винагороду, найчастіше використовуються політичні градієнтні методи, зокрема REINFORCE або його покращення, такі як PPO (Proximal Policy Optimization). Вони добре

підходять для стабільного оновлення параметрів контролера в умовах високої стохастичності нагород і нестаціонарного середовища.

Інші потенційні варіанти включають Actor-Critic схеми або A2C/A3C, які дозволяють ефективно враховувати як оцінку політики, так і цінності станів. Однак для NAS особливо критичним є баланс між стабільністю та варіативністю генерації – тому перевагу часто надають методам із м’яким обмеженням оновлення (як у PPO) і можливістю регулювання ентропії.

Таким чином, вибір алгоритму RL у контексті NAS базується на здатності справлятися з високою складністю простору дій, нестабільністю оцінки нагороди та необхідністю збереження диверсифікації архітектур під час навчання.

1.5 Постановка задачі

З огляду на вищезазначене, основною метою даного дослідження є розробка системи, яка забезпечить автоматизований вибір архітектури нейронної мережі агента в середовищі з підкріплювальним навчанням. Система повинна дозволити динамічно генерувати нові архітектури, оцінювати їх ефективність на основі взаємодії з симуляцією та на цій основі покращувати якість обраних конфігурацій. Ключові задачі, які вирішуються в межах дослідження:

– автоматизоване створення архітектур (кількість шарів, нейронів тощо) нейронних мереж для агента RL;

– запуск навчання агента у середовищі з обраною архітектурою;

– збір та збереження результатів навчання для кожної архітектури;

– аналіз зібраних даних з метою виявлення залежностей між структурою мережі та її ефективністю;

– реалізація логіки NAS-агента, який, використовуючи отримані результати, покращує процес вибору наступної архітектури;

– оцінка продуктивності системи порівняно з фіксованими або ручними конфігураціями.

У результаті реалізовано систему, що поєднує переваги NAS та RL, водночас уникаючи їхніх типових недоліків, таких як надмірні витрати на перенавчання агента чи залежність від ручного відбору моделей. Запропонована архітектура забезпечує баланс між продуктивністю й ефективністю, відкриваючи перспективи її використання у більш складних середовищах або для вирішення задач прикладного рівня.

Реалізація такої системи дозволить зменшити людський фактор у процесі проектування моделей для задач підкріплювального навчання та підвищити якість агентів, особливо в умовах змінного або складного середовища. У перспективі такий підхід може бути масштабований на реальні робототехнічні системи, адаптивне керування або автономні платформи.

2 АНАЛІЗ СЕРЕДОВИЩА

2.1 Методи та підходи до розв’язання задач

Вибір середовища для проведення експериментів у підкріплювальному навчанні має ключове значення, оскільки від характеристик середовища залежать складність задачі, можливості агента та адекватність отриманих результатів. На сьогоднішній день одним із найпопулярніших фреймворків для розробки і тестування алгоритмів reinforcement learning є OpenAI Gym – відкритий набір середовищ, який включає широкий спектр класичних задач із різним рівнем складності. Gym забезпечує легкий доступ до різноманітних середовищ, від простих ігрових до більш складних фізичних симуляцій, що робить його універсальним інструментом для дослідників.

Проте, хоча OpenAI Gym є зручною та поширеною платформою, його базові середовища, як правило, мають обмежену фізичну достовірність і спрощену динаміку. Це накладає певні обмеження на можливості тестування алгоритмів у складних умовах, близьких до реальних. Для більш глибокого і реалістичного моделювання рухів роботизованих агентів і складних фізичних процесів у дослідницькій спільноті дедалі частіше застосовується DM Control Suite – набір середовищ, розроблений компанією DeepMind.

DM Control Suite відзначається високою фізичною достовірністю, складною багатовимірною динамікою, а також широким спектром завдань із чітко сформульованими цілями. На відміну від Gym, середовища DM Control Suite пропонують більш деталізоване моделювання, що дозволяє більш адекватно оцінити поведінку агентів у сценаріях, близьких до реального світу.

Для реалізації експериментів у цій роботі було обрано середовище Cheetah з пакету DM Control Suite (рисунок 2.1). Воно є одним із

найпопулярніших і найбільш широко використовуваних середовищ у сучасних дослідженнях з підкріплювального навчання завдяки поєднанню складної динаміки руху, багатовимірного простору станів і дій, а також чітко визначеної мети для агента. Саме такі характеристики роблять Cheetah ідеальним тестовим полігоном для дослідження ефективності автоматичного підбору архітектури нейронної мережі агента.

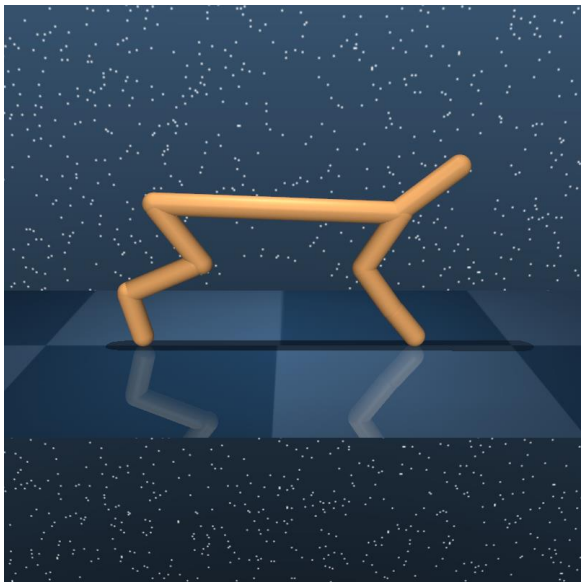

Рисунок 2.1 – Візуалізація симульованого агента Cheetah

Cheetah моделює біг двоногого механічного «гепарда», який повинен навчитися пересуватися вперед із максимальною швидкістю, зберігаючи баланс і не падаючи. Завдання агента – за допомогою керування суглобами тулуба, стегон і ніг, досягти стабільного та швидкого руху вперед. Середовище створене таким чином, щоб нагороджувати агента за прогрес вздовж осі X та карати за неефективні чи небезпечні рухи, що можуть призвести до втрати рівноваги.

Стан агента у середовищі Cheetah описується числовим вектором, який включає такі ознаки:

– позиції та швидкості всіх основних сегментів тіла (наприклад, тулуба, стегон, гомілок);

– кутові положення та швидкості суглобів;

– інформацію про контакт із землею;

– допоміжні ознаки, що характеризують поточний стан механізму та його динаміку.

Зазвичай розмірність простору спостережень для Cheetah складає від 17 до 24 чисел (залежно від конкретної реалізації), що дозволяє детально описати положення і рух усіх частин моделі.

Дії агента – це набір безперервних сигналів (діапазон [-1, 1]), які відповідають моментам сили, що прикладаються до суглобів ніг та тулуба. Для Cheetah зазвичай використовується від 6 до 8 керованих ступенів свободи, тобто агент на кожному кроці приймає рішення про величину зусилля для кожного суглоба окремо.

Винагорода у середовищі Cheetah складається з кількох компонентів:

– основна частина: це приріст координати X (тобто швидкість руху вперед);

– додаткові штрафи можуть нараховуватися за надмірно великі або різкі дії, що імітують енергетичні витрати або ризик пошкодження;

– у випадку падіння агент може отримати значний штраф або завершити епізод достроково.

Винагорода нараховується на кожному кроці симуляції, що стимулює агента досягати максимальної швидкості руху при мінімальних витратах енергії та збереженні стійкості.

Під час кожного епізоду взаємодії агента з середовищем збирається послідовність даних у вигляді пар «спостереження – дія – винагорода». Ці дані використовуються для навчання політики агента за допомогою алгоритмів підкріплювального навчання (наприклад, PPO), а також для оцінки ефективності різних архітектур нейронної мережі.

Оскільки обрана для реалізації RL-алгоритмів бібліотека не передбачає безпосередньої підтримки середовищ із пакету DM Control Suite, для забезпечення коректної інтеграції середовища Cheetah було розроблено

спеціальний інтерфейс-обгортку (wrapper). Ця обгортка вирішує низку технічних задач, зокрема:

– уніфікацію формату спостережень шляхом перетворення словникової структури ознак у вектор, сумісний з нейронною мережею агента;

– приведення простору дій до стандартного діапазону, прийнятного для використаної реалізації алгоритмів;

– обробку завершення епізодів у випадках, коли агент падає або досягає максимальної довжини епізоду;

– збирання додаткової інформації, необхідної для подальшого аналізу, такої як кількість кроків, сумарна винагорода або динаміка зміни станів.

