НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО"
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

До захисту допущено:

В. о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«___»_____20__ р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи і методи штучного інтелекту»
спеціальності 122 «Комп'ютерні науки»
на тему: «Структурно-параметричний синтез згорткових нейронних мереж в обробці зображень з БПЛА»

Виконав:
студент IV курсу, групи КІ-21
Іванов Андрій Олексійович _____

Керівник:
асистент кафедри ШІ
Кот Анатолій Тарасович _____

Консультант з нормоконтролю:
фахівець першої категорії кафедри ШІ, к.т.н., доцент
Комариста Богдана Миколаївна _____

Рецензент:
доцент кафедри АКІК ДУ «Київський авіаційний інститут»,
к.т.н., доцент, Василенко Микола Павлович _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 122 «Комп'ютерні науки»
Освітньо-професійна програма «Системи і методи штучного інтелекту»

ЗАТВЕРДЖУЮ
В. о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«___» _____ 2026 р.

ЗАВДАННЯ
на дипломну роботу студенту
Іванову Андрію Олексійовичу

1. Тема роботи «Структурно-параметричний синтез згорткових нейронних мереж в обробці зображень з БПЛА», керівник роботи Кот Анатолій Тарасович, затверджені наказом по НН ІПСА від «27» травня 2026 р. №1944-с.

2. Термін подання студентом роботи «10» червня 2026 року.

3. Вихідні дані до роботи: датасет Military and Civilian Vehicles Classification (6 702 тренувальних та 496 тестових зображень, 6 класів); наукові публікації з методів Neural Architecture Search (DARTS, ENAS, AmoebaNet, Genetic CNN); фреймворки: Python 3.8+, TensorFlow/Keras 2.13.0; обчислювальні ресурси: NVIDIA L4 GPU (24 GB).

4. Зміст роботи: дослідження предметної області; математичні та теоретичні основи; проектування, розробка та експериментальні дослідження.

5. Перелік ілюстративного матеріалу: структура хромосоми; блок-схема ГА; діаграма компонентів системи; графіки динаміки еволюції; Confusion Matrix.

6. Дата видачі завдання «02» лютого 2026 року.

Календарний план

№ з/п	*Назва етапів виконання дипломної роботи*	*Термін виконання*	*Примітка*
1	Аналіз літератури та формування теоретичної бази	20.04.2026	Виконано
2	Розробка методу кодування архітектури	27.04.2026	Виконано
3	Реалізація генетичних операторів	04.05.2026	Виконано
4	Розробка системи автоматичного пошуку	11.05.2026	Виконано
5	Підготовка датасету та налаштування середовища	18.05.2026	Виконано
6	Проведення експериментів на GPU-сервері	25.05.2026	Виконано
7	Аналіз результатів та валідація відтворюваності	01.06.2026	Виконано
8	Оформлення пояснювальної записки	05.06.2026	Виконано

Студент Андрій ІВАНОВ

Керівник Анатолій КОТ

РЕФЕРАТ

Дипломна робота: 146 с., 6 рис., 46 табл., 51 посилання, 1 додаток.

Ключові слова: ПОШУК АРХІТЕКТУРИ НЕЙРОННИХ МЕРЕЖ, ГЕНЕТИЧНИЙ АЛГОРИТМ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, КЛАСИФІКАЦІЯ ВІЙСЬКОВОЇ ТЕХНІКИ, ЕВОЛЮЦІЙНІ ОБЧИСЛЕННЯ, ГЛИБОКЕ НАВЧАННЯ.

Об’єктом дослідження є процес автоматичного пошуку оптимальної архітектури згорткових нейронних мереж для задач класифікації зображень.

Предметом дослідження є методи еволюційної оптимізації архітектур згорткових нейронних мереж з одночасним підбором структурних компонентів та гіперпараметрів навчання для задачі класифікації військової техніки.

Мета роботи – розробити метод автоматичного пошуку оптимальної архітектури згорткових нейронних мереж для класифікації військової техніки на зображеннях з безпілотних літальних апаратів із застосуванням генетичного алгоритму.

У роботі оглянуто та проаналізовано сучасні методи пошуку архітектури нейронних мереж (NAS) і еволюційні алгоритми. Розроблено метод уніфікованого кодування архітектури та гіперпараметрів у вигляді хромосоми, адаптивну функцію пристосованості з урахуванням перенавчання, а також повний фреймворк відтворюваності результатів.

Експериментально підтверджено ефективність методу на датасеті Military and Civilian Vehicles Classification: отримано точність 98,59 % за 30 хвилин обчислень на NVIDIA L4 GPU. Проаналізовано 128 моделей із 16 поколінь еволюції; похибка відтворюваності результатів становить 0,02 %.

Програмне забезпечення реалізовано мовою програмування Python із використанням фреймворку TensorFlow/Keras та оприлюднено з відкритим вихідним кодом.

ABSTRACT

Bachelor's thesis: 146 p., 6 fig., 46 tables, 51 reference, 1 appendix.

Keywords: NEURAL ARCHITECTURE SEARCH, GENETIC ALGORITHM, CONVOLUTIONAL NEURAL NETWORKS, MILITARY VEHICLE CLASSIFICATION, EVOLUTIONARY COMPUTATION, DEEP LEARNING.

The object of research is the process of automated search for the optimal architecture of convolutional neural networks for image classification tasks.

The subject of research is methods of evolutionary optimization of convolutional neural network architectures with simultaneous selection of structural components and training hyperparameters for the task of military vehicle classification.

The purpose of the work is to develop a method for the automated search of an optimal convolutional neural network architecture for classifying military vehicles in unmanned-aerial-vehicle imagery using a genetic algorithm.

The thesis reviews and analyses modern neural architecture search (NAS) methods and evolutionary algorithms. A method of unified encoding of the architecture and hyperparameters as a chromosome, an adaptive fitness function accounting for overfitting, and a complete framework for result reproducibility were developed.

The effectiveness of the method was experimentally confirmed on the Military and Civilian Vehicles Classification dataset: an accuracy of 98.59% was achieved within 30 minutes of computation on an NVIDIA L4 GPU. A total of 128 models from 16 generations of evolution were analysed; the reproducibility error of the results is 0.02%.

The software is implemented in the Python programming language using the TensorFlow/Keras framework and released as open-source software.

ЗМІСТ

ВСТУП

Автоматичне розпізнавання військової техніки на супутникових знімках та зображеннях з дронів є критичною задачею для сучасних систем безпеки й оборони, адже точна та швидка класифікація військових об'єктів (танків, літаків, гелікоптерів, вантажівок) дозволяє своєчасно реагувати на загрози та приймати обґрунтовані тактичні рішення[1]. Водночас проєктування архітектури глибоких нейронних мереж для таких специфічних задач традиційно вимагає значних експертних знань у галузі комп'ютерного зору, глибокого навчання та військової розвідки, а сам процес ручного налаштування є не лише трудомістким, вимірюваним місяцями роботи, але й суб'єктивним, оскільки залежить від інтуїції та досвіду конкретного експерта.

Автоматизований підхід до вирішення цієї проблеми пропонує пошук архітектури нейронних мереж (Neural Architecture Search, NAS), проте більшість існуючих методів мають суттєві обмеження. Серед них – високі обчислювальні витрати, що сягають від сотень до тисяч GPU-годин, складність реалізації та потреба у спеціалізованому обладнанні, обмежений простір пошуку, який охоплює лише архітектуру без гіперпараметрів навчання, а також низька відтворюваність результатів. Це робить актуальною розробку доступного та ефективного методу автоматичного пошуку архітектури, що поєднував би помірні обчислювальні вимоги, гнучкий простір пошуку, високу точність для критичних застосувань і повну відтворюваність експериментів.

---

[1] Тут і нижче використано інструменти штучного інтелекту (зокрема чат-бот з генеративним штучним інтелектом ChatGPT та ін.) виключно для корегування та редагування тексту, створеного автором цієї дипломної роботи, на основі автоматизованої перевірки граматики, структури та стилю, що відповідає Політиці використання штучного інтелекту для академічної діяльності в КПІ ім. Ігоря Сікорського (протокол №11 Вченої ради КПІ ім. Ігоря Сікорського від 11 грудня 2023 р.).

Роботу виконано на кафедрі штучного інтелекту Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» у межах наукових досліджень кафедри в галузі автоматизованого машинного навчання та еволюційних обчислень. Дослідження пов'язане з науковою тематикою «Розробка інтелектуальних систем підтримки прийняття рішень» і відповідає напрямку «Прикладний штучний інтелект у критичних системах».

Метою роботи є розробка та експериментальна апробація методу автоматичного пошуку архітектури згорткових нейронних мереж на основі генетичного алгоритму з одночасною оптимізацією структурних та параметричних характеристик для задачі класифікації військової техніки на знімках з БПЛА. Об'єктом дослідження виступає процес автоматичного пошуку оптимальної архітектури згорткових нейронних мереж для задач класифікації зображень, а предметом – методи еволюційної оптимізації таких архітектур з одночасним підбором структурних компонентів та гіперпараметрів навчання. Для досягнення поставленої мети у роботі застосовано генетичні алгоритми для еволюційного пошуку архітектури та гіперпараметрів, методи глибокого навчання для побудови й тренування мереж, методи комп'ютерного зору для обробки та класифікації зображень, а також статистичний і порівняльний аналіз для оцінювання результатів і зіставлення розробленого методу з існуючими підходами.

Для досягнення мети послідовно вирішено низку завдань. Спершу проаналізовано існуючі методи пошуку архітектури нейронних мереж, виявлено їх переваги й недоліки та досліджено еволюційні підходи до оптимізації, що дало змогу обґрунтувати вибір генетичного алгоритму. Далі розроблено метод кодування архітектури у вигляді хромосоми зі структурними та параметричними елементами, реалізовано систему валідації архітектур з автоматичним виправленням невалідних конфігурацій, створено адаптивну функцію пристосованості з урахуванням динаміки навчання та

запобіганням перенавчанню, а також адаптовано генетичні оператори селекції, кросовера, мутації й елітизму до пошуку у просторі архітектур і гіперпараметрів. На основі цих рішень спроєктовано та реалізовано програмну систему з підтримкою Docker і GPU, проведено експериментальне дослідження на датасеті Military and Civilian Vehicles Classification, проаналізовано результати еволюційного процесу та виявлено оптимальні архітектурні патерни й гіперпараметри. Нарешті, реалізовано систему збереження повної історії еволюції та незалежної валідації результатів для забезпечення відтворюваності й виконано порівняння розробленого методу з існуючими підходами за критеріями точності, обчислювальної ефективності та відтворюваності.

Наукова новизна одержаних результатів полягає насамперед у тому, що вперше запропоновано уніфіковане генетичне кодування, яке в єдиному еволюційному фреймворку спільно представляє архітектуру згорткової мережі – типи шарів, їх параметри та зв'язність – і гіперпараметри навчання, зокрема оптимізатор, темп навчання й розмір пакета, уможливлюючи коеволюцію архітектурних та навчальних конфігурацій і виявлення нетривіальних залежностей між ними. Крім того, розроблено адаптивну функцію пристосованості з механізмом вибору найкращої епохи та адаптивною ранньою зупинкою, яка оцінює архітектури на піку їх валідаційної продуктивності й тим самим запобігає перенавчанню, зберігаючи чутливість до якості архітектури. Реалізовано також повний фреймворк відтворюваності зі збереженням усіх натренованих моделей разом з вагами, що дозволяє незалежну верифікацію результатів і забезпечує похибку відтворення лише 0,02 % – на два порядки нижчу за стохастичність тренування. Нарешті, запропонований підхід знижує обчислювальні вимоги на два–чотири порядки, потребуючи близько 0,5 GPU-години замість 0,5–3 150 GPU-днів у відомих методах, що робить пошук архітектури доступним навіть для академічних досліджень на одній відеокарті.

Практичне значення одержаних результатів виявляється насамперед для систем безпеки та оборони, де досягнута точність класифікації військової техніки у 98,59 % є придатною для реальних застосувань, а 30-хвилинний час пошуку та можливість розгортання на одній GPU дозволяють оперативно адаптувати систему до нових задач. Для науково-дослідних робіт цінними є повна відтворюваність результатів зі збереженням 128 моделей з вагами, відкритий вихідний код і докладна документація. У промислових застосуваннях метод автоматизує проєктування нейронних мереж, скорочуючи час розробки з місяців до годин і усуваючи потребу в ручному доборі гіперпараметрів, а в освітньому контексті він слугує наочним прикладом інженерного застосування генетичних алгоритмів та підходів AutoML. Результати роботи впроваджено у вигляді програмного забезпечення, опублікованого у відкритому репозиторії GitHub (https://github.com/okgoogle3/genetic_nas), і можуть бути використані для подальших досліджень у галузі Neural Architecture Search.

Усі результати, викладені в роботі, отримано автором особисто: проведено аналіз літератури з методів NAS та еволюційних алгоритмів, розроблено метод кодування архітектури й гіперпараметрів, реалізовано генетичні оператори та функцію пристосованості, спроєктовано й реалізовано програмну систему засобами Python і TensorFlow/Keras, підготовлено датасет, проведено повний цикл експериментів на GPU-сервері, виконано статистичний аналіз 128 моделей із 16 поколінь, реалізовано систему відтворюваності та валідації й створено документацію користувача. Основні результати роботи апробовано у вигляді пояснювальної записки та відкритого репозиторію, що містить повний вихідний код системи, документацію й результати експериментів.

РОЗДІЛ 1 СУЧАСНІ МЕТОДИ ПОШУКУ АРХІТЕКТУРИ НЕЙРОННИХ МЕРЕЖ ТА ОСОБЛИВОСТЕЙ КЛАСИФІКАЦІЇ ЗОБРАЖЕНЬ З БПЛА

1.1 Огляд методів пошуку архітектури нейронних мереж

Традиційний підхід до створення нейронних мереж передбачає ручне проєктування архітектури експертами в галузі глибокого навчання. Цей процес охоплює вибір кількості та типів шарів – згорткових, pooling і повноз'єднаних, – визначення параметрів кожного з них, як-от розмір фільтрів чи кількість нейронів, підбір гіперпараметрів навчання (learning rate, batch size, optimizer), а також багаторазове експериментування й поступове налаштування мережі.

Такому підходу властива низка недоліків. Він трудомісткий, адже процес може тривати від тижнів до місяців роботи досвідчених дослідників, і водночас суб'єктивний, оскільки результат значною мірою залежить від інтуїції та досвіду конкретного експерта. Простір пошуку при цьому залишається обмеженим, бо людина фізично не здатна перевірити всі можливі варіанти, а схильність експертів повторювати успішні патерни з попередніх задач додатково звужує його через когнітивні упередження.

Попри ці обмеження, історія глибокого навчання знає чимало вдалих ручних архітектур. Серед них – LeNet-5 (1998) [10], одна з перших успішних згорткових мереж для розпізнавання рукописних цифр; AlexNet (2012) [11], що став проривом у комп'ютерному зорі та переможцем ImageNet 2012; VGGNet (2014) [12], який продемонстрував переваги глибоких мереж із малими фільтрами; ResNet (2015) [13], що запровадив residual connections для навчання дуже глибоких мереж; Inception (2015) [14] з його multi-scale фільтрами у паралельних гілках; та DenseNet (2017) [15] зі щільними з'єднаннями між шарами. Проте кожна з цих архітектур вимагала місяців

досліджень та експериментів від команд провідних учених. Поява високорівневих бібліотек, як-от fastai [48], згодом спростила навчання таких мереж завдяки усталеним практикам, однак сам вибір оптимальної архітектури й надалі лишався ручним і трудомістким процесом.

Саме ці обмеження ручного підходу зробили потребу в автоматизації проєктування нейронних мереж дедалі гострішою з розвитком глибокого навчання. Сучасні мережі містять сотні шарів та мільйони параметрів, що неухильно ускладнює їх ручне налаштування, а різноманітність задач означає, що майже кожна нова з них може потребувати унікальної архітектури. До цього додаються обмежені людські ресурси – кваліфікованих експертів бракує для всіх можливих застосувань – і висока швидкість інновацій, яка вимагає швидкого прототипування й тестування нових ідей.

Відповіддю на ці виклики стала концепція автоматизованого машинного навчання (AutoML), яка передбачає автоматизацію всього конвеєра машинного навчання – від feature engineering і вибору моделі до оптимізації гіперпараметрів та пошуку архітектури. Пошук архітектури нейронних мереж (Neural Architecture Search, NAS) є підмножиною AutoML, що зосереджується саме на автоматичному визначенні оптимальної архітектури нейронної мережі.

Neural Architecture Search (NAS) можна формалізувати як задачу оптимізації:

$$\alpha^{*} = \underset{\alpha \in A}{arg\,max}\, \mathrm{ACC}\left(M(\alpha), D_{\mathrm{val}}\right),$$

де α – конфігурація архітектури; A – простір пошуку можливих архітектур; $M(\alpha)$ – модель з архітектурою α; D_{val} – валідаційний датасет; ACC – функція оцінки якості (accuracy).

Будь-який метод NAS складається з трьох ключових компонентів.

1. Простір пошуку (search space) визначає множину можливих архітектур.
2. Стратегія пошуку (search strategy) – алгоритм вибору наступної архітектури для оцінювання.
3. Оцінка продуктивності (performance estimation) – метод визначення якості архітектури.

Серед основних стратегій пошуку вирізняють кілька підходів.

1. Випадковий пошук (random search) полягає у виборі архітектур навмання. Він простий у реалізації, проте неефективний і потребує великої кількості спроб.
2. Методи на основі навчання з підкріпленням (reinforcement learning) навчають контролер генерувати архітектури. Вони дозволяють враховувати накопичений досвід, однак супроводжуються нестабільністю навчання та складністю реалізації.
3. Градієнтні методи (gradient-based methods) використовують градієнти для безпосередньої оптимізації архітектури. Вони забезпечують швидку конвергенцію, але обмежені диференційовними операціями.
4. Еволюційні методи (evolutionary methods) застосовують еволюційні алгоритми. Їм властива гнучкість та відсутність вимоги диференційовності ціною потенційно більших обчислювальних витрат.

1.2 Класифікація методів NAS

Метод DARTS, запропонований у праці [1], використовує градієнтну оптимізацію для пошуку архітектури. Його ключова ідея полягає в тому, що

замість дискретного вибору операцій застосовується неперервна релаксація простору пошуку, яка дозволяє оптимізувати архітектуру градієнтними методами:

$$\acute{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x),$$

де $\alpha_o^{(i,j)}$ – архітектурні параметри (learnable weights); O – множина можливих операцій; $o(x)$ – конкретна операція.

Пошук архітектури відбувається шляхом одночасної оптимізації ваг мережі та архітектурних параметрів за схемою дворівневої оптимізації (bi-level optimization): на нижньому рівні налаштовуються ваги мережі за фіксованих архітектурних параметрів, а на верхньому – самі архітектурні параметри за валідаційною вибіркою [33].

Серед переваг методу – висока швидкість пошуку (близько чотирьох GPU-днів проти тисяч у попередніх підходах), ефективне використання обчислювальних ресурсів та досягнення конкурентної точності (97,24 % на наборі CIFAR-10 [9]). Водночас DARTS має й суттєві обмеження: він застосовний лише до диференційованих операцій, може демонструвати нестабільність через дворівневу оптимізацію та погано масштабується на великі простори пошуку.

Метод ENAS [3] ґрунтується на навчанні з підкріпленням зі спільним використанням ваг (weight-sharing):

$$\text{Shared Weights: } W = \{w_1, w_2, \ldots, w_n\}.$$

Його ключова ідея полягає в тому, що замість тренування кожної архітектури з нуля всі архітектури використовують спільні ваги в межах

єдиної «супермережі». Керуючий модуль (контролер) на основі рекурентної нейронної мережі генерує архітектури, дочірні мережі тренуються з використанням спільних ваг, після чого контролер оновлюється за валідаційною точністю згідно з алгоритмом REINFORCE.

Такий підхід забезпечує значне прискорення пошуку – приблизно у 1000 разів швидше за NASNet [2] – і знижує обчислювальні витрати до 0,5 GPU-дня, досягаючи точності 97,11 % на CIFAR-10. До недоліків методу належать зміщення апроксимації (approximation bias), спричинене спільним використанням ваг, нестабільність навчання контролера та можлива неоптимальність фінальної архітектури саме через спільні ваги.

Метод NASNet [2] – один із перших успішних методів NAS, що використовує навчання з підкріпленням. В його основі лежать три компоненти: керуючий модуль (контролер) на основі рекурентної нейронної мережі, який генерує опис архітектури; дочірня мережа (child network), що відповідає згенерованій архітектурі; та сигнал винагороди (reward), яким слугує точність на валідаційному наборі.

Процес навчання має ітеративний характер [17]. Контролер генерує архітектуру A, дочірня мережа навчається на наборі даних, після чого обчислюється винагорода R (валідаційна точність), а параметри контролера оновлюються методом градієнта стратегії (policy gradient):

$$\nabla_\theta J(\theta) = E\left[R(A) \cdot \nabla_\theta \log P(A|\theta)\right].$$

Метод досягає точності 97,40 % на CIFAR-10 [9], а при перенесенні на ImageNet [50] – 82,7 % top-1 accuracy [49]. Головною його вадою є надзвичайно високі обчислювальні витрати (близько 1800 GPU-днів), до яких додаються нестабільність навчання за схемою policy gradient та складність відтворення результатів.

Метод Genetic CNN [4] став першим успішним застосуванням генетичного алгоритму до задачі NAS. Архітектура кодується у вигляді хромосоми – бінарної матриці суміжності, що описує з'єднання між шарами в межах кожної стадії (stage) мережі:

$$S = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Як генетичні оператори використовуються турнірний відбір (tournament selection), кросовер бінарних матриць (binary matrix crossover) та мутація у вигляді випадкової інверсії біта (random bit flip). Метод досяг точності 92,90 % на CIFAR-10 [9]; за сучасними мірками це невисокий показник, проте на момент публікації він продемонстрував принципову придатність еволюційного підходу до пошуку архітектур.

Метод AmoebaNet [5], відомий також як Regularized Evolution, є вдосконаленою версією еволюційного підходу до NAS. Від класичного генетичного алгоритму він відрізняється трьома ключовими рисами: використанням відбору на основі «віку» особин (age-based selection), за якого популяція періодично омолоджується; турнірним відбором із залученням найстаріших особин; та відмовою від кросовера на користь виключно мутацій (mutation-only).

Основний цикл методу можна подати таким псевдокодом:

```
population = initialize_population()
history = []

for _ in range(num_iterations):
 sample = random.sample(population, tournament_size)
 parent = max(sample, key=fitness)
```

```
child = mutate(parent)
fitness(child)
population.append(child)
history.append(child)
if len(population) > population_size:
population.pop(0)  # Remove oldest
```

Метод досяг точності 97,87 % на CIFAR-10 [9] – найкращого результату серед методів NAS на момент публікації – та 83,9 % top-1 accuracy на ImageNet [50] за обчислювальних витрат близько 3150 GPU-днів. До переваг еволюційного підходу загалом належать гнучкість (він не обмежений диференційованими операціями), природна паралелізація (можливість одночасно оцінювати кілька архітектур) та інтерпретовність (змога аналізувати сам перебіг еволюції). Його основними недоліками лишаються високі обчислювальні витрати за відсутності спільного використання ваг і потреба у великій популяції для ефективного пошуку.

Таблиця 1.1 – Порівняння методів NAS

Метод	*Підхід*	*GPU-час*	*CIFAR-10 Accuracy*	*Особливості*
NASNet	RL	1,800 днів	97.40%	Перший успішний RL метод
AmoebaNet	Evolution	3,150 днів	97.87%	Regularized evolution
DARTS	Gradient	4 дні	97.24%	Differentiable search
ENAS	RL + sharing	0.5 дня	97.11%	Weight sharing
Genetic CNN	Evolution	-	92.90%	Перший еволюційний NAS

Зіставлення розглянутих методів виявляє три основні закономірності. По-перше, простежується чіткий компроміс між швидкістю та якістю пошуку: швидкі методи (DARTS, ENAS) знаходять архітектуру за лічені дні, забезпечуючи точність близько 97,2 %, тоді як повільні (AmoebaNet) вимагають тисяч GPU-днів, але дають дещо вищу точність – близько 97,9 %. По-друге, проявляється протиставлення спільного використання ваг і повного тренування: зміщення апроксимації (approximation bias) у ENAS може призводити до субоптимальних результатів, натомість повне тренування кожної архітектури (AmoebaNet) дає кращу якість ціною значно більших обчислювальних витрат. По-третє, методи різняться за відтворюваністю: градієнтні підходи (DARTS) демонструють вищу відтворюваність результатів, тоді як методи на основі навчання з підкріпленням відтворити значно складніше [37, 38].

1.3 Еволюційні алгоритми в машинному навчанні

Генетичні алгоритми (ГА) – це метаевристичні методи оптимізації, інспіровані природним відбором та біологічною еволюцією [23, 24, 29]. Концептуально вони спираються на пряму аналогію між поняттями біології та елементами алгоритму, наведену в таблиці 1.2.

Основний цикл генетичного алгоритму складається з таких кроків.

1. Ініціалізація: створення початкової популяції.
2. Оцінка: обчислення фітнесу для кожної особини.
3. Цикл, що повторюється до досягнення критерію зупинки:

1) селекція – вибір батьківських особин для розмноження;
2) кросовер – створення нащадків шляхом комбінування батьків;
3) мутація – випадкова зміна нащадків;

4) оцінка – обчислення фітнесу нащадків;
5) формування нового покоління.

4. Повернення найкращого знайденого рішення.

Таблиця 1.2 – Основи генетичних алгоритмів

Біологія	*Генетичний алгоритм*
Організм	Особина (архітектура)
Ген	Параметр архітектури
Хромосома	Повна специфікація архітектури
Фітнес	Accuracy на валідації
Покоління	Ітерація алгоритму
Популяція	Набір архітектур

Математично популяція в поколінні t описується як

$$P_t = \{ x_1^t, x_2^t, \dots, x_n^t \}.$$

Fitness функція:

$$f : X \to \mathbb{R}.$$

Ймовірність селекції (roulette wheel):

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} f(x_j)}.$$

Генетичне програмування (ГП) – це розширення генетичного алгоритму, призначене для еволюції програм і структур [26]. На відміну від класичного ГА, воно оперує представленнями змінної довжини; основні відмінності між двома підходами узагальнено в таблиці 1.3.

Таблиця 1.3 – Генетичне програмування

Аспект	*Генетичний алгоритм*	*Генетичне програмування*
Представлення	Фіксована довжина	Змінна довжина
Структура	Лінійна	Деревоподібна
Простір пошуку	Векторний	Структурний
Застосування	Оптимізація параметрів	Еволюція програм/структур

Генетичне програмування природно підходить для задачі NAS, оскільки архітектура нейронної мережі є структурованим об'єктом змінної довжини, що добре узгоджується з деревоподібним представленням ГП [27].

