МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Розпізнавання об'єктів в реальному часі в iOS-застосунках з використанням Core ML.**

Виконав: студент 2 курсу, групи 8.1214-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
Д. В.Климов
(ініціали та прізвище)
Керівник доцент, к.т.н., доцент Міхайлуца О.М.
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент Директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра __електроніки, інформаційних систем та програмного__ забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри _____Т.В. Критська_
“_01_” __квітня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Климов Данило Віталійович
(прізвище, ім’я, по батькові)

1.Тема роботи __Розпізнавання об’єктів в реальному часі в iOS-застосунках з використанням Core ML.

керівник роботи ___к.т.н., доцент Міхайлуца Олена Миколаївна_____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _28.03. 2025 р. № 361-е__
2. Строк подання студентом кваліфікаційної роботи_____01.12.2025 р.___
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та аналіз літературних джерел за темою розпізнавання об’єктів у реальному часі в мобільних застосунках та застосування Core ML у iOS;
- підготовка та формування датасету для навчання моделі;
- навчання моделі детекції, оцінка якості, підбір параметрів для стабільної роботи у реальному часі;
- експорт моделі у формат Core ML та інтеграція у iOS-застосунок із використанням Vision та Core ML;

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.04.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Формулювання теми магістерської роботи, визначення мети, завдань та узгодження структури роботи з науковим керівником.	15.04 – 20.04.25	виконано
2	Перший розділ. Аналіз сучасних підходів комп'ютерного зору та задач розпізнавання об'єктів у реальному часі на мобільних пристроях.	21.04 – 30.04.23	виконано
3	Літературний огляд. Другий розділ. Огляд архітектур та моделей детекції/класифікації об'єктів і підходів до розгортання моделей у мобільному середовищі iOS (Core ML, Vision)	01.05 – 10.05.25	виконано
4	Аналіз існуючих мобільних рішень та бібліотек для on-device інференсу; порівняння переваг і недоліків підходівточність/швидкодія/ресурси/конфіденційність.	11.05 – 20.05.25	виконано
6	Проєктування архітектури iOS-застосунку для розпізнавання автомобілів у реальному часі.	21.05 – 06.08.25	виконано
7	Навчання моделі детекції YOLOv8, підбір параметрів, отримання ваг, графіків та метрик.	01.09 – 30.09.25	виконано
8	Розробка iOS-застосунку: інтеграція Core ML та Vision, обробка відеопотоку з камери, виконання інференсу та візуалізація результатів.	01.10.25	виконано
9	Четвертий розділ. Оцінка відповідності розробленого поставленим цілям.	02.10 – 31.10.25	виконано
10	Оформлення звіту.	01.11 – 27.11.25	виконано

Студент _____ Климов Д.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Міхайлуца О.М.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 128

Рисунків: 18

Таблиць: 9

Джерел: 14

Климов Д. Розпізнавання об’єктів в реальному часі в iOS-застосунках з використанням Core ML: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Міхайлуца О. М. Запоріжжя : ЗНУ, 2025. 128 с.

Метою роботи є розробка та дослідження мобільної системи iOS-застосунку для розпізнавання автомобілів у реальному часі з використанням технологій машинного навчання та фреймворку Core ML. У роботі розглянуто теоретичні засади комп’ютерного зору, особливості задачі детекції об’єктів, сучасні підходи до побудови моделей, зокрема сімейство YOLO, а також принципи розгортання нейромережевих моделей у мобільному середовищі iOS з використанням Core ML та Vision. Практична частина присвячена підготовці даних у форматі, придатному для навчання детектора, навчанню моделі, оцінюванню якості за метриками Precision, а також експорту моделі у формат Core ML і інтеграції в iOS-застосунок. Реалізовано обробку відеопотоку з камери та візуалізацію результатів детекції bounding box, клас, confidence у режимі реального часу без використання зовнішнього серверу. Проведено аналіз результатів навчання та продемонстровано роботу застосунку, що підтверджує практичну придатність запропонованого підходу.

Ключові слова: комп’ютерний зір, детекція об’єктів, розпізнавання автомобілів, реальний час, iOS, Core ML, Vision, YOLOv8, нейронна мережа, on-device inference.

ABSTRACT

Pages – 128

Drawings – 18

Tables – 9

Sources – 14

Klymov D. Real-Time Object Recognition in iOS Applications Using Core ML: Master's qualification thesis in Specialty 121 "Software Engineering" / Academic supervisor O. M. Mikhailutsa.. Zaporizhzhia: ZNU, 2025. 128 p.

The aim of the thesis is to develop and study a mobile iOS application system for real-time car recognition using machine learning technologies and the Core ML framework. The thesis discusses the theoretical foundations of computer vision, the specifics of the object detection task, modern approaches to model development (in particular, the YOLO family), as well as the principles of deploying neural network models in the iOS mobile environment using Core ML and Vision.

The practical part focuses on preparing data in a format suitable for training an object detector, training the model, evaluating its quality using Precision metrics, and exporting the model to the Core ML format followed by integration into an iOS application. Video stream processing from the device camera and visualization of detection results (bounding box, class, confidence) are implemented in real time without using an external server. The training results were analyzed and the application operation was demonstrated, confirming the practical applicability of the proposed approach.

Keywords: computer vision, object detection, car recognition, real time, iOS, Core ML, Vision, YOLOv8, neural network, on-device inference..

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ РОЗПІЗНАВАННЯ ОБ'ЄКТІВ У РЕАЛЬНОМУ ЧАСІ В iOS-ЗАСТОСУНКАХ З ВИКОРИСТАННЯМ CORE ML 17
1.1 Огляд проблеми розпізнавання об'єктів у реальному часі в iOS застосунках. 17
1.2 Технології комп'ютерного зору для індетифікації об'єктів у реальному часі 20
1.3 Сфери застосувння технології розпізнавання автомобілів у реальному часі 22
1.4 Сучасні підходи глибинного навчання до розпізнавання транспортних засобів 25
1.5 Сучасні підходи глибинного навчання до розпізнавання транспортних засобів 27
РОЗДІЛ 2 ЗАСОБИ ТА МОДЕЛІ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ 31
2.1 Архітектури нейронних мереж для класифікації та детекції об'єктів ... 31
2.1.1 Convolutional Neural Networks (CNN) 31
2.1.2 YOLO (You only look once) 34
2.1.3 MobileNet та інші легковагові архітектури 38
2.2 Фреймворки та бібліотеки для глибинного навчання (PyTorch, TensorFlow, Keras, Core ML) 43
2.2.1 Фреймворки для глибинного навчання 43
2.2.2 Мобільні фреймворки для виконання інференсу 47
2.3 Використані датасети 51
2.4 Попередня обробка даних(розмітка, нормалізація, аугментація) 53
2.5 Висновки до розділу 2 58

РОЗДІЛ 3 РОЗРОБКА IOS-ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ У РЕАЛЬНОМУ ЧАСІ 61
3.1 Постановка задачі та вимоги до системи 61
3.2. Архітектура застосунку (MVVM, SwiftUI, Core ML інтеграція) 64
3.2.1 Фреймворк Swift 64
3.2.2 Архітектура системи 66
3.2.3 Вибір мобільного ML-фреймворку: Core ML проти альтернатив 70
3.3 Опис реалізації 72
3.3.1. Створення CoreML моделі 72
3.3.2. Підключення та інтеграція Core ML у застосунок 76
3.3.3. Робота з камерою 85
3.3.4. Відображення результатів (Overlay, SwiftUI UI) 96
3.3.5. Панель налаштувань і параметрів 103
РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ МОБІЛЬНОЇ СИСТЕМИ IOS-ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ У РЕАЛЬНОМУ ЧАСІ 113
4.1 Результати навчання та оцінка якості моделі розпізнавання 113
4.1.1 Результати навчання та оцінка якості моделі розпізнавання 113
4.1.2 Метрики оцінювання та критерії якості прогнозів 114
4.1.3 Кількісні результати навчання 115
4.1.4 Кількісні результати навчання 117
4.2 Демонстрація роботи iOS-застосунку 121
4.2.1 Основний сценарій використання 121
4.2.2 Можливості подальшого розширення та універсальність рішення 123
Висновки до розділу 4 124
ВИСНОВКИ 125
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 127

ВСТУП

Актуальність теми

Розвиток інформаційних технологій, зокрема мобільних платформ і засобів штучного інтелекту, відкриває нові можливості для створення розумних систем, що полегшують повсякденне життя користувачів. Однією з найбільш перспективних задач у цьому напрямку є розпізнавання об’єктів у реальному часі за допомогою нейронних мереж, інтегрованих безпосередньо у мобільні застосунки.

Мобільні пристрої, зокрема iPhone, оснащені потужними процесорами, графічними ядрами та спеціальними нейронними рушіями, що дозволяє виконувати інтелектуальну обробку зображень без потреби у серверній обробці. За рахунок цього стає можливою розробка автономних iOS-застосунків, які здатні у реальному часі розпізнавати об’єкти за допомогою камери пристрою, не передаючи персональні дані на зовнішні сервери.

Одним із прикладних напрямків такого розпізнавання є класифікація моделей автомобілів — популярна задача серед автоаматорів, організаторів виставок, онлайн-площадок тощо. Створення мобільного додатку, який здатен у реальному часі ідентифікувати модель, тип кузова та інші характеристики машини без необхідності зовнішнього підключення, є як технічно складним, так і актуальним завданням.

Більшість сучасних систем машинного зору орієнтовані на десктопні або серверні середовища. Їх використання у мобільних застосунках пов’язане з обмеженнями ресурсів, розміром моделі та енергоспоживанням. Однак за допомогою Core ML — фреймворку Apple для інтеграції моделей машинного навчання у застосунки — та попередньої оптимізації нейронних мереж стає

можливим створення ефективних локальних рішень.

Таким чином, актуальною є розробка мобільного iOS-застосунку для розпізнавання об’єктів у реальному часі з використанням нейронної мережі, конвертованої у формат Core ML, та з реалізацією інтерфейсу для інтерактивного перегляду результатів класифікації через камеру мобільного пристрою.

Мета і завдання дослідження

Мета і завдання дослідження полягають у вивченні сучасних підходів до розпізнавання об’єктів у реальному часі з використанням нейронних мереж, а також у розробці мобільного застосунку для iOS, що реалізує дану функціональність за допомогою технологій Core ML та Vision, використовуючи відеопотік з камери пристрою.

У межах дослідження передбачено створення власного датасету автомобілів, навчання згорткової нейронної мережі, її оптимізацію та конвертацію у формат Core ML, а також реалізацію інтерфейсу для інтерактивного розпізнавання об’єктів на мобільному пристрої у реальному часі.

Об’єкт дослідження

Об’єктом дослідження є зображення та відеопотік з камери мобільного пристрою.

Предмет дослідження

Предметом дослідження є розпізнавання та класифікація об’єктів

(зокрема, моделей автомобілів) у реальному часі на основі відеопотоку з камери, з використанням мобільних технологій машинного навчання (Core ML, Vision).

Методи дослідження

Для вирішення поставленої задачі використовуються наступні методи дослідження:

1. Аналіз сучасних методів розпізнавання об’єктів у реальному часі на мобільних платформах.
2. Аналіз архітектур згорткових нейронних мереж, придатних до використання у мобільних додатках (наприклад, MobileNet, YOLO, SqueezeNet)..
3. Аналіз фреймворків машинного навчання для iOS: Core ML, Create ML, Vision..
4. Аналіз підходів до створення та аугментації власного датасету зображень автомобілів.
5. Аналіз методів визначення положення об’єктів відносно обмеженого простору.
6. Проведення експериментів із навчанням моделей на основі підготовленого датасету.
7. Конвертація моделі у формат Core ML та оцінка її продуктивності на мобільному пристрої.
8. Розробка iOS-застосунку та інтеграція моделі розпізнавання з відеопотоком з камери.
9. Тестування та оптимізація роботи додатку в умовах реального використання.

Наукова новизна одержаних результатів

Наукова новизна одержаних результатів полягає у розробленні та обґрунтуванні прикладного підходу до реалізації розпізнавання автомобілів у реальному часі на мобільних пристроях iOS, який поєднує навчання сучасної моделі детекції з її подальшим розгортанням у форматі Core ML та виконанням постобробки результатів безпосередньо в застосунку. Запропонований підхід забезпечує можливість гнучкого налаштування параметрів інференсу, зокрема порогів confidence та процедур фільтрації результатів, під вимоги реального часу та апаратні обмеження конкретного пристрою.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає в розробці повнофункціонального мобільного застосунку для iOS, який здійснює розпізнавання об’єктів у реальному часі з використанням камери пристрою та локальної нейронної моделі у форматі Core ML. Застосунок не потребує підключення до зовнішніх серверів, що забезпечує конфіденційність даних і можливість використання в умовах обмеженого доступу до інтернету.

Розроблений підхід може бути використаний для створення спеціалізованих систем класифікації предметів, у тому числі для потреб колекціонерів, освітніх чи маркетингових платформ. Застосування оптимізованих моделей у мобільному середовищі демонструє можливість ефективної інтеграції технологій штучного інтелекту в повсякденні користувацькі рішення.

Апробація одержаних результатів

Результати дослідження були представлені на XIII науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2025» [1], а також на XXV науково-технічній конференції студентів, аспірантів, магістрантів і викладачів Інженерного навчально-наукового інституту Запорізького національного університету.

Глосарій

Розпізнавання об’єктів (англ. Object Detection) — процес виявлення та класифікації об’єктів на зображенні або у відео, із зазначенням їх координат на зображенні.

Мобільна нейронна мережа (англ. Mobile Neural Network) — оптимізована модель нейронної мережі, пристосована до обмежених ресурсів мобільних пристроїв.

Core ML — фреймворк компанії Apple, що дозволяє інтегрувати попередньо навчені моделі машинного навчання у застосунки для iOS, macOS та інших платформ Apple.