Таким чином, реалізація обгортки забезпечила можливість поєднати функціональність DM Control Suite із практичними вимогами до структури середовищ у використовуваній RL-бібліотеці.

Усі дані про архітектуру мережі, параметри навчання, отримані винагороди та інші метрики фіксуються у структурованих списках або таблицях для подальшого аналізу та порівняння. Це дозволяє оцінити, які саме архітектурні рішення (кількість шарів, нейронів, тип активації) призводять до кращих результатів саме у середовищі Cheetah.

Середовище Cheetah є класичним тестовим полігоном для задач підкріплювального навчання, оскільки воно вимагає від агента не лише оптимальної політики, а й здатності нейромережі відображати складні залежності між станом і дією. Успішне навчання у цьому середовищі свідчить про здатність агента адаптуватися до високої динамічності, непередбачуваності та багатовимірності задачі.

Завдяки відкритості та стандартизованості DM Control Suite, результати, отримані у середовищі Cheetah, легко порівнювати з іншими роботами у галузі, що робить це середовище ідеальним для тестування нових підходів до автоматичного підбору архітектури нейронної мережі. Крім того, така симуляція дозволяє проводити експерименти без ризику для

реального обладнання, швидко змінювати параметри та масштабувати дослідження.

Таким чином, середовище DM Control Suite: Cheetah забезпечує складну, реалістичну та гнучку платформу для досліджень у сфері підкріплювального навчання. Його структура, організація даних та чітко визначені критерії успіху дозволяють якісно аналізувати вплив різних архітектур нейронної мережі на результативність агентів, що є ключовим для розробки автоматизованих систем підбору архітектури у задачах контролю та робототехніки.

2.2 Формування простору архітектур

Першим етапом при розв’язанні задачі автоматизованого проектування нейронних мереж для задач підкріплювального навчання є формалізація простору пошуку архітектур. Простір архітектур визначає набір можливих конфігурацій моделі, які система має змогу генерувати, оцінювати та використовувати під час навчання. Надто вузький простір може обмежити ефективність моделей, тоді як надто широкий значно збільшити обчислювальні витрати. У цій роботі застосовується збалансований варіант: компактний, але достатньо гнучкий простір, що дозволяє знаходити ефективні архітектури в реалістичних умовах.

Простір визначається трьома ключовими параметрами, які безпосередньо впливають на структуру нейронної мережі:

– кількість прихованих шарів;

– кількість нейронів у кожному шарі;

– функція активації.

Кожен з параметрів може комбінуватись із будь-яким іншим, утворюючи великий, але скінченний набір потенційних архітектур. Наприклад, конфігурація [256, 128] з активацією Leaky ReLU буде відрізнятися від [128, 128, 128] з Swish не лише кількісно, а й якісно: різні

глибини, кількість параметрів та нелінійності можуть призводити до радикально відмінної поведінки агентів. Таким чином, навіть у межах обмежених варіантів можливе створення десятків різних, структурно значущих архітектур.

Варто відзначити, що вибір саме цих трьох параметрів (глибина, ширина та функція активації) обумовлений їх фундаментальним впливом на здатність мережі до узагальнення та ефективного навчання. Глибина визначає складність представленої функції, ширина впливає на кількість інформації, що може бути одночасно опрацьована, а тип активації – на нелінійність, яка необхідна для моделювання складних залежностей. Збалансований вибір цих параметрів дозволяє охопити широкий спектр можливих моделей, що варіюються від простих і швидких до більш складних та потужних.

Виходячи з цього, у даній роботі параметри простору архітектур були підібрані з урахуванням необхідності балансу між виразною здатністю моделі та обчислювальною ефективністю. Зокрема, кількість прихованих шарів обмежено діапазоном від 1 до 5, а ширину шарів – значеннями 64, 128 і 256 нейронів. Таке рішення дозволяє дослідити широкий спектр моделей – від компактних і швидких до більш глибоких і потенційно потужних – водночас уникаючи надмірного ускладнення мережі, яке могло б призвести до перенавчання чи значного збільшення часу навчання.

Окрему увагу було приділено вибору функцій активації, оскільки вони безпосередньо визначають ступінь нелінійності моделі – ключовий чинник у здатності нейронної мережі апроксимувати складні залежності у даних. Активаційні функції також впливають на динаміку поширення градієнтів, що критично важливо для стабільності та ефективності навчання, особливо у глибоких мережах. У межах цього дослідження було обрано три популярні функції: ReLU (рисунок 2.2), Leaky ReLU (рисунок 2.3) та Swish (рисунок 2.4).

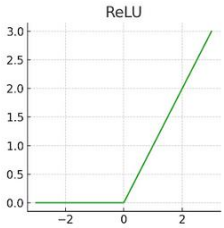

Рисунок 2.2 – Графік активаційної функції ReLU

Функція ReLU (Rectified Linear Unit) є стандартом де-факто у більшості сучасних архітектур завдяки своїй простоті й обчислювальній ефективності, але вона має недолік у вигляді «проблеми загасаючих нейронів» (dead neurons), коли градієнт повністю зникає для негативних вхідних значень.

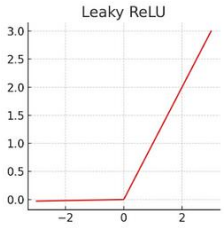

Рисунок 2.3 – Графік активаційної функції Leaky ReLU

Leaky ReLU є модифікацією, що пом’якшує цю проблему, пропускаючи невелике значення градієнта навіть для від’ємних входів, що часто покращує стабільність навчання у складних середовищах.

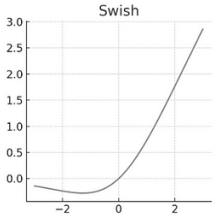

Рисунок 2.4 – Графік активаційної функції Swish

У свою чергу, Swish, запропонована дослідниками з Google, є більш новою активаційною функцією, яка демонструє вищу ефективність у багатьох задачах глибокого навчання завдяки своїй плавності та здатності зберігати інформацію на всьому діапазоні значень.

Завдяки включенню цих трьох функцій у простір пошуку, модель має змогу адаптуватися до різних умов задачі: від швидкого навчання в простіших сценаріях до кращого узагальнення в умовах складної, нестабільної динаміки, як це спостерігається у середовищі Cheetah. Також, генеративний підхід до формування архітектур дозволяє легко масштабувати пошук і інтегрувати додаткові параметри у майбутньому, наприклад, тип нормалізації, регуляризації або навіть складніші типи шарів (згорткові, рекурентні). Така гнучкість робить розроблений простір архітектур придатним для подальшого розвитку і адаптації під різні завдання і типи нейронних мереж.

Використання різноманітних функцій активації сприяє більш ефективному дослідженню простору архітектур під час автоматичного пошуку, оскільки агент може експериментувати з різними комбінаціями для досягнення оптимального балансу між швидкістю збіжності та якістю моделі. Це також підвищує стійкість системи до перенавчання і допомагає знаходити більш стабільні політики у задачах підкріплювального навчання зі складною динамікою.

Крім того, важливим аспектом при формуванні простору архітектур є не лише його виразна гнучкість, а й придатність до ефективного дослідження методами навчання з підкріпленням. Це передбачає, що кількість можливих конфігурацій не повинна бути надмірною, щоб уникнути експоненційного зростання часу пошуку, але водночас має бути достатньою для охоплення широкого спектра потенційно ефективних рішень. Розроблений у цій роботі простір архітектур балансує між повнотою пошуку і обчислювальною доцільністю, що є критично важливим для

практичного застосування методів автоматичного проектування моделей у задачах підкріплювального навчання.

Згенерована модель негайно проходить повний цикл навчання в симуляційному середовищі. Навчання здійснюється за заздалегідь визначеними умовами – фіксована кількість кроків, стабільне середовище, сталі гіперпараметри алгоритму підкріплювального навчання (PPO). Після завершення тренування агент тестується у кількох епізодах, і обчислюється середня сумарна винагорода. Саме це значення використовується як основна метрика оцінки якості архітектури.

Особливістю запропонованого підходу є також те, що простір архітектур не фіксується наперед, а формується поступово, в процесі роботи системи. Кожна нова архітектура додається до історії експериментів, і контролер має змогу в подальших ітераціях враховувати результати попередніх спроб. Це дозволяє зробити формування простору не просто технічним етапом, а інтегрованою частиною процесу навчання та адаптації до середовища.

Описаний механізм дозволяє уникнути непотрібного перебору неефективних варіантів і зосередитися на тих конфігураціях, які потенційно можуть принести високу винагороду. Простір архітектур залишається водночас достатньо компактним для обчислюваного перебору, і водночас достатньо різноманітним для пошуку нових, конкурентних структур.