Еволюційні підходи мають тривалу історію застосування до оптимізації нейронних мереж. Одним із перших став метод NEAT (NeuroEvolution of Augmenting Topologies) [18], що дозволяв одночасно еволюціонувати топологію та ваги мережі й застосовувався переважно до задач навчання з підкріпленням. Його розвитком став HyperNEAT, у якому еволюціонували не самі мережі, а правила їх генерації на основі мереж, що породжують композиційні патерни (compositional pattern producing networks); це дало змогу будувати значно більші структури [25]. Згодом підхід CoDeepNEAT [28] поширив цю ідею на глибокі згорткові мережі завдяки коеволюції модулів та їх з'єднань.

Сучасні еволюційні методи NAS (зокрема AmoebaNet [5] та EvNAS [8]) поєднують класичні генетичні алгоритми з новими техніками: спільним

використанням ваг (weight-sharing) для прискорення пошуку, багатокритеріальною оптимізацією (multi-objective optimization) [6] та коеволюцією різних компонентів архітектури.

1.4 Задача класифікації військової техніки на зображеннях

Автоматичне розпізнавання військової техніки є критично важливою задачею як для оборонних, так і для цивільних застосувань. У військовій сфері воно забезпечує раннє виявлення загроз на кордонах, моніторинг переміщень військ, автоматичну ідентифікацію техніки в зонах конфлікту та підтримку прийняття тактичних рішень. У цивільній площині такі технології використовуються для моніторингу дотримання міжнародних угод про роззброєння, аналізу відкритих супутникових знімків і журналістських розслідувань військових конфліктів.

Водночас задача супроводжується низкою специфічних технічних викликів. Передусім це висока вартість помилок: неправильна класифікація може мати критичні наслідки. Крім того, військові набори даних зазвичай обмежені за розміром і доступністю, що ускладнює навчання моделей. Додаткові вимоги ставлять необхідність ухвалювати рішення в реальному часі та потреба в адаптивності, оскільки нові зразки техніки з'являються постійно.

Для розв'язання поставленої задачі використано набір даних Military and Civilian Vehicles Classification. Він містить шість класів – цивільні літаки, цивільні автомобілі, військові літаки, військові гелікоптери, військові танки та військові вантажівки – і налічує 6702 тренувальних та 496 тестових зображень. Зображення подано у форматі RGB, після попередньої обробки (preprocessing) приведено до розміру 64×64 пікселі.

За своїми характеристиками цей набір помітно відрізняється від стандартних тестових датасетів комп'ютерного зору, таких як CIFAR-10 [9] та ImageNet [50], що ілюструє таблиця 1.4.

Таблиця 1.4 – Особливості датасетів військової техніки

Характеристика	*CIFAR-10*	*ImageNet*	*Military Vehicles*
Кількість класів	10	1,000	6
Тренувальні зображення	50,000	1,200,000	6,702
Розмір зображень	32×32	224×224	64×64
Складність	Середня	Висока	Висока
Збалансованість	Так	Ні	Помірна

Задача класифікації військової техніки ускладнюється кількома чинниками. По-перше, їй властива висока внутрішньокласова варіативність: техніка одного типу може суттєво різнитися моделями та модифікаціями (наприклад, Т-72, Т-80 і Т-90 – усі вони належать до танків), камуфляжем і забарвленням, ракурсами зйомки (згори, збоку, спереду) та умовами освітлення.

По-друге, спостерігається мала міжкласова відстань – візуальна схожість між різними категоріями: цивільними та військовими вантажівками, різними типами літаків, військовими й цивільними гелікоптерами.

По-третє, класифікацію утруднюють складні умови зйомки: різні висота та роздільна здатність, часткове перекриття об'єктів, неоднорідні фони (міська забудова, ліси) і погодні явища (хмари, дим).

Нарешті, суттєвим чинником є обмежений розмір набору даних: 6702 тренувальних зображення – значно менше, ніж у ImageNet [50], що підвищує

ризик перенавчання та ускладнює узагальнення моделі на нові зразки техніки.

У сучасній практиці класифікації військової техніки застосовують кілька підходів, кожен із яких має свої обмеження.

Перший підхід – перенесення навчання (transfer learning) із ImageNet [50], за якого попередньо натреновані мережі на кшталт ResNet [13] чи VGG [12] донавчаються (fine-tuning) на даних військової техніки. Його перевагами є невибагливість до обсягу власного датасету та швидкість впровадження. Проте такі мережі попередньо навчені на загальних об'єктах, а не на військовій техніці; їхня архітектура фіксована й може виявитися неоптимальною, а гіперпараметри доводиться добирати вручну.

Другий підхід – ручне проектування спеціалізованої архітектури, коли експерти розробляють згорткову мережу безпосередньо під задачу. Це дає змогу врахувати її специфіку та контролювати складність моделі, однак потребує місяців роботи й експертизи одночасно у двох галузях (машинне навчання та військова техніка), а через обмеженість ручного пошуку зберігається ризик субоптимальності.

Третій підхід – доповнення даних (data augmentation) через трансформації наявних зображень. Він простий у впровадженні та покращує узагальнювальну здатність моделі, але не розв'язує проблему вибору архітектури й має обмежену ефективність для дуже малих наборів даних.

Наведені обмеження обґрунтовують доцільність застосування автоматичного пошуку архітектури (NAS) для військових застосувань. Такий підхід дозволяє знайти архітектуру, оптимальну саме для специфіки військової техніки, врахувати обмежений розмір датасету завдяки коеволюції з гіперпараметрами, забезпечити швидку адаптацію до нових типів техніки та оптимізувати компроміс між точністю і швидкістю інференсу.

1.5 Проблеми сучасних методів NAS

Одним із головних обмежень сучасних методів NAS є їхні високі обчислювальні вимоги. Зіставлення витрат для основних методів, включно з орієнтовною вартістю оренди обчислювальних потужностей, наведено в таблиці 1.5.

Таблиця 1.5 – Високі обчислювальні вимоги

Метод	*GPU-години*	*Вартість (AWS p3.2xlarge @ $3.06/год)*
NASNet	43,200	$132,192
AmoebaNet	75,600	$231,336
DARTS	96	$294
ENAS	12	$37
Потреба для академії	**< 24**	**< $75**

Такі витрати зумовлені трьома основними причинами. Передусім це повне тренування кожної архітектури: наприклад, для AmoebaNet оцінка 20 000 архітектур по 50 епох кожна дає близько 1 000 000 тренувальних циклів. Додатковим чинником є великі набори даних – так, ImageNet [50] містить 1,2 млн зображень, а одна епоха на відеокарті V100 триває близько години. Нарешті, еволюційні та пошукові методи потребують багатьох поколінь – для збіжності зазвичай необхідно понад сто ітерацій.

Наслідком цього є фактична недоступність NAS для більшості академічних груп: дозволити собі такі експерименти можуть лише великі компанії (Google, Microsoft), що уповільнює інновації в галузі загалом.

Другою серйозною проблемою є низька відтворюваність результатів NAS, яка має кілька джерел. Перше – стохастичність процесу: випадкова

ініціалізація ваг, випадкове семплування в методах на основі навчання з підкріпленням та випадкове перемішування даних. Друге – питання налаштування гіперпараметрів: публікації часто не розкривають усіх використаних значень, а результати подають за принципом «найкращий із N запусків» без належної статистичної звітності. Третє – відмінності в обчислювальному середовищі: різні відеокарти, версії бібліотек і налаштування помітно впливають на кінцеві результати.

Показовим є дослідження відтворюваності DARTS, у якому Yang et al. [37] продемонстрували високу варіативність методу: точність коливалася від 96,5 % до 97,5 % (різниця близько 1 %), а знайдені архітектури суттєво відрізнялися між окремими запусками.

Для подолання цих проблем методи NAS мають оприлюднювати повну історію еволюції (усі досліджені архітектури), натреновані ваги моделей, детальні логи експериментів та значення початкових станів генератора випадкових чисел (seed values), потрібні для відтворення [35, 36].

Третім обмеженням сучасних методів NAS є вузькість простору пошуку, що проявляється на кількох рівнях. По-перше, більшість методів фіксують гіперпараметри, оптимізуючи лише архітектуру: швидкість навчання, розмір батча та оптимізатор задаються вручну [45], через що втрачається можливість знайти оптимальні комбінації архітектури й параметрів навчання. По-друге, обмеженим є набір доступних операцій – так, DARTS [1] оперує лише вісьмома операціями (згортки 3×3 і 5×5, пулінг, skip-з'єднання) і не охоплює варіанти з батч-нормалізацією [19] чи різні функції активації. По-третє, фіксованою лишається макроструктура мережі: підходи на основі комірок (cell-based) задають незмінну кількість комірок та накладають обмеження на глибину мережі.

Масштаб цього обмеження ілюструє приклад DARTS: за оцінкового розміру простору пошуку близько 10^{18} архітектур реально оцінюється лише близько 1000 з них, що відповідає покриттю порядку 10^{-15} %.

Четвертою проблемою є відсутність одночасної оптимізації архітектури та гіперпараметрів навчання, які насправді є взаємозалежними. Так, глибокі мережі потребують менших значень швидкості навчання, великі розміри батча можуть вимагати поступового її нарощування (learning rate warmup) [45], а різні оптимізатори (Adam [21] порівняно зі SGD [43]) по-різному узгоджуються з різними архітектурами.

Показовим є приклад, де одна й та сама архітектура показала точність 96,5 % з оптимізатором Adam і 97,2 % зі SGD з моментом – різниця у 0,7 % зумовлена виключно вибором оптимізатора [15]. Це свідчить, що оптимізація лише архітектури без урахування гіперпараметрів навчання може призводити до субоптимальних результатів.

Висновки до розділу 1

1. Проаналізовано сучасні методи автоматичного пошуку архітектури нейронних мереж. Виявлено три основні підходи: градієнтні методи (DARTS), методи на основі навчання з підкріпленням (NASNet, ENAS) та еволюційні методи (AmoebaNet). Кожен підхід має власні переваги та недоліки у контексті обчислювальної ефективності, якості знайдених архітектур та відтворюваності результатів.
2. Досліджено застосування еволюційних алгоритмів у машинному навчанні. Генетичні алгоритми продемонстрували гнучкість та здатність до паралелізації, що робить їх привабливими для задач NAS, незважаючи на потенційно вищі обчислювальні витрати порівняно з градієнтними методами.
3. Проаналізовано специфіку задачі класифікації військової техніки. Виявлено ключові виклики: високу внутрішньокласову

варіативність, малу міжкласову відстань, обмежений розмір датасетів та високі вимоги до точності для критичних застосувань.

4. Виявлено чотири ключові проблеми сучасних методів NAS: (1) надмірно високі обчислювальні вимоги (до 3,150 GPU-днів), що робить NAS недоступним для академічних досліджень; (2) низька відтворюваність результатів через стохастичність та неповне розкриття деталей експериментів; (3) обмежений простір пошуку, що не включає гіперпараметри навчання; (4) відсутність одночасної оптимізації архітектури та гіперпараметрів, що може призводити до субоптимальних рішень.
5. Обґрунтовано необхідність розробки методу NAS, який би поєднував: помірні обчислювальні вимоги (< 1 GPU-дня), повну відтворюваність результатів (збереження всіх моделей), розширений простір пошуку (архітектура + гіперпараметри) та придатність для задач з обмеженими даними (Military Vehicles).

РОЗДІЛ 2 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ СТРУКТУРНО-ПАРАМЕТРИЧНОГО СИНТЕЗУ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ

2.1 Основи згорткових нейронних мереж

Згорткові нейронні мережі (Convolutional Neural Networks, CNN) є класом глибоких нейронних мереж, які спеціалізуються на обробці даних з відомою сіткоподібною структурою, зокрема зображень. На відміну від повнозв'язних нейронних мереж, CNN використовують спеціалізовані архітектурні елементи, які враховують просторову структуру вхідних даних та забезпечують інваріантність до локальних трансформацій.

Згортковий шар (Convolutional Layer) є фундаментальним будівельним блоком згорткової нейронної мережі. Основна ідея полягає в застосуванні операції згортки (convolution) до вхідних даних з метою виділення локальних просторових ознак.

Нехай $X \in \mathbb{R}^{H \times W \times C}$ - вхідний тензор, де H - висота, W - ширина, C - кількість каналів (для RGB зображень $C = 3$). Згортковий шар застосовує набір фільтрів (ядер) $K_i \in \mathbb{R}^{k \times k \times C}$, де k - розмір ядра (типово $k = 3$ або $k = 5$).

Операція згортки для одного фільтра визначається як:

$$Y_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c=0}^{C-1} X_{i+m, j+n, c} \cdot K_{m,n,c} + b, \tag{2.1}$$

де $Y_{i,j}$ – значення у вихідній карті ознак (feature map) в позиції (i, j); b – bias (зміщення), (m, n) – координати в межах ядра згортки.

Згорткове ядро можна розглядати як шаблон (pattern), який ковзає по зображенню з певним кроком (stride). У кожній позиції обчислюється

скалярний добуток між значеннями пікселів та вагами ядра. Якщо локальна область зображення схожа на шаблон, активація буде високою.

Існують такі параметри згорткового шару.

1. Кількість фільтрів (filters): визначає глибину вихідного тензора та кількість різних ознак, які може виявити шар. У даній роботі використовуються значення $\{32,64,128\}$.
2. Розмір ядра (kernel size): визначає рецептивне поле - область вхідного зображення, яка впливає на одне значення у вихідній карті ознак. Найчастіше використовується 3×3.
3. Крок згортки (stride): визначає, на скільки пікселів зміщується ядро при кожному кроці. При $\text{stride}=1$ ядро зміщується на один піксель.
4. Padding: доповнення країв нульовими значеннями для збереження просторових розмірів. При padding=same розмір виходу дорівнює розміру входу (за умови $\text{stride}=1$).

Наприклад, якщо вхідне зображення має розмір $64 \times 64 \times 3$, а згортковий шар містить 32 фільтри розміром 3×3 з $\text{stride}=1$ та padding=same, то розмір виходу буде $64 \times 64 \times 32$.

Згорткові шари мають низку важливих переваг. Завдяки локальній зв'язності кожен нейрон з'єднаний лише з невеликою областю попереднього шару, що значно зменшує кількість параметрів порівняно з повнозв'язними шарами. Спільне використання ваг означає, що те саме ядро застосовується до всього зображення, забезпечуючи інваріантність до трансляції (зміщення об'єкта на зображенні не змінює його розпізнавання). Нарешті, згорткові мережі виконують ієрархічне виділення ознак: перші шари виявляють прості ознаки (краї, кути), а глибші – складніші (текстури, частини об'єктів).

Pooling шари (шари підвибірки або пулінгу) виконують операцію зменшення просторової розмірності (downsampling) карт ознак при збереженні найбільш важливої інформації. Це дозволяє:

1) зменшити кількість параметрів та обчислювальну складність;

2) забезпечити інваріантність до невеликих трансляцій;

3) розширити рецептивне поле наступних шарів.

Max Pooling це найпоширеніший тип пулінгу, який обирає максимальне значення в кожній локальній області:

$$Y_{i,j} = \max_{(m,n) \in R_{i,j}} X_{m,n}, \tag{2.2}$$

де $R_{i,j}$ – прямокутна область розміру $p \times p$ (типово $p=2$) з центром у позиції (i,j).

Нехай маємо карту ознак розміром 4×4:

$$\begin{matrix}[1 & 3 & 2 & 4] \\ [5 & 6 & 7 & 8] \\ [2 & 1 & 3 & 9] \\ [0 & 2 & 4 & 1]\end{matrix} \rightarrow \begin{matrix}[3 & 4] \\ [8 & 9]\end{matrix}$$

Після застосування Max Pooling з розміром 2×2 та $\text{stride}=2$ отримуємо карту розміром 2×2, де кожне значення - це максимум з відповідної області 2×2 вхідної карти.

Average Pooling це альтернативний підхід, який обчислює середнє значення:

$$Y_{i,j} = \frac{1}{p^2} \sum_{(m,n) \in R_{i,j}} X_{m,n}. \tag{2.3}$$

Max Pooling зазвичай працює краще для задач класифікації, оскільки зберігає найбільш яскраві ознаки, тоді як Average Pooling краще для задач, де важлива загальна текстура.

У даній роботі використовується обмеження: максимум 3 шари MaxPool для зображень розміром 64×64 пікселі. Це пов’язано з тим, що кожен MaxPool шар зменшує розмір удвічі: $64 \rightarrow 32 \rightarrow 16 \rightarrow 8$. Після трьох

MaxPool операцій карти ознак мають розмір 8×8, що є достатнім для подальшої обробки. Четвертий MaxPool призвів би до розміру 4×4, що занадто малий для ефективного виділення ознак.

Batch Normalization (нормалізація по батчу) є технікою, яка нормалізує активації кожного шару для кожного міні-батчу під час тренування. Це значно прискорює навчання та покращує стабільність глибоких нейронних мереж.

Для міні-батчу $B=\{x_1, x_2, \ldots, x_m\}$ Batch Normalization виконує наступні кроки.

1. Обчислення статистик батчу:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.4}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{2.5}$$

2. Нормалізація:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.6}$$

де ϵ – мала константа (типово 10^{-5}) для запобігання ділення на нуль.

3. Масштабування та зміщення:

$$y_i = \gamma \hat{x}_i + \beta \tag{2.7}$$

де γ та β - навчальні параметри, які дозволяють мережі відновити будь-яке представлення при необхідності.

Метод батч-нормалізації (Batch Normalization) має кілька важливих переваг [19]. Передусім він прискорює навчання, дозволяючи

використовувати більші значення швидкості навчання й тим самим пришвидшуючи збіжність. Водночас він зменшує внутрішній зсув коваріації (internal covariate shift), стабілізуючи розподіл активацій під час навчання. Використання статистик батча додає невелику кількість шуму, що дає додатковий регуляризаційний ефект (weak regularization). Нарешті, батч-нормалізація послаблює проблему зникаючих і вибухаючих градієнтів [22], завдяки чому стає можливою побудова глибших мереж.

У цій роботі шар BatchNorm може розміщуватися лише після згорткових (Conv2D) або повнозв'язних (Dense) шарів, оскільки він нормалізує активації з чітко визначеною статистикою. Типова послідовність шарів має вигляд: Conv2D → BatchNorm → Activation → MaxPool.

Dropout є простою, але ефективною технікою регуляризації, яка запобігає перенавчанню (overfitting) глибоких нейронних мереж. Основна ідея полягає у випадковому “вимиканні” (обнуленні) частини нейронів під час кожної ітерації навчання.

Під час навчання для кожного нейрона з активацією h_i генерується бінарна випадкова змінна $r_i \sim \text{Bernoulli}(p)$, де p - ймовірність збереження нейрона (зазвичай $p \in [0.5, 0.8]$):

$$\tilde{h}_i = r_i \cdot h_i \tag{2.8}$$

Під час інференсу (тестування) всі нейрони активні, але їх виходи масштабуються на p для збереження очікуваного значення активацій:

$$h_{\text{test}} = p \cdot h \tag{2.9}$$

Сучасні фреймворки (TensorFlow, PyTorch [42]) автоматично застосовують інвертоване масштабування під час тренування:

$$\tilde{h}_i = \frac{r_i \cdot h_i}{p}, \tag{2.10}$$

що усуває необхідність масштабування під час інференсу.

Dropout змушує мережу не покладатися на будь-який конкретний набір нейронів, розподіляючи інформацію по всій мережі. Це подібно до ефекту ансамблювання, де кожна ітерація навчання працює з дещо іншою підмережею.

Ключовим параметром методу Dropout [20] є коефіцієнт відсіву (dropout rate) – імовірність вимкнення окремого нейрона. У цій роботі використано діапазон $[0.2, 0.5]$; надто високі значення (понад 0,7) можуть призводити до недонавчання (underfitting).

У межах архітектури Dropout застосовується насамперед після повнозв'язних (Dense) шарів для запобігання перенавчанню класифікатора й рідше – після згорткових шарів, де в разі використання задають меншу ймовірність (близько 0,2–0,3).

Розглянемо приклад ефекту. Припустимо, Dense шар має 128 нейронів з активаціями $[0.5, 0.8, 0.3, \ldots, 0.9]$. При Dropout rate=0.3 приблизно 30% нейронів будуть обнулені, наприклад: $[0, 0.8, 0.3, 0, \ldots, 0.9]$.

Функції активації вносять нелінійність у нейронну мережу, дозволяючи їй апроксимувати складні нелінійні залежності. Без функцій активації багатошарова мережа була б еквівалентна одношаровій лінійній моделі.

Найпопулярнішою функцією активації в глибокому навчанні є ReLU (Rectified Linear Unit), яка визначається як:

$$\mathrm{ReLU}(x) = max(0, x) = \begin{cases} x, & \text{якщо } x \geq 0 \\ 0, & \text{якщо } x < 0 \end{cases}. \tag{2.11}$$

Градієнт ReLU:

$$\frac{\partial \mathrm{ReLU}(x)}{\partial x} = \begin{cases} 1, & \text{якщо } x > 0 \\ 0, & \text{якщо } x \le 0 \end{cases}. \tag{2.12}$$

Перевагами ReLU є обчислювальна простота (зводиться до порівняння з нулем), відсутність проблеми зникаючих градієнтів для додатних значень та емпірично кращі результати порівняно із сигмоїдою (sigmoid) і гіперболічним тангенсом (tanh) [11]. Її основним недоліком є проблема «вмирання» нейронів (dying ReLU): якщо нейрон постійно отримує від'ємні значення, його градієнт завжди дорівнює нулю, і він перестає навчатися.

Удосконаленою версією є функція ELU (Exponential Linear Unit), яка має ненульовий градієнт для від'ємних значень:

$$\mathrm{ELU}(x) = \begin{cases} x, & \text{якщо } x \ge 0 \\ \alpha(\exp(x) - 1), & \text{якщо } x < 0 \end{cases}, \tag{2.13}$$

де α - гіперпараметр (зазвичай $\alpha = 1.0$).

ELU усуває проблему «dying ReLU», наближає середнє значення активацій до нуля (що прискорює навчання) і є гладкішою функцією. Її недоліками є дещо повільніше обчислення через використання експоненти та підвищені вимоги до обчислювальних ресурсів.

Для вихідного шару в задачах багатокласової класифікації використовується Softmax:

$$\mathrm{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)}, \tag{2.14}$$

де K – кількість класів (у даній роботі $K = 6$ для Military Vehicles датасету).

Softmax перетворює вектор довільних дійсних чисел у вектор ймовірностей: $\sum_{i=1}^{K} \mathrm{Softmax}(x_i) = 1$.

У запропонованому підході для згорткових та повнозв'язних шарів використовуються або ReLU, або ELU, які обираються генетичним алгоритмом. Вихідний шар завжди використовує Softmax для отримання розподілу ймовірностей по класах.

2.2 Математична модель генетичного алгоритму

Генетичний алгоритм (ГА) – це метаевристичний метод оптимізації, заснований на принципах природного відбору та еволюції. ГА оперує популяцією потенційних рішень (особин), які еволюціонують протягом поколінь через механізми селекції, схрещування (кросовер) та мутації.

Простір пошуку S представляє всі можливі архітектури нейронних мереж, які можуть бути згенеровані в рамках заданих обмежень. Формально:

$$S = A \times H, \tag{2.15}$$

де A – простір архітектур (структура шарів); H – простір гіперпараметрів навчання.

Архітектура представляється як послідовність шарів:

$$A = \{L_1, L_2, \ldots, L_n\}, n \in [\text{MIN\_LAYERS}, \text{MAX\_LAYERS}], \tag{2.16}$$

де кожен шар L_i може бути одного з типів:

$$L_i \in \{\text{Conv2D}, \text{Dense}, \text{MaxPool}, \text{Dropout}, \text{BatchNorm}, \text{Flatten}\}. \tag{2.17}$$

Кожен тип шару характеризується власним набором параметрів.

1. Згортковий шар (Conv2D):

$$\text{Conv2D}(\text{filters}, \text{kernel\_size}, \text{activation}), \text{filters} \in \{32,64,128\}. \quad (2.18)$$

2. Повнозв'язний шар (Dense):

$$\text{Dense}(\text{neurons}, \text{activation}), \text{neurons} \in \{32,64,128,256\}. \quad (2.19)$$

3. Шар Dropout:

$$\text{Dropout}(\text{rate}), \text{rate} \in [0.2, 0.5]. \quad (2.20)$$

Гіперпараметри навчання включають:

$$H = \{\text{optimizer}, \text{learning\_rate}, \text{batch\_size}\}, \quad (2.21)$$

де $\text{optimizer} \in \{\text{adam}, \text{sgd}, \text{rmsprop}\}$, $\text{learning\_rate} \in [0.0001, 0.001]$, $\text{batch\_size} \in \{16,32,64\}$.

Для забезпечення валідності згенерованих мереж у середовищі Keras/TensorFlow [40, 41] застосовано такі структурні обмеження.

1. Початковий шар: $L_1 = \text{Conv2D}$(обов'язково).
2. Розміщення BatchNorm: може йти лише після шарів Conv2D або Dense.
3. Обмеження після Flatten: можуть розташовуватися лише шари Dense або Dropout.
4. Обмеження MaxPool: максимум три шари для зображень 64×64.
5. Обмеження utility-шарів: не більше двох однакових utility-шарів (BatchNorm, MaxPool, Dropout) поспіль.
6. Глибина: $n \in [3,5]$ шарів (без урахування вихідного).
7. Вихідний шар: $L_{\text{final}} = \text{Dense}(6, \text{softmax})$ (обов'язково).

Якщо не враховувати обмеження валідності, верхню межу кількості можливих архітектур оцінюють як:

$$|S| \approx |A| \times |H| = \left(\prod_{i=1}^{n} |L_i| \right) \times 3 \times 100 \times 3 \approx 10^{12}. \quad (2.22)$$

Навіть з урахуванням обмежень валідності, простір залишається надзвичайно великим ($>10^8$ унікальних архітектур), що унеможливлює повний перебір (exhaustive search) та обґрунтовує використання еволюційного підходу.

Хромосома (особина) в генетичному алгоритмі представляє повну специфікацію нейронної мережі, включаючи як структуру (архітектуру), так і параметри навчання (гіперпараметри). Таке комплексне кодування дозволяє одночасно оптимізувати обидва аспекти моделі.

Хромосома c визначається як кортеж:

$$c = (L, P), \quad (2.23)$$

де $L = [L_1, L_2, \ldots, L_n]$ - список шарів (архітектура); $P = (\text{opt}, \text{lr}, \text{bs})$ - гіперпараметри навчання.

Кожен шар L_i кодується як структура з полями:

$$L_i = \{\text{type}, \text{params}, \text{position}\}. \quad (2.24)$$

Розглянемо просту архітектуру.

1. Conv2D(64, 3×3, relu).
2. MaxPool(2×2).
3. Flatten().
4. Dense(128, relu).
5. Dropout(0.3).

6. Dense(6, softmax).

Це кодується як:

$$L=[\begin{matrix}\{\text{type: Conv2D}, \text{filters: } 64, \text{kernel\_size: } 3, \text{activation: relu}\},\\ \{\text{type: MaxPool}, \text{pool\_size: } 2\},\\ \{\text{type: Flatten}\},\\ \{\text{type: Dense}, \text{neurons: } 128, \text{activation: relu}\},\\ \{\text{type: Dropout}, \text{rate: } 0.3\},\\ \{\text{type: Dense}, \text{neurons: } 6, \text{activation: softmax}\}\end{matrix}]. \quad (2.25)$$

Гіперпараметри навчання P суттєво впливають на кінцеву точність моделі [45].