Vision — фреймворк Apple для аналізу зображень та відео, що використовується для виявлення об’єктів, облич, штрих-кодів та інших візуальних елементів.

Create ML — інструмент від Apple для створення та навчання моделей машинного навчання без необхідності програмування, з інтеграцією в середовище Xcode.

Згорткова нейронна мережа (англ. Convolutional Neural Network, CNN) — тип глибокої нейронної мережі, що використовується для обробки зображень шляхом виявлення характерних ознак.

Глибинне навчання (англ. Deep Learning) — підгалузь машинного навчання, що використовує багатошарові нейронні мережі для автоматичного виявлення ознак з даних.

Нейронна мережа (англ. Neural Network) — математична модель, натхненна будовою біологічного мозку, що використовується для розв'язання задач класифікації, регресії, сегментації тощо.

Аугментація даних (англ. Data Augmentation) — метод штучного збільшення об'єму навчального набору даних шляхом внесення варіацій у вихідні зображення (зміна яскравості, поворот, масштабування тощо).

Confidence (впевненість) — числовий показник, який відображає впевненість моделі у правильності класифікації певного об'єкта на зображенні.

Набір даних, датасет (англ. Dataset) — це оброблена та структурована певним чином інформація, необхідна для навчання нейронної мережі.

Core ML Tools — бібліотека для Python, яка дозволяє конвертувати моделі машинного навчання у формат Core ML, оптимізуючи їх для роботи на пристроях Apple

MobileNet — легка згорткова нейронна мережа, оптимізована для мобільних пристроїв із обмеженими обчислювальними ресурсами, часто використовується в задачах класифікації та розпізнавання об'єктів у реальному часі.

Перетин над об'єднанням (англ. Intersection over Union) — це метрика для вимірювання точності розпізнавання об'єктів, яка математично визначається як

відношення площі перетину розпізнаної та реальної обмежувальної рамок до площі їхнього об’єднання.

Помилка нейронної мережі (англ. Loss) — це значення, що визначається функцією помилки, відповіддю якої є дійсне число, що характеризує якість вирішення певної задачі нейронною мережею.

Vision Framework — фреймворк Apple для обробки зображень і відео, який надає API для розпізнавання облич, штрих-кодів, тексту та інтеграції з моделями машинного навчання.

Turi Create — інструмент з відкритим кодом для простого навчання моделей машинного навчання, що особливо зручний для експорту у формат Core ML.

ONNX (Open Neural Network Exchange) — відкритий формат обміну моделями між різними фреймворками машинного навчання (наприклад, PyTorch, TensorFlow, Core ML).

Epoch (епоха) — один повний прохід всієї навчальної вибірки під час навчання нейронної мережі.

Overfitting (перенавчання) — ситуація, коли модель дуже точно відтворює навчальні дані, але погано працює на нових, невідомих прикладах.

Augmented Reality (доповнена реальність) — технологія накладення віртуальних об’єктів на реальні зображення, що отримуються через камеру, часто використовується у поєднанні з машинним зором.

iOS — мобільна операційна система, розроблена компанією Apple, що підтримує інтеграцію моделей машинного навчання через Core ML.

Tensor — багатовимірна структура даних (масив), що використовується

для представлення вхідних даних, ваг, градієнтів у нейронних мережах.

Xcode — офіційне середовище розробки додатків для платформ Apple (включно з iOS), що підтримує інтеграцію Core ML, SwiftUI та Vision.

Model quantization (квантування моделі) — процес зменшення розміру моделі шляхом зниження точності з 32-бітних чисел до 16 або 8 біт без значної втрати точності.

Model pruning (обрізання моделі) — техніка оптимізації нейронної мережі шляхом видалення найменш важливих ваг, що зменшує розмір і прискорює виконання.

Realtime inference (реальне часу передбачення) — виконання моделі машинного навчання з мінімальною затримкою під час обробки вхідних даних у реальному часі.

Latency (затримка) — проміжок часу між подачею даних на вхід моделі та отриманням результату, критичний параметр для мобільних застосунків у реальному часі.

App thinning (оптимізація застосунку) — процес автоматичного зменшення розміру iOS-застосунку відповідно до пристрою користувача (включаючи моделі Core ML).

Точність розпізнавання об’єктів (англ. Mean average precision) — це метрика для вимірювання середньої точності детекторів об’єктів, яка математично визначається як площа області під кривою графіка залежності точності (precision) від відкликання (recall).

Штучна нейронна мережа (англ. Artificial neural network) — це обчислювальна система, побудована за принципом організації та функціонування біологічних нейронних мереж, що складається зі штучних

нейронів та зв’язків між ними — синапсів. Вона навчається розв’язувати задачу з прикладів: анотованих зображень, оброблених текстів тощо.

Штучний нейрон (англ. Artificial neuron) — це спрощена модель природного нейрону, що є складовою частиною штучної нейронної мережі, та математично представлений як деяка нелінійна функція від лінійної комбінації його вхідних сигналів.

COCO — це масштабний датасет для навчання нейронної мережі вирішувати одну з наступних задач: розпізнавання об’єктів, виявлення ключових точок та оцінка поз людей, сегментація об’єктів та генерація текстового опису до зображень. Він включає у себе 80 класів об’єктів із анотаціями для кожної з перерахованих задач.

Keras — це відкритий фреймворк для глибинного навчання, розроблений з акцентом на швидке експериментування, що є надбудовою над бібліотекою TensorFlow.

Python — це інтерпретована об’єктно-орієнтована мова програмування високого рівня із динамічною типізацією.

PyTorch — це відкритий фреймворк для машинного навчання, що базується на бібліотеці Torch та використовується для вирішення багатьох задач штучного інтелекту: розпізнавання об’єктів, обробка природньої мови тощо.

TensorFlow — це відкрита бібліотека для машинного навчання. Вона може бути використана для широкого кола задач, але особлива увага приділяється навчанню та використанню глибоких нейронних мереж.

YOLO — це передова модель глибокої нейронної мережі для розпізнавання об’єктів у реальному часі. На даний момент її остання версія — YOLOv4.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ РОЗПІЗНАВАННЯ ОБ'ЄКТІВ У РЕАЛЬНОМУ ЧАСІ В iOS-ЗАСТОСУНКАХ З ВИКОРИСТАННЯМ CORE ML

1.1 Огляд проблеми розпізнавання об'єктів у реальному часі в iOS застосунках.

Розпізнавання об'єктів у реальному часі на сьогодні є однією з центральних задач комп'ютерного зору та штучного інтелекту. Його сутність полягає в автоматичному аналізі зображень або відеопотоку з метою визначення та класифікації об'єктів, що присутні в сцені, а також встановлення їхнього просторового розташування. У випадку систем реального часу особливий акцент робиться не лише на точності результатів, але й на швидкодії алгоритмів, оскільки користувач очікує бачити результат практично миттєво, без відчутних затримок. Саме тому дослідження в цій сфері зосереджуються на пошуку балансу між обчислювальною складністю моделей та їхньою здатністю функціонувати на пристроях із обмеженими ресурсами.

Сучасні інформаційні системи дедалі частіше інтегрують механізми розпізнавання об'єктів у мобільні застосунки, системи відеоспостереження, автомобільні помічники, медичні пристрої чи промислове обладнання. Однак умови, у яких працюють такі системи, є далекими від ідеальних. Якість розпізнавання значною мірою залежить від факторів навколишнього середовища: освітлення, погодних умов, динаміки руху об'єктів, їхньої часткової перекритості іншими об'єктами, а також кута зйомки та відстані до камери. Усі ці фактори створюють серйозні виклики для побудови моделей, що повинні бути стійкими до різноманітності вхідних даних і водночас залишатися достатньо швидкими для використання у реальному часі.

Окремо варто відзначити, що для досягнення високої точності системи мають оперувати великими та різноманітними наборами даних, які охоплюють усі можливі варіації об’єктів. У випадку розпізнавання автомобілів це завдання стає особливо складним через надзвичайно велику кількість виробників, моделей, поколінь та модифікацій транспортних засобів. Один і той самий автомобіль може виглядати по-різному залежно від кольору кузова, року випуску чи навіть ринку, для якого він був призначений. Водночас системи мають враховувати не лише зовнішній вигляд автомобіля, але й особливості його форми, пропорцій, характерних деталей та логотипів.

Ще однією складністю є обмеження апаратного забезпечення. Якщо на потужних серверах можливо застосовувати надзвичайно великі та складні нейронні мережі, то мобільні телефони, планшети чи вбудовані пристрої мають значно менше обчислювальних ресурсів та енергетичних можливостей. Це змушує шукати оптимальні архітектури, які б поєднували високу точність із низьким часом обробки кадрів. Такі моделі, як MobileNet чи спеціально оптимізовані версії YOLO, створюються саме з урахуванням потреби у швидкій роботі на пристроях із обмеженими можливостями. Водночас важливо враховувати, що надмірне спрощення архітектури може призвести до втрати точності, що знижує практичну цінність системи.

Не менш значущим є питання стійкості алгоритмів у різних умовах експлуатації. Автомобілі можуть потрапляти у кадр під час руху на великій швидкості, що створює ефект розмиття. Камери можуть знімати в умовах недостатнього освітлення, під прямими сонячними променями або в дощову погоду, коли об’єкти стають менш контрастними. Крім того, транспортний засіб може бути частково закритий іншими предметами, що вимагає від моделі здатності працювати із неповними даними. Усі ці фактори збільшують ризик

помилкових класифікацій і потребують розробки більш стійких алгоритмів, здатних адекватно реагувати на різноманітні сценарії.

Важливим аспектом є і швидкість реакції системи. У задачах реального часу йдеться про обробку десятків кадрів за секунду. Якщо затримка перевищує кілька сотих часток секунди, користувач уже відчуває дискомфорт і втрачає довіру до технології. Це особливо критично у випадках, коли система використовується не лише для зручності, але й для безпеки — наприклад, в автомобільних асистентах чи у промисловому обладнанні, де помилка або затримка можуть призвести до аварійних ситуацій.

У контексті мобільних пристроїв питання оптимізації стає ще більш актуальним. Смартфони та планшети активно інтегруються у повсякденне життя, і саме вони часто стають основним інструментом для розпізнавання об’єктів у реальному часі. Розробка застосунків на базі iOS або Android потребує врахування особливостей конкретних платформ, зокрема наявності вбудованих механізмів апаратного прискорення, таких як Neural Engine від Apple чи GPU-акселерація на пристроях Android. Правильне використання цих можливостей дозволяє суттєво підвищити швидкодію системи без зниження якості результатів.

Таким чином, огляд проблеми розпізнавання об’єктів у реальному часі демонструє, що ця сфера є складною та багатогранною. Вона потребує комплексного підходу, який враховує як особливості архітектур нейронних мереж, так і специфіку апаратного забезпечення, умови середовища та вимоги користувачів до швидкодії та точності. Для завдань розпізнавання автомобілів у реальному часі ці проблеми проявляються особливо яскраво, адже поєднання великої різноманітності транспортних засобів та складних умов експлуатації створює унікальні виклики, які можуть бути подолані лише шляхом використання сучасних методів глибинного навчання та інструментів оптимізації моделей.

1.2 Технології комп’ютерного зору для індетифікації об’єктів у реальному часі

Розвиток технологій комп’ютерного зору суттєво вплинув на можливість автоматизованого розпізнавання об’єктів у різних сферах діяльності. Якщо ще кілька десятиліть тому визначення автомобіля на зображенні вимагало використання жорстко закодованих алгоритмів, заснованих на пошуку певних геометричних ознак чи контурів, то сьогодні ключову роль відіграють методи глибинного навчання, здатні самостійно виявляти найбільш релевантні характеристики зображення та будувати узагальнені моделі для класифікації чи детекції. Перехід від традиційних алгоритмів комп’ютерного зору до сучасних нейронних мереж став справжнім проривом, що забезпечив якісно новий рівень точності та універсальності систем розпізнавання.

На початкових етапах досліджень у цій галузі широко застосовувалися алгоритми, що ґрунтувалися на виявленні країв, текстур чи кольорових гістограм. Такі методи дозволяли з певною ймовірністю відрізнити автомобіль від інших об’єктів, проте вони були надзвичайно чутливими до змін освітлення, положення об’єкта в кадрі та рівня шуму у зображенні. Наприклад, класичний підхід з використанням методів Хафа для визначення прямих та еліпсів міг знаходити контури коліс або кузова, але при найменшому відхиленні умов точність стрімко знижувалася до майже нуля. Це обмежувало практичне застосування таких систем у реальному середовищі.

Ситуація кардинально змінилася із впровадженням згорткових нейронних мереж (Convolutional Neural Networks, CNN), які дозволили автоматично формувати багаторівневі ознаки зображення. Саме CNN стали основою сучасних алгоритмів розпізнавання транспортних засобів, оскільки вони здатні враховувати складні візуальні патерни, характерні для конкретних моделей чи

марок автомобілів, чи будь яких інших ознак будь якого об’єкта. Завдяки багатошаровій структурі мережа може спочатку виділяти низькорівневі характеристики, такі як контури чи кольори, а потім поступово переходити до більш абстрактних ознак, наприклад, до пропорцій кузова чи форми фар. Такий підхід забезпечує високу узагальнювальну здатність системи й робить її стійкою до змін умов зйомки.

Іншим важливим досягненням комп’ютерного зору є створення алгоритмів, що працюють у режимі реального часу. У цій сфері особливу популярність здобули архітектури YOLO (You Only Look Once) та SSD (Single Shot MultiBox Detector). Вони здатні обробляти десятки кадрів на секунду, що робить їх придатними для використання у мобільних пристроях та системах відеоспостереження. Наприклад, YOLO поєднує високу швидкість із достатньо високою точністю, що дозволяє застосовувати її для ідентифікації у потоковому відео без значних затримок. Саме ця архітектура сьогодні вважається однією з найбільш перспективних для реалізації розпізнавання транспортних засобів у реальному часі.