У підсумку, сформований простір архітектур виконує роль фундаменту для всієї системи NAS-пошуку: він забезпечує гнучкість, контрольованість та масштабованість експериментів, а також дозволяє будувати статистично обґрунтовані висновки щодо ефективності різних топологій у задачах навчання з підкріпленням. Це також дає змогу адаптувати систему до нових завдань без необхідності суттєвої модифікації базової інфраструктури.

2.3 Reward-система та механізми навчання агента

У контексті автоматизованого проєктування архітектур нейронних мереж за допомогою методів навчання з підкріпленням, система винагороди (reward system) виконує центральну роль. Вона слугує не лише засобом зворотного зв'язку для агента, а й критерієм якості генерованих рішень. Саме через систему винагород агент «дізнається», які архітектури є кращими, ефективнішими чи результативнішими в обраному середовищі. Як наслідок, правильно побудована reward-функція – це основа всього процесу пошуку.

Найпоширенішим підходом є використання винагороди, що базується на сумарній продуктивності агента, тобто середньої сумарної нагороди, яку отримує агент під час тестових або навчальних епізодів. Цей підхід дозволяє оцінювати архітектуру як «чорну скриньку»: незалежно від її внутрішньої структури, важливо лише те, як добре вона справляється із завданням. Саме така метрика забезпечує максимальну відповідність практичним цілям – знайти архітектури, які демонструють найвищу ефективність у конкретному середовищі.

У деяких варіантах задач NAS додатково можуть враховуватись інші аспекти якості моделі, наприклад:

– час навчання або час інференсу (як індикатор ефективності);

– кількість параметрів (компактність моделі);

– обчислювальна складність (наприклад, FLOPs);

– стабільність навчання (варіативність результатів);

– або ж інші метрики, пов’язані з узагальненням чи переносністю.

У таких випадках reward-функція може бути комбінованою, з додатковими штрафами або бонусам. Наприклад, агент може отримувати меншу нагороду, якщо архітектура занадто велика або надто довго навчається. Це дозволяє балансувати між якістю рішення і його вартістю, що особливо важливо у сценаріях з обмеженими ресурсами.

Однак у багатьох практичних реалізаціях, особливо в перших етапах дослідження, нагорода спрощується до єдиної метрики продуктивності, щоб не ускладнювати завдання пошуку додатковими факторами. Такий підхід дозволяє зосередитись на основному – пошуку архітектур, які дають максимальний результат у заданих умовах.

Ще одним важливим моментом є однорідність умов експерименту, яка забезпечує чесне порівняння між архітектурами. Зазвичай для всіх моделей використовуються однакові:

– кількість епох або кроків навчання;

– набір гіперпараметрів PPO;

– початкове середовище;

– випадковий seed або початкові умови.

Це дозволяє ізолювати вплив архітектури як єдиного змінного фактора та робить порівняння коректним.

Таким чином, reward-система в поєднанні з PPO реалізує повний цикл зворотного зв’язку. На першому етапі формується архітектура моделі, яка визначає конфігурацію нейронної мережі. Після цього відбувається її навчання у середовищі підкріплювального навчання, що дозволяє перевірити, наскільки ефективно ця модель справляється із заданим завданням. Далі здійснюється оцінка продуктивності цієї моделі на основі її середньої сумарної винагороди за серію епізодів. Отримане значення винагороди використовується як зворотний зв’язок для системи: воно визначає, наскільки вдалою була обрана архітектура. На основі цього зворотного зв’язку оновлюється політика, яка відповідає за вибір наступних архітектур у майбутніх епізодах.

Навіть за умови використання простої reward-функції, яка оцінює лише підсумкову ефективність, така система вже є достатньо потужною для того, щоб поступово наближатися до оптимальних архітектур, здатних до ефективного навчання в реальних задачах.

Варто також зазначити, що вибір простоти reward-функції є компромісом між точністю оцінки архітектури і складністю навчального процесу. Занадто складні або багатокомпонентні функції винагород можуть призводити до проблем з конвергенцією, оскільки агенту складніше розпізнавати, які саме аспекти архітектури впливають на результати. Тому на початкових етапах досліджень акцент робиться на максимально чітких і однозначних метриках, які забезпечують стабільне і зрозуміле навчання контролера. Надалі, накопичивши достатньо експериментальних даних, стає можливим поступово ускладнювати reward-функцію, вводячи додаткові компоненти для врахування таких характеристик, як енергоефективність чи здатність до узагальнення.

2.4 Побудова середовища взаємодії

Для реалізації автоматичного підбору архітектур у рамках підкріплювального навчання було створено інтегроване середовище взаємодії, що поєднує симуляції DeepMind Control Suite із сучасними алгоритмами навчання агентів та контролерів. Завдяки розробленій обгортці, що забезпечує сумісність середовища з популярними RL-бібліотеками, зокрема Stable-Baselines3, реалізується гнучкість у виборі та налаштуванні алгоритмів, таких як PPO та REINFORCE.

Stable-Baselines3 – це широко використовувана відкрита бібліотека на Python, яка надає реалізації сучасних алгоритмів навчання з підкріпленням із акцентом на простоту використання, ефективність та модульність. Використання цієї бібліотеки значно спрощує процес розробки та експериментів, забезпечуючи надійні та оптимізовані реалізації, що прискорюють навчання агентів у складних середовищах.

Такий підхід дозволяє одночасно навчати агента у середовищі та контролер, який оптимізує архітектуру нейронної мережі, створюючи ефективний цикл зворотного зв’язку між процесом пошуку і оцінкою

продуктивності. Таке середовище дає змогу абстрагуватись від низькорівневих деталей реалізації, зосередившись на побудові архітектур і оцінці їх продуктивності в єдиному узгодженому фреймворку.

Середовище призначене для автоматичного підбору архітектур нейронних мереж, які будуть використані як політики для навчання агентів. Кожен епізод у цьому середовищі відповідає повному циклу: побудови архітектури моделі політики, її тренування у фіксованому симульованому середовищі (наприклад, cheetah-run), та подальшої оцінки її ефективності. Такий підхід забезпечує безшовну інтеграцію процесу створення конфігурації з її практичною перевіркою, усуваючи необхідність проміжних збережень або ручного втручання.

У рамках одного епізоду агент-контролер генерує повну конфігурацію нейронної мережі, яка використовується як політика: визначається кількість шарів, число нейронів у кожному з них, а також тип активаційної функції. Після цього архітектура компілюється у повноцінну модель, яка тестується у середовищі управління. Такий підхід дозволяє одразу оцінювати повні архітектури з урахуванням взаємозв’язків між усіма її компонентами, а не окремі локальні рішення.

Далі, створена політика навчається у вибраному середовищі з використанням стандартного алгоритму підкріплювального навчання (у нашому випадку, PPO). Навчання триває фіксовану кількість епізодів, що дозволяє забезпечити однакові умови для всіх моделей. Після завершення тренування агент виконує серію тестових епізодів, у яких вимірюється сумарна винагорода. Це значення фіксується як показник якості архітектури, яка була згенерована в поточному епізоді.

Таким чином, середовище реалізує повний цикл взаємодії:

– від формування архітектури моделі;

– через її навчання на стандартному симульованому завданні;

– до кількісної оцінки її ефективності у формі середньої сумарної винагороди за тестові епізоди.

Особливістю реалізованого середовища є також те, що при порівнянні з базовими (ручними) архітектурами використовується однаковий обсяг ресурсів: фіксована кількість кроків навчання, ідентичне симульоване середовище та однаковий набір гіперпараметрів (наприклад, learning rate, batch size, entropy coefficient тощо). Це дозволяє здійснювати коректне порівняння та об’єктивно оцінювати, наскільки згенеровані контролером архітектури є конкурентними або кращими за заздалегідь задану модель. Реалізація у лістингу 2.1.

Лістинг 2.1 – Програмний код базової (еталонної) архітектури

```
baseline_architecture = {
    'pi': [64, 64],
    'vf': [64, 64],
    'activation': 'relu'
}
```

Ця архітектура є типовим прикладом, що часто використовується у багатьох стандартних реалізаціях алгоритмів із бібліотеки Stable-Baselines3. Вона збалансована за складністю і обчислювальними витратами: має по два шари у політичній і value-мережі з помірною кількістю нейронів (64), а також класичну активаційну функцію ReLU, що демонструє хорошу продуктивність у широкому спектрі задач.