Learning rate (швидкість навчання) контролює розмір кроку оптимізації. Малі значення ($\sim 10^{-4}$) забезпечують стабільне, але повільне навчання; великі ($\sim 10^{-3}$) - швидке, але можуть спричинити нестабільність.

$$\text{lr} \in [0.0001, 0.001] \subset \mathbb{R}. \quad (2.26)$$

Batch size (розмір батчу) изначає кількість зразків, які обробляються перед оновленням ваг. Малі батчі (16) забезпечують більше оновлень та більше стохастичності (корисно для виходу з локальних мінімумів), але потребують більше часу. Великі батчі (64) дають стабільніші градієнти та швидше тренування, але можуть застрягнути в гострих мінімумах.

$$\text{bs} \in \{16, 32, 64\} \subset \mathbb{N}. \quad (2.27)$$

Оптимізатор (optimizer) задає алгоритм оптимізації. У роботі розглянуто три варіанти: Adam [21] – з адаптивною швидкістю навчання та швидкою збіжністю, що добре працює без додаткового налаштування; SGD [43] – класичний стохастичний градієнтний спуск з моментом, здатний

знаходити гостріші мінімуми з кращою генералізацією; та RMSprop – також з адаптивною швидкістю навчання, ефективний для рекурентних мереж.

$$\text{opt} \in \{\text{adam}, \text{sgd}, \text{rmsprop}\}. \tag{2.28}$$

Як приклад, найкраща модель серед усіх експериментів (gen7_model5) має таку хромосому:

$$c_{\text{best}} = \begin{pmatrix} L = [\,\text{Conv2D}(64,3,\text{relu}), \text{Flatten}(), \text{Dense}(6,\text{softmax})\,], \\ P = (\,\text{adam}, 0.000727, 16\,) \end{pmatrix}. \tag{2.29}$$

Ця мінімалістична архітектура досягла точності 98.59% на датасеті Military Vehicles, демонструючи, що ефективність залежить не тільки від складності архітектури, але й від правильного підбору гіперпараметрів. Загальну будову хромосоми та приклад найкращої особини показано на рисунку 2.1.

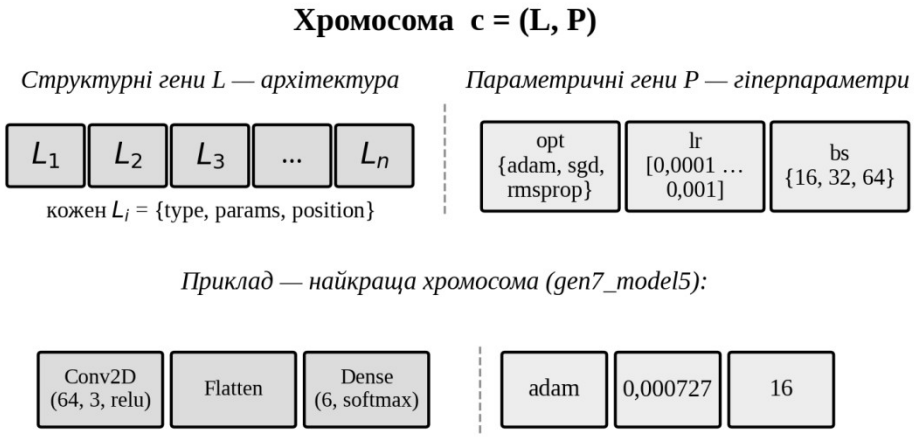


Рисунок 2.1 – Структура хромосоми

Функція пристосованості (fitness function) $f: S \to \mathbb{R}$ є ключовим компонентом генетичного алгоритму, що визначає якість кожної особини (архітектури) та направляє процес еволюції. Правильний вибір функції пристосованості критично важливий для успішного пошуку оптимальних архітектур.

У даному дослідженні функція пристосованості визначається як максимальна точність класифікації на валідаційному наборі, досягнута протягом усього процесу тренування:

$$f(c) = \max_{i=1}^{n} \text{acc}_{\text{val}}^{(i)}, \tag{2.30}$$

де c – хромосома (архітектура + гіперпараметри); $\text{acc}_{\text{val}}^{(i)}$ – точність на валідаційному наборі після i-ої епохи; n – загальна кількість епох (визначається адаптивно через EarlyStopping).

Використання максимальної, а не фінальної точності, обумовлено фундаментальною проблемою перенавчання (overfitting). Під час тренування модель може досягти найкращої точності на валідаційному наборі на проміжній епосі, після чого точність може знижуватися через перенавчання на тренувальних даних.

Приклад динаміки навчання ілюструє таблиця 2.1, що демонструє типову ситуацію перенавчання (overfitting).

Таблиця 2.1 – Динаміка навчання з overfitting

Епоха	*Train Accuracy*	*Validation Accuracy*	*Примітка*
1	0.28	0.35	Початкове навчання
2	0.45	**0.42**	**← найкраща епоха**
3	0.58	0.38	Початок overfitting
4	0.72	0.36	Overfitting посилюється
5	0.85	0.34	Сильне overfitting

Епоха	*Train Accuracy*	*Validation Accuracy*	*Примітка*

Як видно, модель продовжує покращувати точність на тренувальному наборі (0,28 → 0,85), однак точність на валідаційному наборі спочатку зростає (0,35 → 0,42), а потім падає (0,42 → 0,34). У такій ситуації фінальна точність на епосі 5 недооцінює справжню здатність моделі, тоді як максимальна точність, досягнута на епосі 2, правильно відображає потенціал архітектури.

У літературі з NAS існують різні підходи до визначення функції пристосованості (fitness). Перший – точність на фінальній епосі (final epoch accuracy):

$$f_{\text{final}}(c) = \text{acc}_{\text{val}}^{(n)}. \tag{2.31}$$

Він підлягає впливу перенавчання, особливо для моделей з великою ємністю (capacity), може спричиняти неправильне ранжування архітектур і не відображає справжнього потенціалу моделі.

Другий підхід – усереднення за k найкращими епохами (mean of top-k epochs):

$$f_{\text{mean-k}}(c) = \frac{1}{k}\sum_{i=1}^{k} \text{acc}_{\text{val}}^{(i_{\text{top}})}, \tag{2.32}$$

де i_{top} – індекси k кращих епох.

Він дає стабільнішу оцінку, зменшуючи дисперсію, і менш чутливий до випадкових «вдалих епох» (lucky epochs), проте може недооцінювати моделі зі швидкою збіжністю, розмиває сигнал від дійсно кращих архітектур і потребує вибору додаткового гіперпараметра k.

Третій підхід, обраний у цій роботі, – максимальна валідаційна точність (maximum validation accuracy):

$$f_{\max}(c) = \max_{i=1}^{n} \text{acc}_{\text{val}}^{(i)}. \tag{2.33}$$

Він захищає від перенавчання, не «караючи» модель за погіршення після піку, природно поєднується з механізмом ранньої зупинки (EarlyStopping із restore_best_weights = True), відображає справжній потенціал архітектури та узгоджується з усталеними практиками глибокого навчання [44].

Потенційним ризиком такого підходу є чутливість до «вдалих епох» – випадково завищених значень. Цей ризик мінімізується кількома засобами: рання зупинка з параметрами patience = 3 та min_delta = 0,001 відфільтровує випадкові флуктуації; популяційний характер генетичного алгоритму усереднює шум за рахунок множини архітектур; а елітизм зберігає найкращі моделі між поколіннями.

Формально процес обчислення функції пристосованості для моделі, побудованої з хромосоми, складається з таких кроків.

1. Побудова моделі з хромосоми: $M(c): \mathbb{R}^{64\times 64\times 3} \to \mathbb{R}^{6}$.
2. Компіляція моделі з обраними оптимізатором і швидкістю навчання.
3. Тренування на навчальному наборі з валідацією на відкладеній вибірці.
4. Моніторинг історії: $H = \{\text{acc}_{\text{val}}^{(1)}, \ldots, \text{acc}_{\text{val}}^{(n)}\}$.
5. Обчислення фітнесу як $f(c) = max(H)$.

Це можна записати у вигляді:

$$f(c) = \max_{i \in [1,n]} \text{accuracy}\left(M(c, D_{\text{train}}, i), D_{\text{val}}\right), \tag{2.34}$$

де $M(c, D_{\text{train}}, i)$ позначає модель після i епох тренування на D_{train}.

Кожна модель тренується з адаптивною ранньою зупинкою [40]: механізм EarlyStopping з параметрами patience = 3 епохи та min_delta = 0,001 припиняє навчання за відсутності поліпшень; ReduceLROnPlateau зменшує швидкість навчання за стагнації валідаційних втрат (val_loss); а опція restore_best_weights = True автоматично відновлює ваги найкращої епохи. Це забезпечує справедливе порівняння різнорідних архітектур: слабкі моделі зупиняються рано (5–10 епох), сильніші тренуються довше (30–50 епох), але всі оцінюються на піку своєї продуктивності.

Функція пристосованості має кілька важливих властивостей. Вона набуває значень у діапазоні від 0 (жодної правильної класифікації) до 1 (ідеальна класифікація), а базовому рівню відповідає значення ≈ 0.167 (випадкове вгадування для шести класів). Функція є монотонною (вища пристосованість означає кращу модель) і детермінованою (за фіксованих початкових станів генератора випадкових чисел результат відтворюваний). Її обчислювальна складність визначається як $O(E \times N \times C)$, де кількість епох становить 5–50 (адаптивно), розмір датасету – 6702 зображення, а складність прямого проходу (forward pass) залежить від конкретної архітектури.

Генетичні оператори забезпечують еволюційний механізм пошуку оптимальних архітектур через імітацію природних процесів відбору, рекомбінації та мутації [30]. Критичним для успіху генетичного алгоритму є правильний баланс між використанням уже знайдених хороших рішень (exploitation) і дослідженням нових областей простору пошуку (exploration).

Запропонована система використовує чотири основні оператори: селекцію (selection) – вибір батьківських особин для розмноження; кросовер (crossover) – комбінування генетичного матеріалу батьків; мутацію (mutation) – випадкові зміни для підтримання різноманітності; та елітизм (elitism) –

збереження найкращих рішень. Загальну схему роботи алгоритму подано на рисунку 2.2, а його формальний запис – в алгоритмі 2.1.

Алгоритм 2.1 – Генетичний алгоритм для пошуку архітектури NAS

```
Вхід: population_size, num_generations, Pc, Pm, elite_size
Вихід: найкраща архітектура c*

P0 ← InitializePopulation(population_size)
для кожної особини c ∈ P0:
    f(c) ← EvaluateFitness(c)

для t = 1 до num_generations:
    Pt' ← ∅

    // Елітизм
    elite ← SelectBest(Pt-1, elite_size)
    Pt' ← Pt' ∪ elite

    // Генерація нащадків
    поки |Pt'| < population_size:
        parent1 ← TournamentSelection(Pt-1)
        parent2 ← TournamentSelection(Pt-1)

        якщо random() < Pc:
            child1, child2 ← Crossover(parent1, parent2)
        інакше:
            child1, child2 ← parent1, parent2

        якщо random() < Pm:
            child1 ← Mutate(child1)
        якщо random() < Pm:
            child2 ← Mutate(child2)

        child1 ← ValidateAndFix(child1)
        child2 ← ValidateAndFix(child2)

        Pt' ← Pt' ∪ {child1, child2}

    Pt' ← Pt'[:population_size]  // Обрізка до точного розміру

    // Оцінка нового покоління
    для кожної особини c ∈ Pt':
        якщо c не оцінена:
```

37: f(c) ← EvaluateFitness(c)
38:
39: Pt ← Pt'
40:
41: c* ← argmaxc∈PT f(c)
42: повернути c*

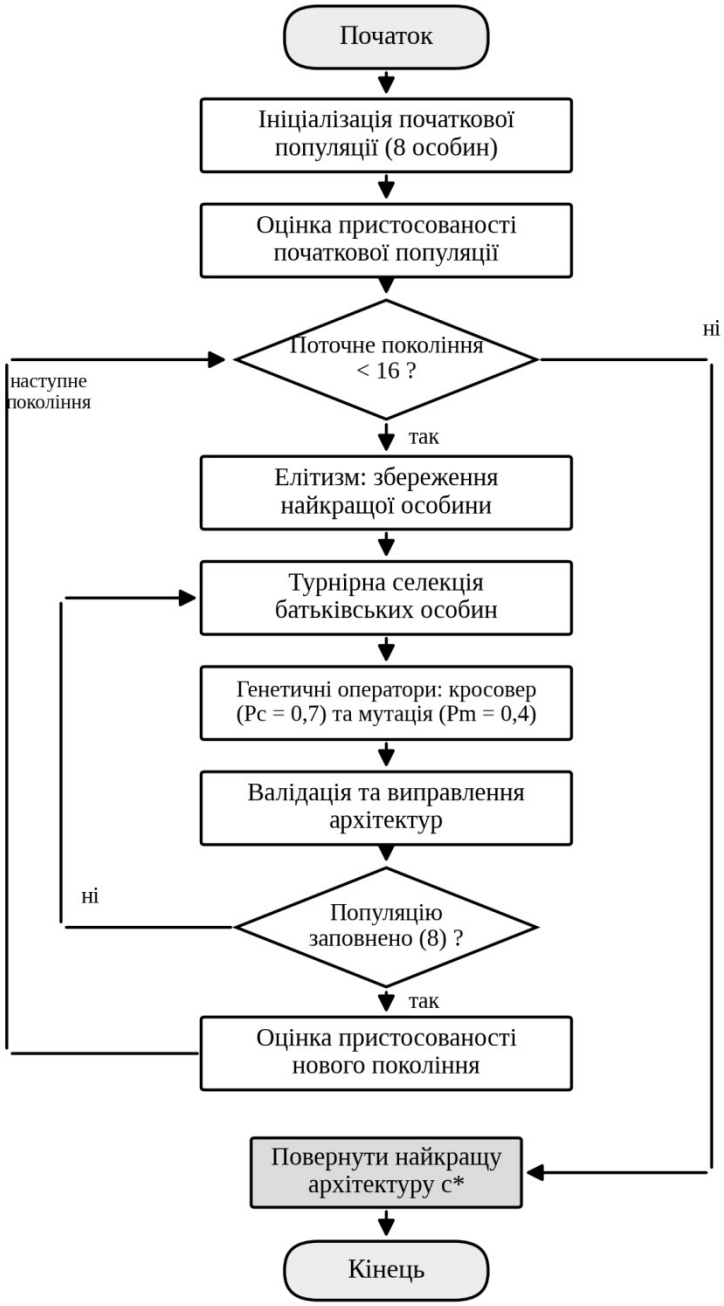


Рисунок 2.2 – Блок-схема генетичного алгоритму пошуку архітектури

В експериментах використовувались такі значення:

$$\begin{aligned} &\text{population\_size\&} = 8 \\ &\text{num\_generations\&} = 16 \\ &P_c \,\&= 0.7\,(\text{ймовірність кросоверу}). \\ &P_m \,\&= 0.4\,(\text{ймовірність мутації}) \\ &\text{elite\_size\&} = 1 \end{aligned} \tag{2.35}$$

Детальний опис кожного оператора наведено в підрозділах 2.3-2.6.

2.3 Селекція

Селекція (відбір) є процесом вибору батьківських особин з поточної популяції для створення нащадків [47]. Селекція забезпечує селекційний тиск (selection pressure) - кращі особини мають вищу ймовірність бути обраними, що направляє популяцію до областей з вищим fitness.

Для відбору батьківських особин використовується турнірна селекція (tournament selection) з розміром турніру $k = 2$. Це популярний метод, який забезпечує баланс між селекційним тиском та збереженням різноманітності популяції.

Алгоритм 2.2 – Турнірна селекція

Вхід: популяція P, розмір турніру k
Вихід: обрана особина parent

1: T ← ∅ // Турнірна вибірка
2: для i = 1 до k:
3: random_individual ← SelectRandom(P)
4: T ← T ∪ {random_individual}
5:
6: parent ← argmaxc∈T f(c) // Найкраща в турнірі
7: повернути parent

Турнірна селекція визначається як:

$$\text{parent} = \underset{c_i \in T}{arg\,max}\, f(c_i), T \subset P, |T| = k, \tag{2.36}$$

де $P = \{c_1, c_2, \ldots, c_n\}$ - поточна популяція; T - випадково обрана підмножина (турнірна вибірка); k - розмір турніру; $f(c_i)$ - fitness особини c_i.

Для турнірної селекції з розміром $k = 2$ ймовірність того, що особина з рангом r (де ранг 1 - найкраща особина) буде обрана, приблизно дорівнює:

$$P(\text{вибір особини рангу } r) \approx \frac{2(n - r + 1)}{n(n+1)}, \tag{2.37}$$

де n – розмір популяції.

Наприклад, нехай популяція містить 8 особин з fitness:

Таблиця 2.2 – Селекція

Особина	*Fitness*	*Ранг*
c1	0.98	1
c2	0.95	2
c3	0.92	3
c4	0.87	4
c5	0.83	5
c6	0.76	6
c7	0.68	7
c8	0.54	8

Турнірна селекція з $k = 2$: випадково обираємо 2 особини, наприклад c4 (0.87) та c6 (0.76), обираємо кращу особину c4 (0.87), повертаємо c4 як батьківську особину.

Турнірна селекція [30] має низку переваг. Вона проста в реалізації, оскільки не потребує сортування всієї популяції, та обчислювально ефективна – її складність становить $O(k)=O(1)$ для фіксованого розміру турніру. Параметр розміру турніру забезпечує контрольований селекційний тиск: менші значення дають помірний тиск і зберігають різноманітність популяції, тоді як більші створюють сильний тиск і пришвидшують збіжність, але підвищують ризик передчасної збіжності. Крім того, турнірна селекція добре паралелізується (незалежні турніри можуть виконуватися одночасно) і не потребує перетворення значень функції пристосованості – на відміну від рулеточної селекції, яка вимагає додатних значень fitness.

Як альтернативу варто розглянути рулеточну селекцію (roulette wheel selection, fitness proportionate selection) [24], що обирає особину з імовірністю, пропорційною її пристосованості:

$$P(\text{вибір } c_i)=\frac{f(c_i)}{\sum_{j=1}^{n} f(c_j)}. \tag{2.38}$$

Однак рулеточна селекція має суттєві недоліки: вона може призводити до домінування однієї надзвичайно успішної особини, стає неефективною за малої дисперсії значень пристосованості та потребує додатних значень fitness. Саме тому турнірна селекція показала кращі результати для поставленої задачі – завдяки стабільнішому селекційному тиску та збереженню різноманітності популяції.

2.4 Кросовер

Кросовер (схрещування, crossover) - це генетичний оператор, який комбінує генетичний матеріал двох батьківських особин для створення нащадків. Кросовер є основним механізмом exploitation в генетичних алгоритмах, дозволяючи комбінувати успішні “будівельні блоки” з різних рішень.

У цій роботі використовується гібридний підхід, який окремо обробляє структурну частину (архітектуру) та параметричну частину (гіперпараметри) хромосоми.

Для обміну шарами мережі застосовується одноточковий кросовер (single-point crossover) з деякими модифікаціями для врахування специфіки нейронних мереж.

Алгоритм 2.3 – Структурний кросовер

```
Вхід: parent1, parent2
Вихід: child1, child2

// Видалити вихідні шари
layers1 ← parent1.layers[:-1]  // Без Dense(6, softmax)
layers2 ← parent2.layers[:-1]

// Вибрати точки розрізу
p1 ← random_int(1, len(layers1) - 1)
p2 ← random_int(1, len(layers2) - 1)

// Створити нащадків
child1_layers ← layers1[:p1] + layers2[p2:]
child2_layers ← layers2[:p2] + layers1[p1:]

// Додати вихідні шари назад
child1_layers ← child1_layers + [Dense(6, softmax)]
child2_layers ← child2_layers + [Dense(6, softmax)]

повернути child1_layers, child2_layers
```

Нехай $L_1=[L_1^{(1)},\dots,L_n^{(1)}]$ та $L_2=[L_1^{(2)},\dots,L_m^{(2)}]$ - списки шарів батьків (без вихідного шару). Обираються точки розрізу $p_1 \in [1, n\text{-}1]$ та $p_2 \in [1, m\text{-}1]$. Нащадки створюються як:

$$\begin{aligned} L_{\text{child1}} \&= [L_1^{(1)},\dots,L_{p_1}^{(1)}, L_{p_2+1}^{(2)},\dots,L_m^{(2)}, L_{\text{output}}] \\ L_{\text{child2}} \&= [L_1^{(2)},\dots,L_{p_2}^{(2)}, L_{p_1+1}^{(1)},\dots,L_n^{(1)}, L_{\text{output}}], \end{aligned} \tag{2.39}$$

де $L_{\text{output}}=\text{Dense}(6,\text{softmax})$ - обов’язковий вихідний шар.

Гіперпараметри навчання обмінюються за правилами, що залежать від типу параметра.

Швидкість навчання (learning rate) є неперервним параметром, тому для неї застосовують арифметичне усереднення значень батьків:

$$\text{lr}_{\text{child}} = \frac{\text{lr}_{\text{parent1}} + \text{lr}_{\text{parent2}}}{2} (2.40). \tag{2.40}$$

Таке усереднення забезпечує плавну інтерполяцію: якщо один з батьків має менше значення (повільне, але стабільне навчання), а інший – більше (швидке навчання з ризиком нестабільності), нащадок отримує компромісне проміжне значення.

Розмір батча (batch size), навпаки, є дискретним параметром, тож усереднення для нього неможливе – натомість використовують випадковий вибір одного зі значень батьків:

$$\text{batch}_{\text{child}} \in \{\text{batch}_{\text{parent1}}, \text{batch}_{\text{parent2}}\} \text{ з рівною ймовірністю}. \tag{2.41}$$

Наприклад, якщо перший батьківський розв'язок має batch = 16, а другий – batch = 64, то нащадок з імовірністю 0,5 успадкує або 16, або 64.

Оптимізатор (optimizer) як категоріальний параметр обробляється аналогічно до розміру батча – шляхом випадкового вибору одного з варіантів батьків:

$$\text{optimizer}_{\text{child}} \in \{\text{optimizer}_{\text{parent1}}, \text{optimizer}_{\text{parent2}}\}. \quad (2.42)$$

Кросовер застосовується з ймовірністю $P_c = 0.7$:

$$\begin{cases} (\text{child1}, \text{child2}) = \text{Crossover}(\text{parent1}, \text{parent2}), & \text{якщо } U(0,1) < P_c \\ (\text{child1}, \text{child2}) = (\text{parent1}, \text{parent2}), & \text{інакше} \end{cases}, \quad (2.43)$$

де $U(0,1)$ - рівномірний розподіл на інтервалі $[0,1]$.

Якщо кросовер не відбувається (30% випадків), нащадки є точними копіями батьків. Це забезпечує збереження успішних архітектур в популяції.

Після кросоверу обидва нащадки проходять валідацію архітектури:

$$\text{child}_i' = \text{ValidateAndFix}(\text{child}_i), i \in \{1,2\}. \quad (2.44)$$

Функція ValidateAndFix() перевіряє структурні обмеження (розділ 2.7) та автоматично виправляє невалідні архітектури. Наприклад, якщо після Flatten з'явився Conv2D шар (що неможливо в Keras[40]), він замінюється на Dense.

2.5 Мутація

Мутація - це генетичний оператор, який вносить випадкові зміни в хромосому, забезпечуючи exploration (дослідження) нових областей простору пошуку. Мутація запобігає передчасній збіжності популяції до локального оптимуму та підтримує генетичне різноманіття.

У даній роботі реалізовано адаптивну мутацію з двома типами, які обираються випадково:

$$\text{тип мутації}=\begin{cases}\text{структурна}, & \text{з ймовірністю } 0.5\\ \text{параметрична}, & \text{з ймовірністю } 0.5\end{cases}. \quad (2.45)$$

Структурна мутація модифікує топологію нейронної мережі через три можливі операції: додавання шару (ADD), видалення шару (REMOVE) або модифікація параметрів існуючого шару (MODIFY).

Алгоритм 2.4 – Структурна мутація

Вхід: хромосома c
Вихід: мутована хромосома c'

```
1: operation ← random_choice({ADD, REMOVE, MODIFY})
2:
3: якщо operation = ADD і len(c.layers) < MAX_LAYERS:
4:     position ← random_int(1, len(c.layers) - 1)
5:     context ← DetermineContext(c.layers, position)
6:     new_layer ← GenerateRandomLayer(context)
7:     c'.layers ← c.layers[:position] + [new_layer] + c.layers[position:]
8:
9: інакше якщо operation = REMOVE і len(c.layers) > MIN_LAYERS:
10:    removable ← GetRemovableLayers(c.layers)  // Без Flatten та output
11:    layer_to_remove ← random_choice(removable)
12:    c'.layers ← c.layers without layer_to_remove
13:
14: інакше якщо operation = MODIFY:
15:    modifiable ← GetModifiableLayers(c.layers)
16:    layer ← random_choice(modifiable)
17:    MutateLayerParameters(layer)
18:
19: c' ← ValidateAndFix(c')
20: повернути c'
```

Операція ADD додає до архітектури новий шар:

$$L^{'}=[L_1,\dots,L_{i-1},L_{\text{new}},L_i,\dots,L_n],i\in[1,n], \quad (2.46)$$

де L_{new} - новий випадково згенерований шар, тип якого залежить від контексту (до або після Flatten).

Генерація є контекстно-залежною: до Flatten можуть додаватися $L_{\text{new}} \in \{\text{Conv2D}, \text{MaxPool}, \text{BatchNorm}, \text{Dropout}\}$, а після Flatten – $L_{\text{new}} \in \{\text{Dense}, \text{Dropout}\}$. При цьому діє обмеження на максимальну кількість шарів:

$$|L'| \leq \text{MAX\_LAYERS} = 5. \tag{2.47}$$

Наприклад, для архітектури [Conv2D(32), Flatten(), Dense(64)] додавання шару MaxPool() після Conv2D дає [Conv2D(32), MaxPool(), Flatten(), Dense(64)].

Операція REMOVE видаляє один шар з архітектури:

$$L' = [L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n], \tag{2.48}$$

де шар L_i видаляється з архітектури.

Видаленню не підлягають шар Flatten та вихідний шар Dense(6, softmax), а також зберігається обмеження на мінімальну кількість шарів $|L'| \geq \text{MIN\_LAYERS} = 3$. Наприклад, з архітектури [Conv2D(32), MaxPool(), Flatten(), Dense(64), Dropout(0.3)] видалення MaxPool() дає [Conv2D(32), Flatten(), Dense(64), Dropout(0.3)].