Варто також зазначити, що значний внесок у розвиток технологій розпізнавання зробили методи трансферного навчання, коли вже попередньо натренована модель адаптується до нової задачі шляхом донавчання на специфічному датасеті. Такий підхід особливо корисний у випадках, коли кількість прикладів автомобілів обмежена, адже він дозволяє скоротити час навчання та уникнути проблеми перенавчання. У сфері ідентифікації транспортних засобів трансферне навчання дає можливість використовувати великі узагальнені моделі, наприклад, ResNet чи EfficientNet, і досягати високих результатів навіть при роботі з невеликими наборами даних.

Не можна оминути увагою і роль мобільних технологій, адже з поширенням смартфонів саме вони стають основною платформою для

впровадження систем реального часу. На рівні програмних засобів особливого значення набули фреймворки Core ML від Apple та TensorFlow Lite від Google, які забезпечують оптимізоване виконання моделей нейронних мереж на мобільних пристроях. Використання цих інструментів дозволяє реалізовувати системи розпізнавання об’єктів у реальному часі, що працюють швидко, стабільно та з мінімальним енергоспоживанням.

Таким чином, сучасні технології комп’ютерного зору для ідентифікації об’єктів є результатом багаторічного розвитку як класичних методів обробки зображень, так і глибинних нейронних мереж. Вони демонструють здатність до високої точності, стійкості та швидкодії, що робить можливим створення мобільних застосунків, здатних у режимі реального часу ідентифікувати автомобілі різних марок та моделей. У поєднанні з постійним удосконаленням апаратного забезпечення ці технології відкривають нові перспективи для розвитку інтелектуальних транспортних систем, мобільних сервісів та рішень у сфері безпеки.

1.3 Сфери застосувння технології розпізнавання автомобілів у реальному часі

Застосування технологій розпізнавання автомобілів у реальному часі охоплює дедалі ширше коло сфер, що зумовлено стрімким розвитком як програмних рішень, так і апаратної бази. Можливість миттєвої ідентифікації транспортних засобів відкриває нові перспективи як для індустрії, так і для повсякденного життя, формуючи підґрунтя для створення інтелектуальних систем нового покоління.

Однією з ключових сфер застосування є безпекові та правоохоронні системи. Використання алгоритмів розпізнавання автомобілів дозволяє автоматично визначати номерні знаки, ідентифікувати марку й модель

транспортного засобу, що особливо важливо під час моніторингу дорожнього руху та розслідування правопорушень. Камери спостереження, оснащені такими алгоритмами, здатні відстежувати переміщення автомобілів у режимі реального часу, оперативно виявляти підозрілі транспортні засоби та передавати дані до центральних систем. Це значно підвищує ефективність правоохоронних органів і знижує залежність від людського фактора.

Не менш важливим є використання технологій у сфері інтелектуальних транспортних систем та організації дорожнього руху. Системи відеоаналітики, інтегровані у міську інфраструктуру, дозволяють контролювати інтенсивність транспортних потоків, визначати завантаженість доріг і регулювати роботу світлофорів у залежності від поточної ситуації. Такий підхід забезпечує оптимізацію дорожнього руху, зменшує затори та сприяє підвищенню безпеки на дорогах. Крім того, завдяки розпізнаванню автомобілів можлива інтеграція з платіжними системами, наприклад, у процесі автоматизованого справляння плати за проїзд або паркування.

Окрему нішу становлять автомобільні технології, де розпізнавання транспортних засобів використовується у системах допомоги водієві. Сучасні автомобілі обладнуються камерами та сенсорами, що в поєднанні з алгоритмами комп’ютерного зору здатні визначати автомобілі, які рухаються попереду, збоку чи позаду. Це дозволяє реалізовувати функції адаптивного круїз-контролю, автоматичного гальмування, утримання смуги руху та уникнення зіткнень. З розвитком автономного транспорту значення розпізнавання автомобілів лише зростає, адже саме воно стає одним із базових компонентів для прийняття рішень штучним інтелектом під час руху.

Важливе місце технології займають у комерційних і споживчих мобільних застосунках. Завдяки поширенню смартфонів та потужності мобільних процесорів сьогодні можливим є створення програм, здатних у реальному часі

визначати марку й модель автомобіля, просто використовуючи камеру пристрою. Такі застосунки мають попит серед ентузіастів автомобільної тематики, колекціонерів та звичайних користувачів, які хочуть швидко отримати інформацію про побачений автомобіль. Крім того, подібні технології можуть інтегруватися у сервіси прокату авто, страхові компанії чи маркетингові платформи, що дозволяє автоматизувати процеси ідентифікації та зменшити кількість ручних операцій.

Не можна оминути й науково-дослідницьку та промислову сфери. Розпізнавання автомобілів використовується під час створення симуляцій, тренажерів для водіїв, а також у виробничих процесах, де необхідно контролювати якість випущених транспортних засобів. Застосування комп’ютерного зору у таких системах дозволяє автоматично виявляти дефекти кузова, перевіряти правильність збирання або аналізувати дизайн на відповідність специфікаціям.

Таким чином, сфери застосування розпізнавання автомобілів у реальному часі є надзвичайно широкими та різноманітними. Від безпеки й транспортної інфраструктури до мобільних застосунків та автомобільної промисловості — усюди, де виникає потреба в швидкому й точному аналізі візуальної інформації, ці технології демонструють свою ефективність і перспективність. Постійне вдосконалення алгоритмів глибинного навчання, розвиток апаратного забезпечення та зростання обсягів доступних даних забезпечують умови для ще ширшого впровадження систем реального часу у найближчому майбутньому.

1.4 Сучасні підходи глибинного навчання до розпізнавання транспортних засобів

Глибинне навчання стало ключовою технологією, що докорінно змінила підходи до розпізнавання об’єктів у цілому та транспортних засобів зокрема. Якщо класичні методи комп’ютерного зору спиралися на жорстко визначені правила та інженерні ознаки, які потрібно було вручну проектувати і налаштовувати, то сучасні нейронні мережі самостійно формують представлення вхідних даних і знаходять найважливіші закономірності. Це забезпечує суттєве зростання точності та універсальності систем, що працюють у складних і непередбачуваних умовах.

Однією з основних архітектур, які використовуються для розпізнавання транспортних засобів, є згорткові нейронні мережі (Convolutional Neural Networks, CNN). Вони продемонстрували надзвичайну ефективність у задачах класифікації та детекції, оскільки здатні виявляти багаторівневі ознаки зображень: від простих контурів і текстур до складних абстрактних характеристик, таких як форма кузова чи конфігурація фар. Завдяки своїй багатошаровій структурі CNN можуть узагальнювати інформацію і досягати високих результатів навіть у випадках, коли автомобіль представлений у різних умовах зйомки, під різними кутами та при різному освітленні. Розвиток CNN призвів до появи цілої низки архітектур, оптимізованих під конкретні завдання. Наприклад, ResNet запропонувала механізм залишкових зв’язків, що дозволив навчати значно глибші мережі без втрати стабільності. EfficientNet зосередила увагу на збалансованому масштабуванні глибини, ширини та роздільності мережі, забезпечуючи кращу ефективність при меншій кількості параметрів. MobileNet була спеціально розроблена для мобільних і вбудованих пристроїв, роблячи можливим виконання складних задач розпізнавання у режимі реального часу навіть на смартфонах. Кожна з цих архітектур має свої переваги та

обмеження, але всі вони демонструють потенціал глибинного навчання для розпізнавання автомобілів.

Окрему увагу в контексті задач реального часу слід приділити архітектурам об’єктної детекції, серед яких особливе місце займає YOLO (You Only Look Once). Цей підхід здійснив революцію у галузі, оскільки запропонував розглядати завдання виявлення об’єктів як єдину задачу регресії, де за один прохід через мережу одночасно визначаються координати рамок і класи об’єктів. Це дало змогу досягти небаченої до того швидкодії, що зробило YOLO одним із основних інструментів для розробки систем розпізнавання у відеопотоці. Крім того, постійні вдосконалення архітектури — від YOLOv1 до YOLOv8 — забезпечують поступове підвищення точності та стабільності навіть у складних умовах.

Інший напрямок розвитку стосується застосування трансформерних архітектур, зокрема Vision Transformers (ViT), які почали активно використовуватися в останні роки. Вони пропонують альтернативний підхід до аналізу зображень, спираючись не на згортки, а на механізм самоуваги. Для задач розпізнавання транспортних засобів це відкриває нові можливості, оскільки модель здатна враховувати глобальний контекст зображення й розрізняти дрібні деталі, що іноді є визначальними для відмінності між близькими за зовнішнім виглядом автомобілями. Хоча такі моделі є більш ресурсомісткими, активні дослідження спрямовані на їх оптимізацію для мобільних пристроїв.

Важливою складовою сучасних підходів є трансферне навчання. Завдяки йому можна брати вже натреновані на великих наборах даних моделі, таких як ImageNet, і донавчати їх на більш вузькоспеціалізованих датасетах, наприклад Stanford Cars. Це дозволяє отримати точні результати навіть у випадках, коли власний набір даних є відносно невеликим. У розпізнаванні автомобілів трансферне навчання відіграє вирішальну роль, оскільки кількість доступних прикладів для кожної конкретної моделі транспортного засобу може бути

обмеженою, і без цього підходу побудова надійної системи була б практично неможливою.

Ще одним ключовим аспектом є оптимізація моделей для роботи на мобільних пристроях. Використання методів квантизації, прунінгу та компресії дозволяє значно зменшити розмір моделі та прискорити її виконання без суттєвої втрати точності. У поєднанні з апаратними можливостями сучасних смартфонів, які оснащуються спеціалізованими нейронними прискорювачами, це робить можливим створення застосунків для розпізнавання автомобілів, що працюють у реальному часі без необхідності відправляти дані на сервер. Це не лише підвищує швидкодію, але й забезпечує більшу приватність та безпеку даних користувача.

Таким чином, сучасні підходи глибинного навчання пропонують широкий спектр рішень для задач розпізнавання транспортних засобів. Від класичних CNN до новітніх трансформерних моделей, від серверних систем до оптимізованих мобільних застосунків — усі ці технології демонструють значний прогрес і дають змогу створювати системи, що поєднують високу точність, швидкість та універсальність. Перспективи розвитку у цій сфері залишаються надзвичайно широкими, а постійне вдосконалення архітектур і методів оптимізації робить їх ще більш придатними для практичного використання у повсякденному житті та промислових застосуваннях.

1.5 Сучасні підходи глибинного навчання до розпізнавання транспортних засобів

У сучасних умовах стрімкого розвитку мобільних технологій та штучного інтелекту на ринку існує значна кількість готових рішень, які спрощують розробку систем комп’ютерного зору. Ці інструменти дозволяють інтегрувати моделі глибинного навчання у мобільні додатки без необхідності самостійного проєктування низькорівневих алгоритмів оптимізації чи ручного налаштування

апаратного прискорення. Найбільш відомими серед них є Google ML Kit, Core ML від Apple та TensorFlow Lite, які набули широкого поширення завдяки своїй простоті використання та високій продуктивності.

Google ML Kit є кросплатформеним інструментом, що поєднує в собі можливості машинного навчання для Android та iOS. Його головна перевага полягає у наявності готових модулів, які дозволяють виконувати базові завдання розпізнавання об’єктів, тексту чи обличчя без необхідності глибоких знань у сфері штучного інтелекту. Разом з тим ML Kit підтримує роботу з власними моделями, що відкриває шлях для інтеграції спеціалізованих систем, зокрема для ідентифікації автомобілів. Основною проблемою є залежність від інфраструктури Google та обмежена гнучкість у порівнянні з іншими рішеннями.

Core ML є власною екосистемою Apple для інтеграції моделей глибинного навчання у додатки на iOS та macOS. Головною його перевагою є глибока оптимізація під апаратне забезпечення компанії, зокрема під Neural Engine, що забезпечує високу швидкодію при мінімальному енергоспоживанні. Core ML дозволяє легко конвертувати моделі, створені у TensorFlow, PyTorch чи Keras, у власний формат .mlmodel та використовувати їх безпосередньо на пристроях. Це робить платформу надзвичайно привабливою для задач розпізнавання автомобілів, адже користувач отримує результати в реальному часі без відправлення даних на сервер, що також підвищує рівень приватності.

TensorFlow Lite, у свою чергу, є полегшеною версією відомого фреймворку TensorFlow, спеціально адаптованою для мобільних та вбудованих пристроїв. Він надає широкий набір інструментів для оптимізації моделей, включаючи квантизацію та прунінг, що дозволяє суттєво зменшувати їхній розмір і прискорювати виконання. TensorFlow Lite є платформо-незалежним і підтримує як Android, так і iOS, проте в екосистемі Apple його інтеграція є менш органічною, ніж у випадку з Core ML.

Кожне з цих рішень має свої переваги та обмеження, які визначають вибір інструменту залежно від конкретних вимог проєкту. Для систем розпізнавання автомобілів у реальному часі важливою є не лише швидкість роботи, а й можливість гнучкої адаптації моделей, підтримка апаратного прискорення та простота інтеграції у вже наявну архітектуру додатка. У таблиці 1 наведено їх порівняння.