Така конфігурація слугує точкою відліку, яка дозволяє порівняти якість автоматично знайдених моделей з ручним проєктуванням, і водночас є достатньо простою, щоб уникнути надмірної переобученості або нестабільного навчання. Якщо згенеровані архітектури демонструють кращі результати в однакових умовах, це свідчить про ефективність підходу до автоматичного проєктування.

Отримана система інтегрує механізми побудови, навчання і оцінки моделей в одному замкнутому циклі, придатному як для навчання

контролера, так і для великомасштабного тестування різних стратегій проектування архітектур у задачах підкріплювального навчання.

Ще однією важливою особливістю є незалежність середовища від конкретного середовища завдання: реалізація дозволяє легко змінити базову симуляцію (наприклад, з cheetah-run на walker-walk або reacher-easy), що надає змогу вивчати узагальнювальні властивості знайдених архітектур. Система модульна й не прив'язана до одного конкретного домену чи алгоритму – вона лише координує генерацію, тренування й оцінку моделі згідно з заданими правилами.

Оскільки весь цикл автоматизовано, можливо запускати великомасштабні експерименти без втручання людини: наприклад, перебирати тисячі варіантів архітектур у режимі безперервної генерації, навчаючи кожну та порівнюючи їхню ефективність. Це створює основу для побудови повноцінного NAS-фреймворку (Neural Architecture Search), адаптованого під задачі з підкріпленням, де оцінка моделі значно дорожча, ніж у класичних задачах класифікації.

Для підвищення ефективності та прозорості такого процесу, фреймворк також включає розширені механізми моніторингу та аналізу. Для забезпечення більш гнучкого аналізу реалізоване середовище підтримує логування ключових показників під час кожного епізоду: фіксуються згенеровані конфігурації архітектур, фінальні значення винагород і втрати контролера. Це дозволяє не лише оцінювати загальну ефективність, а й аналізувати динаміку навчання, стабільність процесу та фактори, що впливають на якість архітектур.

Крім того, реалізовано збереження найкращої моделі контролера, що дає змогу повторно запускати процес генерації архітектур для подальших тестувань або перенавчання. Повна автоматизація процесу і візуалізація навчання створюють умови для масштабованого дослідження різних стратегій побудови архітектур у задачах підкріплення.

3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Архітектура контролера LSTM для генерації параметрів агента

В процесі автоматичного підбору архітектури агента важливо забезпечити гнучкий та послідовний механізм генерації параметрів, який дозволяє враховувати взаємозв’язки між окремими елементами конфігурації. Контролер у даній реалізації виконує роль генератора, що на кожній ітерації формує повний набір параметрів для побудови нової архітектури, зокрема: кількість шарів, кількість нейронів у кожному шарі, а також тип функції активації. Саме тому вибір ефективної моделі контролера є ключовим аспектом успішної реалізації системи.

Для реалізації контролера було використано архітектуру LSTM (Long Short-Term Memory) – тип рекурентної нейронної мережі, що добре підходить для обробки послідовної інформації. У даному випадку LSTM обробляє вхідний вектор та генерує представлення, яке одночасно використовується для оцінки ймовірностей усіх трьох параметрів конфігурації: кількості шарів, кількості нейронів і функції активації. Хоча генерація не є покроковою у часі, саме рекурентна структура дозволяє контролеру мати потенціал до урахування послідовної структури даних у майбутніх модифікаціях.

Такий підхід забезпечує більшу гнучкість у порівнянні зі звичайними лінійними моделями, дозволяючи контролеру адаптуватися до зворотного зв’язку у вигляді винагороди. Це дає змогу ефективніше шукати архітектури, які забезпечують кращу продуктивність агента у середовищі.

Далі в цьому розділі буде детально розглянута структура реалізації контролера, механізм генерації параметрів та ключові особливості обраного підходу.

3.1.1 Опис моделі LSTM

LSTM, на відміну від класичних RNN, у яких інформація про попередні стани швидко втрачається через проблему затухання градієнта, LSTM оснащено трьома видами «воріт» – вхідними, забувними та вихідними – які дозволяють вибірково зберігати ключову інформацію та ігнорувати менш важливі сигнали. Завдяки такій архітектурі мережі LSTM здатні навчатися правильно інтерпретувати й утримувати довготривалі залежності, що робить їх незамінними при обробці природних мов, часових рядів і будь-яких послідовних або контекстуально залежних даних.

Завдяки цим властивостям LSTM широко застосовуються у задачах обробки природних мов, часових рядів та інших контекстно залежних даних. У моїій роботі модель LSTM використовується в ролі контролера для автоматичного підбору архітектури агента у середовищі навчання з підкріпленням.

Контролер на основі LSTM отримує на вхід початковий стан та намагається згенерувати параметри нейронної мережі агента – такі як кількість шарів, розмір кожного шару та функцію активації. Важливо, що в моїй реалізації ці параметри генеруються паралельно, а не строго послідовно. Тобто контролер формує спільне внутрішнє представлення через LSTM, з якого одночасно отримуються логіти – це необроблені числові оцінки, які ще не є ймовірностями – для кожного з параметрів. Такий підхід дозволяє LSTM підтримувати узгодженість вибору, базуючись на єдиному контексті, але не зобов’язує мережу генерувати параметри крок за кроком.

Ця спільна динаміка прихованих станів дає змогу контролеру враховувати взаємозв’язки між різними параметрами, що допомагає уникнути створення суперечливих чи неефективних конфігурацій архітектури. Наприклад, якщо контролер вирішив обрати велику кількість

шарів, він може зменшити кількість нейронів у кожному, щоб уникнути надмірної складності, або навпаки.

Крім того, LSTM відрізняються високою стійкістю до шуму в даних і здатні адаптуватися до невеликих випадкових відхилень у послідовностях, що важливо у випадку стохастичного семплінгу параметрів контролером. Це означає, що навіть якщо окремі кроки генерації здійснюються із деякою випадковістю або додатковим шумом (наприклад, для забезпечення експлоративної складової), мережа зможе коригувати свої внутрішні стани так, щоб кінцевий набір параметрів залишався узгодженим і близьким до тих конфігурацій, які в минулому давали високі результати.

Отже, використання LSTM у ролі контролера забезпечує баланс між гнучкістю та стабільністю при пошуку оптимальних архітектур, враховуючи всю історію прийнятих рішень у вигляді внутрішнього стану мережі, що позитивно впливає на якість та ефективність процесу автоматичного підбору нейронних мереж.

3.1.2 Структура контролера

Контролер реалізовано як клас Controller, всередині якого основним блоком є багатошаровий LSTM. На вході контролера кожен крок отримує тензор розмірності (1, 1, input_size), де input_size=10 – це умовний розмір «порожнього» вхідного вектора. Це дозволяє ініціювати процес семплінгу параметрів буквально «з нуля», без попереднього контексту.

```
hidden = self.init_hidden()
x = torch.zeros(1, 1, 10)
```

Використовуються три LSTM-шари з розмірністю прихованого стану hidden_size=32. Саме ця кількість шарів та розмір схованого вектора дозволяють контролеру зберігати досить багатий контекст попередніх рішень, не витрачаючи надто багато ресурсів на надмірну складність. Після

LSTM йдуть три окремі fully-connected шари (fc_layers, fc_units, fc_activation), кожен з яких видає логіти для одного з трьох параметрів:

– fc_layers: генерує логіти для кількості шарів;

– fc_units: для кількості нейронів у кожному шарі;

– fc_activation: для вибору функції активації.

Реалізація у лістингу 3.1.

Лістинг 3.1 – Вивід логітів з моделі

```
out_layers, out_units, out_activation, _ = self.forward(x, hidden)
layers_logits = out_layers.squeeze() / temperature
units_logits = out_units.squeeze() / temperature
activation_logits = out_activation.squeeze() / temperature
```

Для перетворення логітів у дискретні параметри архітектури використовується softmax, після чого здійснюється семплінг за допомогою multinomial sampling. Реалізація у лістингу 3.2.

Лістинг 3.2 – Обчислення ймовірностей та ентропії

```
out_layers, out_units, out_activation, _ = self.forward(x, hidden)
layers_logits = out_layers.squeeze() / temperature
units_logits = out_units.squeeze() / temperature
activation_logits = out_activation.squeeze() / temperature
```

Далі, на основі ймовірностей відбувається семплінг дискретних значень параметрів, і створюється опис архітектури. Реалізація у лістингу 3.3.