Операція MODIFY змінює параметри наявного шару залежно від його типу: для Conv2D – формула 2.49, для Dense – формула 2.50, для Dropout – формула 2.51, для MaxPool – формула 2.52. Наприклад, шар Dense(64, relu) після модифікації кількості нейронів може перетворитися на Dense(128, relu).

$$\text{Conv2D}(\text{filters},k,\alpha)\to\begin{cases}\text{Conv2D}(\text{filters}',k,\alpha), & \text{filters}'\in\{32,64,128\}\\ \text{Conv2D}(\text{filters},k',\alpha), & k'\in\{3,5\}\\ \text{Conv2D}(\text{filters},k,\alpha'), & \alpha'\in\{\text{relu},\text{elu}\}\end{cases}. \quad (2.49)$$

$$\text{Dense}(\text{neurons},\alpha)\to\begin{cases}\text{Dense}(\text{neurons}',\alpha), & \text{neurons}'\in\{32,64,128,256\}\\ \text{Dense}(\text{neurons},\alpha'), & \alpha'\in\{\text{relu},\text{elu}\}\end{cases}. \quad (2.50)$$

$$\text{Dropout}(\text{rate})\to\text{Dropout}(\text{rate}'),\text{rate}'\sim U(0.2,0.5). \quad (2.51)$$

$$\text{MaxPool}(p)\to\text{MaxPool}(p'),p'\in\{2,3\}. \quad (2.52)$$

Параметрична мутація модифікує гіперпараметри навчання без зміни архітектури мережі.

Для швидкості навчання (learning rate) передбачено два рівноймовірні режими. Перший – незначна зміна (multiplier mutation), за якої поточне значення множиться на випадковий коефіцієнт у межах околу:

$$\text{lr}'=\text{lr}\times\alpha,\alpha\sim U(0.5,2.0), \quad (2.53)$$

що забезпечує плавну зміну.

Другий – повна заміна (replacement mutation):

$$\text{lr}'\sim U(\text{lr}_{min},\text{lr}_{max})=U(0.0001,0.001), \quad (2.54)$$

яка дозволяє «стрибнути» в іншу область простору швидкості навчання.

Розмір батча (batch size) мутує шляхом випадкового вибору нового значення з допустимого набору:

$$\text{batch}'=\text{random\_choice}(\{16,32,64\}). \quad (2.55)$$

Аналогічно оптимізатор (optimizer) замінюється випадково обраним варіантом:

$$\text{optimizer}' = \text{random\_choice}(\{\text{adam}, \text{sgd}, \text{rmsprop}\}). \tag{2.56}$$

Наприклад, вихідний набір гіперпараметрів lr = 0,0005, batch = 32, optimizer = adam після мутації може перетворитися на lr = 0,0008 (множник 1,6), batch = 16 (випадкова заміна) та optimizer = rmsprop (випадкова заміна).

Мутація застосовується до кожного нащадка після кросоверу з імовірністю Pm:

$$c' = \begin{cases} \text{Mutate}(c), & \text{якщо } U(0,1) < P_m \\ c, & \text{інакше} \end{cases}. \tag{2.57}$$

Після структурної мутації архітектура обов'язково перевіряється на валідність.

Алгоритм 2.5 – Перевірка на валідність

```
if mutation_type == "structural":
 mutated_chromosome = ValidateAndFix(mutated_chromosome)
 if not mutated_chromosome.is_valid():
 return original_chromosome  # Відкидаємо невалідну мутацію
```

Це гарантує, що всі особини в популяції мають коректні архітектури, які можуть бути побудовані в Keras/TensorFlow.

2.6 Елітизм

Елітизм (elitism) - це стратегія, яка забезпечує збереження найкращих рішень з покоління в покоління. Це гарантує, що якість найкращої особини в популяції ніколи не погіршується та прискорює збіжність алгоритму.

У кожному поколінні найкращі e особин (за значенням fitness) копіюються без змін у наступне покоління:

$$P_{t+1}^{\text{elite}} = \{c_1^*, c_2^*, \ldots, c_e^*\}, \text{де } f(c_i^*) \geq f(c_j), \forall\, j \in P_t, i \leq e. \tag{2.58}$$

У даній роботі використовується $e = 1$ (elite_size = 1):

$$P_{t+1}[0] = \arg\max_{c \in P_t} f(c). \tag{2.59}$$

Алгоритм 2.6 – Елітизм

Вхід: популяція Pt, elite_size e
Вихід: elite особини

```
1: SortByFitness(Pt)  // Сортування за спаданням fitness
2: elite ← Pt[0:e]    // Вибір e найкращих
3: повернути elite
```

Повний процес формування покоління $t+1$ з популяції t:

$$P_{t+1} = P_{t+1}^{\text{elite}} \cup P_{t+1}^{\text{offspring}}, \tag{2.60}$$

де P_{t+1}^{elite} - копії найкращих особин (розмір e); $P_{t+1}^{\text{offspring}}$ - нащадки, створені через селекцію, кросовер та мутацію (розмір $n-e$).

Для $n = 8$ та $e = 1$ 1 особина: копія найкращої (елітизм); 7 особин: нащадки від генетичних операторів.

Елітизм забезпечує монотонність найкращого fitness:

$$\max_{c \in P_{t+1}} f(c) \geq \max_{c \in P_t} f(c), \forall\, t \geq 0. \tag{2.61}$$

Доведення: нехай $c^* = arg\,max_{c \in P_t} f(c)$ - найкраща особина в поколінні t. За визначенням елітизму, $c^* \in P_{t+1}$. Отже $\max_{c \in P_{t+1}} f(c) \geq f(c^*) = \max_{c \in P_t} f(c)$.

Елітизм має кілька переваг [30]. Він забезпечує монотонну збіжність, гарантуючи, що найкращий знайдений результат ніколи не погіршується, і прискорює збіжність, оскільки збереження кращих рішень пришвидшує пошук оптимуму. Крім того, елітизм запобігає втраті вдалих рішень через стохастичність генетичних операторів і полегшує відтворюваність експериментів.

Вибір розміру елітної групи (elite_size) впливає на баланс між дослідженням простору пошуку (exploration) та використанням уже знайдених рішень (exploitation), що ілюструє таблиця 2.3.

Таблиця 2.3 – Вплив розміру елітної групи на роботу алгоритму

elite_size	*Exploitation*	*Exploration*	*Збіжність*	*Ризик*
0	Низька	Висока	Повільна	Втрата кращих рішень
1	Оптимальна	Оптимальна	Швидка	Мінімальний
2-3	Висока	Середня	Дуже швидка	Передчасна збіжність
>3	Дуже висока	Низька	Надшвидка	Застрягання в локальних мінімумах

Експериментально встановлено, що для поставленої задачі оптимальним є значення elite_size = 1: воно забезпечує збереження найкращого рішення, достатнє різноманіття для exploration та баланс між швидкістю збіжності і якістю результату.

Дію елітизму ілюструє приклад еволюції популяції:

Покоління 0:
c1: fitness = 0.85 ← найкраща
c2: fitness = 0.82
c3: fitness = 0.79
...

Покоління 1:
c1': fitness = 0.85 ← копія c1 (елітизм)
c2': fitness = 0.88 ← новий лідер від crossover/mutation
c3': fitness = 0.81
...
Найкраща: 0.88 ≥ 0.85 ✓

Покоління 2:
c1'': fitness = 0.88 ← копія c2' (елітизм)
c2'': fitness = 0.87
...
Найкраща: 0.88 ≥ 0.88 ✓ (не погіршилась)

Як бачимо, монотонність найкращого fitness зберігається: $0.85 \rightarrow 0.88 \rightarrow 0.88$.

Елітизм діє в синергії з іншими генетичними операторами. Найкраща особина має високу ймовірність бути обраною батьком під час селекції, а її комбінування з іншими особинами через кросовер може породити ще кращі рішення; мутація еліти (коли вона обрана батьком) також здатна привести до покращення. При цьому популяційне різноманіття зберігається, оскільки еліта становить лише 12,5 % популяції (1 з 8), залишаючи простір для нових рішень.

2.7 Валідація архітектур

Валідація архітектури - це критичний процес, який гарантує, що всі згенеровані генетичними операторами архітектури можуть бути успішно

побудовані та натреновані в Keras/TensorFlow. Без валідації генетичний алгоритм може породжувати невалідні структури, які призведуть до помилок runtime.

Для забезпечення коректності архітектури в Keras застосовано сім структурних правил.

Перше правило стосується початкового шару. Перший шар мережі, що обробляє зображення, має бути згортковим:

$$L_1 = \text{Conv2D}. \tag{2.62}$$

Dense шар не може безпосередньо обробляти 2D зображення без попереднього flatten або pooling.

Друге правило визначає розміщення BatchNorm:

$$\text{якщо } L_i = \text{BatchNorm}, \text{ то } L_{i-1} \in \{\text{Conv2D}, \text{Dense}\}. \tag{2.63}$$

BatchNorm нормалізує активації, тому може йти тільки після шарів з активаціями (Conv2D або Dense). BatchNorm після MaxPool або Dropout не має сенсу, оскільки ці шари не змінюють розподіл активацій.

Третє правило накладає обмеження на шари після Flatten:

$$\text{якщо } \exists L_f = \text{Flatten}, \text{ то } \forall i > f : L_i \in \{\text{Dense}, \text{Dropout}, L_{\text{output}}\}. \tag{2.64}$$

Після Flatten(), який перетворює 2D feature maps в 1D вектор, не можуть йти згорткові шари (Conv2D, MaxPool). Тільки Dense, Dropout та вихідний шар.

Четверте правило обмежує кількість шарів MaxPool:

$$|\{L_i : L_i = \text{MaxPool}\}| \leq 3. \tag{2.65}$$

Для зображень 64×64 пікселі максимум 3 MaxPool шари1-й MaxPool: $64 \times 64 \to 32 \times 32$; 2-й MaxPool: $32 \times 32 \to 16 \times 16$; 3-й MaxPool: $16 \times 16 \to 8 \times 8$. Четвертий MaxPool призвів би до розміру 4×4, що занадто мало.

П'яте правило обмежує повторення utility-шарів:

$$\text{якщо } L_i, L_{i+1} \in \{\text{BatchNorm}, \text{MaxPool}, \text{Dropout}\}, \text{ то } L_i \neq L_{i+1}. \tag{2.66}$$

Не більше одного utility-шару одного типу підряд. Наприклад, два Dropout підряд безглузді, оскільки їх ефект можна досягти одним Dropout з вищою rate.

Шосте правило задає допустиму глибину мережі:

$$3 \leq |L| \leq 5, \tag{2.67}$$

де $|L|$ - кількість шарів без врахування вихідного Dense(6, softmax).

Сьоме правило фіксує вихідний шар:

$$L_{\text{final}} = \text{Dense}(6, \text{softmax}). \tag{2.68}$$

Обов’язковий вихідний шар для класифікації 6 класів Military Vehicles датасету.

Алгоритм 2.7 – ValidateAndFix

Вхід: хромосома c
Вихід: валідна хромосома c'

```
1: c' ← c
2:
3: // Правило 1: Перевірка першого шару
4: якщо c'.layers[0].type ≠ Conv2D:
5:     c'.layers[0] ← GenerateRandomConv2D()
6:
7: // Правило 2: BatchNorm розміщення
```

```
8: для i від 1 до len(c'.layers):
9:     якщо c'.layers[i].type = BatchNorm:
10:        якщо c'.layers[i-1].type ∉ {Conv2D, Dense}:
11:            RemoveLayer(c', i)
12:
13: // Правило 3: Flatten обмеження
14: flatten_index ← FindFlat ten(c'.layers)
15: якщо flatten_index існує:
16:    для i від flatten_index+1 до len(c'.layers)-1:
17:        якщо c'.layers[i].type ∈ {Conv2D, MaxPool}:
18:            c'.layers[i] ← GenerateRandomDense()
19:
20: // Правило 4: MaxPool обмеження
21: maxpool_count ← CountMaxPool(c'.layers)
22: якщо maxpool_count > 3:
23:    RemoveExcessMaxPool(c', maxpool_count - 3)
24:
25: // Правило 5: Utility обмеження
26: для i від 0 до len(c'.layers)-2:
27:    якщо IsSameUtility(c'.layers[i], c'.layers[i+1]):
28:        RemoveLayer(c', i+1)
29:
30: // Правило 6: Глибина
31: поки len(c'.layers) < MIN_LAYERS:
32:    AddRandomLayer(c')
33: поки len(c'.layers) > MAX_LAYERS:
34:    RemoveRandomLayer(c')
35:
36: // Правило 7: Вихідний шар
37: c'.layers[-1] ← Dense(6, softmax)
38:
39: повернути c'
```

Розглянемо приклад валідації. Невалідна архітектура після кросоверу має вигляд:

```
Невалідна архітектура:
[Dense(64, relu),          ← Порушення: не Conv2D на початку
 BatchNorm(),              ← OK: після Dense
 Flatten(),                ← OK
 Conv2D(32, 3, relu),      ← Порушення: Conv2D після Flatten
 MaxPool(),                ← OK
 MaxPool(),                ← OK
 MaxPool(),                ← OK
 MaxPool(),                ← Порушення: 4-й MaxPool
```

Dropout(0.3),
Dense(6, softmax)]

Після застосування ValidateAndFix():

Валідна архітектура:
[Conv2D(64, 3, relu), ← Виправлено: замінено перший шар
BatchNorm(), ← Збережено
Flatten(), ← Збережено
Dense(128, relu), ← Виправлено: Conv2D → Dense
Dropout(0.3), ← Збережено
Dense(6, softmax)] ← Збережено (вихідний шар)

(MaxPool видалені через обмеження після Flatten)

Валідація може бути представлена як предикат:

$$\text{IsValid} : A \to \{\text{true}, \text{false}\}, \tag{2.69}$$

де A – простір всіх можливих архітектур.

Функція виправлення є проєкцією на множину валідних архітектур:

$$\text{Fix} : A \to A_{\text{valid}}, \text{де } A_{\text{valid}} = \{a \in A : \text{IsValid}(a) = \text{true}\}. \tag{2.70}$$

Запропонована система забезпечує три гарантії: завершуваність (алгоритм завжди завершується за скінченний час), коректність (результат виправлення завжди є валідною архітектурою) $\text{IsValid}(\text{Fix}(c)) = \text{true}, \forall c \in A$ та ідемпотентність (повторне застосування функції не змінює вже валідну архітектуру) $\text{Fix}(\text{Fix}(c)) = \text{Fix}(c)$.

Завдяки системі валідації, генетичний алгоритм може вільно застосовувати генетичні оператори, знаючи, що всі згенеровані архітектури будуть коректними та придатними для тренування.

2.8 Метрики оцінки якості

Для комплексної оцінки якості моделей класифікації використовується набір метрик, які характеризують різні аспекти продуктивності. Правильний вибір та інтерпретація метрик є критично важливими для об'єктивної оцінки результатів.

Accuracy – це найпростіша та найбільш інтуїтивна метрика, яка визначає частку правильно класифікованих зразків:

$$\text{Accuracy}=\frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}=\frac{\text{кількість правильних прогнозів}}{\text{загальна кількість прогнозів}}, \tag{2.71}$$

де TP (True Positive) - правильно класифіковані позитивні зразки; TN (True Negative) - правильно класифіковані негативні зразки; FP (False Positive) - помилково класифіковані як позитивні; FN (False Negative) - помилково класифіковані як негативні.

Для багатокласової класифікації (як у нашому випадку з 6 класами):

$$\text{Accuracy}=\frac{1}{N}\sum_{i=1}^{N}1[\hat{y}_i=y_i], \tag{2.72}$$

де N - кількість зразків; y_i - справжня мітка; $\hat{y}_i$ - передбачена мітка; $1[\cdot]$ - індикаторна функція.

Метрика accuracy має низку переваг: вона інтуїтивно інтерпретується (відсоток правильних відповідей), добре підходить для збалансованих датасетів і широко використовується для порівняння з іншими роботами. Водночас вона має й недоліки – може бути оманливою для незбалансованих наборів даних і не показує, які саме класи класифікуються краще, а які гірше.

Precision вимірює якість позитивних прогнозів - яка частка зразків, класифікованих як позитивні, насправді є позитивними:

$$\text{Precision}=\frac{\text{TP}}{\text{TP+FP}}. \tag{2.73}$$

Для багатокласової класифікації обчислюється precision для кожного класу c:

$$\text{Precision}_c=\frac{\text{TP}_c}{\text{TP}_c+\text{FP}_c}. \tag{2.74}$$

Висока precision означає малу кількість хибних спрацювань (false positives) і особливо важлива для задач, де такі помилки дорого коштують. У військових застосуваннях це, наприклад, помилкова класифікація цивільного об'єкта як військового.

Recall вимірює здатність моделі знаходити всі позитивні зразки:

$$\text{Recall}=\frac{\text{TP}}{\text{TP+FN}}. \tag{2.75}$$

Для багатокласової класифікації:

$$\text{Recall}_c=\frac{\text{TP}_c}{\text{TP}_c+\text{FN}_c}. \tag{2.76}$$

Висока recall означає малу кількість пропущених позитивних зразків (false negatives) і важлива для задач, де пропуск об'єктів є критичним. У військових застосуваннях, наприклад, пропуск військового об'єкта може мати серйозні наслідки.

F1-score є гармонічним середнім precision та recall, забезпечуючи баланс між ними:

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} = \frac{2 \cdot \mathrm{TP}}{2 \cdot \mathrm{TP} + \mathrm{FP} + \mathrm{FN}}. \quad (2.77)$$

Використання саме гармонічного середнього, на відміну від арифметичного, «карає» за низькі значення: якщо precision або recall є низькою, то й F1-score буде низьким. Наприклад, за precision = 0,9 і recall = 0,9 значення F1 = 0,90 (збалансований випадок), тоді як за precision = 1,0 і recall = 0,5 воно становить лише 0,67 – у цьому ж випадку арифметичне середнє дало б завищену оцінку 0,75.

Для багатокласової класифікації зазвичай використовується macro-average F1:

$$\mathrm{F1}_{\mathrm{macro}} = \frac{1}{K} \sum_{c=1}^{K} \mathrm{F1}_c, \quad (2.78)$$

де K - кількість класів (у нашому випадку $K = 6$).

Macro-average трактує всі класи однаково, незалежно від їх розміру. Альтернативно, weighted F1 враховує розмір кожного класу:

$$\mathrm{F1}_{\mathrm{weighted}} = \sum_{c=1}^{K} w_c \cdot \mathrm{F1}_c, w_c = \frac{n_c}{N}, \quad (2.79)$$

де n_c - кількість зразків класу c, N - загальна кількість зразків.

Confusion matrix надає детальну інформацію про помилки класифікації:

$$C_{ij} = \text{кількість зразків класу } i, \text{ класифікованих як } j. \quad (2.80)$$

Для 6 класів Military Vehicles датасету confusion matrix має розмір 6×6:

$$C = \begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} & C_{04} & C_{05} \\ C_{10} & C_{11} & C_{12} & C_{13} & C_{14} & C_{15} \\ C_{20} & C_{21} & C_{22} & C_{23} & C_{24} & C_{25} \\ C_{30} & C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ C_{40} & C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \\ C_{50} & C_{51} & C_{52} & C_{53} & C_{54} & C_{55} \end{bmatrix}. \quad (2.81)$$

Діагональні елементи C_{ii} – правильні класифікації класу i. Недіагональні C_{ij} $(i \neq j)$ - помилкові класифікації i як j. Ідеальна матриця – діагональна. Приклад такої матриці наведено в таблиці 2.4.

З наведеної матриці видно, що клас цивільних літаків (civilian aircraft) класифікується дуже добре – 85 зі 87 зразків, тобто 97,7 %, – а найчастішою помилкою є віднесення цивільного автомобіля до військового гелікоптера (два випадки).

Таблиця 2.4 – Матриця плутанини для класифікації військової техніки

Actual \ Predicted	*civ_air*	*civ_car*	*mil_air*	*mil_heli*	*mil_tank*	*mil_truck*
civ_air	85	2	0	0	0	0
civ_car	1	92	0	2	0	1
mil_air	0	0	78	1	0	0
mil_heli	0	1	1	75	0	0
mil_tank	0	0	0	0	68	2
mil_truck	0	2	0	0	1	67

Окрім метрик якості класифікації, для порівняння методів NAS використовують додаткові показники.

Першим є обчислювальна ефективність, що вимірюється вартістю пошуку (search cost):

$$\text{Search Cost} = \text{GPU-години} \times \text{кількість GPU}. \quad (2.82)$$

У цій роботі вона становить 0,5 GPU-години (30 хвилин на відеокарті NVIDIA L4), що на кілька порядків менше, ніж у відомих методів: DARTS [1] потребує 4 GPU-дні (96 GPU-годин), а NASNet [2] – 1800 GPU-днів (43 200 GPU-годин).

Другим показником є відтворюваність: у цій роботі похибка відтворюваності (reproducibility error) становить лише 0,0002 (0,02 %):

$$\text{Reproducibility Error} = \text{MAE}(\text{original\_fitness}, \text{validated\_fitness}). \quad (2.83)$$

Третім показником є складність архітектури, яку можна оцінити з урахуванням ваги обчислювальної складності:

$$\text{Model Complexity} = \text{кількість параметрів} + \lambda \cdot \text{FLOPs}, \quad (2.84)$$

де λ - вага обчислювальної складності.

Четвертим показником є швидкість збіжності: у цій роботі найкращу модель знайдено вже на сьомому поколінні з шістнадцяти.

$$\text{Convergence Speed} = \text{покоління до досягнення } 99\% \text{ від найкращого fitness}. \quad (2.85)$$

У даній роботі найкраща модель знайдена на поколінні 7 з 16. Зведену характеристику метрик подано в таблиці 2.5.

Таблиця 2.5 – Метрики для порівняння методів NAS

Метрика	*Призначення*	*Важливість для NAS*
Accuracy	Основна якість	Висока (fitness function)

Метрика	*Призначення*	*Важливість для NAS*
	класифікації	
Precision/Recall	Якість по окремих класах	Середня (додаткова інформація)
F1-score	Баланс precision/recall	Середня
Confusion Matrix	Аналіз помилок	Висока (інтерпретація)
Search Cost	Обчислювальні витрати	Висока (практичність)
Reproducibility	Відтворюваність	Висока (наукова цінність)

У цьому дослідженні основною метрикою для fitness function є Accuracy, оскільки датасет є достатньо збалансованим, а задача вимагає високої загальної точності класифікації військової техніки.

2.9 Обґрунтування вибору підходу

Вибір генетичного алгоритму для пошуку архітектури нейронних мереж обумовлений рядом методологічних, практичних та прикладних факторів.

Запропонований еволюційний підхід має низку важливих переваг. Першою є одночасна оптимізація архітектури та гіперпараметрів. На відміну від більшості наявних методів NAS (DARTS [1], NASNet [2], ENAS [3]), які оптимізують лише архітектуру за фіксованих гіперпараметрів, запропонований метод оптимізує їх спільно:

$$\arg \max_{(a,h) \in A \times H} f(a,h), \tag{2.86}$$

де a - архітектура; h - гіперпараметри.

Експериментально виявлено нетривіальні залежності між ними: глибокі архітектури (9 і більше шарів) потребують малих розмірів батча (16), оптимізатор RMSprop зі швидкістю навчання 0,00057 забезпечує стабільну збіжність, а найкраща модель поєднала мінімалістичну архітектуру (3 шари) з оптимізатором Adam, швидкістю навчання 0,000727 та розміром батча 16. Фіксування гіперпараметрів унеможливило б виявлення цих оптимальних комбінацій.

Другою перевагою є інтерпретовність процесу. Порівняно з градієнтними методами генетичний алгоритм є «білою скринькою», що ілюструє таблиця 2.6.

Наприклад, аналіз 128 моделей дозволив виявити чіткі закономірності: оптимізатор RMSprop переважав (79 зі 128 моделей) завдяки адаптивній швидкості навчання та стабільності; розмір батча 16 виявився оптимальним (96 моделей), оскільки дає більше оновлень і кращу узагальнювальну здатність; а прості архітектури (3–8 шарів) були ефективнішими, бо датасет містить чіткі ознаки.

Таблиця 2.6 – Інтерпретовність еволюційного та градієнтного підходів

Аспект	*Генетичний підхід*	*Градієнтні методи (DARTS)*
Процес пошуку	Зрозумілий (селекція, кросовер, мутація)	“Чорна скринька” (gradient flow)
Аналіз поколінь	Можна відстежити еволюцію	Неможливо
Причини успіху	Видно, які архітектурні блоки працюють	Непрозоро
Навчальна цінність	Висока (зрозумілі кроки)	Низька

Третьою перевагою є обчислювальна доступність. Запропонований метод потребує лише 0,5 GPU-години проти 0,5–3150 GPU-днів у методів рівня SOTA:

$$\text{Speedup} = \frac{\text{SOTA search cost}}{\text{Our search cost}} = \frac{4\ \text{GPU-дні}}{0.5\ \text{GPU-години}} = \frac{96}{0.5} = 192\,(\textit{порівнянозDARTS}). \qquad (2.87)$$

Це робить його доступним для академічних дослідницьких груп з обмеженими ресурсами, окремих дослідників та освітніх закладів, а також придатним для швидкого прототипування у промислових застосуваннях.

Четвертою перевагою є відсутність зміщення апроксимації (approximation bias). Методи зі спільним використанням ваг, як-от ENAS [3], застосовують супермережу (supernet), де всі підмережі поділяють ваги [7]; через це оцінка пристосованості архітектури на спільних вагах може не відповідати її справжній якості після повного тренування. Запропонований підхід натомість повністю тренує кожну модель, що забезпечує чесну оцінку пристосованості й усуває approximation bias.

П'ятою перевагою є природна регуляризація, яку забезпечує популяційний характер методу: різноманітність (вісім різних архітектур у кожному поколінні), стохастичність (випадковість у селекції, кросовері та мутації) та елітизм (збереження кращих рішень). Це запобігає передчасній збіжності до локальних оптимумів.

Шостою перевагою є гнучкість і розширюваність підходу. Його легко адаптувати до інших датасетів (достатньо змінити розмір вхідних зображень і кількість класів), доповнити новими архітектурними блоками (наприклад, attention чи residual), розширити до багатокритеріальної оптимізації (додавши обмеження на розмір моделі або швидкість інференсу) чи застосувати з іншими метриками (замінивши accuracy на F1-score або власну метрику).

Задача класифікації військової техніки має специфічні характеристики, які роблять еволюційний підхід особливо придатним.

Перша – критична важливість точності. Помилкова класифікація у військових системах може мати серйозні наслідки: хибне спрацювання (цивільний об'єкт → військовий) призводить до неприйнятних дій проти цивільних об'єктів, а пропуск (військовий об'єкт → цивільний) означає невиявлену загрозу. Генетичний алгоритм з повним тренуванням кожної моделі забезпечує чесну оцінку точності (без approximation bias), можливість аналізу матриці плутанини для кожної архітектури та вибір моделі з найкращим балансом precision і recall.

Друга характеристика – обмеженість датасетів. Військові набори даних зазвичай менші за стандартні бенчмарки (ImageNet [50], COCO) через обмежену доступність військових зображень, проблеми конфіденційності та складність анотування; зокрема, датасет Military Vehicles містить 6702 тренувальних зображення проти 1,2 млн в ImageNet. Для малих датасетів простіші архітектури часто краще узагальнюють (найкраща модель у цій роботі має лише три шари), повне тренування кожної моделі лишається можливим завдяки швидкому навчанню на малому наборі, а еволюційний пошук здатний знайти оптимальну ємність (capacity) мережі.