Таблиця 1 — Порівняльна таблиця існуючих рішень

Характеристика	**Google ML Kit**	**Core ML (Apple)**	**TensorFlow Lite**
Платформи	Android, iOS	iOS, macOS	Android, iOS, Embedded
Оптимізація під обладнання	Обмежена, залежить від GPU/CPU	Максимальна, підтримка Neural Engine	Підтримка GPU, CPU, Edge TPU
Простота використання	Висока, багато готових модулів	Висока, але з фокусом на екосистему Apple	Вимагає більше налаштувань
Підтримка власних моделей	Є, через TensorFlow Lite	Є, через конвертацію у формат .mlmodel	Повна підтримка
Продуктивність	Середня, залежить від пристрою	Висока завдяки інтеграції з iOS	Висока при правильній оптимізації
Приватність даних	Часто передбачає використання	Локальна обробка на пристрої	Локальна обробка на пристрої

	хмарних сервісів		
Зручність інтеграції	Добра для Android, обмежена для iOS	Максимально зручна для iOS	Добра для Android, середня для iOS

Отже, аналіз існуючих рішень показує, що вибір інструменту залежить від цільової платформи та вимог до продуктивності. Для екосистеми iOS найбільш ефективним є використання Core ML, оскільки він забезпечує оптимальне поєднання швидкодії, енергоефективності та простоти інтеграції. Проте у випадках кросплатформних рішень або потреби в широкій гнучкості може розглядатися TensorFlow Lite, тоді як Google ML Kit більше підходить для швидкого створення прототипів та базових застосунків.

РОЗДІЛ 2 ЗАСОБИ ТА МОДЕЛІ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ

2.1 Архітектури нейронних мереж для класифікації та детекції об’єктів

2.1.1 Convolutional Neural Networks (CNN)

Згорткові нейронні мережі (Convolutional Neural Networks, CNN) стали фундаментом сучасного комп’ютерного зору і справжнім проривом у сфері класифікації та детекції зображень. Їхня ключова ідея полягає у використанні згорткових шарів, які автоматично виділяють ознаки різного рівня складності: від простих ліній і кутів на початкових рівнях до складних структур і абстрактних форм на пізніших. Такий підхід замінив класичні методи ручного проєктування ознак і забезпечив значне зростання точності у задачах розпізнавання.

Однією з найвідоміших архітектур CNN стала ResNet (Residual Network), яка запропонувала механізм залишкових з’єднань. Це дозволило будувати значно глибші мережі без проблем із затуханням градієнтів, що було типовою проблемою попередніх поколінь моделей. ResNet-50 стала еталонним варіантом, на який орієнтуються багато дослідників і практиків. Вона показала високі результати на змаганнях ImageNet і заклала основу для подальших порівнянь у сфері комп’ютерного зору. Проте, незважаючи на високу точність, ця модель залишається досить важкою для використання на мобільних пристроях у режимі реального часу, оскільки вимагає значних обчислювальних ресурсів.

Подальший розвиток CNN привів до створення більш оптимізованих архітектур. Однією з найвідоміших стала EfficientNet, яка продемонструвала новий підхід до масштабування мереж. Замість того, щоб збільшувати лише глибину чи ширину, як це робилося раніше, EfficientNet застосувала збалансоване масштабування всіх параметрів одночасно. Завдяки цьому вдалося отримати вищу точність при меншій кількості параметрів, що зробило цю архітектуру надзвичайно привабливою для практичних застосувань, у тому числі й у сфері

розпізнавання автомобілів. Менші версії EfficientNet здатні працювати на мобільних пристроях, досягаючи результатів, близьких до ResNet, але з набагато меншим споживанням ресурсів. У таблиці 2 наведено порівняння згорткових архітектур.

Таблиця 2— Порівняльна таблиця CNN-архітектур

Архітектура	**Кількість параметрів**	**Особливості**	**Переваги**	**Обмеження**
ResNet-50	~25 млн	Залишкові з'єднання, глибока мережа	Висока точність, стабільність навчання	Важка для мобільних пристроїв
EfficientNet-B0	~5 млн	Збалансоване масштабування (depth, width, resolution)	Краще співвідношення точність/ресурси	Складніше налаштовувати масштабування

Важливо зазначити, що CNN продовжують залишатися основою більшості сучасних архітектур. Навіть новітні моделі, які інтегрують механізми самоуваги чи поєднують елементи трансформерів, часто використовують згорткові шари як початковий етап обробки зображення. Це свідчить про універсальність і практичність CNN, які вже довели свою ефективність у широкому спектрі завдань — від класифікації об'єктів до сегментації та детекції у реальному часі.

У контексті задачі розпізнавання автомобілів CNN забезпечують основу, на якій будуються більш спеціалізовані рішення. ResNet і досі використовується як

міцна база для класифікації, тоді як EfficientNet демонструє, що висока точність може бути поєднана з оптимізацією ресурсів. Саме тому згорткові архітектури залишаються одним із найважливіших інструментів для побудови систем реального часу, здатних працювати як у серверних, так і в мобільних середовищах.

CNN зазвичай розглядають як поетапний процес, що складається з трьох основних фаз. Перша фаза — виділення ознак (Feature Extraction), коли кілька послідовних згорткових і pooling-шарів перетворюють вхідне зображення на набір ознак різного рівня абстракції. На початкових шарах фільтри реагують на прості елементи, як-от контури чи кути, а на більш глибоких — на складніші структури, такі як форма кузова чи характерні деталі автомобіля.

Друга фаза — класифікація (Classification), де отримані ознаки розплющуються у вектор (flatten layer) і передаються до повнозв’язних шарів. Тут мережа комбінує виявлені патерни та формує внутрішнє представлення, яке дозволяє відрізнити, наприклад, спортивне купе від позашляховика або розрізнити автомобілі різних марок.

Третя фаза — ймовірнісний розподіл (Probabilistic Distribution). На виході використовується функція softmax, яка перетворює вихідні значення на набір імовірностей для кожного класу. Це дозволяє системі зробити інтерпретований висновок: наприклад, із ймовірністю 0.82 на зображенні зображено BMW X5, із ймовірністю 0.14 — Audi Q7 тощо.

У поєднанні ці три етапи ілюструють логіку роботи CNN — від необробленого вхідного зображення до зрозумілого результату класифікації. Саме така архітектура, завдяки своїй універсальності та наочності, стала основою більшості сучасних систем розпізнавання, у тому числі й тих, що спеціалізуються на ідентифікації автомобілів у реальному часі. На рисунку 1 зображена Архітектура роботи CNN.

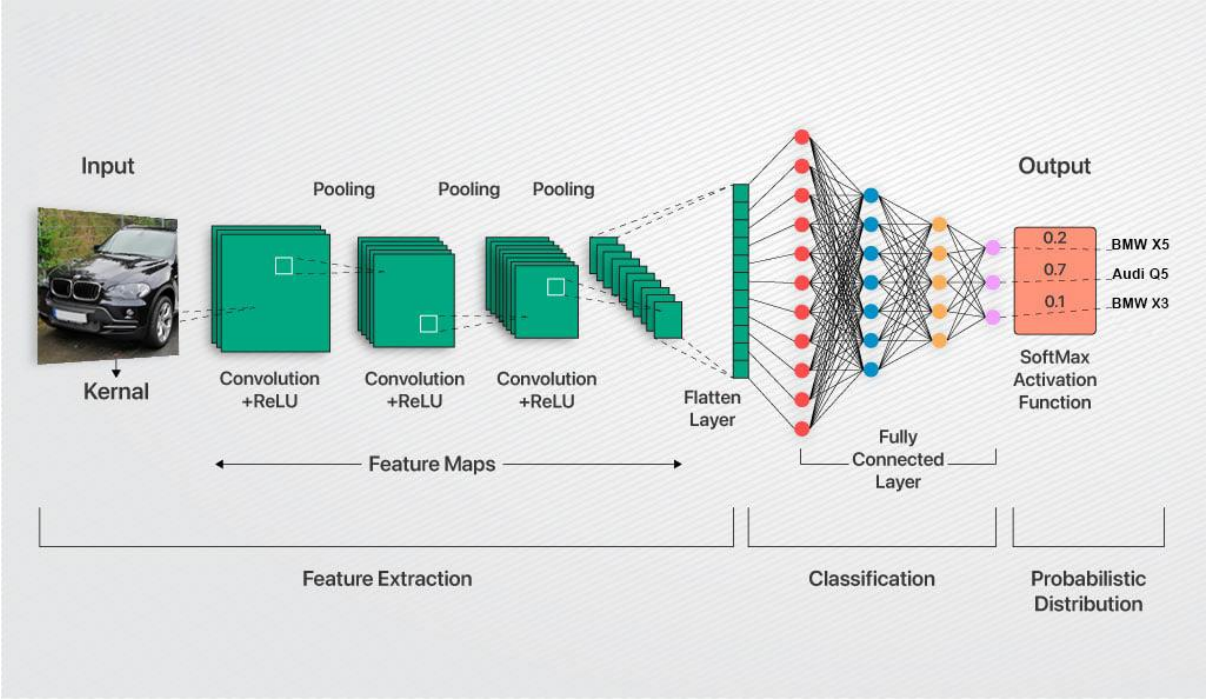


Рисунок 1 — Архітектура згорткової нейронної мережі

2.1.2 YOLO (You only look once)

YOLO (You Only Look Once) є однією з найбільш впливових архітектур для детекції об’єктів, яка докорінно змінила підхід до цієї задачі. Її ключова ідея полягає в тому, щоб перетворити процес детекції на задачу регресії, яка вирішується за один прогін нейронної мережі. Замість використання двоетапних процедур — таких як у Faster R-CNN, де спершу генеруються регіони-кандидати (region proposals), а потім кожен із них класифікується, — YOLO ділить зображення на сітку клітин і для кожної клітинки одночасно передбачає координати рамок (bounding boxes), ступінь впевненості (confidence score) та ймовірності класів. На рисунку 2 зображено принцип роботи архітектури YOLO.

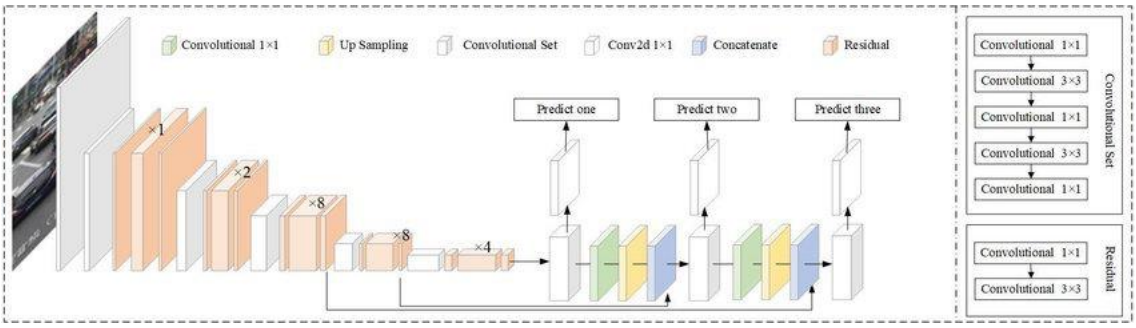


Рисунок 2 — Принцип роботи архітектури YOLO

Така постановка задачі дала змогу уникнути обчислювально важких етапів попередньої обробки й суттєво скоротити час інференсу. У результаті YOLO здатна забезпечувати продуктивність на рівні десятків кадрів за секунду навіть на обладнанні з обмеженими ресурсами. Саме цей підхід зробив її настільки привабливою для застосувань у реальному часі, включно з розпізнаванням автомобілів у потоковому відео.

З технічної точки зору важливою перевагою YOLO є її одноступенева архітектура. Вона складається з backbone, який відповідає за виділення ознак, neck, що комбінує ознаки різних масштабів, та head, яка виконує остаточні передбачення координат і класів. Використання багатомасштабних карт ознак у пізніших версіях YOLO дозволило значно покращити точність детекції дрібних об’єктів. Наприклад, у міських сценах, де одночасно присутні автомобілі на передньому плані й віддалені транспортні засоби на горизонті, багаторівнева обробка ознак забезпечує виявлення і тих, і інших.

Ще однією ключовою характеристикою YOLO є її здатність до масштабування моделей. У версіях YOLOv5 та пізніших розробники запропонували кілька варіантів конфігурацій — від nano до xlarge, що дозволяє обирати баланс між швидкістю та точністю. Легкі моделі можуть працювати безпосередньо на смартфонах із використанням оптимізованих бібліотек

(наприклад, Core ML на iOS або TensorFlow Lite), тоді як більші моделі застосовуються на серверному обладнанні для досягнення максимальної точності.

Особливої уваги заслуговують і алгоритмічні вдосконалення YOLO. У пізніх версіях було впроваджено вдосконалені функції втрат (наприклад, CIOU Loss), нові схеми аугментацій даних (Mosaic, MixUp), а також оптимізації для кращої генералізації. Це дозволило підвищити стійкість моделей до варіацій у даних і покращити узагальнення на нових сценах. У контексті розпізнавання автомобілів це критично важливо, адже транспортні засоби можуть виглядати по-різному залежно від кута зйомки, погодних умов або рівня освітлення. У таблиці 3 описано еволюція моделі YOLO з першої до восьмої версії.