Лістинг 3.3 – Семплінг параметрів та побудова архітектури

```
units_idx = torch.multinomial(units_probs, 1).item()
```

Продовження лістингу 3.3

```
    activation_idx = torch.multinomial(activation_probs, 1).item()
    layers_idx = torch.multinomial(layers_probs, 1).item()
    units_options = [64, 128, 256]
    activations = ['relu', 'leaky_relu', 'swish']
    units = units_options[units_idx]
    activation_func = activations[activation_idx]
    num_layers = layers_idx + 1
```

Для логарифмічної оцінки дій контролера розраховуються логарифми ймовірностей обраних варіантів. Ці значення пізніше використовуються в обчисленні градієнта під час оптимізації. Реалізація у лістингу 3.4.

Лістинг 3.4 – Логарифмічні ймовірності та підсумковий результат

```
    log_prob_layers = torch.log(layers_probs[layers_idx])
    log_prob_units = torch.log(units_probs[units_idx])
    log_prob_activation = torch.log(activation_probs[activation_idx])
    log_prob_sum = log_prob_layers + log_prob_units + log_prob_activation
    pi = [units] * num_layers
    vf = [units] * num_layers
    architecture = dict(pi=pi, vf=vf, activation=activation_func)
    return architecture, log_prob_sum, total_entropy
```

Отже, контролер у кожному циклі проходить три етапи:

– прямий прохід через LSTM;

– перетворення виходів у логіти;

– обчислення softmax + мультино-міальний відбір для отримання дискретного параметру, а також розрахунок log-prob та ентропії для подальшої оптимізації.

Ця архітектура дозволяє побудувати повноцінну нейронну політику, здатну генерувати комплексні та взаємопов’язані конфігурації агента, враховуючи історію попередніх виборів та зберігаючи гнучкість через експлорацію (параметр epsilon) і регулювання розподілу (температура).

3.1.3 Особливості реалізації

У процесі розробки контролера я реалізувала низку важливих модифікацій порівняно зі стандартним підходом, що дозволили зробити процес генерації архітектур більш гнучким і стабільним. Зокрема, я додала можливість регулювання ступеня випадковості при виборі параметрів за допомогою температурного семплінгу та ε-жадібності. Це дає змогу контролювати перехід між детермінованим вибором та більш випадковими стратегіями, підвищуючи здатність моделі досліджувати нові архітектури й уникати застрягання в локальних оптимумах.

Крім того, я включила у функцію втрат додатковий регуляризаційний доданок – ентропію вихідного розподілу. Завдяки цьому контролер не надто швидко зосереджується на одній комбінації параметрів, зберігаючи варіативність у виборах. Формула функції втрат має вигляд:

```
loss = -log_prob * total_reward - entropy_coef * entropy
```

Ще однією важливою модифікацією стало введення штрафу за повторення однієї й тієї ж архітектури. Якщо протягом п’яти ітерацій контролер генерує ідентичну конфігурацію (я це перевіряю через серіалізацію словника у формат JSON), отримана нагорода навмисно зменшується на 50%. Це заохочує систему шукати нові рішення й активніше досліджувати простір архітектур. Реалізація у лістингу 3.5.

Лістинг 3.5 – Штраф за повторення архітектури протягом кількох ітерацій

```
last_arch_str = None
```

Продовження лістингу 3.5

```
same_count = 0
if arch_str == last_arch_str:
   same_count += 1
else:
   same_count = 1
   last_arch_str = arch_str
...

if same_count >= 5:
    print(f"Architecture repeated {same_count} times.
Penalizing reward.")
    total_reward *= 0.5
```

Для підтримки стабільності результатів на етапі порівняння моделей я використовувала фіксацію початкового стану генераторів випадкових чисел. Це дозволяє гарантувати відтворюваність експериментів і справедливість у порівнянні. Для цього наприкінці реалізації контролера викликається функція, зображена у лістингу 3.6.

Лістинг 3.6 – Встановлення фіксованого зерна для відтворюваності експериментів

```
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
```

Фіксація зерна є загальноприйнятою практикою у дослідженнях зі штучного інтелекту. Вона значно знижує вплив випадковості та дозволяє точніше аналізувати ефективність підходів.

Усі ці рішення спрямовані на досягнення балансу між експлорацією та експлуатацією, забезпечення стійкості до коливань у нагороді, а також збереження високої варіативності згенерованих конфігурацій.

3.2 Інтеграція контролера з навчальним середовищем

Інтеграція контролера з навчальним середовищем є ключовим етапом у реалізації системи автоматизованого пошуку нейронних архітектур. Контролер, який виконує роль агента підкріплювального навчання, взаємодіє з середовищем, формуючи послідовність рішень, що визначають структуру нейронної мережі. Цей процес потребує чіткої організації обміну інформацією між компонентами системи, забезпечуючи безперервний цикл генерації, тренування та оцінювання архітектур.

Важливою складовою інтеграції є визначення протоколів передачі даних та форматів опису архітектур, що дозволяють гнучко адаптувати контролер під різні завдання та умови середовища. Крім того, інтеграція передбачає налаштування параметрів навчання агента, таких як кількість ітерацій, розмір пакету даних для тренування, а також критерії зупинки або переходу до наступного етапу.

Забезпечення ефективної взаємодії між контролером і середовищем дозволяє оптимізувати процес пошуку архітектур, підвищуючи якість кінцевих моделей та знижуючи обчислювальні витрати за рахунок зменшення кількості ітерацій і непотрібних повторних тренувань.

3.2.1 Формування архітектури агентів

Контролер, реалізований у вигляді LSTM-моделі, генерує параметри нейронної мережі агента: кількість прихованих шарів, розмірність кожного шару та тип функції активації. Кожен параметр семплюється із відповідного ймовірнісного розподілу, після чого всі значення об’єднуються у словник, що визначає архітектуру. Цей словник слугує вхідними даними для ініціалізації агента, адаптованого до поточної конфігурації. Контролер постійно оновлює свої параметри на основі отриманого зворотного зв’язку, що дозволяє поступово покращувати якість генерованих архітектур.

3.2.2 Побудова агента у середовищі

Після отримання параметрів від контролера ініціалізується новий агент, який реалізує алгоритм PPO. Ініціалізація враховує як структуру, сформовану контролером, так і простори спостережень та дій конкретного середовища. Це забезпечує узгодженість між входами/виходами нейронної мережі агента та специфікацією середовища.

При кожній зміні архітектури створюється новий екземпляр агента – старий повністю знищується. Таким чином гарантовано, що жодні ваги або конфігураційні залишки попередніх запусків не впливають на нове навчання. Цей підхід забезпечує чистий старт для кожної нової архітектури, що критично важливо для об’єктивної оцінки її ефективності. Крім того, він дозволяє уникнути накопичення помилок і потенційних конфліктів між різними конфігураціями. Реалізація у лістингу 3.7.

Лістинг 3.7 – Код функції для створення та тренування агента

```
def train_agent(env, architecture, total_timesteps=5000):
    vec_env = DummyVecEnv([lambda: env])
    pi_layers = architecture['pi']
    vf_layers = architecture['vf']
    activation = architecture['activation']
    policy_kwargs = dict(
        net_arch=[dict(pi=pi_layers, vf=vf_layers)],
        activation_fn=get_activation_function(activation)
    )
    print("Training with architecture:", architecture)
    model = PPO("MlpPolicy", vec_env,
policy_kwargs=policy_kwargs, verbose=1)
    model.learn(total_timesteps=total_timesteps)
    return model
```

3.2.3 Робота з векторним середовищем

Для запуску агента використовується обгортка DummyVecEnv з бібліотеки stable-baselines3. Вона дозволяє управляти однією або кількома копіями середовища у вигляді векторної структури. Це спрощує управління станами середовища, автоматично виконує скидання на початку кожного епізоду та забезпечує повторюваність експериментів.

```
from stable_baselines3.common.vec_env import DummyVecEnv
vec_env = DummyVecEnv([lambda: env])
```

Використання векторного середовища дозволяє уникнути помилок, пов’язаних із залишковими станами середовища, та покращує стабільність навчання при частій зміні архітектури.

3.2.4 Інтеграція контролера та середовища як замкнений цикл

Процес побудований як замкнене коло:

– контролер формує параметри архітектури;

– на основі цих параметрів створюється агент;

– агент навчається у середовищі, взаємодіючи з ним та отримуючи зворотний зв’язок;

– отримана нагорода передається назад контролеру для оновлення його параметрів та покращення генерації архітектур.

Таким чином, кожна ітерація циклу інтегрує як пошук нових архітектур, так і оцінку їх ефективності у середовищі.

3.3 Процес навчання контролера

Контролер на кожному кроці LSTM формує ймовірнісний розподіл параметру, враховуючи контекст попередніх виборів. Це дозволяє

моделювати залежності між параметрами та отримувати корельовані конфігурації.