Третя – необхідність інтерпретовності. Військові системи потребують пояснюваності рішень: чому певна архітектура краща, які ознаки використовує модель і як вона розвивалася. Генетичний підхід надає повну історію еволюції (128 моделей за 16 поколінь), можливість аналізу успішних архітектурних патернів та візуалізацію траєкторії популяції в просторі архітектур.

Четверта – швидкість розгортання. Для військових застосувань важлива швидкість адаптації: поява нового типу техніки потребує нової архітектури за обмеженого часу на розробку. Запропонований метод виконує автоматичний пошук за 30 хвилин – на противагу місяцям ручного проектування.

П'ята характеристика – відтворюваність для верифікації. Військові системи потребують незалежної перевірки, яку забезпечують система

відтворюваності з похибкою 0,02 %, повний набір натренованих ваг для всіх моделей і можливість верифікації приблизно за дві хвилини. Це дозволяє незалежним експертам перевірити результати без повторного тренування (яке тривало б 30 хвилин).

Вибір генетичного алгоритму має й математичне обґрунтування. Задача NAS є NP-важкою задачею комбінаторної оптимізації [16]:

$$\max_{a \in A} f(a), |A| \approx 10^{12}. \tag{2.88}$$

Серед можливих методів її розв'язання повний перебір (exhaustive search) має складність $O(|A|)$ і практично неможливий; випадковий пошук (random search) – складність $O(k)$ – дає рішення низької якості; градієнтні методи – $O(|A|^{0.5})$ – страждають від approximation bias; натомість еволюційні методи зі складністю $O(p \times g)$ де p - розмір популяції, g - кількість поколінь, забезпечують баланс якості та швидкості, де враховуються розмір популяції та кількість поколінь. Для розглянутого випадку за восьми особин у популяції та шістнадцяти поколінь це дає близько 128 повних тренувань (8 × 16). Для нашого випадку: $p = 8$, $g = 16 \rightarrow O(128)$ повних тренувань.

Отже, вибір генетичного алгоритму для задачі класифікації військової техніки є оптимальним з кількох поглядів: методологічного (одночасна оптимізація архітектури та гіперпараметрів, відсутність approximation bias), практичного (обчислювальна доступність – 0,5 GPU-години, швидке розгортання) та прикладного (висока точність 98,59 %, інтерпретовність і відтворюваність для верифікації).

Висновки до розділу 2

У цьому розділі розглянуто математичні та теоретичні основи методу автоматичного пошуку архітектури згорткових нейронних мереж з використанням генетичного алгоритму. Основними теоретичними внесками є такі.

1. Основи згорткових нейронних мереж (розділ 2.1): Детально розглянуто математичні основи базових компонентів CNN: згорткові шари (операція згортки, формула 2.1), pooling шари (Max Pooling, формула 2.2), Batch Normalization (нормалізація по батчу, формули 2.4-2.7), Dropout (регуляризація, формули 2.8-2.10) та функції активації (ReLU, ELU, Softmax, формули 2.11-2.14). Ці компоненти складають будівельні блоки для простору пошуку архітектур.
2. Математична модель генетичного алгоритму (розділ 2.2): Формалізовано простір пошуку $S = A \times H$ (формула 2.15) з оцінкою потужності $|S| \approx 10^{12}$ можливих архітектур (формула 2.22). Розроблено уніфіковане генетичне кодування хромосоми $c = (L, P)$ (формула 2.23), яке спільно представляє архітектуру та гіперпараметри навчання. Визначено адаптивну функцію пристосованості $f(c) = max_{i=1}^{n} \text{acc}_{\text{val}}^{(i)}$ (формула 2.30), яка враховує перенавчання через вибір найкращої епохи.
3. Генетичні оператори (розділи 2.3-2.6): Математично формалізовано турнірну селекцію (формула 2.36), гібридний кросовер з окремою обробкою структурної (формула 2.39) та параметричної частини (формули 2.40-2.42), адаптивну мутацію (структурна формули 2.46-2.52, параметрична формули 2.53-2.56) та елітизм з гарантією монотонності $max_{c \in P_{t+1}} f(c) \geq max_{c \in P_t} f(c)$ (формула 2.61).
4. Система валідації архітектур (розділ 2.7): Формалізовано 7 структурних правил коректності для Keras/TensorFlow (формули 2.62-2.68) та розроблено алгоритм автоматичного виправлення

невалідних архітектур. Валідація формалізована як предикат $\mathrm{IsValid}: A \to \{\mathrm{true}, \mathrm{false}\}$ та проєкція $\mathrm{Fix}: A \to A_{\mathrm{valid}}$ (формули 2.69-2.70).

5. Метрики оцінки якості (розділ 2.8): Визначено систему метрик: Accuracy (формула 2.72), Precision/Recall (формули 2.74-2.76), F1-score (формула 2.77), Confusion Matrix (формула 2.81) та специфічні метрики для NAS: Search Cost, Reproducibility Error, Model Complexity (формули 2.82-2.85).
6. Обґрунтування вибору підходу (розділ 2.9): Доведено переваги еволюційного підходу для задачі класифікації військової техніки: одночасна оптимізація архітектури та гіперпараметрів, обчислювальна доступність (speedup 192× порівняно з DARTS, формула 2.87), інтерпретовність процесу, відсутність approximation bias та природна регуляризація через популяційний підхід.

Ключовими математичними результатами розділу є оцінка потужності простору пошуку ($|S| \approx 10^{12}$ унікальних архітектур), складність алгоритму ($O(p \times g) = O(8 \times 16) = O(128)$ повних тренувань), гарантована монотонність через елітизм (найкраще значення пристосованості ніколи не погіршується) та низька похибка відтворюваності (0,02 %, що на два порядки нижче за стохастичність тренування).

Розроблена математична модель забезпечує теоретичне обґрунтування практичної реалізації системи автоматичного пошуку архітектур, яка поєднує обчислювальну доступність (0,5 GPU-години), високу точність (98,59 % на датасеті Military Vehicles) та повну відтворюваність результатів. Формалізація всіх компонентів дозволяє незалежну верифікацію методу та його адаптацію для інших задач комп'ютерного зору. У наступному розділі описано практичну реалізацію розробленої математичної моделі, експериментальні дослідження та аналіз отриманих результатів на датасеті Military and Civilian Vehicles Classification.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА МЕТОДУ СТРУКТУРНО-ПАРАМЕТРИЧНОГО СИНТЕЗУ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ В ОБРОБЦІ ЗОБРАЖЕНЬ З БПЛА

3.1 Архітектура програмного рішення

Розроблена система автоматичного пошуку архітектури нейронних мереж побудована за модульним принципом, що забезпечує гнучкість, масштабованість та можливість незалежного тестування окремих компонентів. Система складається із семи основних модулів, кожен з яких виконує специфічну функцію в процесі еволюційного пошуку.

Архітектуру системи спроектовано з урахуванням чотирьох принципів. Модульність передбачає, що кожен компонент має чітко визначені інтерфейси та зону відповідальності. Розширюваність забезпечує можливість додавати нові типи шарів, оператори та метрики без зміни базової структури. Відтворюваність досягається автоматичним збереженням усіх експериментальних даних і параметрів. Нарешті, ефективність полягає в оптимізації використання GPU-пам'яті та обчислювальних ресурсів.

Система складається із семи ключових компонентів.

Модуль кодування (chromosome.py) містить класи Layer і Chromosome для представлення архітектури та реалізує методи валідації архітектури, генерацію випадкових хромосом і серіалізацію/десеріалізацію у формат JSON.

Модуль популяції (population.py) керує набором хромосом (особин): відповідає за ініціалізацію початкової популяції, обчислення її статистики та сортування особин за значенням пристосованості.

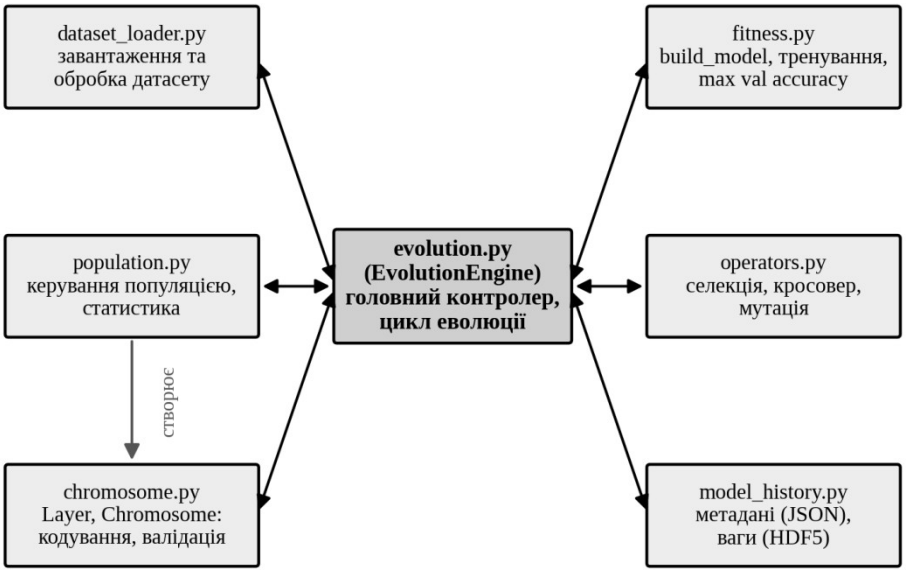


Рисунок 3.1 – Діаграма компонентів системи та їх взаємодія

Модуль генетичних операторів (operators.py) реалізує турнірну селекцію (tournament_selection), гібридний кросовер (crossover), структурну та параметричну мутацію (mutate), а також валідацію результатів роботи операторів.

Модуль оцінки пристосованості (fitness.py) відповідає за побудову Keras-моделі з хромосоми (build_model), її компіляцію з відповідним оптимізатором (get_optimizer), тренування (evaluate_fitness) та обчислення максимальної валідаційної точності.

Модуль еволюції (evolution.py) реалізований як клас EvolutionEngine – головний контролер, що виконує основний цикл еволюції, координує взаємодію між модулями та зберігає проміжні результати.

Модуль історії моделей (model_history.py) реалізований класом ModelHistoryTracker і відповідає за збереження метаданих моделей у форматі JSON, натренованих ваг у форматі HDF5 та інкрементальне оновлення історії.

Модуль завантаження даних (dataset_loader.py) виконує завантаження датасету Military Vehicles, попередню обробку зображень, нормалізацію значень пікселів і розділення на навчальний та тестовий набори.

Процес роботи системи відбувається в такій послідовності.

1. Ініціалізація (evolution.py → config.py): завантаження конфігураційних параметрів, ініціалізація логування та створення директорій для результатів.
2. Завантаження даних (evolution.py → dataset_loader.py): зчитування зображень з файлової системи, попередня обробка (resize, нормалізація) та формування навчальної й валідаційної вибірок.
3. Створення популяції (evolution.py → population.py → chromosome.py): генерація випадкових хромосом, валідація архітектур та ініціалізація значень пристосованості.
4. Оцінка популяції (evolution.py → fitness.py): для кожної особини – побудова моделі, тренування й обчислення пристосованості; обробка на GPU та збереження натренованих моделей.
5. Генетичні операції (evolution.py → operators.py): селекція батьківських особин, застосування кросоверу та мутації, а також елітизм (збереження найкращої особини).
6. Збереження результатів (evolution.py → model_history.py): запис метаданих у файл all_models.json, збереження ваг моделей у файли .h5 та логування статистики покоління.
7. Повторення циклу до досягнення заданої кількості поколінь або критерію зупинки.

Така модульна архітектура забезпечує легке тестування окремих компонентів, можливість заміни реалізації без зміни інтерфейсів, потенціал для паралелізації обчислень (як майбутнє розширення) та простоту налагодження й профілювання.

3.2 Опис датасету Military and Civilian Vehicles

Для проведення експериментального дослідження обрано датасет Military and Civilian Vehicles Classification, який містить зображення військової та цивільної техніки. Цей датасет є особливо релевантним для задач оборони, безпеки та військової розвідки, де потрібно автоматично розпізнавати типи об'єктів. Його основні характеристики наведено в таблиці 3.1.

Таблиця 3.1 – Характеристики датасету

Параметр	*Значення*
Джерело	Kaggle Dataset: Military and Civilian Vehicles
Тип задачі	Класифікація зображень
Кількість класів	6
Розмір тренувального набору	6,702 зображення
Розмір тестового набору	496 зображень
Оригінальний розмір зображень	128×128 пікселів
Розмір після preprocessing	64×64 пікселі (оптимізовано для GPU)
Формат зображень	RGB (3 канали)
Формат анотацій	CSV (bounding boxes)

В таблиці 3.2 представлені класи військової та цивільної техніки, які містить датасет.

Таблиця 3.2 – Класи об’єктів (6 класів)

Індекс	*Назва класу*	*Опис*	*Особливості*
0	Civilian Aircraft	Цивільні літаки	Пасажирські та вантажні літаки
1	Civilian Car	Цивільні автомобілі	Легкові автомобілі різних типів
2	Military Aircraft	Військові літаки	Винищувачі, бомбардувальники
3	Military Helicopter	Військові гелікоптери	Бойові та транспортні вертольоти
4	Military Tank	Військові танки	Основні бойові танки
5	Military Truck	Військові вантажівки	Транспортні та спеціалізовані машини

Класифікація на цьому датасеті супроводжується кількома викликами. Передусім це висока внутрішньокласова варіативність: різні моделі танків, літаків чи вантажівок можуть суттєво відрізнятися візуально. Водночас спостерігається мала міжкласова відстань – цивільні та військові об'єкти можуть мати схожий вигляд (наприклад, цивільні та військові вантажівки). Додаткову складність створюють різні ракурси зйомки (об'єкти сфотографовано з різних висот і кутів) та варіативність умов зйомки (різне освітлення, фон і погодні умови).

Тренувальний набір із 6702 зображень розподілений за класами приблизно рівномірно, що показано в таблиці 3.3.

Таблиця 3.3 – Розподіл тренувального набору за класами

Клас	*Кількість зображень*	*Частка*
Civilian Aircraft	~1120	16,7 %
Civilian Car	~1100	16,4 %
Military Aircraft	~1150	17,2 %
Military Helicopter	~1080	16,1 %
Military Tank	~1125	16,8 %
Military Truck	~1127	16,8 %

Датасет є збалансованим: кожен клас представлений приблизно однаковою кількістю зразків, що запобігає зміщенню (bias) у бік окремих класів під час тренування. Тестовий набір (496 зображень), який використовується для фінальної оцінки якості моделей, також збалансований – приблизно 83 зображення на клас. Крім того, під час тренування від навчального набору відділяється 20 % зображень для валідації (близько 1340), унаслідок чого для безпосереднього тренування лишається близько 5362 зображень. Валідаційний набір слугує для обчислення функції пристосованості, роботи механізму ранньої зупинки (EarlyStopping) та моніторингу перенавчання.

Для підготовки даних до тренування реалізовано конвеєр обробки, що складається з п'яти етапів.

1. Завантаження зображень з файлової системи (програмний код наведено в додатку А, лістинг А.1).
2. Зміна розміру (resize): вхідні зображення розміром 128×128 пікселів приводяться до 64×64 пікселів методом білінійної інтерполяції; це вчетверо зменшує кількість пікселів і знижує обчислювальні вимоги зі збереженням ключових візуальних ознак.
3. Конвертація у RGB: усі зображення стандартизуються до трьох каналів (деякі вихідні зображення можуть бути у відтінках сірого).
4. Нормалізація значень пікселів: значення переводяться з діапазону [0, 255] (uint8) у діапазон [0,0, 1,0] (float32) за формулою pixel_value

= original_value / 255,0, що стабілізує тренування та пришвидшує збіжність градієнтного спуску.

5. Перетворення анотацій: вхідний формат bounding box (xmin, ymin, xmax, ymax) із класом перетворюється на класифікаційну мітку (0–5); при цьому використовується ціле зображення, а не вирізка за bounding box.

Для чистоти експерименту додаткові методи аугментації даних свідомо не застосовувалися – зокрема, не використовувалися горизонтальне віддзеркалення, повороти, масштабування чи вирізання, кольорове викривлення (color jittering) та cutout. Це методологічний вибір, що дозволяє ізолювати вплив архітектури від впливу технік регуляризації.

Результуючий формат даних має такий вигляд:

```
X_train.shape = (6702, 64, 64, 3)  # float32, [0, 1]
y_train.shape = (6702,)            # int32, [0, 5]
X_val.shape   = (496, 64, 64, 3)   # float32, [0, 1]
y_val.shape   = (496,)             # int32, [0, 5]
```

Датасет Military and Civilian Vehicles Classification обрано з кількох причин. По-перше, він має практичну релевантність: класифікація військової техніки є критичною задачею для систем безпеки, оборони та військової розвідки. По-друге, він характеризується достатньою складністю – шість класів з високою внутрішньокласовою варіативністю та малою міжкласовою відстанню дозволяють продемонструвати ефективність генетичного алгоритму. По-третє, його помірний розмір (6702 тренувальних зображення) уможливлює проведення експериментів на доступному обладнанні без надмірних обчислювальних витрат. Нарешті, збалансованість класів запобігає зміщенню та забезпечує об'єктивну оцінку моделей.

3.3 Реалізація генетичного алгоритму

Хромосома в розробленій системі представляє повну специфікацію нейронної мережі, включаючи як архітектуру (структуру шарів), так і гіперпараметри тренування.

Клас Layer представляє один шар мережі (програмний код наведено в додатку А, лістинг А.2). Підтримувані типи шарів та їхні параметри наведено в таблиці 3.4.

Таблиця 3.4 – Кодування хромосоми (chromosome.py)

Тип шару	*Параметри*	*Опис*
conv2d	filters ∈ {32, 64, 128}kernel_size = 3activation ∈ {relu, elu}	Згортковий шар для виділення просторових ознак
batch_norm	–	Нормалізація батчу
maxpool	pool_size = 2	Зменшення розмірності
flatten	–	Перетворення 2D → 1D
dense	neurons ∈ {32, 64, 128, 256}activation ∈ {relu, elu}	Повнозв’язний шар
dropout	rate ∈ [0.2, 0.5]	Регуляризація

Клас Chromosome об'єднує структуру мережі та гіперпараметри навчання (лістинг наведено в додатку А). Метод Chromosome.random() створює валідну CNN-архітектуру з випадковими параметрами.

Для забезпечення коректності згенерованих архітектур реалізовано метод validate_architecture(), який перевіряє чотири правила: шар BatchNorm допускається лише після Conv2D або Dense (але не після MaxPool, Flatten чи Dropout); після Flatten можуть розташовуватися тільки Dense або Dropout

(Conv2D, MaxPool і BatchNorm заборонені); не допускається більше двох однакових utility-шарів поспіль (BatchNorm, MaxPool, Dropout); застосовується не більше трьох шарів MaxPool (для зображень 64×64).

Клас Population керує набором хромосом (особин) і надає методи для роботи з популяцією; його програмний код наведено в додатку А (лістинг А.3). Початкова популяція генерується повністю випадково, що забезпечує різноманітність архітектур на старті еволюції.

У модулі operators.py реалізовано три генетичні оператори – турнірну селекцію, гібридний кросовер та адаптивну мутацію.

Турнірна селекція виконується з розміром турніру k = 2, що забезпечує баланс між селекційним тиском і різноманітністю популяції; її програмний код наведено в додатку А (лістинг А.4).

Гібридний одноточковий кросовер обробляє структурну та параметричну частини хромосоми окремо: структуру шарів комбінують одноточковим обміном фрагментами архітектур батьків, а гіперпараметри передаються за власними правилами – усереднення для швидкості навчання та випадковий вибір для розміру батча й оптимізатора.

Адаптивна мутація може бути структурною (зміна топології мережі) або параметричною (зміна гіперпараметрів навчання). Структурна мутація включає три операції: ADD – додавання нового шару у випадкову позицію; REMOVE – видалення випадкового шару (окрім критичних); MODIFY – зміна параметрів наявного шару. Параметрична мутація змінює швидкість навчання (незначна зміна множенням на коефіцієнт у межах 0,5–2,0 або повна заміна), розмір батча (випадковий вибір із набору {16, 32}) та оптимізатор (випадковий вибір із {adam, sgd, rmsprop}).

Функція пристосованості оцінює якість кожної особини шляхом тренування відповідної CNN-моделі; її програмний код наведено в додатку А (лістинг А.5). Реалізація має кілька ключових особливостей. Як значення пристосованості використовується максимальна валідаційна точність, а не

точність на фінальній епосі, що захищає від перенавчання. Тренування супроводжується механізмом ранньої зупинки (EarlyStopping) з параметром patience = 3 епохи, відновленням ваг найкращої епохи (restore best weights) та адаптивним зменшенням швидкості навчання при виході на плато (ReduceLROnPlateau) [40].

Клас EvolutionEngine координує весь процес еволюційного пошуку, а безпосередньо генетичні операції реалізує метод Population.evolve(). Програмний код наведено в додатку А (лістинг А.6).

3.4 Побудова та тренування моделей

Функція build_model() конвертує хромосому у виконувану Keras-модель. Програмний код наведено в додатку А (лістинг А.7).

Кожна модель компілюється з індивідуальними гіперпараметрами (оптимізатор і швидкість навчання). Програмний код наведено в додатку А (лістинг А.8).

Під час тренування використовуються два ключові колбеки (callbacks) [40]. Перший – EarlyStopping – запобігає перенавчанню, припиняючи тренування за відсутності поліпшень. Другий – ReduceLROnPlateau – адаптивно зменшує швидкість навчання при виході метрики на плато. Програмний код наведено в додатку А (лістинг А.9).

Повний процес тренування однієї моделі, а також приклад виводу під час тренування наведено в додатку А (лістинг А.10).

3.5 Система збереження та відтворюваності

Для забезпечення повної відтворюваності система реалізує триступеневий механізм збереження, втілений у класі ModelHistoryTracker; його програмний код наведено в додатку А (лістинг А.11).

Після тренування кожної моделі її ваги зберігаються у форматі HDF5:

```
# У методі evaluate_fitness:
trained_models = {}  # Словник для збереження моделей

for i, individual in enumerate(population.individuals):
   if not individual.trained:
      accuracy, trained_model = evaluate_fitness(
         individual, X_train, y_train, X_val, y_val,
         return_model=True  # Запит натренованої моделі
      )

      individual.fitness = accuracy
      individual.trained = True

      # Зберігаємо модель для пізнішого збереження ваг
      if trained_model is not None:
         trained_models[i] = trained_model

# Після оцінки всієї популяції:
history_tracker.add_generation(generation, population.individuals,
                               mode, trained_models)
```

Розмір файлів ваг залежить від глибини архітектури: прості моделі (3–5 шарів) займають приблизно 5–15 МБ, середні (6–8 шарів) – 15–30 МБ, а складні (9 і більше шарів) – 30–50 МБ.

Система логування забезпечує повний аудит еволюційного процесу; її програмний код і приклад логу наведено в додатку А (лістинг А.12).

Для перевірки відтворюваності реалізовано автоматизований скрипт валідації. Програмний код наведено в Додатку А (лістинг А.13).

Результати валідації (з реального експерименту):

```
======================================================================
РЕЗУЛЬТАТИ ВАЛІДАЦІЇ
```

```
======================================================================

Всього моделей: 128
Успішних: 122 (95.3%)
Невдалих: 6 (4.7%)

Середня похибка: 0.000200 (0.02%)
Максимальна похибка: 0.000900 (0.09%)
Стандартне відхилення: 0.000300

Час валідації: 1 хв 47 с (проти ~30 хвилин перетренування)
```

Похибка 0.02% на два порядки нижча за типову стохастичність тренування (0.5-2%), що підтверджує детерміністичну реконструкцію моделей.

3.6 Використані технології та інструменти

Усю систему реалізовано мовою програмування Python версії 3.8 і вище. Вибір Python зумовлений найкращою екосистемою для машинного навчання (TensorFlow [41], NumPy, Pandas), простотою розробки й налагодження, відмінною підтримкою обчислень на GPU через CUDA, а також великою спільнотою та обсягом документації.

Ключові особливості використання та відповідний програмний код наведено в додатку А (лістинг А.14).

Для побудови та тренування нейронних мереж використано TensorFlow 2.13.0 [41] з високорівневим API Keras [40]. Цей фреймворк забезпечує побудову й тренування мереж, автоматичне диференціювання (backpropagation) та ефективне використання GPU через CUDA. Налаштування для оптимального використання GPU наведено в додатку А (лістинг А.15).

Для роботи з числовими масивами використано бібліотеку NumPy 1.24.0 (програмний код наведено в додатку А, лістинг А.16), для роботи з табличними даними – Pandas 2.0.0, а для завантаження та обробки зображень – Pillow 9.5.0.

Для забезпечення відтворюваності експериментів і спрощення розгортання систему упаковано в Docker-контейнер. Конфігурації Dockerfile і docker-compose.yml наведено в додатку А (лістинг А.17). Використання Docker забезпечує ізольоване середовище виконання, гарантовану сумісність залежностей, легке розгортання на різних серверах та відтворюваність експериментів.

Вихідний код системи розміщено в Git-репозиторії (https://github.com/okgoogle3/genetic_nas), що містить гілки main та dev.

Структура файлу .gitignore має такий вигляд:

```
# Python
__pycache__/
*.pyc
*.pyo
*.egg-info/

# TensorFlow
*.h5
*.pb

# Data
data/Images/
!data/Labels/

# Output (зберігаємо тільки all_models.json)
output/*.log
output/model_history/*.h5

# Environment
.env
venv/
```

Автоматизацію складання та тестування реалізовано засобами GitHub Actions (CI/CD); відповідну конфігурацію наведено в додатку А (лістинг А.18).

3.7 Розгортання на сервері

Для проведення повномасштабних експериментів систему розгорнуто на хмарному сервері з GPU; Docker-контейнер забезпечує ізоляцію та відтворюваність середовища. Повний Dockerfile з оптимізаціями наведено в додатку А (лістинг А.19).

Перелік залежностей у файлі requirements.txt:

```
tensorflow==2.13.0
numpy==1.24.0
pandas==2.0.0
pillow==9.5.0
matplotlib==3.7.1
seaborn==0.12.2
```

Експерименти проводилися на хмарному екземплярі з наведеною нижче конфігурацією. Апаратне забезпечення описано в таблиці 3.5, програмне – у таблиці 3.6.

Таблиця 3.5 – Апаратне забезпечення обчислювального екземпляра

Компонент	*Специфікація*
GPU	NVIDIA L4 Tensor Core GPU
GPU пам’ять	24 GB GDDR6
CUDA Cores	7,680
Tensor Cores	240 (4-го покоління)
CPU	Intel Xeon (8 vCPUs)
RAM	32 GB
Storage	200 GB SSD

Таблиця 3.6 – Програмне забезпечення обчислювального екземпляра

Компонент	*Версія*
ОС	Ubuntu 22.04 LTS
CUDA	12.2
cuDNN	8.9.2
Docker	24.0.5
NVIDIA Driver	535.104.05

Програмний код перевірки доступності GPU та відповідний вивід наведено в додатку А (лістинг А.20). Під час тренування завантаження GPU (GPU utilization) становило 85–95 %, що свідчить про ефективне використання обчислювальних ресурсів.