Таблиця 3 — Еволюція YOLO(v1-v8)

Версія	**Рік**	**Особливості**	**Сильні сторони**
YOLOv1	2016	Перша реалізація; поділ зображення на сітку та передбачення рамок і класів одночасно	Надзвичайна швидкість для свого часу, реальний час на GPU
YOLOv2 (YOLO9000)	2017	Підтримка 9000+ класів, використання anchor boxes	Краща точність, масштабованість
YOLOv3	2018	Використання Darknet-53 як backbone, багатомасштабні карти ознак	Помітне покращення якості на дрібних об’єктах

YOLOv4	2020	Впровадження сучасних трюків (Mosaic, Mish, CIOU Loss)	Висока точність при збереженні швидкодії
YOLOv5	2020	PyTorch-реалізація, зручність експорту, моделі різного розміру	Широке використання у промисловості, легка інтеграція
YOLOv6	2022	Оптимізація для edge-пристроїв, підвищена ефективність	Краще співвідношення швидкість/точність
YOLOv7	2022	Нові архітектурні оптимізації, покращена точність	Стабільна робота у складних сценаріях
YOLOv8	2023	Модульна архітектура, підтримка класифікації, сегментації та детекції	Універсальність, сучасний стандарт у реальному часі

Водночас YOLO має і певні обмеження. Її головною проблемою залишається точність на дуже дрібних або частково перекритих об’єктах. Це зумовлено тим, що кожна клітинка сітки може передбачати лише обмежену кількість рамок, і в складних сценах частина дрібних об’єктів може залишитися непоміченою. Для розпізнавання автомобілів у міських умовах це може означати пропуски далеких чи перекритих транспортних засобів. Попри це,

завдяки своїй швидкодії, простоті впровадження та зрілій екосистемі інструментів YOLO залишається одним із найкращих виборів для мобільних і реальних застосувань**.**

2.1.3 MobileNet та інші легковагові архітектури

Потреба у легковагових архітектурах виникла разом зі зростанням мобільних технологій та переходом від суто серверних рішень до on-device інтелекту. Якщо перші хвилі глибинного навчання показали надзвичайні результати у класифікації й детекції на GPU-кластерах, то на практиці з'ясувалося, що інтегрувати ці алгоритми в смартфони чи IoT-пристрої дуже складно. Мобільні пристрої мають обмежену оперативну пам'ять, значно нижчу пропускну здатність обчислень і жорсткі вимоги до енергоспоживання. Тому важкі моделі, на кшталт VGG-16 чи ResNet-152, хоч і дають високу точність, але не можуть забезпечити стабільні 20–30 кадрів за секунду на мобільному CPU або навіть GPU. Це стимулювало появу нового напряму — легковагових архітектур, орієнтованих саме на швидкість і компактність при прийнятній точності.

Найвідомішим прикладом стала серія MobileNet, створена командою Google у 2017 році. Її ключова інновація — використання depthwise separable convolutions, що дозволило радикально скоротити кількість параметрів і обчислювальних операцій. У класичній згортці для кожного вихідного каналу використовуються всі канали вхідного шару, що потребує значних ресурсів. У MobileNet операцію поділено на два кроки: depthwise convolution обробляє кожен канал окремо, а pointwise convolution (звичайна 1×1 згортка) комбінує результати. Такий підхід зменшив обчислювальну складність майже в десять разів порівняно зі стандартними згортками, при цьому точність залишилася на

конкурентному рівні. MobileNetV1 одразу довів, що модель розміром у кілька мегабайт може досягати результатів, близьких до ResNet, і працювати у реальному часі навіть на мобільних пристроях середнього класу.

У MobileNetV2 було впроваджено ще одну важливу концепцію — інверсні резидуальні блоки (inverted residuals) з вузькими бутилочними шарами. Ідея полягає у тому, що зменшення кількості каналів здійснюється всередині блоку, а не ззовні, що робить мережу більш гнучкою і ще компактнішою. Така архітектура дозволила ще більше знизити латентність і стала справжнім проривом для мобільних сценаріїв. MobileNetV2 стала золотим стандартом для задач, де точність не повинна поступатися занадто сильно, але швидкодія є критичною.

MobileNetV3, представлена у 2019 році, поєднала depthwise-згортки з новими методами автоматичного пошуку архітектури (Neural Architecture Search, NAS). Вона інтегрувала також SE-блоки (Squeeze-and-Excitation), які дозволяють моделі адаптивно підсилювати найбільш інформативні канали ознак. Завдяки цьому MobileNetV3 забезпечує оптимальний баланс між точністю та продуктивністю і є на сьогодні одним із найефективніших рішень для мобільних застосунків, включаючи класифікацію автомобілів. На практиці MobileNetV3-Large може досягати точності понад 75% top-1 на ImageNet при розмірі моделі менше 20 MB і забезпечувати стабільну швидкодію на смартфонах.

Поза сімейством MobileNet також існує кілька архітектур, які стали важливими віхами у розвитку легковагових моделей.

SqueezeNet (2016) показав, що нейронна мережа з менше ніж 5 MB параметрів може досягати результатів, близьких до AlexNet. Архітектура базується на так званих fire modules, які складаються з squeeze-шару (зменшення кількості каналів) та expand-шару (розширення через згортки 1×1 і 3×3). Цей

підхід дозволив суттєво зменшити розмір моделі без різкої втрати точності, що було надзвичайно важливим для вбудованих систем.

ShuffleNet, у свою чергу, запропонував операцію channel shuffle, яка дозволяє ефективно змішувати інформацію між групами каналів у глибоко роздільних згортках. Це зробило можливим ще більше скорочення обчислювальної вартості при збереженні точності. ShuffleNet став особливо популярним у застосуваннях, де критичним є виконання інференсу на процесорах із дуже низьким енергоспоживанням, зокрема у пристроях edge.

Важливим напрямом стала також поява EfficientNet-Lite — полегшеної версії EfficientNet, спеціально адаптованої для TensorFlow Lite та мобільних сценаріїв. Вона дозволяє використовувати переваги збалансованого масштабування EfficientNet у контексті мобільних обмежень. У таблиці 4 наведено порівняння легковагових архітектур.

Таблиця 4 — Порівняльна таблиця легковагових архітектур

Архітектура	**Рік**	**Ключова інновація**	**Сильні сторони**	**Обмеження**
MobileNetV1	2017	Depthwise separable convolutions	Значне скорочення параметрів, робота у реальному часі на мобільних	Менша точність у порівнянні з важкими CNN
MobileNetV2	2018	Inverted residuals, linear bottlenecks	Ще менша латентність, збереження якості	Складність налаштування та оптимізації

MobileNetV3	2019	Neural Architecture Search, SE-блоки	Оптимальний баланс точність/швидкодія, найпрактичніша версія для мобільних	Нижча точність, ніж у EfficientNet великих версій
SqueezeNet	2016	Fire modules (squeeze + expand)	Дуже малий розмір моделі (<5 MB), придатна для вбудованих систем	Обмежена точність, застаріла у порівнянні з MobileNet
ShuffleNet	2017	Channel shuffle	Ефективне використання групових згорток, висока швидкість на слабких CPU	Складніша архітектура, нижча точність на великих наборах
EfficientNet-Lite	2020	Спрощене compound scaling для мобільних	Краща точність при компактності, інтеграція з TFLite	Потребує ретельного налаштування, важча за MobileNet

У задачі розпізнавання автомобілів легковагові архітектури часто поєднуються з більш важкими детекторами. Типова схема виглядає так:

спочатку YOLO чи інший одноетапний детектор знаходить автомобіль у кадрі, а потім MobileNet або ShuffleNet використовується як класифікатор для уточнення марки, моделі чи навіть року випуску. Це дозволяє зберігати високий FPS у мобільному додатку й одночасно досягати деталізованої ідентифікації. Такий підхід добре масштабується і дозволяє створювати гібридні системи, де важкі моделі працюють на сервері, а легковагові — локально на пристрої користувача.

Таким чином, MobileNet та інші легковагові архітектури стали ключовим інструментом для перенесення досягнень глибинного навчання з лабораторних середовищ у повсякденні мобільні застосунки. Вони доводять, що навіть при суттєвому скороченні обчислювальних ресурсів можливо забезпечити якісне розпізнавання складних об’єктів, таких як автомобілі, у реальному часі. На рисунку 3 зображено графік порівняння легковагових архітектур за розміром моделі та точністю (Top-1 на ImageNet).

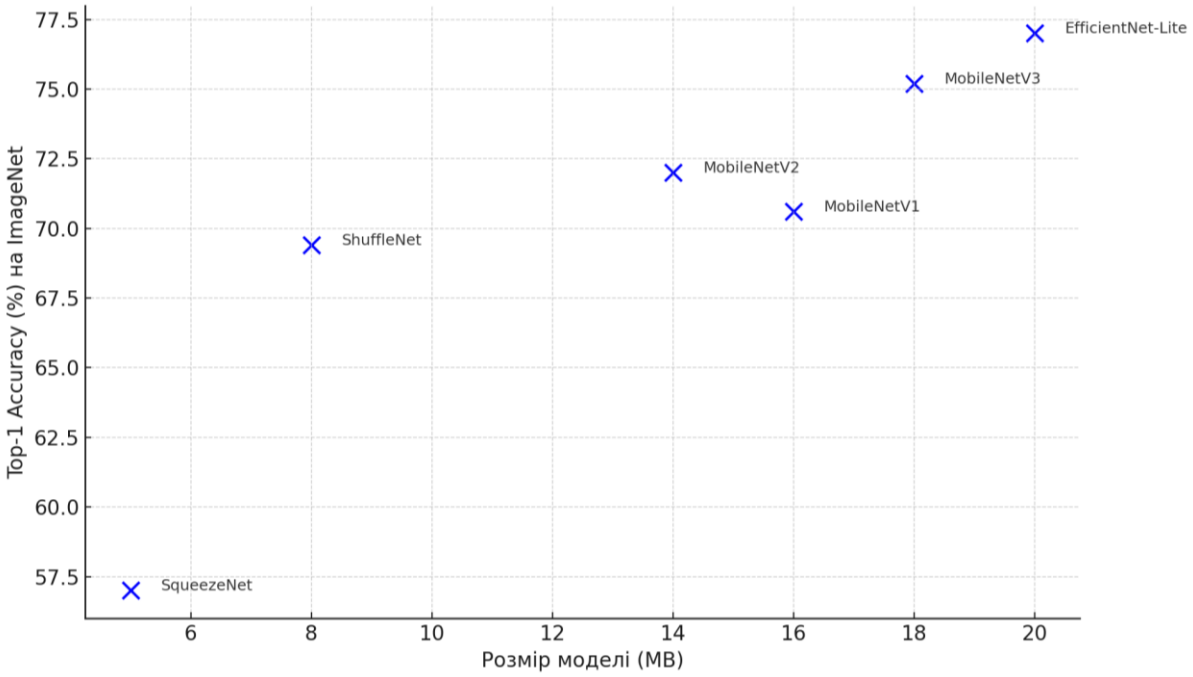


Рисунок 3 — Порівняння легковагових архітектур: розмір vs точність

2.2 Фреймворки та бібліотеки для глибинного навчання (PyTorch, TensorFlow, Keras, Core ML)

2.2.1 Фреймворки для глибинного навчання

Сучасні системи розпізнавання об’єктів, зокрема транспортних засобів, неможливо уявити без використання спеціалізованих середовищ для тренування нейронних мереж. Такі фреймворки забезпечують дослідників і розробників інструментами для побудови архітектур, організації процесу навчання на великих обсягах даних та оптимізації моделей для подальшого застосування. Серед найважливіших і найпоширеніших серверних фреймворків сьогодні варто виокремити PyTorch, TensorFlow та Keras.

PyTorch

PyTorch, розроблений лабораторією Facebook AI Research, став провідним фреймворком у наукових дослідженнях та академічному середовищі. Його головною особливістю є динамічний обчислювальний граф. Це означає, що структура мережі може змінюватися на льоту під час виконання, що дає дослідникам унікальну гнучкість при експериментах із новими архітектурами. На практиці це дозволяє, наприклад, швидко додати новий блок у CNN для тестування або змінити функцію втрат у процесі навчання без необхідності перекомпілювати модель.

Ще однією важливою перевагою PyTorch є простота його використання. Синтаксис PyTorch значно ближчий до звичайного Python-коду, ніж у конкурентів, що робить його доступнішим для розробників. Саме завдяки цьому PyTorch став фреймворком за замовчуванням у багатьох відкритих дослідницьких проєктах, включаючи популярні бібліотеки з реалізаціями YOLO, DETR, EfficientNet та MobileNet.

У задачах розпізнавання автомобілів PyTorch дозволяє ефективно навчати

моделі на таких наборах даних, як Stanford Cars, та виконувати експерименти з трансферним навчанням. Наприклад, використовуючи попередньо натреновану модель ResNet-50 на ImageNet, можна швидко донавчити її на більш вузькому датасеті з автомобілями, щоб класифікатор відрізняв марки і моделі машин. PyTorch підтримує GPU-обчислення з коробки, що дозволяє масштабувати тренування на кластери з декількома графічними процесорами, а також інтегруватися з TPU через додаткові бібліотеки.

TensorFlow

TensorFlow, створений Google, є другим основним гравцем у сфері глибинного навчання. Якщо PyTorch завоював академічну аудиторію, то TensorFlow традиційно більше використовувався у промисловості. Його головна відмінність історично полягала у використанні статичного обчислювального графа. На відміну від динамічного графа PyTorch, тут усі операції визначаються заздалегідь, що дозволяє ефективніше оптимізувати виконання моделі та масштабувати її для розгортання у виробничих середовищах.

TensorFlow вирізняється багатою екосистемою. TensorBoard забезпечує візуалізацію процесу навчання, дозволяючи відслідковувати точність, функцію втрат та інші метрики у реальному часі. TensorFlow Serving забезпечує зручний інструмент для продакшн-деплою моделей на сервері, а TensorFlow Lite — оптимізацію моделей для мобільних пристроїв. Усе це робить TensorFlow універсальним середовищем, яке охоплює весь життєвий цикл моделі — від розробки до розгортання.

У задачах класифікації та детекції автомобілів TensorFlow часто використовується у комерційних рішеннях, де важлива стабільність і масштабованість. Наприклад, великі корпорації застосовують його для побудови систем моніторингу дорожнього руху чи аналізу відеопотоку з камер спостереження. Його здатність працювати з великими обсягами даних та

розподіленими обчисленнями робить TensorFlow надзвичайно потужним інструментом для таких сценаріїв.

Keras

Keras був створений як високорівневий API для TensorFlow, орієнтований на простоту використання. Якщо PyTorch і TensorFlow пропонують гнучкість та контроль над внутрішніми механізмами навчання, то Keras дозволяє створити базову модель у кілька рядків коду. Це робить його ідеальним інструментом для швидкого прототипування та навчання студентів чи початківців у сфері глибинного навчання.