Після повного семплювання конфігурації контролер отримує логарифм ймовірності (log_prob) згенерованої архітектури, що є важливим для подальшого оновлення. Реалізація у лістингу 3.8.

Лістинг 3.8 – Обчислення log_prob для згенерованої архітектури

```
    layers_probs = torch.softmax(out_layers.squeeze(), dim=0)
    units_probs = torch.softmax(out_units.squeeze(), dim=0)
    activation_probs = torch.softmax(out_activation.squeeze(),
dim=0)
    units_idx = torch.multinomial(units_probs, 1).item()
    activation_idx = torch.multinomial(activation_probs,
1).item()
    layers_idx = torch.multinomial(layers_probs, 1).item()
    log_prob_layers = torch.log(layers_probs[layers_idx])
    log_prob_units = torch.log(units_probs[units_idx])
    log_prob_activation =
torch.log(activation_probs[activation_idx])
    log_prob_sum = log_prob_layers + log_prob_units +
log_prob_activation
```

Для підтримки балансу між дослідженням нових архітектур і експлуатацією вже знайдених ефективних рішень у функцію втрат контролера включено два основних компоненти: винагороду агента (R), яка відображає сумарну якість згенерованої архітектури після навчання у середовищі, та ентропію розподілу параметрів, яка виступає регуляризатором, що заохочує різноманітність виборів і запобігає надмірній детермінованості контролера. Формально функція втрат містить суму негативного логарифму ймовірності, помноженого на винагороду, і додатковий штраф за низьку ентропію з коефіцієнтом, що регулює її внесок.

Основний цикл навчання контролера організований у вигляді повторюваного процесу. На кожній ітерації контролер генерує архітектуру, після чого створюється агент з цією конфігурацією, який проходить навчання у середовищі. Потім агент оцінюється, і отримана винагорода використовується для обчислення функції втрат.

Оновлення параметрів контролера виконується за допомогою Adam-оптимізатора, який застосовує алгоритм зворотного поширення помилки на основі обчисленої функції втрат. Ентропійна регуляризація допомагає зберігати баланс між експлуатацією найкращих архітектур та дослідженням нових.

Під час навчання ведеться логування ключових метрик – винагороди агента та значення функції втрат, що дозволяє візуалізувати прогрес у вигляді графіків. При цьому зберігається найкращий стан контролера, який досягнув максимальної винагороди, для подальшого використання.

Наприкінці циклу відбувається відбір топ-архітектур на основі отриманих винагород, що дозволяє вибрати найперспективніші конфігурації для подальшого аналізу або впровадження.

Загалом, такий процес навчання контролера забезпечує ефективний баланс між дослідженням нових архітектур і використанням уже знайдених хороших варіантів, підвищуючи продуктивність та стабільність NAS-процесу.

3.4 Проблеми та виклики реалізації

У процесі реалізації системи автоматичного пошуку архітектур виникла низка технічних і методологічних викликів, які впливали як на стабільність навчання, так і на якість фінальних результатів. Найбільш критичні труднощі стосувалися стабільності навчання контролера, проблем сумісності з обраними середовищами та бібліотеками, а також забезпечення справедливого оцінювання згенерованих архітектур.

Однією з головних складностей стала нестабільність навчання LSTM-контролера. На ранніх етапах навчання спостерігалося значне розкидання у виборі архітектур, а також відсутність тенденції до покращення середньої нагороди. Це зумовлювалося надто великою початковою швидкістю навчання (0.1). Крім того, на перших етапах не використовувались регулятори ентропії та температури. Унаслідок цього контролер швидко фіксувався на локальних максимумах і повторно генерував однакові архітектури.

Додатково, у процесі навчання виникали труднощі з обчисленням градієнтів, що проявлялися у вигляді різких коливань у втраті та нестабільності кроків оновлення. Це було пов’язано з невдалим поєднанням гіперпараметрів на ранніх етапах – особливо високим значенням швидкості навчання, а також з особливостями LSTM, який чутливо реагує на накопичення похибок у градієнтному проході. Зниження швидкості навчання та додавання ентропійної регуляції частково усунули цю проблему та стабілізували динаміку оновлення ваг. Для вирішення цієї проблеми було знижено швидкість навчання до 0.001 після першого мільйона кроків, а також додано ентропійний доданок і температурну регуляцію, що дозволило суттєво покращити варіативність і загальну продуктивність генерації.

Іншою важливою проблемою стала складність інтеграції з середовищем. Через тривалий час обчислень і завантаження середовища навчання стало вузьким місцем у продуктивності. Було проведено оптимізацію коду – зокрема, переорганізацію циклів генерації та збереження станів – що дозволило суттєво скоротити час очікування без втрати точності.

Окрему складність становила інтеграція функціональності для тренування агента за алгоритмом PPO. Використання оригінальної бібліотеки від DeepMind виявилося неможливим через обмеження macOS, яка не дозволяла коректне встановлення залежностей бібліотеки Acme.

Унаслідок цього було прийнято рішення реалізувати PPO через сумісну обгортку для gym із використанням бібліотеки stable-baselines3. Це потребувало додаткового часу на налаштування сумісності, оскільки повернення параметрів між середовищем і агентом відрізнялося за форматом. Після кількох невдалих спроб вдалося створити стабільну обгортку, яка забезпечила коректну взаємодію між контролером і середовищем.

Оцінка якості згенерованих архітектур також виявилася непростою. У процесі виявилось, що оцінка однієї і тієї ж архітектури може суттєво відрізнятись у різних запусках через стохастичність середовища та навчання. Тому було введено фіксоване початкове зерно (seed) як для базової, так і для згенерованих архітектур, що дозволило уніфікувати умови тестування. Крім того, для точнішої оцінки кожна потенційно найкраща архітектура проганялася тричі, а нагорода усереднювалась, щоб уникнути випадкових піків або спадів.

Ще одним викликом стало застрягання у повторному виборі однієї й тієї ж архітектури, що призводило до втрати варіативності. Алгоритм вважав одну з архітектур оптимальною й продовжував її генерувати в десятках наступних епізодів, знижуючи загальну ефективність пошуку. Для вирішення цієї проблеми було реалізовано штраф за дублікати (same_count), а також додано ентропію та температурне масштабування, що дозволило зберегти баланс між експлуатацією успішних рішень та дослідженням нових.

3.5 Аналіз продуктивності реалізованої системи

У процесі оцінки продуктивності реалізованої системи ключовим критерієм виступає співвідношення між витратами обчислювальних ресурсів та ефективністю пошуку архітектур, що демонструють високі результати в середовищі навчання з підкріпленням. Поточна реалізація

функціонує виключно на центральному процесорі, без використання графічних прискорювачів, що істотно впливає на часові характеристики при виконанні повного циклу NAS-процедури.

Середній час генерації, навчання та оцінки однієї популяції з 45 архітектур становить приблизно 1 хвилину 40 секунд. З огляду на те, що кожна архітектура проходить повноцінне навчання агента на горизонті в 5000 кроків, загальний обсяг симуляцій для одного циклу досягає 225 тисяч кроків. Унаслідок цього, система демонструє лінійне зростання часу виконання зі збільшенням кількості кандидатних архітектур, що виступає природним обмеженням масштабованості в умовах фіксованої обчислювальної потужності.

Також варто зазначити, що частина ресурсів витрачається на навчання агентів, архітектури яких з низькою ймовірністю зможуть досягти прийнятного рівня продуктивності. У поточному вигляді ці епізоди обробляються повністю, що знижує загальну ефективність. Це відкриває можливість для реалізації евристик ранньої зупинки або відсіювання неефективних структур до завершення повного епізоду, з метою економії часу та обчислювальних витрат.

Обмеження продуктивності також зумовлені відсутністю векторизованої обробки агентів і послідовним виконанням епізодів у рамках одного потоку. Паралелізація середовищ або використання пакетного навчання (batch RL) може суттєво зменшити загальний час експерименту при збереженні якості результатів. Окрім того, можлива оптимізація механізму оновлення параметрів, зокрема за рахунок більш ефективного зберігання історій, уникнення надмірних перерахунків та використання апаратних інструкцій SIMD.

З точки зору пам’яті, система не є обмеженою в обсязі – архітектурні конфігурації мають компактне представлення, а агенти зберігають лише базові параметри мережі. Тим не менш, за умов масштабування системи до сотень архітектур або переходу до складніших середовищ із

багатоваріантною динамікою, буде доцільно реалізувати політики кешування результатів та повторного використання попередньо обчислених траєкторій.