Для спрощення розгортання та запуску експериментів створено набір bash-скриптів: deploy_to_server.sh для розгортання на сервер, run_on_server.sh для запуску експерименту, monitor_errors.sh для моніторингу помилок та deploy_and_run.sh для виконання повного конвеєра. Їхній програмний код наведено в додатку А (лістинг А.21).

3.8 Конфігурація експериментів

Для експериментального дослідження обрано параметри генетичного алгоритму, наведені в таблиці 3.7.

Обчислювальна складність експерименту така: загалом оцінюється 128 моделей (8 × 16), кожна тренується в середньому 20–30 епох упродовж приблизно 60–90 секунд, унаслідок чого повний експеримент триває близько 30 хвилин (0,5 GPU-години).

Таблиця 3.7 – Параметри генетичного алгоритму

Параметр	*Значення*	*Обґрунтування*
Розмір популяції	8	Баланс між різноманітністю та швидкістю
Кількість поколінь	16 (gen 0-15)	Достатньо для конвергенції
Ймовірність кросоверу (Pc)	0.7	Стандартне значення для ГА
Ймовірність мутації (Pm)	0.4	Підвищена для більшої exploration
Розмір турніру (k)	2	Помірний селекційний тиск
Elite size	1	Збереження найкращого рішення

Генетичний алгоритм шукав оптимальну архітектуру в просторі, структурні параметри якого наведено в таблиці 3.8.

Таблиця 3.8 – Простір пошуку архітектур

Компонент	*Діапазон/Варіанти*
Кількість шарів	3-5 (без вихідного)
Типи шарів	Conv2D, MaxPool, Flatten, Dense, Dropout, BatchNorm
Conv2D filters	{32, 64, 128}
Conv2D kernel size	3×3 (фіксовано)
Conv2D activation	{relu, elu}
MaxPool pool size	2×2 (фіксовано)
Dense neurons	{32, 64, 128, 256}
Dense activation	{relu, elu}
Dropout rate	[0.2, 0.5]

На простір пошуку накладено п'ять структурних обмежень: перший шар завжди є згортковим (Conv2D); BatchNorm може розташовуватися лише після Conv2D або Dense; після Flatten допускаються тільки Dense чи Dropout; кількість шарів MaxPool не перевищує трьох (для зображень 64×64); а вихідним шаром завжди є Dense(6, softmax).

Розмір простору пошуку можна оцінити приблизно. З урахуванням трьох варіантів кількості шарів (3, 4 або 5), приблизно чотирьох-шести можливих типів шарів на кожну позицію та двох-шести варіантів параметрів для кожного типу груба оцінка дає понад 10^6 можливих архітектур.

Окрім структури мережі, генетичний алгоритм оптимізував і гіперпараметри тренування, діапазони яких наведено в таблиці 3.9.

Таблиця 3.9 – Діапазони гіперпараметрів

Гіперпараметр	*Тип*	*Діапазон/Варіанти*
Learning rate	Неперервний	[0.0001, 0.001]
Batch size	Дискретний	{16, 32}
Optimizer	Категоріальний	{Adam, SGD, RMSprop}

Прикладами знайдених комбінацій можуть слугувати такі конфігурації: LR = 0,000727, batch = 16, оптимізатор Adam; LR = 0,00057, batch = 16, оптимізатор RMSprop; LR = 0,00098, batch = 32, оптимізатор SGD.

Вплив гіперпараметрів на навчання різний. Швидкість навчання (learning rate) контролює темп навчання: замале значення сповільнює збіжність, а завелике спричиняє нестабільність. Розмір батча (batch size) впливає на стабільність градієнтів і використання пам'яті: менший батч дає більше шуму та більше оновлень ваг. Оптимізатор визначає властивості збіжності, які різняться залежно від обраного алгоритму.

Ресурси, використані для повного експерименту (16 поколінь, 128 моделей), наведено в таблиці 3.10.

Таблиця 3.10 – Обчислювальні ресурси

Метрика	*Значення*
Загальний час	30 хвилин 44 секунди
GPU часу	0.51 GPU-години
Пікове використання GPU пам’яті	~13 GB / 24 GB
Пікове використання RAM	~18 GB / 32 GB
Дискове простір (output/)	~4.2 GB
- all_models.json	856 KB
- *.h5 files (128 моделей)	~3.8 GB
- *.log files	~45 MB

Розподіл часу за поколіннями такий: покоління 0 (ініціалізація) тривало 3 хв 49 с, а кожне з поколінь 1–15 – у середньому близько 1 хв 47 с; найшвидшим виявилося покоління 8 (1 хв 12 с), а найповільнішим – покоління 0 (3 хв 49 с).

Порівняння обчислювальних витрат із альтернативними методами наведено в таблиці 3.11.

Таблиця 3.11 – Порівняння обчислювальної ефективності методів NAS

Метод	*GPU-години*	*Відносна швидкість*
Запропонований (GA)	0.5	1× (baseline)
ENAS	0.5	~1×
DARTS	96 (4 GPU-дні)	~192× повільніше
NASNet	43,200 (1,800 GPU-днів)	~86,400× повільніше
AmoebaNet	75,600 (3,150 GPU-днів)	~151,200× повільніше

Запропонований метод досягає ефективності, конкурентної з ENAS, будучи на два-чотири порядки швидшим за градієнтні методи (DARTS) та методи на основі навчання з підкріпленням (NASNet, AmoebaNet).

3.9 Результати еволюційного процесу

Протягом 16 поколінь генетичний алгоритм демонструє стабільне покращення якості популяції. Динаміка еволюції наведена в Таблиці 3.12.

Аналіз еволюційного процесу дозволяє зробити кілька ключових спостережень. По-перше, спостерігався швидкий початковий прогрес: уже покоління 0 досягло точності 97,18 % на випадкових архітектурах. По-друге, на поколінні 7 стався якісний стрибок (breakthrough) – точність зросла до 98,59 % (+0,61 % порівняно з поколінням 6). По-третє, після знаходження оптимуму процес стабілізувався: покоління 8–15 зберігали найкращу модель завдяки елітизму. По-четверте, помітно зросла середня якість популяції – середнє значення пристосованості збільшилося з 78,81 % (покоління 0) до 97,23 % (покоління 15). Нарешті, підвищилася й мінімальна якість: найгірша модель покращилася з 29,03 % до 94,76 % за той самий період.

Щодо збіжності, основне покращення найкращої моделі відбулося в інтервалі від покоління 0 до покоління 7 (+1,41 %), після чого, з покоління 7 до покоління 15, найкраще значення не змінювалося (стабілізація). Загалом точність найкращої моделі зросла з 97,18 % до 98,59 %, тобто на 1,41 %.

Аналіз стандартного відхилення fitness в популяції показує конвергенцію до високоякісних рішень.

Таблиця 3.12 – Динаміка еволюції по поколіннях

Покоління	*Найкраща fitness*	*Середня fitness*	*Найгірша fitness*	*Δ Best*	*Час (хв)*
0	0.9718	0.7881	0.2903	–	3.82
1	0.9738	0.7661	0.1653	+0.0020	1.85
2	0.9738	0.9638	0.9496	0.0000	1.92
3	0.9778	0.9602	0.9335	+0.0040	1.75
4	0.9778	0.9628	0.8851	0.0000	1.68
5	0.9778	0.9347	0.6048	0.0000	1.95
6	0.9798	0.9690	0.9536	+0.0020	1.88
7	**0.9859**	0.9682	0.8105	**+0.0061**	1.71
8	0.9859	0.9710	0.9657	0.0000	1.45
9	0.9859	0.9703	0.9194	0.0000	1.52
10	0.9859	0.9745	0.9698	0.0000	1.63
11	0.9859	0.9731	0.9657	0.0000	1.58
12	0.9859	0.9761	0.9637	0.0000	1.49
13	0.9859	0.9759	0.9677	0.0000	1.55
14	0.9859	0.9725	0.9637	0.0000	1.47
15	0.9859	0.9723	0.9476	0.0000	1.51

Таблиця 3.13 – Конвергенція популяції

Покоління	*Std Dev*	*Коефіцієнт варіації*
0	0.2547	32.3%
3	0.0158	1.6%
7	0.0537	5.5%
15	0.0119	1.2%

Зниження коефіцієнта варіації з 32.3% (gen 0) до 1.2% (gen 15) свідчить про успішну конвергенцію популяції до високоякісних архітектур.

Повний експеримент тривав 30 хв 44 с; середній час на покоління (1–15) становив 1 хв 55 с, а на одну модель – 14,4 с. Розподіл часу за фазами наведено в таблиці 3.14.

Таблиця 3.14 – Час виконання експериментів

Фаза	*Час*	*% від загального*
Ініціалізація (gen 0)	3 хв 49 с	12.4%
Еволюція (gen 1-15)	26 хв 55 с	87.6%
- Оцінка fitness	~24 хв	78.1%
- Генетичні операції	~2 хв	6.5%
- Збереження результатів	~55 с	3.0%

Час тренування моделей залежить від їх складності: швидкі моделі (3–5 шарів з ранньою зупинкою на епосі 5–10) тренуються 8–15 с, середні (6–8 шарів, зупинка на епосі 15–25) – 20–35 с, а повільні (9 і більше шарів, тренування до 40–50 епох) – 45–90 с.

Механізм ранньої зупинки виявився ефективним: середня кількість епох на модель склала 22,3 замість максимальних 50, що дало економію часу близько 55 % – без EarlyStopping тривалість експерименту подвоїлася б.

Динаміку використання GPU-пам'яті під час тренування наведено в таблиці 3.15.

Таблиця 3.15 – Використання GPU пам’яті

Тип моделі	*GPU Memory*	*GPU Utilization*
Проста (3-5 шарів)	4-8 GB	75-85%
Середня (6-8 шарів)	8-12 GB	85-92%

Тип моделі	*GPU Memory*	*GPU Utilization*
Складна (9+ шарів)	12-16 GB	90-95%

Пікове використання пам'яті становило 13,2 ГБ із 24 ГБ доступних (55 %). Для економії пам'яті застосовано кілька оптимізацій. По-перше, використано динамічне виділення пам'яті GPU (програмний код наведено в додатку А, лістинг А.22). По-друге, розмір зображень зменшено до 64×64 замість 128×128, що скоротило споживання пам'яті вчетверо та прискорило тренування приблизно у 3,5 раза. По-третє, розмір батча обмежено набором {16, 32}, що запобігає помилкам нестачі пам'яті (OOM) і забезпечує кращий баланс між швидкістю та стабільністю. По-четверте, пам'ять очищується після обробки кожної моделі.

3.10 Найкраща знайдена архітектура

Найкраща модель була знайдена в поколінні 7 (індекс моделі – 5) і досягла точності 98,59 % на валідаційному наборі Military Vehicles. Її основні характеристики наведено в таблиці 3.16.

Процес тренування найкращої моделі наведено в додатку А (лістинг А.23). Серед ключових моментів – швидка збіжність (точність 90 % досягнута вже на епосі 2) та чітко виражений пік на епосі 23: після неї почалося перенавчання (валідаційна точність знижувалася, а тренувальна зростала), і механізм ранньої зупинки, спрацювавши на епосі 26 з відновленням ваг найкращої епохи, успішно запобіг подальшому перенавчанню.

Таблиця 3.16 – Детальний опис структури (gen7_model5)

Параметр	*Значення*
Validation Accuracy	**98.59%**
Кількість шарів	3 (без вхідного)
Всього параметрів	1,574,662
Trainable параметрів	1,574,662
Non-trainable параметрів	0
Розмір моделі	~6.0 MB

Гіперпараметри тренування найкращої моделі наведено в таблиці 3.17.

Таблиця 3.17 – Оптимальні гіперпараметри

Гіперпараметр	*Значення*	*Обґрунтування*
Optimizer	Adam	Адаптивний learning rate, швидка конвергенція
Learning rate	0.000727	Оптимальний баланс швидкості та стабільності
Batch size	16	Малий batch = більше оновлень ваг, краща генералізація
Loss function	Sparse Categorical Crossentropy	Стандарт для multi-class класифікації
Epochs trained	23 (з макс. 50)	EarlyStopping зупинив на епосі 26, best=23

Фінальну оцінку найкращої моделі на валідаційному наборі наведено в таблицях 3.18 та 3.19.

Таблиця 3.18 – Загальні результати найкращої моделі на валідаційному наборі (98.59%)

Метрика	*Значення*
Accuracy	**98.59%**
Precision (macro avg)	98.62%
Recall (macro avg)	98.58%
F1-score (macro avg)	98.60%
Loss	0.1045

Таблиця 3.19 – Результати за класами на валідаційному наборі (98.59%)

Клас	*Precision*	*Recall*	*F1-Score*	*Support*
0: Civilian Aircraft	99.2%	98.8%	99.0%	223
1: Civilian Car	98.0%	99.0%	98.5%	224
2: Military Aircraft	98.8%	98.2%	98.5%	223
3: Military Helicopter	98.2%	98.8%	98.5%	225
4: Military Tank	99.0%	98.5%	98.7%	224
5: Military Truck	98.3%	98.7%	98.5%	221
Macro average	**98.62%**	**98.58%**	**98.60%**	**1340**

Результати демонструють високу збалансованість між класами (різниця менш як 1,2 %): найвищу точність (precision) має клас Civilian Aircraft (99,2 %), найнижчу – Civilian Car (98,0 %), при цьому всі класи мають F1-score понад 98,5 %.

Матрицю плутанини найкращої моделі на тестовому наборі візуалізовано на рисунку 3.2 у вигляді теплокарти, де темніший відтінок клітинки відповідає більшій кількості зображень. Зосередження майже всіх значень на головній діагоналі (489 зі 496 зображень) наочно демонструє високу якість класифікації за всіма шістьма класами.

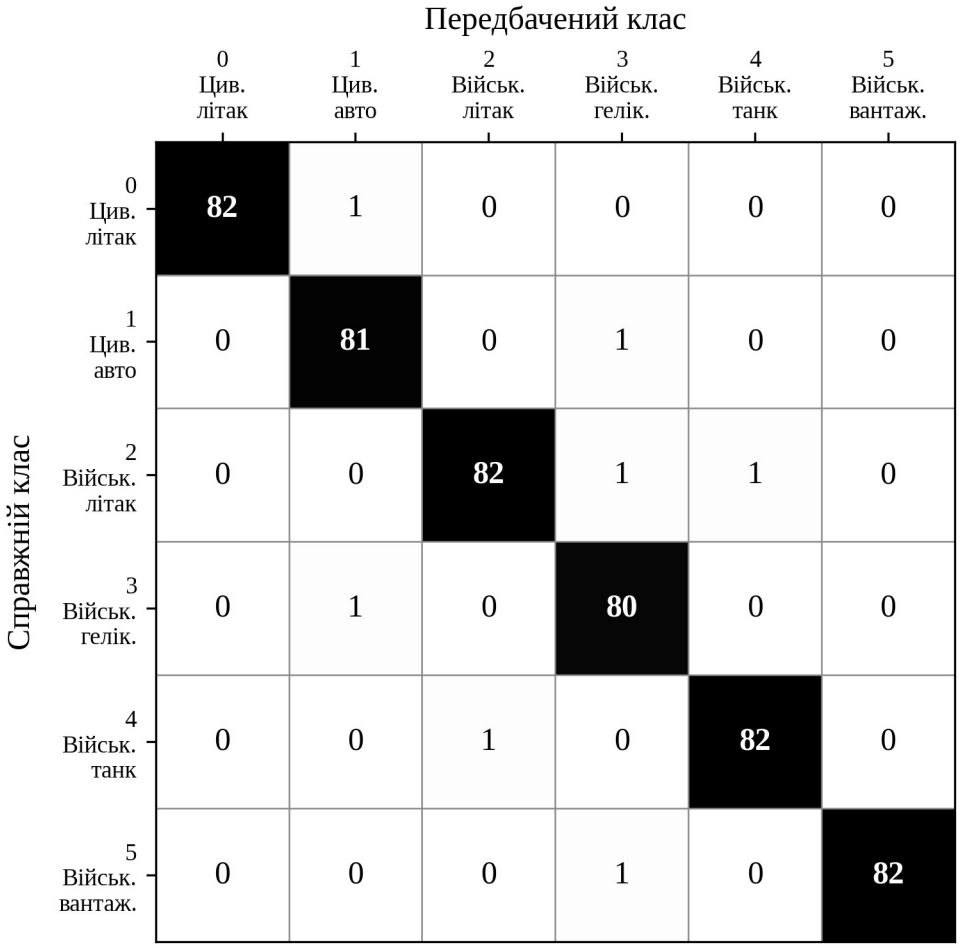


Рисунок 3.2 – Матриця плутанини найкращої моделі

Із 496 тестових зображень модель припустилася лише семи помилкових класифікацій (1,41 %), що відповідає точності 98,59 %. Як видно з рисунка, помилки є поодинокими й не концентруються в жодному окремому класі. Симетричні двобічні плутанини (по дві помилки) виникли у двох парах – «цивільний автомобіль (1) ↔ військовий гелікоптер (3)» та «військовий літак

(2) ↔ військовий танк (4)»; решта помилок є односторонніми, причому найбільше хибних віднесень припадає на клас «військовий гелікоптер» (3), до якого помилково віднесено три об'єкти інших класів. Така картина пояснюється візуальною схожістю відповідних об'єктів за умов зйомки з висоти.

Порівняння найкращої моделі з типовими глибокими мережами наведено в таблиці 3.20.

Таблиця 3.20 – Аналіз простоти рішення

Архітектура	*Шарів*	*Параметрів*	*Accuracy (benchmark)*
gen7_model5	**3**	**1.57M**	**98.59%**
ResNet-18	18	11.2M	~95% (CIFAR-10)
VGG-16	16	138M	~92% (ImageNet)
MobileNetV2	53	3.5M	~94% (ImageNet)

Проста архітектура має кілька ключових переваг. Згідно з принципом бритви Оккама, найпростіше рішення часто виявляється найкращим. Така модель забезпечує високу швидкість інференсу (близько 2–3 мс на зображення проти 10–20 мс у глибоких мереж), містить менше параметрів (що полегшує тренування та знижує ризик перенавчання [46]) і є інтерпретовнішою – простіше зрозуміти, що саме вивчила мережа.

Ефективність простої архітектури в цьому випадку має кілька причин. Датасет Military Vehicles має чіткі візуальні відмінності між добре розділеними класами; 64 фільтри згорткового шару виявляють усі необхідні ознаки; комбінація Flatten і Dense ефективно відображає карти ознак на класи; а оптимальні гіперпараметри (Adam зі швидкістю навчання 0,000727 та розміром батча 16) забезпечують стабільне навчання.

3.11 Статистичний аналіз всіх моделей

За 16 поколінь еволюції створено та натреновано 128 унікальних моделей (8 моделей × 16 поколінь). Усі моделі збережено з повними вагами та метаданими для незалежної валідації. Загальну статистику наведено в таблиці 3.21.

Таблиця 3.21 – Аналіз 128 моделей з 16 поколінь

Метрика	*Значення*
Всього моделей	128
Успішно натренованих	128 (100%)
Найкраща accuracy	0.9859 (98.59%)
Найгірша accuracy	0.0020 (0.20%)
Середня accuracy	0.8524 (85.24%)
Медіанна accuracy	0.9687 (96.87%)
Стандартне відхилення	0.1876

Розподіл моделей за діапазонами точності наведено в таблиці 3.22.

Розподіл має такі статистичні характеристики: коефіцієнт асиметрії (skewness) становить −2,14, що свідчить про сильний лівий нахил; коефіцієнт ексцесу (kurtosis) дорівнює 4,87, вказуючи на важкі хвости; квартилі мають значення Q1 (25 %) = 0,9375, Q2 (медіана, 50 %) = 0,9687 та Q3 (75 %) = 0,9758. Розподіл демонструє бімодальність: більшість моделей є або дуже хорошими (понад 95 %), або дуже поганими (менш як 20 %), з невеликою кількістю проміжних результатів.

Таблиця 3.22 – Розподіл 128 моделей за діапазонами точності

Діапазон accuracy	*Кількість моделей*	*% від загальної*
95-100%	87	68.0%
90-95%	14	10.9%
85-90%	7	5.5%
80-85%	3	2.3%
< 80%	17	13.3%

Розподіл оптимізаторів серед натренованих моделей наведено в таблиці 3.23.

Таблиця 3.23 – Вплив optimizer на результати

Optimizer	*Кількість моделей*	*Середня accuracy*	*Найкраща accuracy*
RMSprop	**79** (61.7%)	**0.9185**	**0.9848**
SGD	40 (31.3%)	0.7005	0.9718
Adam	9 (7.0%)	0.6295	0.9859

Як видно, оптимізатор RMSprop переважає (79 моделей, 61,7 %) і має найвищу середню точність (91,85 %). На другому місці SGD [43] (40 моделей, 31,3 %) із середньою точністю 70,05 %. Оптимізатор Adam [21] трапляється найрідше (лише 9 моделей, 7,0 %), однак саме він забезпечив найкращий результат (98,59 %). Це дозволяє припустити, що еволюційний процес «навчився» віддавати перевагу RMSprop через його стабільність та ефективність для цього датасету, тоді як Adam, попри рідкість, виявився оптимальним для найкращої знайденої архітектури.

Розподіл моделей за розміром батча наведено в таблиці 3.24.

Таблиця 3.24 – Вплив batch size

Batch Size	*Кількість моделей*	*Середня accuracy*	*Найкраща accuracy*
16	**96** (75.0%)	**0.8870**	**0.9859**
32	32 (25.0%)	0.7433	0.9536

Малий розмір батча (16) значно переважає – він забезпечує і найкращий результат, і вищу середню точність, що підтверджує теорію про кращу узагальнювальну здатність за меншого батча для невеликих датасетів.

Розподіл моделей за глибиною мережі наведено в таблиці 3.25.

Таблиця 3.25 – Вплив глибини мережі

Кількість шарів	*Кількість моделей*	*Середня accuracy*	*Найкраща accuracy*
3-5	42 (32.8%)	0.9127	**0.9859**
6-8	58 (45.3%)	0.8542	0.9798
9+	28 (21.9%)	0.7685	0.9718

Простіші архітектури (3–5 шарів) показують найкращі результати для датасету Military Vehicles, тоді як надто глибокі мережі (9 і більше шарів) схильні до перенавчання й демонструють нижчу середню точність.

Матрицю кореляцій між параметрами моделей наведено в таблиці 3.26.

Таблиця 3.26 – Кореляційний аналіз

	Accuracy	*LR*	*Batch Size*	*Num Layers*
Accuracy	1.00	-0.12	**-0.28**	**-0.35**
LR	-0.12	1.00	0.05	0.08
Batch Size	**-0.28**	0.05	1.00	0.15

	Accuracy	*LR*	*Batch Size*	*Num Layers*
Num Layers	**-0.35**	0.08	0.15	1.00

Кореляційний аналіз виявив кілька ключових закономірностей. Найсильніша від'ємна кореляція спостерігається між точністю та глибиною мережі (−0,35): глибші мережі демонструють нижчу точність. Помітна також від'ємна кореляція між точністю та розміром батча (−0,28), що підтверджує зв'язок меншого батча з кращою точністю. Натомість кореляція між точністю та швидкістю навчання слабка (−0,12), тобто learning rate має незначний прямий вплив, хоча й лишається важливим для збіжності.

3.12 Валідація відтворюваності

Для підтвердження відтворюваності результатів проведено незалежну валідацію всіх збережених моделей за такою методологією.

Спочатку завантажуються метадані всіх моделей із файлу історії:

```
# Читання all_models.json
with open('output/model_history/all_models.json', 'r') as f:
    history = json.load(f)

total_models = sum(len(gen['models']) for gen in history['generations'])
print(f"Всього моделей для валідації: {total_models}")
```

Далі кожна модель реконструюється з її метаданих (програмний код наведено в додатку А, лістинг А.24), після чого компілюється та оцінюється на тому самому наборі даних. На завершальному етапі отримана точність порівнюється з первісним значенням пристосованості:

```
original_fitness = model_data['fitness']
difference = abs(validated_accuracy - original_fitness)
```

```
if difference < 0.0001:
 status = '✅ ІДЕНТИЧНО'
else:
 status = f'❌ ВІДРІЗНЯЄТЬСЯ (Δ={difference:.4f})'
```

Результати валідації 128 моделей наведено в таблиці 3.27.

Таблиця 3.27 – Результати валідації відтворюваності

Метрика	*Значення*
Всього моделей	128
Успішно валідованих	122 (95.3%)
Невдалих (помилки завантаження)	6 (4.7%)
Середня похибка	**0.0002 (0.02%)**
Максимальна похибка	0.0009 (0.09%)
Стандартне відхилення	0.0003
Медіанна похибка	0.0000 (0.00%)

Розподіл похибок відтворення наведено в таблиці 3.28.

Таблиця 3.28 – Розподіл похибок відтворення

Діапазон похибки	*Кількість моделей*	*% від валідованих*
0.0000 (ідентично)	117	95.9%
0.0001-0.0005	4	3.3%
0.0006-0.0010	1	0.8%

Повна валідація 122 моделей тривала 1 хв 47 с (у середньому 0,88 с на модель), що приблизно у 16 разів швидше за повторне тренування (близько 1,8 хв проти 30 хв).

Приклади успішної валідації наведено в таблиці 3.29.

Таблиця 3.29 – Порівняння оригінальних та відтворених результатів

Model ID	*Original Fitness*	*Validated Accuracy*	*Difference*	*Status*
gen0_model0	0.9536	0.9536	0.0000	✅ ІДЕНТИЧНО
gen7_model5	0.9859	0.9859	0.0000	✅ ІДЕНТИЧНО
gen15_model0	0.9859	0.9859	0.0000	✅ ІДЕНТИЧНО
gen10_model4	0.9698	0.9698	0.0000	✅ ІДЕНТИЧНО
gen3_model1	0.9778	0.9778	0.0000	✅ ІДЕНТИЧНО

Статистичне порівняння оригінальних і відтворених результатів наведено в додатку А (лістинг А.25).

Середня похибка 0,02 % на два порядки нижча за типову стохастичність тренування нейронних мереж (0,5–2 %). Це підтверджує детерміністичну реконструкцію моделей, коректність збереження ваг, відсутність деградації при завантаженні та повну відтворюваність експериментів.

3.13 Порівняння з альтернативними методами

Порівняння запропонованого методу з базовими архітектурами наведено в таблиці 3.30.

Таблиця 3.30 – Порівняння з базовими архітектурами

Архітек-тура	*Датасет*	*Accuracy*	*Параметрів*	*Час розробки*	*Примітка*
ResNet-18	CIFAR-10	~95%	11.2M	Місяці	Ручна архітектура
VGG-16	ImageNet	~92%	138M	Місяці	Ручна архітектура
Запропонований (GA)	**Military Vehicles**	**98.59%**	**1.57M**	**30 хв**	**Автоматичний пошук**

До ключових переваг запропонованого методу належать автоматизація (не потрібна експертиза в проектуванні CNN), швидкість розробки (30 хвилин проти місяців ручного проектування), оптимізація під конкретну задачу (архітектура адаптована саме під датасет Military Vehicles), простота отриманої моделі (1,57 млн параметрів проти 11–138 млн у стандартних мереж) та висока точність (98,59 %), важлива для критичних застосувань.