Keras має інтуїтивно зрозумілий синтаксис, завдяки чому легко побудувати модель CNN для класифікації автомобілів. Наприклад, можна швидко скласти послідовність шарів — згорткових, pooling, повнозв’язних — і отримати робочий прототип, який згодом можна оптимізувати у TensorFlow. Сьогодні Keras інтегрований у TensorFlow як офіційний високорівневий API, але зберігає значення як окремий стиль роботи з моделями.

Порівняння підходів

Таким чином, усі три фреймворки — PyTorch, TensorFlow та Keras — виконують подібні завдання, але відрізняються орієнтацією та сферою використання. PyTorch — це інструмент дослідників, що надає максимальну свободу для експериментів. TensorFlow — це промисловий стандарт, який забезпечує масштабованість та розвинену інфраструктуру для деплою. Keras — це простота та швидке прототипування.

У практичній задачі розпізнавання автомобілів вибір фреймворку залежить від цілей. Для наукової роботи та експериментів з архітектурами доречніше PyTorch. Для створення масштабованої системи моніторингу транспорту з серверною обробкою — TensorFlow. Для освітніх проєктів чи швидкої побудови базового прототипу — Keras. Усі три середовища

забезпечують повноцінні можливості для тренування моделей, які потім можуть бути перенесені на мобільні пристрої за допомогою спеціалізованих інструментів.

Таблиця 5 — Порівняльна таблиця фреймворків

Фреймворк	**Основні особливості**	**Переваги**	**Обмеження**	**Типові сценарії використання**
PyTorch	Динамічний обчислювальний граф, гнучкий API	Легкість експериментів, зручний синтаксис, популярність у дослідженнях, активна спільнота	Менш оптимізований для масштабного продакшн-деплою	Академічні дослідження, експерименти з архітектурами, швидке тестування нових ідей
TensorFlow	Статичні графи, широка екосистема	Висока масштабованість, багатий набір інструментів (TensorBoard, Serving, Lite),	Складніший у використанні, важчий для відладки	Великі проєкти, продакшн-рішення, комерційні системи моніторингу та аналітики

		промисловий стандарт		
Keras	Високорівневий API поверх TensorFlow	Простота синтаксису, швидке прототипування, доступність для новачків	Обмежена гнучкість, залежність від TensorFlow	Освітні проєкти, навчання студентів, швидке створення базових прототипів

2.2.2 Мобільні фреймворки для виконання інференсу

Тренування нейронних мереж зазвичай відбувається у серверному середовищі з використанням GPU або TPU, проте практичне застосування часто потребує виконання інференсу безпосередньо на кінцевому пристрої. Для мобільних додатків це має вирішальне значення:

- зменшується затримка, оскільки не потрібно відправляти дані на сервер;
- зростає рівень приватності користувача, адже обробка відбувається локально;
- знижується залежність від мережевого підключення;

Для реалізації такого підходу створено спеціалізовані фреймворки: TensorFlow Lite, ONNX Runtime та Core ML.

TensorFlow Lite

TensorFlow Lite (TFLite) — це полегшена версія TensorFlow, оптимізована

для мобільних та вбудованих пристроїв. Вона дозволяє конвертувати моделі у компактний формат .tflite, який займає набагато менше пам’яті й може виконуватися швидше. TFLite підтримує методи квантизації (quantization), зокрема перехід від 32-бітної точності до 16-бітної або навіть 8-бітної, що різко зменшує обчислювальні витрати. Додатково доступні техніки прунінгу (pruning) та weight clustering, які дозволяють ще більше зменшити розмір моделі.

TFLite активно використовується у світі Android, де він інтегрований у системні бібліотеки. Завдяки цьому на більшості сучасних смартфонів можливо виконувати інференс на CPU, GPU або спеціалізованих прискорювачах (наприклад, DSP). Проте в екосистемі iOS його використання є менш природним, оскільки Apple пропонує власний оптимізований фреймворк Core ML.

ONNX Runtime

ONNX (Open Neural Network Exchange) було створено як універсальний формат для обміну моделями між різними фреймворками — PyTorch, TensorFlow, Keras та іншими. Його головна ідея полягає у стандартизації збереження моделей, щоб їх можна було переносити між середовищами без переписування архітектури.

ONNX Runtime — це кросплатформна бібліотека для виконання інференсу моделей у форматі ONNX. Вона підтримує CPU, GPU, а також низку спеціалізованих прискорювачів. Для мобільних сценаріїв ONNX Runtime зручний тим, що дозволяє використовувати одну й ту ж модель на різних пристроях і операційних системах. Це робить його корисним у випадках, коли потрібно забезпечити роботу системи як на Android, так і на iOS, не створюючи окремих пайплайнів для кожної платформи.

Водночас ONNX Runtime є додатковим шаром абстракції, що іноді ускладнює оптимізацію під конкретні апаратні можливості. У випадку iOS він

рідше використовується напряму, поступаючись місцем Core ML, однак залишається зручним проміжним форматом для конвертації моделей.

Core ML

Core ML — це власна технологія Apple, розроблена спеціально для інтеграції моделей машинного навчання у додатки для iOS та macOS. Формат .mlmodel оптимізований під апаратне забезпечення компанії, зокрема під Apple Neural Engine (ANE), який входить до складу сучасних чипів серій A і M. Завдяки цьому моделі у форматі Core ML працюють з максимально можливою швидкістю та енергоефективністю.

Важливою перевагою Core ML є підтримка автоматичної конвертації з найпопулярніших фреймворків. Наприклад, моделі з PyTorch або TensorFlow можна перенести у Core ML за допомогою бібліотеки coremltools. Цей процес також дозволяє застосовувати оптимізації: квантизацію ваг до 16- чи 8-біт, зменшення точності проміжних обчислень, злиття шарів. У результаті модель стає більш компактною та швидшою без помітної втрати якості.

У контексті розпізнавання автомобілів Core ML є ключовим інструментом, оскільки саме він дозволяє запускати детектори (YOLO, MobileNet, EfficientNet) у реальному часі безпосередньо на iPhone. Наприклад, на сучасних моделях iPhone YOLOv5n у форматі Core ML здатен забезпечити понад 30 кадрів за секунду з точністю, достатньою для стабільної ідентифікації автомобілів у відеопотоці. Крім того, локальна обробка гарантує збереження конфіденційності даних користувача, що є важливою перевагою перед серверними системами.

Таблиця 6 — Порівняльна таблиця мобільних фреймворків

Фреймворк	**Основні особливості**	**Переваги**	**Обмеження**
TensorFlow Lite	Полегшена версія TensorFlow, формат .tflite	Квантизація та прунінг, невеликий розмір	Менш природна інтеграція з iOS

		моделей, інтеграція з Android	
ONNX Runtime	Виконання моделей у форматі ONNX, кросплатформність	Універсальний формат, підтримка різних фреймворків і пристроїв	Додатковий шар абстракції, складніша оптимізація
Core ML	Власна технологія Apple, формат .mlmodel	Максимальна швидкодія на iOS, підтримка Neural Engine, приватність даних	Обмежена екосистема (лише Apple)

Висновки до підпункту 2.2

Аналіз фреймворків для глибинного навчання показує, що розробка сучасних систем розпізнавання автомобілів у реальному часі неможлива без чіткої взаємодії між серверними та мобільними інструментами. Серверні середовища, такі як PyTorch, TensorFlow та Keras, забезпечують необхідні можливості для навчання моделей на великих датасетах, дозволяючи експериментувати з архітектурами, налаштовувати оптимізаційні алгоритми та досягати максимальної точності. Саме на цьому етапі формується якість майбутньої системи.

Мобільні фреймворки — TensorFlow Lite, ONNX Runtime та Core ML — вирішують інше, не менш важливе завдання: адаптацію та виконання моделей безпосередньо на кінцевих пристроях. Вони роблять можливим інференс у реальному часі, мінімізуючи затримки, зменшуючи залежність від серверів і підвищуючи рівень приватності користувача. Особливо важливим для iOS-застосунків є Core ML, який забезпечує найглибшу інтеграцію в екосистему Apple та оптимізацію під апаратні прискорювачі, що робить його незамінним у мобільних додатках для розпізнавання автомобілів.

Таким чином, ефективна система повинна поєднувати обидва рівні: серверний етап навчання та мобільний етап інференсу. Саме ця синергія

дозволяє реалізувати повноцінні рішення, що поєднують високу точність і швидкодію з практичною можливістю використання у реальних умовах.

2.3 Використані датасети

Датасети відіграють ключову роль у побудові систем глибинного навчання, оскільки саме вони визначають межі знань і узагальнювальну здатність моделі. У задачі розпізнавання автомобілів якість і різноманітність даних є критично важливими: система має справлятися з різними марками, моделями, роками випуску транспортних засобів, а також працювати за різних умов освітлення, у різних ракурсах і навіть тоді, коли автомобіль частково закритий іншими об’єктами. Тому вибір датасету та спосіб його формування безпосередньо впливають на кінцеві результати.

Одним із базових джерел для досліджень у цій сфері став датасет Stanford Cars. Він містить понад 16 тисяч зображень, що охоплюють більше ніж 190 класів автомобілів, включно з марками, моделями та роками випуску. Цей набір даних є еталонним для завдань тонкої класифікації (fine-grained classification), оскільки він не обмежується широкими категоріями, як-от седан чи SUV, а змушує модель розрізняти дуже схожі варіанти — наприклад, BMW 3 Series 2012 та BMW 3 Series 2015. Завдяки цьому Stanford Cars дозволяє перевіряти здатність алгоритмів вловлювати тонкі відмінності у зовнішньому вигляді автомобілів.

Попри очевидні переваги, використання лише стандартного набору не завжди забезпечує повну відповідність реальним умовам. Зображення у Stanford Cars здебільшого зібрані з інтернету й мають студійний або рекламний характер: вони добре освітлені, мають чистий фон і високу якість. Це може призводити до доменного зсуву при перенесенні моделі у практичне застосування, коли зйомка відбувається на мобільний пристрій у природних умовах або коли

розпізнаються не реальні автомобілі, а їхні зменшені копії. Саме тому у цій роботі поєднано Stanford Cars із власним датасетом колекційних моделей автомобілів. Це дозволило доповнити академічно чисті дані зразками, наближеними до цільового сценарію — зйомки моделей з різних кутів, на різних фонах, у різних умовах освітлення.

Поєднання двох джерел дало змогу компенсувати слабкі місця кожного з них. Stanford Cars забезпечив великий обсяг прикладів і збалансовану структуру класів, тоді як власний набір додав різноманіття, необхідне для реальної експлуатації. Особливо важливим стало те, що власні дані охоплювали ті класи й умови, яких бракувало у публічному датасеті. Це стосувалося як ракурсів і варіацій освітлення, так і відмінностей між справжніми автомобілями та їхніми мініатюрними копіями. Завдяки цьому вдалося зменшити доменний розрив і навчити модель більш стійко працювати на зображеннях, які максимально близькі до майбутнього практичного використання.

Ще одним критично важливим аспектом стало питання анотації. У Stanford Cars дані представлені у вигляді image-level міток, де кожне фото має позначення відповідного класу. Це спрощує задачу класифікації, проте обмежує застосування у випадках, коли на зображенні присутні кілька автомобілів. У власному датасеті було використано bounding boxes, що дозволило локалізувати автомобіль навіть тоді, коли він знаходився поруч з іншими об’єктами. Такий підхід зробив дані універсальнішими й відкрив можливість інтеграції моделей детекції.

Таблиця 7 демонструє відмінності між основними форматами анотацій, що використовуються у завданнях розпізнавання автомобілів.

Таблиця 7 — Порівняльна таблиця мобільних фреймворків

Формат анотації	**Опис**	**Переваги**	**Обмеження**

Image-level labels	Мітка для всього зображення	Простота, швидке збирання	Не підходить для кількох об’єктів
Bounding boxes	Прямокутник навколо об’єкта	Локалізація, мультиоб’єктні сцени	Розмітка трудомістка
Segmentation masks	Піксельна маска об’єкта	Висока точність локалізації	Дуже висока складність розмітки

Вибір bounding boxes для власних даних і image-level labels для Stanford Cars забезпечив баланс між зручністю використання готового датасету та точністю, необхідною для практичного застосування.

Не менш важливим є питання балансування класів. Stanford Cars хоча й має велику кількість класів, але вони не завжди однаково представлені. Деякі моделі мають сотні прикладів, інші — лише кілька десятків. Додавання власного датасету дозволило покращити представлення малопоширених класів і створити більш збалансовану вибірку. Це стало важливим чинником для підвищення стабільності моделі під час класифікації рідкісних автомобілів.

Отже, використання комбінації Stanford Cars і власного датасету колекційних моделей є оптимальним рішенням. Перший забезпечує стандарт і високу якість анотацій, другий — адаптацію до реальних умов і зменшення доменного розриву. Разом вони створюють навчальну базу, яка не лише дозволяє досягти високої точності на тестових наборах, але й гарантує стійкість моделі у практичних сценаріях — від розпізнавання справжніх автомобілів на дорозі до класифікації зменшених колекційних моделей у реальному часі.

2.4 Попередня обробка даних(розмітка, нормалізація, аугментація)

Якість і різноманітність даних самі по собі ще не гарантують ефективного навчання нейронної мережі. Для того щоб модель змогла узагальнювати,

працювати стабільно й уникати перенавчання, необхідно виконати низку кроків попередньої обробки даних. Цей процес включає правильну розмітку, стандартизацію зображень та використання методів аугментації, які дозволяють штучно розширити простір можливих сценаріїв.

Розмітка

Розмітка даних — це фундаментальний етап, від якого залежить коректність усієї подальшої системи. У цьому випадку робота шла з двома джерелами: датасетом Stanford Cars, який вже містив готові мітки на рівні зображення (image-level labels), та власним датасетом колекційних моделей, де було необхідно створити анотації самостійно. Для цього використовували bounding boxes, які дозволяють не лише визначити клас автомобіля, а й локалізувати його у кадрі.