Отже, у поточному вигляді система забезпечує адекватний рівень продуктивності для експериментів малого та середнього масштабу. Водночас вона зберігає потенціал для суттєвої оптимізації – як на рівні організації обчислень, так і за рахунок вдосконалення логіки прийняття рішень щодо доцільності навчання конкретних архітектур. У перспективі це дозволить перейти до більш глибоких архітектурних просторів без суттєвого зростання часу виконання.

3.6 Вплив гіперпараметрів контролера на якість вибору

Поведінка контролера безпосередньо залежить від значень гіперпараметрів, які визначають як процес генерації архітектур, так і динаміку його навчання. Емпіричний аналіз показав, що початкові параметри, зокрема швидкість навчання на рівні 0.1, призводили до нестабільного оновлення ваг та хаотичної динаміки вибору архітектур. Відповідно, на початкових етапах спостерігалася значна варіативність результатів та невисока якість згенерованих моделей, що виражалося як у коливанні нагород, так і у повторенні архітектур із подібними конфігураціями.

Зміна швидкості навчання до 0.001 вже після кількох ітерацій дозволила суттєво знизити коливання під час оновлення параметрів контролера, що забезпечило плавніше збіження. На цьому етапі якість вибору архітектур зросла: стабільність досягнутих нагород покращилася.

Крім цього, було експериментально введено додаткові гіперпараметри – температуру (temperature) та ентропійну регуляризацію (entropy coefficient), які не були присутні на початкових етапах. Їхнє впровадження було мотивоване необхідністю уникнення

надмірного повторення однакових рішень, що спостерігалося у відсутності механізмів стохастичного розширення простору пошуку. Збільшення значень як температури, так і коефіцієнта ентропії дозволило суттєво розширити область досліджуваних варіантів і зменшити частоту повторень. Це позитивно вплинуло на процес оптимізації: зросла різноманітність архітектур, що оцінювалися, і було досягнуто вищих значень сумарної нагороди.

На графіку (рисунок 3.1) простежується значна варіативність значень нагород за епізод, що проявляється у вигляді коливань: періоди зростання змінюються спадом. Така динаміка є природною та навіть бажаною в умовах активного дослідження простору архітектур. Збереження високої варіативності свідчить про те, що контролер не «застряг» у локальних максимумах і продовжує досліджувати нові потенційно ефективні рішення, а не лише експлуатує вже відомі. Це стало можливим завдяки введенню температурної регуляції та ентропійного доданку у функцію втрат, що забезпечує необхідний баланс між експлуатацією й дослідженням.

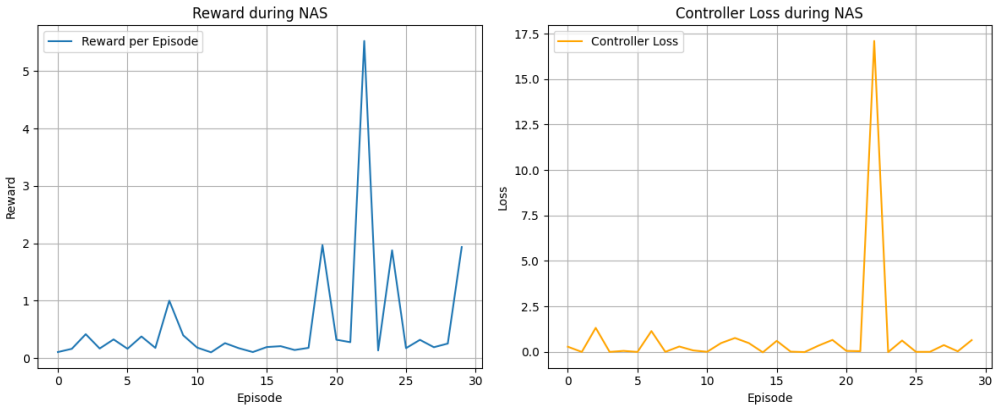

Рисунок 3.1 – Динаміка винагород і втрат контролера під час NAS

З погляду навчальної динаміки, така поведінка вказує на збереження потенціалу до генерації нових архітектур, що є критично важливим на пізніх етапах навчання, коли ризик переобучення контролера особливо високий.

Таким чином, аналіз засвідчує критичну роль гіперпараметрів контролера у формуванні ефективної динаміки пошуку. Занадто висока швидкість навчання на ранніх етапах гальмувала збіжність, тоді як збалансоване використання температурного масштабування та ентропійної регуляризації підвищило як експлоративність, так і якість остаточного вибору. Без цих механізмів контролер мав схильність до конвергенції у вузький клас повторюваних архітектур, що суттєво знижувало потенціал системи до пошуку оптимальних конфігурацій.

4 СТАБІЛЬНІСТЬ РЕЗУЛЬТАТІВ І УЗАГАЛЬНЕННЯ

Одним з ключових аспектів оцінки якості системи нейроеволюційного пошуку архітектур є перевірка стабільності отриманих результатів та здатності моделі до узагальнення. Після завершення основного етапу навчання контролера, було обрано кілька найкращих згенерованих архітектур, які демонстрували найвищу усереднену нагороду на валідаційному середовищі. Щоб мати об’єктивне уявлення про продуктивність цих архітектур, проведено серію контрольованих експериментів, в яких найкращі архітектури порівнювалися з базовою за фіксованих умов.

Для забезпечення чесності порівняння та виключення стохастичних факторів, як базова, так і топова архітектури були навчені на одному і тому самому початковому зерні (random seed). Кожна з моделей тренувалася протягом 100 000 кроків, після чого проводилася оцінка їхньої ефективності шляхом виконання 100 епізодів у середовищі без подальшого навчання. В результаті було отримано середнє значення нагороди за епізод, яке стало основою для порівняння.

В якості базової архітектури було обрано класичну конфігурацію: дві приховані шари по 64 нейрони як для політики (pi), так і для функції цінності (vf), з активацією ReLU. Така архітектура часто використовується в стандартних реалізаціях PPO, оскільки забезпечує прийнятний компроміс між обчислювальною складністю і здатністю до узагальнення. Вона була обрана як початкова точка для навчання контролера, а також як орієнтир для подальшого порівняння.

Для об’єктивної оцінки ефективності контролера було проведено серію незалежних запусків, у яких аналізувалися результати генерації архітектур та їх якість у порівнянні з базовою моделлю. У кожному з п’яти запусків контролер генерував по 30 архітектур. З них відбиралося 5 найкращих на основі середньої нагороди після

навчання (100 тис. кроків) та подальшої оцінки на 100 епізодах. Всі архітектури (як згенеровані, так і базова) навчалися та оцінювалися з однаковим сидом, щоб виключити стохастичні відхилення.

У більшості запусків згенеровані архітектури демонстрували значно вищі нагороди порівняно з базовою (рисунок 4.1). Найбільше зростання (більше ніж на 90%) спостерігалося у першому та третьому експерименті. Це свідчить про ефективність пошуку контролером, який здатен знайти рішення, що краще адаптовані до середовища.

Запуск	Середня нагорода (топ-5 арх.)	Середня нагорода (базова)
1	140.2	68.0
2	98.4	94.0
3	160.1	75.0
4	99.0	70.0
5	104.0	81.0
Середнє	**120.3**	**77.6**

Рисунок 4.1 – Порівняння нагород топ-5 архітектур контролера та базової моделі (5 запусків)

Варто також зазначити, що в другому запуску перевага топових архітектур була менш вираженою. Це може бути зумовлено як внутрішніми флуктуаціями стохастичного середовища, так і тимчасовим застряганням контролера у локальному максимумі. Водночас навіть у таких випадках, як показано у попередніх розділах, використання температурного регулювання та ентропійних штрафів дозволяє з часом вийти на більш варіативні та ефективні архітектури.

Загалом, середнє покращення нагороди між топ-архітектурами та базовою склало понад 42.7 одиниці, або приблизно +55% приросту. Такий

результат є важливою демонстрацією стабільності системи, навіть попри початкову стохастичність та флуктуації в окремих епізодах. З метою перевірки стабільності результатів, обрані топові архітектури були додатково протестовані на інших seed. За результатами п’яти запусків з різними фіксованими зернами (рисунок 4.2) було отримано незначне коливання в середніх нагородах, що підтверджує здатність архітектур до узагальнення та відсутність переобучення на окремому наборі випадкових параметрів.