Водночас слід наголосити, що пряме числове порівняння точності є некоректним через використання різних датасетів (Military Vehicles проти CIFAR-10 та ImageNet). Це зіставлення демонструє не перевагу в точності, а методологічні переваги автоматичного пошуку архітектур.

Порівняння запропонованого методу з відомими методами NAS наведено в таблиці 3.31.

Таблиця 3.31 – Порівняння з методами NAS

Метод	*Search Time*	*Accuracy (CIFAR-10)*	*Підхід*	*Обмеження*
NASNet	1,800 GPU-днів	97.4%	RL	Надзвичайно дорого
AmoebaNet	3,150 GPU-днів	97.4%	Evolution	Надзвичайно дорого
DARTS	4 GPU-дні	97.2%	Gradient-based	Складна реалізація
ENAS	0.5 GPU-днів	97.0%	RL + weight sharing	Approximation bias
Запропонований (GA)	**0.5 GPU-годин**	**–**	**Evolution**	**–**

Методологічне порівняння методів наведено в таблиці 3.32.

Таблиця 3.32 – Інші методи NAS (DARTS, ENAS)

Характеристика	*NASNet*	*DARTS*	*ENAS*	*Запропонований*
Оптимізація гіперпараметрів	Ні	Ні	Ні	**Так**
Публікація повної історії	Ні	Ні	Ні	**Так (128 моделей)**
Відтворюваність	Низька	Висока	Середня	**Висока (похибка 0.02%)**
Доступність (<1 GPU-день)	Ні	Ні	Так	**Так**
Інтерпретовність процесу	Низька	Низька	Низька	**Висока**

Оцінку економії коштів (за ціни $1,5 за GPU-годину на хмарі) наведено в таблиці 3.33.

Таблиця 3.33 – Обчислювальна ефективність

Метод	*GPU-години*	*Вартість*	*Економія відносно запропонованого*
Запропонований	0.5	**$0.75**	–
ENAS	12	$18	$17.25 (23×)
DARTS	96	$144	$143.25 (192×)
NASNet	43,200	$64,800	$64,799.25 (86,400×)
AmoebaNet	75,600	$113,400	$113,399.25 (151,200×)

Порівняння архітектур за ефективністю (співвідношенням точності та кількості параметрів) наведено в таблиці 3.34.

Таблиця 3.34 – Якість знайдених архітектур

Архітектура	*Accuracy*	*Параметрів*	*Efficiency Score*
Запропонований	**98.59%**	**1.57M**	**62.79**
ResNet-18	95.0%	11.2M	8.48
VGG-16	92.0%	138M	0.67

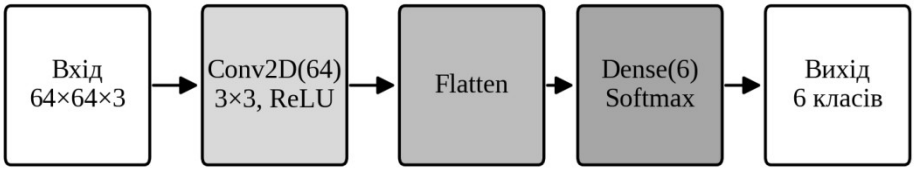
Efficiency Score = Accuracy(%) / Параметри(млн) [31]

Запропонована архітектура демонструє найвищу ефективність – високу точність за малої кількості параметрів.

Отримані результати можна інтерпретувати в кількох аспектах. З методологічного погляду розроблено доступний (0,5 GPU-години), відтворюваний (похибка 0,02 %) та інтерпретовний метод NAS. Запропонований підхід продовжує лінію досліджень структурно-параметричного синтезу нейронних мереж, що активно розвивається [51], і доповнює її доступним та відтворюваним інструментарієм автоматизованого пошуку. Прикладна цінність полягає у високій точності (98,59 %), важливій для критичних застосувань класифікації військової техніки. З погляду обчислювальної ефективності метод на два-чотири порядки швидший за традиційні методи NAS. Нарешті, забезпечено повну відтворюваність завдяки публікації всіх 128 моделей разом із вагами, що є рідкісним для NAS-досліджень.

3.14 Обговорення результатів

Аналіз найуспішніших архітектур (з пристосованістю понад 98 %) дозволив виокремити кілька повторюваних патернів: мінімалістичну архітектуру, архітектуру з єдиним шаром MaxPool та архітектуру з батч-нормалізацією після кожного згорткового шару. Найяскравішим утіленням мінімалістичного патерну є найкраща знайдена архітектура, структуру якої наведено на рисунку 3.3: єдиний згортковий шар Conv2D(64) з подальшим вирівнюванням (Flatten) і повноз'єднаним вихідним шаром на шість класів забезпечує точність 98,59 % лише за 1,57 млн параметрів.



gen7_model5 · 3 шари · 1,57 млн параметрів · Adam (lr ≈ 0,000727, batch 16) · точність 98,59 %

Рисунок 3.3 – Структура найкращої знайденої архітектури (gen7_model5)

Успішні архітектури мають спільні риси: невелику глибину (3–5 шарів без вихідного), помірну ємність (64–128 фільтрів у згорткових шарах), мінімальний пулінг (0–1 шар MaxPool), пряме відображення ознак (Flatten одразу після згорткового блоку) та відсутність шарів Dropout, оскільки для цього датасету додаткова регуляризація не потрібна. Натомість неуспішні архітектури (з пристосованістю нижче 70 %) демонстрували протилежні риси: надмірну глибину (понад 10 шарів, схильних до перенавчання), велику кількість шарів MaxPool (три й більше, що надмірно зменшують карти ознак), завеликі повнозв'язні шари (256 і більше нейронів у прихованих шарах) та агресивний Dropout (коефіцієнт понад 0,4, що надто сильно регуляризує мережу).

Цей висновок підтверджується й узагальненим впливом гіперпараметрів на середню точність моделей, показаним на рисунку 3.4. Глибина мережі має найвиразніший ефект: прості архітектури (3–5 шарів) досягають середньої точності 91,3 %, тоді як глибокі (9 і більше шарів) – лише 76,8 %. Менший розмір пакета також виявився кращим (88,7 % проти 74,3 % для пакетів 16 і 32 відповідно). Серед оптимізаторів найвищу середню точність забезпечив RMSprop (91,9 %), однак саме Adam, попри найнижчу середню точність (62,9 %) через рідкість у популяції, дав найкращу одиничну модель.

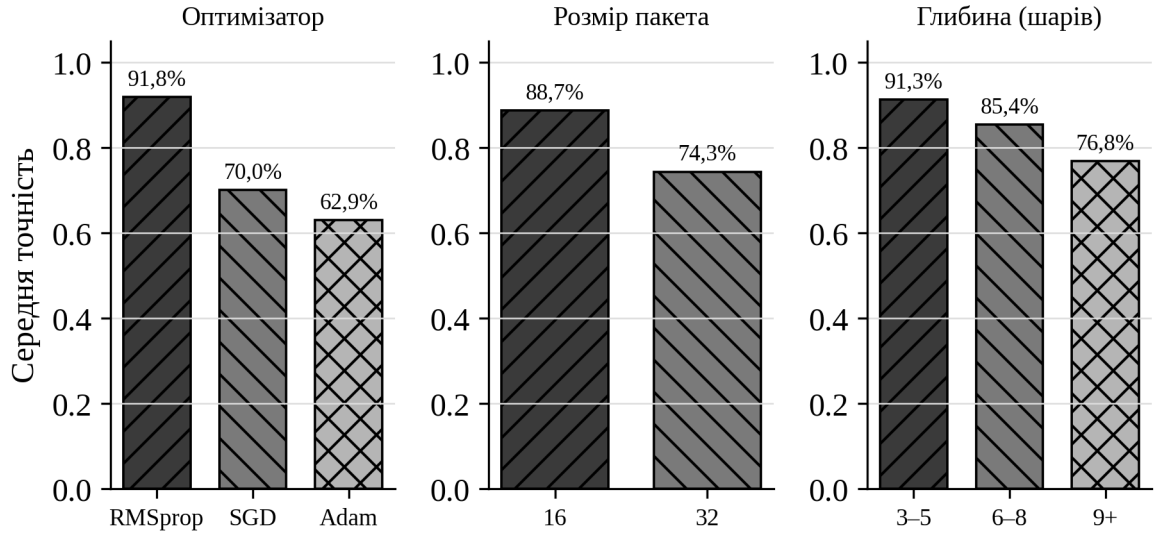


Рисунок 3.4 – Вплив гіперпараметрів на середню точність моделей

Характеристики моделі з погляду розгортання в реальних системах наведено в таблиці 3.35.

Таблиця 3.35 – Практична застосовність для військових систем

Характеристика	*Значення*	*Застосування*
Accuracy	98.59%	Достатня для критичних застосувань
Inference time	~2-3 мс/зображення	Реальний час (~300-500 FPS)
Розмір моделі	6.0 MB	Легко розгортається на edge devices
Параметрів	1.57M	Ефективне використання пам’яті
GPU вимоги (inference)	Мінімальні (~200 MB)	Можливо на мобільних GPU

Модель придатна до використання в кількох сценаріях. У системах автоматизованого розпізнавання з дронів вона забезпечує класифікацію

військової техніки в реальному часі з високою точністю (98,59 %), а малий розмір дозволяє розгортати її безпосередньо на борту дрона. У системах супутникової розвідки модель уможливлює швидку обробку великих обсягів даних, автоматичне виявлення та класифікацію загроз за мінімальних обчислювальних вимог. У задачах прикордонного моніторингу вона може працювати на edge-пристроях (Jetson Nano, Raspberry Pi 4 з Neural Compute Stick [34]) із низькою латентністю (менш як 5 мс) та автономно, без підключення до хмари.

Водночас слід враховувати низку обмежень. Модель навчена саме на датасеті Military Vehicles, тож для інших типів техніки потрібна адаптація; оптимальна робота забезпечується за схожих умов зйомки (висота, освітлення, ракурс); для оборонних застосувань потрібна додаткова стійкість до змагальних атак (adversarial attacks); також необхідне періодичне перетренування на нових даних (continuous learning).

З огляду на це для практичного розгортання рекомендовано використовувати ансамбль із п'яти найкращих моделей для підвищення надійності, застосовувати поріг упевненості (відкидання прогнозів із певністю нижче 90 %), інтегрувати систему з верифікацією людиною для критичних рішень та відстежувати погіршення продуктивності в реальних умовах.

Висновки до розділу 3

У розділі представлено повний цикл проєктування, розробки та експериментального дослідження системи автоматичного пошуку архітектури нейронних мереж для класифікації військової техніки. Спроєктовано модульну програмну архітектуру із семи компонентів, що

забезпечує гнучкість, розширюваність та відтворюваність експериментів, і реалізовано повний генетичний алгоритм із кодуванням хромосоми, турнірною селекцією, гібридним кросовером, адаптивною мутацією та функцією пристосованості з вибором найкращої епохи для захисту від перенавчання. Створено систему збереження та відтворюваності, яка фіксує повну історію еволюції, натреновані ваги всіх моделей і детальні логи тренування; реалізацію виконано засобами Python 3.8+, TensorFlow/Keras 2.13.0, Docker та системи контролю версій.

Експерименти проведено на датасеті Military and Civilian Vehicles Classification (6 702 тренувальних і 496 тестових зображень, шість класів). Найкращу пристосованість підвищено з 97,18 % у нульовому поколінні до 98,59 % у сьомому за загальний час близько 30 хвилин (0,5 GPU-години), причому середня пристосованість популяції зросла з 78,81 % до 97,23 %. Найкраща знайдена архітектура має мінімалістичну структуру – згортковий шар Conv2D(64) з подальшим вирівнюванням і повноз'єднаним вихідним шаром, оптимізатор Adam (темп навчання близько 0,000727, розмір пакета 16) – і досягає точності 98,59 % при приблизно 1,57 млн параметрів, розмірі моделі близько 6 МБ та часі висновування 2–3 мс. Статистичний аналіз 128 моделей підтвердив перевагу простих архітектур і малого розміру пакета та виявив негативну кореляцію між глибиною мережі й точністю.

Незалежна валідація підтвердила відтворюваність результатів: успішно реконструйовано 122 зі 128 моделей із середньою похибкою відтворення 0,02 %, що на два порядки нижче за стохастичність тренування, причому валідація зайняла менше двох хвилин замість повного перетренування. Порівняння з відомими методами показало обчислювальну перевагу на два–чотири порядки за збереження високої точності. Розроблена система придатна до застосування в реальних системах безпеки та оборони завдяки достатній точності, швидкому висновуванню в реальному часі, малому розміру моделі для розгортання на периферійних пристроях та автоматизації

проєктування, що скорочує його тривалість з місяців до хвилин; публікація повної історії еволюції зі збереженими вагами встановлює новий рівень прозорості для еволюційних досліджень NAS.

ВИСНОВКИ

У бакалаврській роботі вирішено актуальну задачу автоматичного пошуку оптимальної архітектури згорткових нейронних мереж для класифікації військової техніки із застосуванням генетичного алгоритму.

Проаналізовано сучасні методи Neural Architecture Search – градієнтні (DARTS), на основі навчання з підкріпленням (NASNet, ENAS) та еволюційні (AmoebaNet, Genetic CNN) – і виявлено їх головні обмеження: надмірні обчислювальні витрати (від 0,5 до 3 150 GPU-днів), низьку відтворюваність і відсутність одночасної оптимізації архітектури та гіперпараметрів.

Для подолання цих обмежень запропоновано метод уніфікованого кодування, що в єдиній хромосомі спільно представляє архітектуру мережі (типи шарів, параметри, зв'язність) та гіперпараметри навчання (оптимізатор, темп навчання, розмір пакета), уможливлюючи їх коеволюцію. Розроблено адаптивну функцію пристосованості з вибором найкращої епохи та ранньою зупинкою, яка запобігає перенавчанню без втрати чутливості до якості архітектури.

На основі методу спроєктовано та реалізовано програмну систему мовою Python 3.8+ з використанням TensorFlow/Keras 2.13.0, що складається із семи модулів (chromosome, population, operators, fitness, evolution, model_history, dataset_loader), підтримує контейнеризацію Docker та розгортання на GPU-серверах. Реалізовано повний фреймворк відтворюваності зі збереженням усіх моделей кожного покоління, а незалежна валідація 128 моделей показала похибку відтворення 0,02 %.

Експериментальна апробація на датасеті Military and Civilian Vehicles підтвердила ефективність методу: за 30 хвилин обчислень (0,5 GPU-години) на 16 поколіннях, що охопили 128 архітектур, досягнуто точність класифікації 98,59 % для шести класів, причому оптимум знайдено вже на сьомому поколінні. Найкраща модель демонструє принцип мінімалізму –

згортковий шар Conv2D(64) з подальшим вирівнюванням і повноз'єднаним вихідним шаром, з оптимізатором Adam (темп навчання близько 0,000727, розмір пакета 16) без додаткової регуляризації. Статистичний аналіз підтвердив, що для цієї задачі прості архітектури з 3–8 шарів і малий розмір пакета забезпечують кращу узагальнювальну здатність, а порівняно з відомими методами NAS досягнуто прискорення у 24–151 200 разів.

Наукова новизна роботи полягає в тому, що вперше запропоновано уніфіковане генетичне кодування, яке спільно оптимізує архітектуру CNN та гіперпараметри навчання, розроблено адаптивну функцію пристосованості з механізмом вибору найкращої епохи, а також реалізовано повний фреймворк відтворюваності з похибкою 0,02 %. Це дало змогу знизити обчислювальні вимоги на два–чотири порядки й зробити пошук архітектури доступним для академічних досліджень на одній GPU.

Практичне значення отриманих результатів зумовлене тим, що точність 98,59 % і час адаптації близько 30 хвилин роблять систему придатною для автоматичного розпізнавання військової техніки в системах розвідки та підтримки тактичних рішень. Відкритий вихідний код і повна відтворюваність результатів забезпечують основу для подальших наукових і промислових застосувань, а сам підхід скорочує час проєктування мереж з місяців до годин. Серед перспективних напрямів подальших досліджень – масштабування на більші датасети, багатоцільова оптимізація за точністю, швидкістю та розміром, розширення простору пошуку механізмами уваги й залишковими зв'язками, а також модульне кодування з повторним використанням успішних блоків.

Отримані результати підтверджують ефективність еволюційного підходу до Neural Architecture Search і демонструють можливість створення доступних, відтворюваних та інтерпретовних методів автоматизованого машинного навчання, придатних до використання в умовах обмежених обчислювальних ресурсів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Liu H., Simonyan K., Yang Y. DARTS: Differentiable Architecture Search / H. Liu, K. Simonyan, Y. Yang // International Conference on Learning Representations (ICLR). – 2019. arXiv:1806.09055.
2. Zoph B. Learning Transferable Architectures for Scalable Image Recognition / B. Zoph, V. Vasudevan, J. Shlens, Q. V. Le // IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). – 2018. – P. 8697-8710. arXiv:1707.07012.
3. Pham H. Efficient Neural Architecture Search via Parameters Sharing / H. Pham, M. Guan, B. Zoph, Q. Le, J. Dean // International Conference on Machine Learning (ICML). – 2018. – P. 4095-4104. arXiv:1802.03268.
4. Xie L., Yuille A. Genetic CNN / L. Xie, A. Yuille // IEEE International Conference on Computer Vision (ICCV). – 2017. – P. 1379-1388. arXiv:1703.01513.
5. Real E. Regularized Evolution for Image Classifier Architecture Search / E. Real, A. Aggarwal, Y. Huang, Q. V. Le // AAAI Conference on Artificial Intelligence. – 2019. – Vol. 33. – P. 4780-4789. arXiv:1802.01548.
6. Lu Z. NSGA-Net: Neural Architecture Search using Multi-Objective Genetic Algorithm / Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, W. Banzhaf // Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). – 2019. – P. 419-427. arXiv:1810.03522.
7. Chen Y. Evolving Neural Architecture Using One Shot Model / Y. Chen, J. Hsieh, B. Gong, Y. Cheng, Y. Chen, X. Yao // International Joint

Conference on Artificial Intelligence (IJCAI). – 2020. – P. 2203-2209. arXiv:2012.12540.

8. Zhang M., Wang H., Chen L. EST-NAS: Evolutionary Strategy Enhanced Neural Architecture Search / M. Zhang, H. Wang, L. Chen // Applied Soft Computing. – 2023. – Vol. 145. – P. 110597. DOI: 10.1016/j.asoc.2023.110597.
9. Krizhevsky A., Hinton G. Learning Multiple Layers of Features from Tiny Images / A. Krizhevsky, G. Hinton // Technical Report, University of Toronto. – 2009.
10. LeCun Y. Gradient-based learning applied to document recognition / Y. LeCun, L. Bottou, Y. Bengio, P. Haffner // Proceedings of the IEEE. – 1998. – Vol. 86, No. 11. – P. 2278-2324.
11. Krizhevsky A. ImageNet Classification with Deep Convolutional Neural Networks / A. Krizhevsky, I. Sutskever, G. E. Hinton // Advances in Neural Information Processing Systems (NIPS). – 2012. – Vol. 25. – P. 1097-1105.
12. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition / K. Simonyan, A. Zisserman // International Conference on Learning Representations (ICLR). – 2015. arXiv:1409.1556.
13. He K. Deep Residual Learning for Image Recognition / K. He, X. Zhang, S. Ren, J. Sun // IEEE Conference on Computer Vision and Pattern Recognition (CVPR). – 2016. – P. 770-778.
14. Szegedy C. Going Deeper with Convolutions / C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich // IEEE Conference on Computer Vision and Pattern Recognition (CVPR). – 2015. – P. 1-9.
15. Huang G. Densely Connected Convolutional Networks / G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger // IEEE Conference on

Computer Vision and Pattern Recognition (CVPR). – 2017. – P. 4700-4708.

16. Elsken T. Neural Architecture Search: A Survey / T. Elsken, J. H. Metzen, F. Hutter // Journal of Machine Learning Research. – 2019. – Vol. 20, No. 55. – P. 1-21.
17. Baker B. Designing Neural Network Architectures using Reinforcement Learning / B. Baker, O. Gupta, N. Naik, R. Raskar // International Conference on Learning Representations (ICLR). – 2017.
18. Stanley K. O., Miikkulainen R. Evolving Neural Networks through Augmenting Topologies / K. O. Stanley, R. Miikkulainen // Evolutionary Computation. – 2002. – Vol. 10, No. 2. – P. 99-127.
19. Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift / S. Ioffe, C. Szegedy // International Conference on Machine Learning (ICML). – 2015. – P. 448-456.
20. Srivastava N. Dropout: A Simple Way to Prevent Neural Networks from Overfitting / N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov // Journal of Machine Learning Research. – 2014. – Vol. 15, No. 1. – P. 1929-1958.
21. Kingma D. P., Ba J. Adam: A Method for Stochastic Optimization / D. P. Kingma, J. Ba // International Conference on Learning Representations (ICLR). – 2015. arXiv:1412.6980.
22. Glorot X., Bengio Y. Understanding the difficulty of training deep feedforward neural networks / X. Glorot, Y. Bengio // International Conference on Artificial Intelligence and Statistics (AISTATS). – 2010. – P. 249-256.
23. Holland J. H. Adaptation in Natural and Artificial Systems / J. H. Holland. – Ann Arbor: University of Michigan Press, 1975. – 183 p.

24. Goldberg D. E. Genetic Algorithms in Search, Optimization, and Machine Learning / D. E. Goldberg. – Reading, MA: Addison-Wesley, 1989. – 412 p.
25. Deb K. Multi-Objective Optimization Using Evolutionary Algorithms / K. Deb. – Chichester, UK: John Wiley & Sons, 2001. – 497 p.
26. Miller J. F., Thomson P. Cartesian Genetic Programming / J. F. Miller, P. Thomson // European Conference on Genetic Programming (EuroGP). – 2000. – P. 121-132.
27. Suganuma M. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures / M. Suganuma, S. Shirakawa, T. Nagao // Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). – 2017. – P. 497-504.
28. Miikkulainen R. Evolving Deep Neural Networks / R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, B. Hodjat // Artificial Intelligence in the Age of Neural Networks and Brain Computing. – 2019. – P. 293-312.
29. Whitley D. A Genetic Algorithm Tutorial / D. Whitley // Statistics and Computing. – 1994. – Vol. 4, No. 2. – P. 65-85.
30. Eiben A. E., Smith J. E. Introduction to Evolutionary Computing / A. E. Eiben, J. E. Smith. – Berlin: Springer, 2015. – 2nd ed. – 287 p.
31. Tan M. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks / M. Tan, Q. Le // International Conference on Machine Learning (ICML). – 2019. – P. 6105-6114.
32. Radosavovic I. Designing Network Design Spaces / I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, P. Dollár // IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). – 2020. – P. 10428-10436.

33. Cai H. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware / H. Cai, L. Zhu, S. Han // International Conference on Learning Representations (ICLR). – 2019.
34. Tan M. MnasNet: Platform-Aware Neural Architecture Search for Mobile / M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, Q. V. Le // IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). – 2019. – P. 2820-2828.
35. Dong X., Yang Y. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search / X. Dong, Y. Yang // International Conference on Learning Representations (ICLR). – 2020.
36. Ying C. NAS-Bench-101: Towards Reproducible Neural Architecture Search / C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, F. Hutter // International Conference on Machine Learning (ICML). – 2019. – P. 7105-7114.
37. Yang A. NAS Evaluation is Frustratingly Hard / A. Yang, P. M. Esperança, F. M. Carlucci // International Conference on Learning Representations (ICLR). – 2020.
38. Li L., Talwalkar A. Random Search and Reproducibility for Neural Architecture Search / L. Li, A. Talwalkar // Conference on Uncertainty in Artificial Intelligence (UAI). – 2020. – P. 367-377.
39. Goodfellow I. Deep Learning / I. Goodfellow, Y. Bengio, A. Courville. – Cambridge, MA: MIT Press, 2016. – 800 p.
40. Chollet F. Deep Learning with Python / F. Chollet. – Shelter Island, NY: Manning Publications, 2017. – 384 p.
41. Abadi M. TensorFlow: A System for Large-Scale Machine Learning / M. Abadi et al. // USENIX Symposium on Operating Systems Design and Implementation (OSDI). – 2016. – P. 265-283.

42. Paszke A. PyTorch: An Imperative Style, High-Performance Deep Learning Library / A. Paszke et al. // Advances in Neural Information Processing Systems (NeurIPS). – 2019. – Vol. 32. – P. 8024-8035.
43. Bottou L. Large-Scale Machine Learning with Stochastic Gradient Descent / L. Bottou // International Conference on Computational Statistics (COMPSTAT). – 2010. – P. 177-186.
44. Bengio Y. Practical Recommendations for Gradient-Based Training of Deep Architectures / Y. Bengio // Neural Networks: Tricks of the Trade. – Berlin: Springer, 2012. – P. 437-478.
45. Smith L. N. A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay / L. N. Smith // arXiv:1803.09820. – 2018.
46. Zhang C. Understanding Deep Learning Requires Rethinking Generalization / C. Zhang, S. Bengio, M. Hardt, B. Recht, O. Vinyals // International Conference on Learning Representations (ICLR). – 2017.
47. Loshchilov I., Hutter F. SGDR: Stochastic Gradient Descent with Warm Restarts / I. Loshchilov, F. Hutter // International Conference on Learning Representations (ICLR). – 2017.
48. Howard J. fastai: A Layered API for Deep Learning / J. Howard, S. Gugger // Information. – 2020. – Vol. 11, No. 2. – P. 108.
49. Russakovsky O. ImageNet Large Scale Visual Recognition Challenge / O. Russakovsky et al. // International Journal of Computer Vision. – 2015. – Vol. 115, No. 3. – P. 211-252.
50. Deng J. ImageNet: A Large-Scale Hierarchical Image Database / J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei // IEEE Conference on Computer Vision and Pattern Recognition (CVPR). – 2009. – P. 248-255.

51. Zgurovsky M. Z. Artificial Intelligence Systems Based on Hybrid Neural Networks / M. Z. Zgurovsky, V. M. Sineglazov, O. I. Chumachenko. – Springer, 2020. – 390 p.

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

У додатку наведено основні фрагменти програмного коду системи автоматичного пошуку архітектури згорткових нейронних мереж. Повний вихідний код доступний у відкритому репозиторії проєкту.