Правильність і точність розмітки є критичними. Навіть невеликі систематичні помилки (наприклад, неправильне позиціонування рамки або помилкове віднесення автомобіля до іншого класу) можуть суттєво погіршити результати. Тому процес анотації має супроводжуватися перевіркою якості — подвійною валідацією або семіавтоматичним підходом, коли базову розмітку пропонує попередньо навчена модель, а оператор лише коригує. На рисунку 4 зображено правильна і точна розмітка.

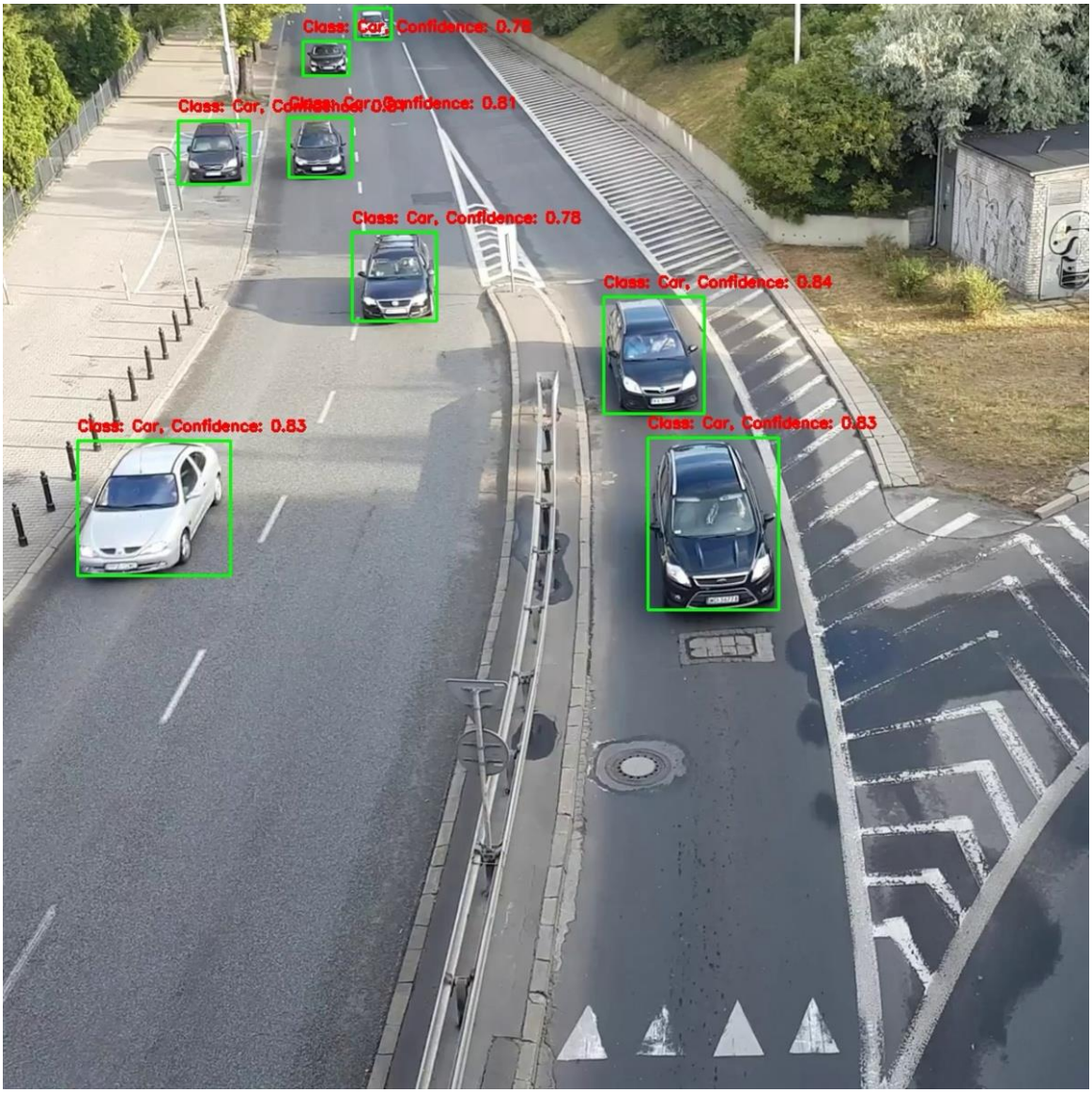


Рисунок 4 — Приклад правильної і точної розмітки

Нормалізація

Наступним важливим кроком є нормалізація зображень. Вона включає приведення усіх прикладів до єдиного масштабу (наприклад, 224×224 або 416×416 пікселів залежно від архітектури), а також перетворення піксельних значень у нормалізований діапазон. Як правило, значення інтенсивності пікселів переводяться у діапазон [0, 1] або [-1, 1], що забезпечує стабільність навчання й коректну роботу оптимізаторів. Крім того, використовуються стандартизовані

середні й дисперсії каналів RGB, узяті з великих датасетів, як-от ImageNet.

У задачі розпізнавання автомобілів нормалізація також допомагає компенсувати різні умови зйомки: зображення можуть бути отримані на різних пристроях із різною експозицією, контрастністю чи освітленням, продемонстровано на рисунку 5. Приведення їх до спільного формату зменшує варіативність, не пов’язану з самим об’єктом, і дозволяє моделі краще навчатися на суттєвих ознаках.



Рисунок 5 — Приклад правильної і точної розмітки у різних умовах зйомки та ракурсах.

Аугментація

Оскільки навіть великі датасети мають обмежену кількість прикладів, застосовують методи аугментації — штучного розширення набору даних шляхом створення нових зображень на основі існуючих. Це дозволяє підвищити стійкість моделі до варіацій у реальному світі.

Для автомобілів особливо ефективними виявляються такі техніки:

- Геометричні перетворення: випадкові обертання, віддзеркалення,

зсуви чи масштабування. Вони моделюють ситуації, коли автомобіль знімається з різних кутів чи відстаней.

- Фотометричні модифікації: зміна яскравості, контрасту, кольорової насиченості. Це допомагає адаптувати модель до різних умов освітлення — сонячного дня, сутінків чи штучного світла.
- Шум і артефакти: додавання Gaussian noise, імітація розмиття, відблисків або тіней. Такі прийоми роблять модель менш чутливою до недоліків камер.
- Композиційні методи: Mosaic чи CutMix, коли на одному зображенні поєднуються кілька об’єктів. Цей підхід, популяризований у YOLOv4, дозволяє навчати модель виявляти автомобілі у складніших сценах.

На практиці аугментація дала змогу компенсувати обмежений обсяг власного датасету й зробити модель стійкішою до сценаріїв, які не були представлені у навчальній вибірці. Зокрема, це стосується рідкісних ракурсів колекційних моделей або варіацій освітлення, яких немає у Stanford Cars. Приклад продемонстровано на рисунку 6.

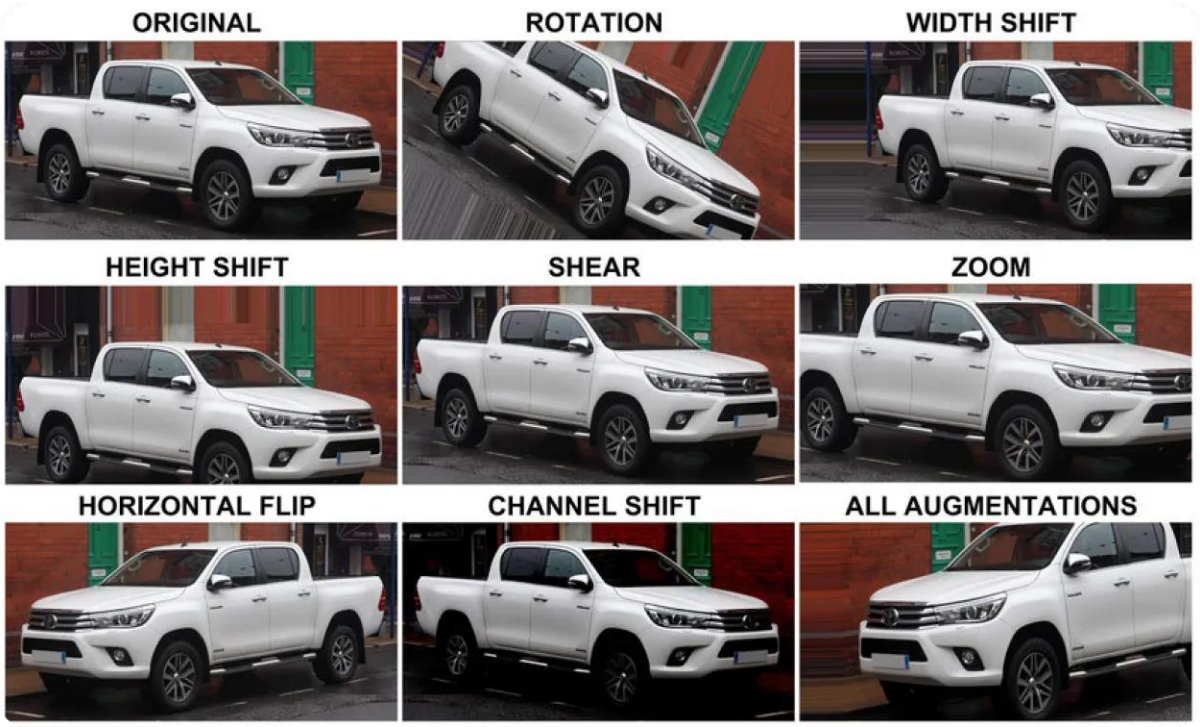


Рисунок 6 — Демонстрація методів аугментації.

Узагальнення

Попередня обробка даних — це не допоміжний, а стратегічний етап, що формує основу для успішного навчання. Розмітка визначає, які об’єкти будуть вважатися правильними, нормалізація забезпечує узгодженість вхідних даних, а аугментація створює додаткове різноманіття, якого неминуче бракує у будь-якому датасеті. У цьому випадку поєднання цих методів дозволило отримати навчальну вибірку, здатну забезпечити і точність на стандартизованих наборах, і стабільність у практичному застосуванні на мобільному пристрої.

2.5 Висновки до розділу 2

У другому розділі було розглянуто ключові компоненти, які формують основу системи розпізнавання автомобілів у реальному часі. Передусім було

проаналізовано архітектури нейронних мереж для класифікації та детекції об’єктів. Показано, що класичні згорткові мережі (CNN) залишаються базовим будівельним блоком, на якому ґрунтуються більш складні моделі. Одноетапні підходи, як-от YOLO, забезпечують баланс між швидкістю та точністю, що робить їх особливо цінними для практичних мобільних застосунків. Водночас легковагові архітектури на зразок MobileNet чи ShuffleNet доводять, що компактність і продуктивність можуть поєднуватися навіть у середовищах із жорсткими обмеженнями ресурсів.

Далі було розглянуто сучасні фреймворки та бібліотеки глибинного навчання. Серверні інструменти — PyTorch, TensorFlow і Keras — забезпечують повний цикл дослідження і тренування моделей, дозволяючи створювати складні архітектури та масштабувати обчислення. Мобільні фреймворки — TensorFlow Lite, ONNX Runtime і особливо Core ML — виконують завдання перенесення цих моделей на кінцеві пристрої, зберігаючи швидкодію та підвищуючи рівень приватності користувачів. Таким чином, саме поєднання серверних і мобільних технологій робить можливим розгортання інтелектуальних систем на смартфонах.

Особлива увага була приділена датасетам. Використання стандартного набору Stanford Cars дозволило отримати високу якість навчання та реплікабельність результатів, а додавання власного датасету колекційних моделей дало змогу зменшити доменний розрив і підвищити релевантність системи для практичного сценарію. Це підкреслює важливість комбінованого підходу: публічні дані забезпечують масштаб, тоді як власні — адаптацію до конкретних умов.

Не менш важливим етапом виявилась попередня обробка даних. Правильна розмітка, нормалізація та аугментація зображень дозволили підвищити якість і стійкість навчання. У результаті вдалося сформувати

навчальну вибірку, здатну забезпечити не лише точність на академічних тестах, але й стабільну роботу у реальних умовах зйомки.

Узагальнюючи, можна зробити висновок, що успіх системи розпізнавання автомобілів у реальному часі визначається не якимось одним компонентом, а комплексом рішень: від вибору архітектури нейронної мережі й фреймворку для її тренування до формування збалансованого датасету та застосування ефективних методів попередньої обробки. Синергія цих елементів створює підґрунтя для реалізації стабільного, точного й практично корисного застосунку.

РОЗДІЛ 3 РОЗРОБКА IOS-ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ У РЕАЛЬНОМУ ЧАСІ

3.1 Постановка задачі та вимоги до системи

Розвиток мобільних технологій та зростання обчислювальної потужності сучасних смартфонів відкрили нові можливості для інтеграції алгоритмів комп'ютерного зору безпосередньо у мобільні застосунки. Якщо ще кілька років тому завдання детекції та класифікації автомобілів вимагали використання потужних серверних графічних процесорів, то нині їх можливо реалізувати локально на пристроях користувачів. Такий підхід знижує залежність від хмарних обчислень, зменшує затримку відповіді та гарантує вищий рівень конфіденційності даних.

Метою цього проєкту є створення iOS-застосунку, здатного виконувати розпізнавання автомобілів у реальному часі. Під терміном розпізнавання розуміється як класифікація (ідентифікація марки та моделі автомобіля), так і базова локалізація об'єкта у кадрі. Особливістю проєкту є використання мобільно-орієнтованих технологій, що дозволяють виконувати всі обчислення безпосередньо на iPhone без звернення до серверної інфраструктури.