Seed	Середня нагорода (100 епізодів)
0	142.1
11	139.4
21	141.3
42	140.8
77	143.0
Середнє	**141.3 ± 1.2**

Рисунок 4.2 – Варіація результатів при зміні seed для топової архітектури

Ці результати демонструють високий рівень стабільності системи – як у процесі навчання, так і під час незалежної оцінки. Контролер навчався у стохастичному середовищі, однак завдяки запровадженим регуляторам (температура, ентропія, штраф за дублікати), йому вдалося зберегти продуктивну варіативність архітектур. Це особливо важливо для задач, де важлива не лише ефективність, але й адаптивність до змін середовища.

Також спостерігалося, що деякі архітектури, які не демонстрували найвищі результати в окремих проходах, після усереднення показували стабільно добру продуктивність. Це стало додатковим аргументом на

користь багаторазового тестування та усереднення нагород, що було реалізовано як частина остаточної стратегії відбору.

Таким чином, результати підтверджують, що система здатна не лише знаходити сильні архітектури у межах одного запуску, а й формувати архітектурні рішення, стійкі до змін стохастичних факторів, що робить її придатною до використання в більш широких сценаріях застосування.

ВИСНОВКИ

Дане дослідження було присвячене розробці комплексної системи автоматизованого підбору архітектури нейронної мережі для агентів підкріплювального навчання у симульованих середовищах DM Control Suite, зокрема на прикладі завдання Cheetah. Результати проведеної роботи демонструють успішне досягнення поставлених цілей та відкривають перспективи для подальшого вдосконалення і розширення підходу.

У ході дослідження було реалізовано ефективний підхід до організації експериментів із навчання агентів у середовищах з високою динамікою. Було створено програмний інструментарій для автоматичної генерації різних архітектур нейронних мереж, їх навчання за допомогою алгоритмів підкріплювального навчання (PPO) та збору результатів для подальшого порівняльного аналізу.

Застосування сучасних підходів до автоматизованого підбору архітектури (Neural Architecture Search) дозволило значно підвищити ефективність пошуку оптимальних рішень без необхідності ручного втручання чи підбору гіперпараметрів. Система була побудована таким чином, щоб забезпечити гнучкість і масштабованість експериментів, а також можливість швидкого додавання нових середовищ для тестування універсальності підходу.

Серцевиною розробленої системи став контролер на основі рекурентної нейронної мережі (LSTM), який генерував параметри архітектури для агентів. Далі кожна згенерована архітектура тестувалася у середовищі Cheetah, а отримана винагорода використовувалася як єдиний критерій якості. Такий підхід дозволив агенту самостійно знаходити найбільш ефективні конфігурації мережі для конкретного завдання, а також накопичувати статистику для подальшого аналізу впливу різних параметрів на результативність.

Впровадження розробленої системи автоматичного підбору архітектури дозволяє значно знизити вплив людського фактору у процесі проектування агентів підкріплювального навчання, підвищити якість отриманих моделей та скоротити час на проведення експериментів. Такий підхід відкриває перспективи для застосування у більш складних або реальних задачах, наприклад, у сфері робототехніки, автономного керування чи оптимізації інженерних систем.

Подальші напрямки розвитку системи включають розширення набору тестових середовищ (наприклад, Walker, Hopper та інші з DM Control Suite), а також інтеграцію додаткових критеріїв оцінки, таких як обчислювальна складність, швидкість навчання чи стійкість до змін у середовищі. Перспективним є впровадження більш складних NAS-алгоритмів та використання мета-навчання для підвищення швидкості та якості пошуку.

Загалом, результати проведеного дослідження демонструють високий потенціал використання сучасних методів автоматизованого підбору архітектури та підкріплювального навчання для побудови ефективних агентів у складних симульованих середовищах. Це створює підґрунтя для подальших досліджень і практичного застосування у галузях, де потрібна адаптація моделей до змінних умов та високих вимог до продуктивності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Nandy A., Biswas M. Applying Python to Reinforcement Learning. *Reinforcement Learning*. 2017. URL: http://dx.doi.org/10.1007/978-1-4842-3285-9_4 (дата звернення: 20.05.2025).

2. Dalamagkidis K., Kolokots D. Reinforcement Learning for Building Environmental Control. *Reinforcement Learning*. 2008. URL: http://dx.doi.org/10.5772/5286 (дата звернення: 20.05.2025).

3. Nandy A., Biswas M. RL Theory and Algorithms. *Reinforcement Learning*. 2017. URL: http://dx.doi.org/10.1007/978-1-4842-3285-9_2 (дата звернення: 20.05.2025).

4. White C., Safari M., Sukthanker R., Ru B. Neural Architecture Search: Insights from 1000 Papers. *Machine Learning*. 2022. URL: https://doi.org/10.48550/arXiv.2301.08727 (дата звернення: 20.05.2025).

5. Cassimon A., Mercelis S., Mets K. Scalable Reinforcement Learning-based Neural Architecture Search. *Neural Computing and Applications*. 2024. URL: https://doi.org/10.1007/s00521-024-10445-2 (дата звернення: 20.05.2025).

6. Mellor J., Turner J., Storkey A. Neural architecture search without training. *Reinforcement Learning*. 2021. URL: https://proceedings.mlr.press/v139/mellor21a.html (дата звернення: 20.05.2025).

7. Zoph B., Le Q. V. Neural architecture search with reinforcement learning. *Machine Learning*. 2016. URL: https://doi.org/10.48550/arXiv.1611.01578 (дата звернення: 03.06.2025).

8. Pham H., Guan M. Y., Zoph B., Le Q. V., Dean J. Efficient neural architecture search via parameter sharing. *Machine Learning*. 2018. URL: https://doi.org/10.48550/arXiv.1802.03268 (дата звернення: 03.06.2025).

9. Tassa Y., Doron Y., Muldal A. dm_control: Software and tasks for continuous control. *Robotics.* 2020. URL: https://doi.org/10.48550/arXiv.2006.12983 (дата звернення: 03.06.2025).

10. Miao Y., Song X., Co-Reyes J. D. Differentiable Architecture Search for Reinforcement Learning. *Machine Learning*. 2021. URL: https://doi.org/10.48550/arXiv.2106.02229 (дата звернення: 03.06.2025).

11. Xie L., Chen X., Bi K. Weight-Sharing Neural Architecture Search: A Battle to Shrink the Optimization Gap. *Computer Vision and Pattern Recognition.* 2020. URL: https://doi.org/10.48550/arXiv.2008.01475 (дата звернення: 03.06.2025).

12. Pourchot A., Ducarouge A., Sigaud O. To Share or Not To Share: A Comprehensive Appraisal of Weight-Sharing. *Machine Learning*. 2020. URL: https://doi.org/10.48550/arXiv.2002.04289 (дата звернення: 03.06.2025).

13. Yu K., Ranftl R., Salzmann M. An Analysis of Super-Net Heuristics in Weight-Sharing NAS. *Machine Learning*. 2021. URL: https://doi.org/10.48550/arXiv.2110.01154 (дата звернення: 03.06.2025).

14. Ghasemi M., Moosavi A. H., Ebrahimi D. Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges. *Artificial Intelligence*. 2024. URL: https://doi.org/10.48550/arXiv.2411.18892 (дата звернення: 10.06.2025).

15. Holder J. P., Rajan R., Song X. Automated Reinforcement Learning (AutoRL): A Survey and Open Problems. *Journal of Artificial Intelligence Research*. 2022. URL: https://doi.org/10.1613/jair.1.13596 (дата звернення: 10.06.2025).

16. Meng F., Wang C. A., Zhang L. Evolution and Efficiency in Neural Architecture Search: Bridging the Gap Between Expert Design and Automated Optimization. *Neural and Evolutionary Computing*. 2024. URL: https://doi.org/10.48550/arXiv.2403.17012 (дата звернення: 10.06.2025).

17. Wang Y., He H., Wen C, Tan X. Truly Proximal Policy Optimization. *Machine Learning*. 2019. URL: https://doi.org/10.48550/arXiv.1903.07940 (дата звернення: 10.06.2025).

18. Luo F.-M., Xu T., Lai H., Chen X.-H. A Survey on Model-based Reinforcement Learning. *Machine Learning*. 2022. URL: https://doi.org/10.48550/arXiv.2206.09328 (дата звернення: 10.06.2025).

19. Deng Z., Jiang J., Long G., Zhang C. Causal Reinforcement Learning: A Survey. *Machine Learning*. 2023. URL: https://doi.org/10.48550/arXiv.2307.01452 (дата звернення: 10.06.2025).

20. Avval S., Eskue N., Groves R., Yaghoubi V. *Artificial Intelligence Review*. 2025. URL: https://doi.org/10.1007/s10462-024-11058-w (дата звернення: 10.06.2025).