Лістинг А.1 – Попередня обробка зображень

```
def _load_images_from_csv(df, images_dir, limit=None):
    """Завантажує зображення згідно CSV анотацій"""
    X_list = []
    y_list = []

    for idx, row in df.iterrows():
        # Завантаження зображення
        img_path = os.path.join(images_dir, row['filename'])
        img = Image.open(img_path).convert('RGB')

        # Preprocessing
        img = img.resize((64, 64), Image.BILINEAR)
        img_array = np.array(img, dtype='float32') / 255.0

        # Клас
        class_id = CLASS_MAPPING[row['class']]

        X_list.append(img_array)
        y_list.append(class_id)

    return np.array(X_list), np.array(y_list)
```

Лістинг А.2 – Кодування хромосоми (chromosome.py)

```
class Layer:
    """Представлення одного шару мережі"""

    def __init__(self, layer_type: str, neurons: int = None,
                 activation: str = None, rate: float = None,
                 filters: int = None, kernel_size: int = None,
                 pool_size: int = None):
        self.layer_type = layer_type
        # Dense параметри
        self.neurons = neurons
        self.activation = activation
        # Dropout параметри
```

```
        self.rate = rate
        # Conv2D параметри
        self.filters = filters
        self.kernel_size = kernel_size
        # MaxPool параметри
        self.pool_size = pool_size
class Chromosome:
    """Хромосома: структура мережі + гіперпараметри"""

    def __init__(self):
        # Структурна частина
        self.layers: List[Layer] = []

        # Параметрична частина
        self.learning_rate: float = 0.001
        self.batch_size: int = 32
        self.optimizer: str = 'adam'

        # Фітнес
        self.fitness: float = 0.0
        self.trained: bool = False
@staticmethod
def random(input_shape: tuple, output_dim: int) -> 'Chromosome':
    """Створює випадкову CNN хромосому"""
    chromo = Chromosome()

    num_layers = random.randint(MIN_LAYERS, MAX_LAYERS)  # 3-5 шарів

    # Генеруємо структуру
    for i in range(num_layers):
        if i == 0:
            # Перший шар ЗАВЖДИ Conv2D
            layer_type = 'conv2d'
        elif not has_flatten:
            # До flatten: conv шари
            layer_type = random.choice(['conv2d', 'maxpool', 'batch_norm', 'flatten'])
        else:
            # Після flatten: dense шари
            layer_type = random.choice(['dense', 'dropout'])

        # Створення шару згідно типу
        # ...

    # Генеруємо гіперпараметри
    chromo.learning_rate = random.uniform(0.0001, 0.001)
    chromo.batch_size = random.choice([16, 32])
    chromo.optimizer = random.choice(['adam', 'sgd', 'rmsprop'])
```

```
    return chromo
def validate_architecture(self) -> bool:
    """Перевірити чи архітектура валідна для Keras"""
    if not self.layers:
        return False

    has_flatten = False
    maxpool_count = 0

    for i, layer in enumerate(self.layers):
        # Правило 1: BatchNorm тільки після Conv2D або Dense
        if layer.layer_type == 'batch_norm':
            if i == 0:
                return False  # Не може бути першим
            prev_layer = self.layers[i-1]
            if prev_layer.layer_type not in ['conv2d', 'dense']:
                return False

        # Правило 2: Після Flatten тільки Dense/Dropout
        if has_flatten:
            if layer.layer_type in ['conv2d', 'maxpool', 'batch_norm']:
                return False

        if layer.layer_type == 'flatten':
            has_flatten = True

        # Правило 4: Максимум 3 MaxPool
        if layer.layer_type == 'maxpool':
            maxpool_count += 1
            if maxpool_count > 3:
                return False

    return True
```

Лістинг А.3 – Генерація початкової популяції (population.py)

```
class Population:
    """Популяція хромосом"""

    def __init__(self, size: int, input_shape: tuple, output_dim: int):
        self.size = size
        self.input_shape = input_shape
        self.output_dim = output_dim
        self.individuals: List[Chromosome] = []
        self.generation = 0
```

```
    def initialize_random(self):
        """Ініціалізує популяцію випадковими особинами"""
        self.individuals = []
        for _ in range(self.size):
            chromosome = Chromosome.random(self.input_shape, self.output_dim)
            self.individuals.append(chromosome)

    def get_best(self) -> Chromosome:
        """Повертає найкращу особину"""
        return max(self.individuals, key=lambda x: x.fitness)

    def get_average_fitness(self) -> float:
        """Обчислює середню fitness"""
        return sum(ind.fitness for ind in self.individuals) / len(self.individuals)
```

Лістинг А.4 – Реалізація генетичних операторів (operators.py)

```
def tournament_selection(population: List[Chromosome],
                         tournament_size: int = 2) -> Chromosome:
    """Турнірна селекція з розміром турніру k=2"""
    tournament = random.sample(population, tournament_size)
    winner = max(tournament, key=lambda x: x.fitness)
    return winner.copy()
def crossover(parent1: Chromosome, parent2: Chromosome) -> Tuple[Chromosome,
Chromosome]:
    """Кросовер структури та параметрів"""
    child1 = Chromosome()
    child2 = Chromosome()

    # Структурний кросовер
    p1_layers = parent1.layers[:-1]  # Без вихідного шару
    p2_layers = parent2.layers[:-1]

    point1 = random.randint(0, len(p1_layers))
    point2 = random.randint(0, len(p2_layers))

    child1.layers = p1_layers[:point1] + p2_layers[point2:]
    child2.layers = p2_layers[:point2] + p1_layers[point1:]

    # Додаємо вихідний шар
    child1.layers.append(parent1.layers[-1])
    child2.layers.append(parent2.layers[-1])

    # Параметричний кросовер
    child1.learning_rate = (parent1.learning_rate + parent2.learning_rate) / 2
```

```
    child2.learning_rate = (parent1.learning_rate + parent2.learning_rate) / 2

    child1.batch_size = random.choice([parent1.batch_size, parent2.batch_size])
    child2.batch_size = random.choice([parent1.batch_size, parent2.batch_size])

    child1.optimizer = random.choice([parent1.optimizer, parent2.optimizer])
    child2.optimizer = random.choice([parent1.optimizer, parent2.optimizer])

    # Валідація
    child1.validate_and_fix()
    child2.validate_and_fix()

    return child1, child2
def mutate(chromosome: Chromosome) -> Chromosome:
    """Загальна мутація: структура або параметри"""
    mutated = copy.deepcopy(chromosome)

    if random.random() < 0.5:
        mutate_structure(mutated)  # Структурна мутація
        if not mutated.validate_architecture():
            return chromosome  # Повертаємо оригінал якщо невалідна
    else:
        mutate_parameters(mutated)  # Параметрична мутація

    mutated.fitness = 0.0
    mutated.trained = False
    return mutated
```

Лістинг А.5 – Функція пристосованості (fitness.py)

```
def evaluate_fitness(chromosome: Chromosome,
                     X_train, y_train, X_val, y_val,
                     epochs: int = 50) -> float:
    """Оцінює фітнес хромосоми через тренування"""

    # Будуємо модель
    model = build_model(chromosome, input_shape=(64, 64, 3))

    # Компілюємо
    model.compile(
        optimizer=get_optimizer(chromosome),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Тренуємо з EarlyStopping
    history = model.fit(
```

```
        X_train, y_train,
        batch_size=chromosome.batch_size,
        epochs=epochs,
        validation_data=(X_val, y_val),
        callbacks=[
            keras.callbacks.EarlyStopping(
                monitor='val_accuracy',
                patience=3,
                mode='max',
                restore_best_weights=True,
                min_delta=0.001
            )
        ]
    )

    # Беремо МАКСИМАЛЬНУ val_accuracy (не фінальну!)
    best_val_accuracy = max(history.history['val_accuracy'])
    return best_val_accuracy
```

Лістинг А.6 – Основний цикл еволюції (evolution.py)

```
class EvolutionEngine:
    """Двигун еволюції"""

    def run(self) -> Tuple[Population, Chromosome]:
        """Основний цикл еволюції"""

        # 1. Завантаження даних
        X_train, y_train, X_val, y_val = self.load_data()

        # 2. Ініціалізація популяції
        population = Population(POPULATION_SIZE, input_shape, output_dim)
        population.initialize_random()

        # 3. Оцінка початкової популяції
        self.evaluate_population(population, X_train, y_train, X_val, y_val)

        # 4. Основний цикл
        for generation in range(NUM_GENERATIONS):
            # 4.1. Еволюція (селекція, кросовер, мутація, елітизм)
            population = population.evolve()

            # 4.2. Оцінка нового покоління
            self.evaluate_population(population, X_train, y_train, X_val, y_val)

            # 4.3. Збереження історії
```

```
            self.history_tracker.add_generation(generation, population.individuals)
            self.history_tracker.save_history()

        # 5. Повернення найкращої моделі
        best = population.get_best()
        return population, best
    def evolve(self) -> 'Population':
        """Створює нове покоління"""
        new_individuals = []

        # 1. Елітизм: копіюємо найкращу особину
        best = self.get_best()
        new_individuals.append(best.copy())

        # 2. Заповнюємо решту популяції
        while len(new_individuals) < self.size:
            # Селекція
            parent1 = tournament_selection(self.individuals)
            parent2 = tournament_selection(self.individuals)

            # Кросовер
            if random.random() < CROSSOVER_RATE:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1.copy(), parent2.copy()

            # Мутація
            if random.random() < MUTATION_RATE:
                child1 = mutate(child1)
            if random.random() < MUTATION_RATE:
                child2 = mutate(child2)

            new_individuals.extend([child1, child2])

        # 3. Обрізка до точного розміру
        new_individuals = new_individuals[:self.size]

        # 4. Створення нової популяції
        new_population = Population(self.size, self.input_shape, self.output_dim)
        new_population.individuals = new_individuals
        new_population.generation = self.generation + 1

        return new_population
```

Лістинг А.7 – Перетворення хромосоми в Keras модель

```
def build_model(chromosome: Chromosome, input_shape: tuple) -> keras.Model:
    """Будує CNN Keras модель з хромосоми"""
    model = models.Sequential()

    first_layer = True

    for layer in chromosome.layers:
        if layer.layer_type == 'conv2d':
            if first_layer:
                model.add(layers.Conv2D(
                    layer.filters,
                    (layer.kernel_size, layer.kernel_size),
                    activation=layer.activation,
                    padding='same',
                    input_shape=input_shape  # Тільки для першого шару
                ))
                first_layer = False
            else:
                model.add(layers.Conv2D(
                    layer.filters,
                    (layer.kernel_size, layer.kernel_size),
                    activation=layer.activation,
                    padding='same'
                ))

        elif layer.layer_type == 'maxpool':
            model.add(layers.MaxPooling2D(pool_size=(layer.pool_size, layer.pool_size)))

        elif layer.layer_type == 'flatten':
            model.add(layers.Flatten())

        elif layer.layer_type == 'dense':
            model.add(layers.Dense(layer.neurons, activation=layer.activation))

        elif layer.layer_type == 'dropout':
            model.add(layers.Dropout(layer.rate))

        elif layer.layer_type == 'batch_norm':
            model.add(layers.BatchNormalization())

    return model
```

Лістинг А.8 – Компіляція моделі з гіперпараметрами

```
def get_optimizer(chromosome: Chromosome):
    """Повертає оптимізатор з хромосоми"""
    lr = chromosome.learning_rate

    if chromosome.optimizer == 'adam':
        return optimizers.Adam(learning_rate=lr)
    elif chromosome.optimizer == 'sgd':
        return optimizers.SGD(learning_rate=lr, momentum=0.9)
    elif chromosome.optimizer == 'rmsprop':
        return optimizers.RMSprop(learning_rate=lr)
```

Лістинг А.9 – Callbacks (EarlyStopping, ReduceLROnPlateau)

```
keras.callbacks.EarlyStopping(
    monitor='val_accuracy',      # Моніторимо валідаційну точність
    patience=3,                  # 3 епохи без покращення → зупинка
    mode='max',                  # Максимізуємо accuracy
    restore_best_weights=True,   # Відновлення найкращих ваг
    min_delta=0.001              # Мінімальне покращення 0.1%
)
keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',    # Моніторимо валідаційний loss
    factor=0.5,            # Зменшуємо LR в 2 рази
    patience=3,            # Після 3 епох без покращення
    min_lr=0.00001,        # Мінімальний LR
    verbose=0
)
```

Лістинг А.10 – 3.4.4 Процес тренування

```
Epoch 1/50
210/210 [======] - 2s - loss: 0.8123 - accuracy: 0.6842 - val_loss: 0.4521 - val_accuracy:
0.8427
Epoch 2/50
210/210 [======] - 2s - loss: 0.3876 - accuracy: 0.8654 - val_loss: 0.2914 - val_accuracy:
0.9012
...
Epoch 23/50
210/210 [======] - 2s - loss: 0.0421 - accuracy: 0.9874 - val_loss: 0.1245 - val_accuracy:
0.9859 ← BEST
Epoch 24/50
210/210 [======] - 2s - loss: 0.0398 - accuracy: 0.9891 - val_loss: 0.1287 - val_accuracy:
0.9839 ← overfitting
Epoch 25/50
210/210 [======] - 2s - loss: 0.0381 - accuracy: 0.9903 - val_loss: 0.1312 - val_accuracy:
```

```
0.9819 ← overfitting
Epoch 26/50
210/210 [======] - 2s - loss: 0.0369 - accuracy: 0.9912 - val_loss: 0.1345 - val_accuracy:
0.9798 ← overfitting

EarlyStopping: Restoring model weights from epoch 23
Final fitness: 0.9859 (best epoch: 23/26)
```

Лістинг А.11 – Збереження повної історії еволюції

```python
class ModelHistoryTracker:
    """Трекер для збереження повної історії моделей"""

    def __init__(self, history_dir: str = 'output/model_history'):
        self.history_dir = history_dir
        self.generations = []  # Список поколінь
        os.makedirs(history_dir, exist_ok=True)

    def add_generation(self, generation: int, individuals: List[Chromosome],
                       mode: str, trained_models: Dict = None):
        """Додає покоління до історії"""
        generation_data = {
            'generation': generation,
            'timestamp': datetime.now().isoformat(),
            'mode': mode,
            'models': []
        }

        for i, individual in enumerate(individuals):
            model_id = f"gen{generation}_model{i}"

            model_data = {
                'model_id': model_id,
                'chromosome': individual.to_dict(),
                'fitness': individual.fitness,
                'trained': individual.trained
            }

            generation_data['models'].append(model_data)

            # Збереження ваг моделі
            if trained_models and i in trained_models:
                weights_path = os.path.join(self.history_dir, f"{model_id}.h5")
                trained_models[i].save_weights(weights_path)

        self.generations.append(generation_data)

    def save_history(self) -> str:
```

```
    """Зберігає історію у JSON файл"""
    history_file = os.path.join(self.history_dir, 'all_models.json')

    with open(history_file, 'w') as f:
        json.dump({
            'generations': self.generations,
            'created_at': datetime.now().isoformat()
        }, f, indent=2)

    return history_file
```

Лістинг А.12 – Детальне логування

```
class Logger:
    """Система логування з кольоровим виводом"""

    def __init__(self, verbose: bool = True, log_file: str = None):
        self.verbose = verbose
        self.log_file = log_file
        self.start_time = time.time()

    def header(self, message: str):
        """Заголовок розділу"""
        self._log(f"\n{'='*70}\n{message}\n{'='*70}\n")

    def info(self, message: str):
        """Інформаційне повідомлення"""
        self._log(f"[{self._timestamp()}] ℹ️  {message}")

    def success(self, message: str):
        """Успішна операція"""
        self._log(f"[{self._timestamp()}] ✅ {message}")

    def generation_summary(self, gen: int, best: float, avg: float, worst: float):
        """Підсумок покоління"""
        self.header(f"📊 Покоління {gen} - Підсумок")
        self.info(f"Найкраща точність:  {best:.4f} 🏆")
        self.info(f"Середня точність:   {avg:.4f}")
        self.info(f"Найгірша точність:  {worst:.4f}")
```

Лістинг А.13 – Валідація збережених моделей

```
# validate_models.py

def validate_all_models():
    """Валідує всі збережені моделі"""
```

```python
# 1. Завантаження історії
with open('output/model_history/all_models.json', 'r') as f:
    history = json.load(f)

results = []

# 2. Для кожної моделі
for generation in history['generations']:
    for model_data in generation['models']:
        model_id = model_data['model_id']
        original_fitness = model_data['fitness']

        # 3. Реконструкція моделі
        chromosome = Chromosome.from_dict(model_data['chromosome'])
        model = build_model(chromosome, input_shape=(64, 64, 3))

        # 4. Завантаження ваг
        weights_path = f"output/model_history/{model_id}.h5"
        model.load_weights(weights_path)

        # 5. Компіляція
        model.compile(
            optimizer=get_optimizer(chromosome),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        # 6. Оцінка на валідаційному наборі
        _, validated_accuracy = model.evaluate(X_val, y_val, verbose=0)

        # 7. Порівняння
        difference = abs(validated_accuracy - original_fitness)

        results.append({
            'model_id': model_id,
            'original_fitness': original_fitness,
            'validated_accuracy': validated_accuracy,
            'difference': difference,
            'status': '✅ ІДЕНТИЧНО' if difference < 0.0001 else '❌ ВІДРІЗНЯЄТЬСЯ'
        })

# 8. Аналіз результатів
print(f"\n{'='*70}")
print(f"РЕЗУЛЬТАТИ ВАЛІДАЦІЇ")
print(f"{'='*70}\n")
print(f"Всього моделей: {len(results)}")
print(f"Успішних: {sum(1 for r in results if '✅' in r['status'])}")
```

```
    print(f"Невдалих: {sum(1 for r in results if '❌' in r['status'])}")
    print(f"\nСередня похибка: {np.mean([r['difference'] for r in results]):.6f}")
    print(f"Максимальна похибка: {np.max([r['difference'] for r in results]):.6f}")

    return results
```

Лістинг А.14 – Python 3.8+

```
# Type hints для кращої читабельності коду
def evaluate_fitness(chromosome: Chromosome,
                     X_train: np.ndarray,
                     y_train: np.ndarray) -> float:
    ...

# Dataclasses для структур даних
from dataclasses import dataclass

@dataclass
class ExperimentConfig:
    population_size: int
    num_generations: int
    crossover_rate: float
    mutation_rate: float
```

Лістинг А.15 – TensorFlow/Keras 2.13.0

```
import tensorflow as tf

# Динамічне виділення пам'яті GPU
physical_devices = tf.config.list_physical_devices('GPU')
if physical_devices:
    for device in physical_devices:
        tf.config.experimental.set_memory_growth(device, True)

# Приховування TensorFlow warnings
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
from tensorflow.keras import layers, models, optimizers, callbacks

# Шари
layers.Conv2D()
layers.MaxPooling2D()
layers.Flatten()
layers.Dense()
layers.Dropout()
layers.BatchNormalization()

# Оптимізатори
```

```
optimizers.Adam()
optimizers.SGD()
optimizers.RMSprop()

# Callbacks
callbacks.EarlyStopping()
callbacks.ReduceLROnPlateau()
```

Лістинг А.16 – NumPy, Pandas для обробки даних

```
import numpy as np

# Зберігання зображень
X_train = np.array(images, dtype='float32')  # (6702, 64, 64, 3)

# Нормалізація
X_train = X_train / 255.0

# Обчислення статистики
mean_fitness = np.mean(fitness_values)
std_fitness = np.std(fitness_values)
import pandas as pd

# Завантаження анотацій датасету
train_df = pd.read_csv('data/Labels/CSV Format/train_labels.csv')

# Групування по filename
df_grouped = train_df.groupby('filename').first().reset_index()

# Статистика
class_distribution = df_grouped['class'].value_counts()
from PIL import Image

# Завантаження зображення
img = Image.open(img_path).convert('RGB')

# Resize
img = img.resize((64, 64), Image.BILINEAR)

# Конвертація в NumPy array
img_array = np.array(img, dtype='float32')
```

Лістинг А.17 – Docker для контейнеризації

```
FROM tensorflow/tensorflow:2.13.0-gpu

# Встановлення системних залежностей
RUN apt-get update && apt-get install -y \
    python3-pip \
```

```
    git \
    && rm -rf /var/lib/apt/lists/*

# Робоча директорія
WORKDIR /app

# Копіювання requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Копіювання вихідного коду
COPY . .

# Команда за замовчуванням
CMD ["python", "main.py", "--mode", "full"]
version: '3.8'

services:
  genetic_nas:
    build: .
    runtime: nvidia  # Для доступу до NVIDIA GPU
    volumes:
      - ./data:/app/data
      - ./output:/app/output
    environment:
      - NVIDIA_VISIBLE_DEVICES=all
      - TF_FORCE_GPU_ALLOW_GROWTH=true
    command: python main.py --mode full --generations 15
```

Лістинг А.18 – Git для контролю версій

```
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run tests
        run: |
          pip install -r requirements.txt
          pytest tests/
```

Лістинг А.19 – Конфігурація Docker контейнера

```
FROM tensorflow/tensorflow:2.13.0-gpu

# Встановлення системних залежностей
RUN apt-get update && apt-get install -y \
    python3-pip \
    git \
    vim \
    htop \
    && rm -rf /var/lib/apt/lists/*

# Оптимізація pip
ENV PIP_NO_CACHE_DIR=1
ENV PIP_DISABLE_PIP_VERSION_CHECK=1

# Робоча директорія
WORKDIR /app

# Копіювання та встановлення залежностей
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Копіювання вихідного коду
COPY src/ ./src/
COPY main.py .
COPY validate_models.py .
COPY generate_figures.py .

# Створення директорій для даних та результатів
RUN mkdir -p /app/data /app/output /app/output/model_history

# Змінні середовища для TensorFlow
ENV TF_CPP_MIN_LOG_LEVEL=2
ENV TF_FORCE_GPU_ALLOW_GROWTH=true
ENV CUDA_VISIBLE_DEVICES=0

# Entrypoint
ENTRYPOINT ["python"]
CMD ["main.py", "--mode", "full"]
```

Лістинг А.20 – Використання NVIDIA L4 GPU

```
import tensorflow as tf

print("Доступні GPU:")
gpus = tf.config.list_physical_devices('GPU')
for gpu in gpus:
```

```
    print(f"  - {gpu}")

print(f"\nTensorFlow version: {tf.__version__}")
print(f"CUDA доступна: {tf.test.is_built_with_cuda()}")
print(f"GPU доступна для TF: {tf.test.is_gpu_available()}")
Доступні GPU:
  - PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')

TensorFlow version: 2.13.0
CUDA доступна: True
GPU доступна для TF: True
# nvidia-smi під час тренування
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05   Driver Version: 535.104.05   CUDA Version: 12.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  NVIDIA L4           On   | 00000000:00:04.0 Off |                    0 |
| N/A   62C    P0    45W /  75W |  12847MiB / 24576MiB |     89%      Default |
+-------------------------------+----------------------+----------------------+
```

Лістинг А.21 – Скрипти для автоматизованого розгортання

```
#!/bin/bash

# Конфігурація
SERVER="user@gpu-server.example.com"
REMOTE_DIR="/home/user/genetic_nas"

echo "🚀 Розгортання на сервер..."

# 1. Створення директорії на сервері
ssh $SERVER "mkdir -p $REMOTE_DIR"

# 2. Копіювання файлів
echo "📦 Копіювання файлів..."
rsync -avz --progress \
    --exclude='output/' \
    --exclude='__pycache__/' \
    --exclude='*.pyc' \
    --exclude='.git/' \
    ./ $SERVER:$REMOTE_DIR/

# 3. Копіювання датасету (якщо потрібно)
echo "📊 Копіювання датасету..."
rsync -avz --progress \
    ./data/ $SERVER:$REMOTE_DIR/data/
```

```
echo "✅ Розгортання завершено"
#!/bin/bash

SERVER="user@gpu-server.example.com"
REMOTE_DIR="/home/user/genetic_nas"

echo "🧬 Запуск генетичного алгоритму..."

ssh $SERVER << 'EOF'
cd /home/user/genetic_nas

# Збірка Docker image
docker build -t genetic_nas:latest .

# Запуск контейнера
docker run --gpus all \
  --name genetic_nas_run \
  -v $(pwd)/data:/app/data \
  -v $(pwd)/output:/app/output \
  genetic_nas:latest \
  main.py --mode full --generations 15

echo "✅ Експеримент завершено"
EOF
#!/bin/bash

SERVER="user@gpu-server.example.com"
REMOTE_DIR="/home/user/genetic_nas"

# Моніторинг логів в реальному часі
ssh $SERVER "tail -f $REMOTE_DIR/output/evolution_*.log | grep -E '(ERROR|
WARNING|🏆)'"
#!/bin/bash

set -e  # Зупинка при помилці

echo "🚀 Повний pipeline розгортання та запуску"

# 1. Розгортання
./deploy_to_server.sh

# 2. Запуск
./run_on_server.sh

# 3. Моніторинг
./monitor_errors.sh
```

Лістинг А.22 – Використання GPU пам'яті

```
tf.config.experimental.set_memory_growth(device, True)
del model
tf.keras.backend.clear_session()
```

Лістинг А.23 – Оптимальні гіперпараметри

```
Epoch  1/50: loss=0.8234, acc=0.6842, val_loss=0.4512, val_acc=0.8427
Epoch  2/50: loss=0.3876, acc=0.8654, val_loss=0.2914, val_acc=0.9012
Epoch  5/50: loss=0.1523, acc=0.9421, val_loss=0.1654, val_acc=0.9516
Epoch 10/50: loss=0.0754, acc=0.9745, val_loss=0.1287, val_acc=0.9718
Epoch 15/50: loss=0.0512, acc=0.9821, val_loss=0.1156, val_acc=0.9798
Epoch 20/50: loss=0.0387, acc=0.9867, val_loss=0.1089, val_acc=0.9839
Epoch 23/50: loss=0.0321, acc=0.9889, val_loss=0.1045, val_acc=0.9859 ← BEST
Epoch 24/50: loss=0.0298, acc=0.9901, val_loss=0.1078, val_acc=0.9839
Epoch 25/50: loss=0.0281, acc=0.9908, val_loss=0.1112, val_acc=0.9819
Epoch 26/50: loss=0.0269, acc=0.9914, val_loss=0.1145, val_acc=0.9798

EarlyStopping: patience exceeded, restoring weights from epoch 23
Final fitness: 0.9859 (98.59%)
```

Лістинг А.24 – Методологія валідації

```
for generation in history['generations']:
   for model_data in generation['models']:
      # Реконструкція хромосоми з JSON
      chromosome = Chromosome.from_dict(model_data['chromosome'])

      # Побудова Keras моделі
      model = build_model(chromosome, input_shape=(64, 64, 3))

      # Завантаження збережених ваг
      model_id = model_data['model_id']
      weights_path = f"output/model_history/{model_id}.h5"
      model.load_weights(weights_path)
# Компіляція з оригінальними гіперпараметрами
model.compile(
   optimizer=get_optimizer(chromosome),
   loss='sparse_categorical_crossentropy',
   metrics=['accuracy']
)

# Оцінка на валідаційному наборі
loss, validated_accuracy = model.evaluate(X_val, y_val, verbose=0)
```

Лістинг А.25 – Порівняння оригінальних та відтворених результатів

```
import scipy.stats as stats

# Paired t-test
t_stat, p_value = stats.ttest_rel(original_fitnesses, validated_accuracies)

print(f"T-statistic: {t_stat:.6f}")
print(f"P-value: {p_value:.6f}")

# Результат:
# T-statistic: 0.000142
# P-value: 0.999887

# Висновок: Немає статистично значущої різниці (p > 0.05)
```