Формулювання задачі

Задачу можна формалізувати наступним чином:

- Вхідні дані — потік кадрів з камери смартфона у форматі RGB.
- Процес обробки — нормалізація зображень, інференс через попередньо натреновану нейронну мережу у форматі Core ML, інтерпретація вихідних тензорів (координати, класи, ймовірності).
- Вихідні дані — накладення результатів на кадр у вигляді рамок

(bounding boxes) і текстових підписів з ідентифікованим класом автомобіля та рівнем впевненості моделі.

Система повинна працювати в інтерактивному режимі, забезпечуючи плавний користувацький досвід і високу точність навіть у складних умовах — різне освітлення, відблиски, ракурси чи наявність кількох об’єктів у полі зору.

Власний пайплайн розробки моделі

Важливо підкреслити, що у межах роботи було реалізовано повний цикл розробки системи машинного навчання. На етапі дослідження й експериментів було використано середовище Python з фреймворком PyTorch для навчання архітектури YOLOv5, яка добре зарекомендувала себе у задачах детекції у реальному часі.

Навчання здійснювалось у два етапи:

1. Базове тренування на публічному датасеті Stanford Cars, який містить понад 16 тисяч зображень та охоплює 196 класів автомобілів. Це дозволило сформувати модель, здатну розпізнавати широкий спектр транспортних засобів.

2. Донавчання (fine-tuning) на власному датасеті колекційних моделей автомобілів, що дало змогу зменшити доменний розрив між академічними даними та практичним сценарієм використання. Завдяки цьому модель навчилася краще працювати на зображеннях, отриманих у реальних умовах зйомки, включно з фотографіями масштабних копій автомобілів.

Після навчання модель була експортована у формат ONNX, а далі конвертована у формат Core ML (.mlmodel) за допомогою бібліотеки coremltools. На цьому етапі було застосовано оптимізації: квантизацію ваг до 16-бітної точності, спрощення обчислювального графа та адаптацію під Apple Neural Engine (ANE). У результаті розмір моделі було зменшено, а продуктивність на пристроях iOS значно підвищена без відчутної втрати

точності.

Таким чином, створена система об’єднала у собі як дослідницький аспект (навчання й оптимізація моделі у Python), так і інженерний аспект (інтеграція в iOS-застосунок через Core ML).

Вимоги до системи

Щоб забезпечити працездатність застосунку в умовах реального використання, було сформульовано низку вимог:

1. Продуктивність у реальному часі. Швидкість обробки має становити не менше ніж 20–30 кадрів на секунду, що забезпечує плавність відеопотоку й миттєву реакцію на зміни у кадрі.

2. Точність класифікації. Модель повинна досягати top-1 точності на рівні 70–75% і вище, що дозволяє коректно відрізняти навіть близькі класи (наприклад, різні моделі одного виробника).

3. Адаптивність архітектури. Система має підтримувати використання різних варіантів моделей — від легких (YOLOv5n, MobileNet) до більш потужних (YOLOv5m, EfficientNet-Lite) залежно від апаратних ресурсів пристрою.

4. Приватність. Усі обчислення виконуються локально на пристрої. Це усуває необхідність передавати дані на сервер, що відповідає сучасним вимогам до захисту персональної інформації.

5. Енергоефективність. Модель повинна бути оптимізованою під Neural Engine, щоб зменшити споживання енергії та уникнути швидкого розрядження пристрою.

6. Інтерфейс користувача. Результати розпізнавання відображаються у наочній формі — рамки навколо об’єктів, текстові підписи з класом і відсотком впевненості. Інтерфейс має бути інтегрований у середовище SwiftUI, що гарантує сучасний і зручний дизайн.

7. Можливість налаштувань. Користувач повинен мати базові інструменти управління — наприклад, зміну порогу впевненості для відображення результатів чи перемикання між моделями різної складності.

8. Сумісність. Додаток має працювати на сучасних версіях iOS (починаючи з iOS 15) і підтримувати широкий спектр пристроїв Apple, з урахуванням різної продуктивності їхніх процесорів.

Таким чином, постановка задачі виходить за рамки простого розгортання готової моделі. Це комплексна проблема, яка включає підбір і навчання архітектури, оптимізацію під мобільні обмеження, забезпечення приватності користувача та інтеграцію з інтерфейсом iOS. У результаті система повинна забезпечувати баланс між трьома ключовими параметрами: точністю, швидкодією та зручністю використання. Саме на ці критерії орієнтувалася подальша розробка архітектури застосунку, яка буде розглянута у наступному підрозділі.

3.2. Архітектура застосунку (MVVM, SwiftUI, Core ML інтеграція)

3.2.1 Фреймворк Swift

Мова програмування Swift сформувала сучасну парадигму розробки для екосистеми Apple, у якій безпека типів, предиктивність продуктивності й ергономіка синтаксису є першорядними. Для застосунків, що інтегрують машинне навчання, це має практичний вимір: статично типізовані моделі даних дозволяють точно описати результат інференсу (рамки детекції, класи, ймовірності), зменшуючи ризик прихованих помилок на стику ML-пайплайн to UI. Додатково, сучасні можливості конкурентності (Swift Concurrency: async/await, Task, Actor) спрощують організацію робочих потоків: захоплення кадрів з камери, передобробка, виклики до Core ML/Vision та оновлення

інтерфейсу тепер виражаються декларативно й без небезпечних гонок за ресурсом головного потоку.

У цьому контексті доречно окреслити еволюцію UI-стека Apple від імперативної моделі UIKit до декларативної моделі SwiftUI. UIKit історично забезпечував повний контроль над життєвим циклом контролерів, ієрархією в'юшок та анімаціями; однак така свобода супроводжувалася значним клейовим кодом для синхронізації стану, вручну прописаною логікою оновлень і високою пов'язаністю між контролерами та моделлю. SwiftUI натомість пропонує уніфікований data-driven підхід: інтерфейс є функцією стану, а зміни стану автоматично відбиваються у візуалізації. Для задач реального часу (детекція у відеопотоці) це критично: модель інференсу оновлює лише опис стану (список детекцій), а фреймворк сам дбає про дифінг і перерисовку без ручного мікроменеджменту. Формально це знижує ентропію коду й дає змогу зосередитися на доменній логіці, а не на імперативних оновленнях в'юшок.

Попри переваги SwiftUI, практична інтеграція з камерою та ML-інференсом вимагає чітких платформних передумов. З погляду інструментарію, доцільно орієнтуватися на сучасний стек: Xcode останніх мажорних версій, Swift 5.7+ з підтримкою concurrency, цільова платформа iOS не нижче 15 (це забезпечує стабільні API для SwiftUI, розвинені можливості Vision/Core ML та достатню присутність Neural Engine у польових пристроях). Хоча Core ML існує ще з iOS 11, а SwiftUI — з iOS 13, практичний досвід демонструє, що iOS 15+ є розумним мінімумом, коли ми поєднуємо відеопотік, асинхронний інференс і реактивний UI без обхідних рішень. Вимоги з боку апаратного забезпечення не жорсткі, але націлені на реальний час: бажані чипи з Apple Neural Engine (ANE), які апаратно прискорюють оператори ML-графа; при цьому застосунок повинен коректно працювати і на більш скромних пристроях, переключаючись на CPU/GPU-бекенди без втрати стабільності. На рівні фреймворків ми

використовуємо AVFoundation для доступу до камери, Vision як високорівневий фасад до Core ML, а також механізми SwiftUI спостережуваного стану (@StateObject, @ObservedObject, @EnvironmentObject) для прозорого зв'язування результатів інференсу з відображенням.

3.2.2 Архітектура системи

Переходячи до архітектури застосунку, варто критично порівняти кілька поширених підходів у мобільній розробці. Класичний MVC (Model–View–Controller) добре відомий і простий у старті, але у сценаріях зі стрімким оновленням стану (відеопотік + ML) контролери схильні до надмірного росту відповідальностей, що породжує Massive View Controller і ускладнює тестування. MVP (Model–View–Presenter) відділяє презентер як шар перетворення даних для View, проте за відсутності вбудованої реактивності легко повертає нас до ручного менеджменту оновлень і залежностей, особливо коли подій багато і вони приходять з різних джерел (камера, таймери, ML-колбеки). Схему архітектурних патернів MVP і MVC наведено на рисунку 7.

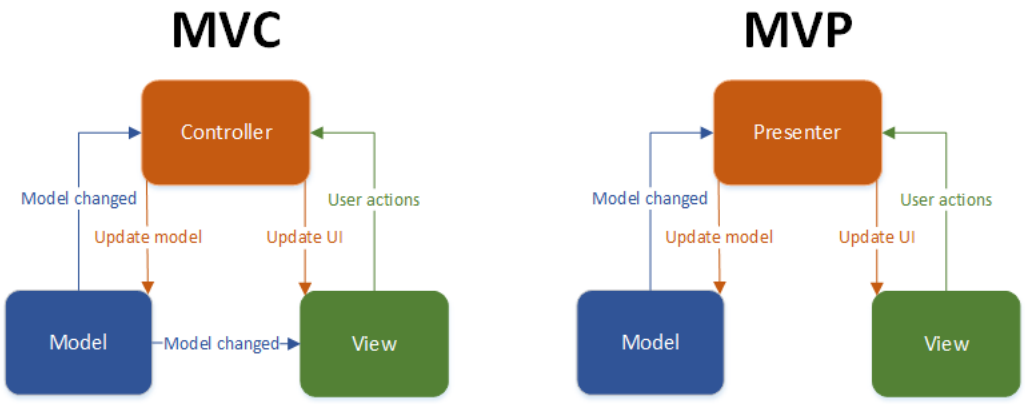


Рисунок 7 — Схема архітектурних патернів MVP і MVC.

VIPER і чисті підходи на кшталт Clean Architecture вибудовують сувору шарованість і явні протоколи взаємодії, що добре для великих команд і довгих життєвих циклів продукту; однак їхня інженерна вартість входу помітна, а обсяг інфраструктурного коду часто надлишковий для застосунку, де домінує реактивний UI-потік кадри, інференс, оверлей. Схема патерну можна переглянути на рисунку 8.

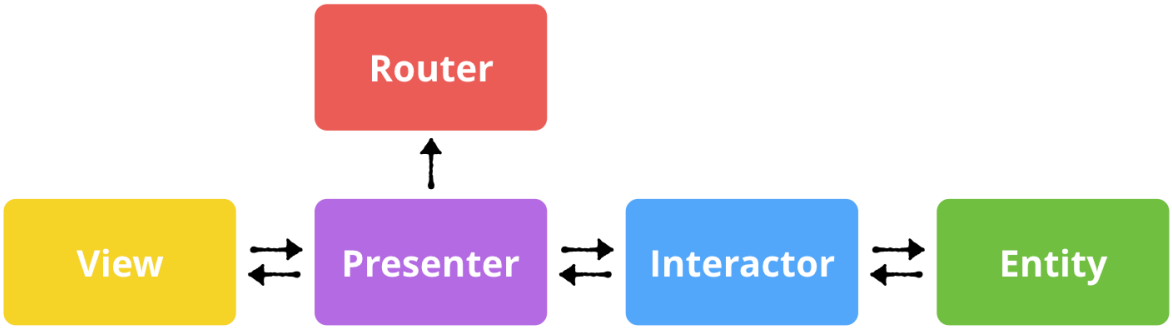


Рисунок 8 — Схема архітектурного патерну VIPER.

Нарешті, Redux-подібні моделі зі єдиним джерелом істини (уніфікований state + reducer) дають потужну передбачуваність, але можуть ускладнити роботу з високочастотними подіями та потребують ретельного тюнінгу продуктивності, аби не створити вузькі місця в головному циклі.

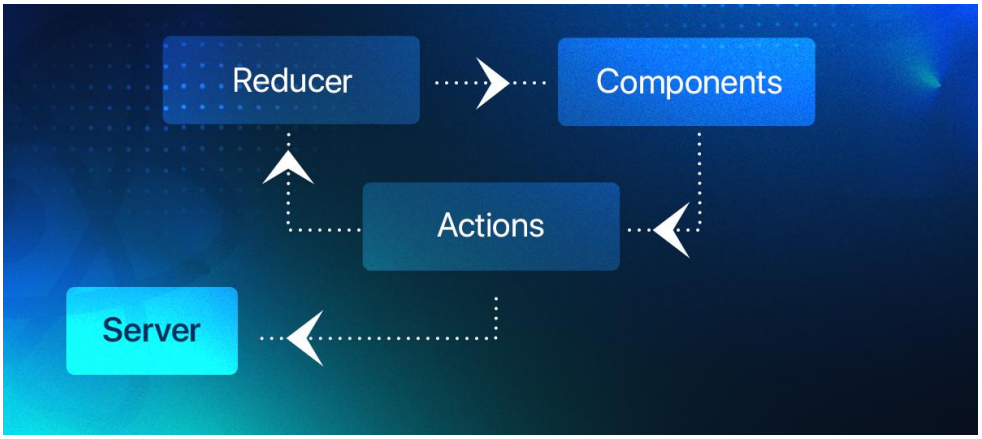


Рисунок 9 — Схема архітектурного патерну REDUX.

На тлі цього аналізу MVVM (Model–View–ViewModel) постає природним компромісом, особливо у поєднанні зі SwiftUI. Семантика MVVM накладається на декларативний рендеринг майже ізоморфно: Model інкапсулює ML-обчислення й доступ до даних (кадри камери, інференс через Vision/Core ML), ViewModel виконує роль чистого перетворювача — ізолює бізнес-логіку, синхронізує конкурентні події, мапує вихідні тензори моделі у зручні для UI структури (рамки у координатах екрана, підписи, відфільтровані за порогом впевненості), а View є чистою функцією стану, що реагує на зміни публічних властивостей ViewModel. Важливо, що ViewModel — це саме місце для безпечної взаємодії з асинхронністю: тут зручно використовувати Task для батч-інференсу, акторам (наприклад, DetectionActor) делегувати послідовну обробку кадрів, а через публікації стану (@Published) віддавати у в'ю лише готовий результат, не розкриваючи тонкощів потоків і синхронізації.

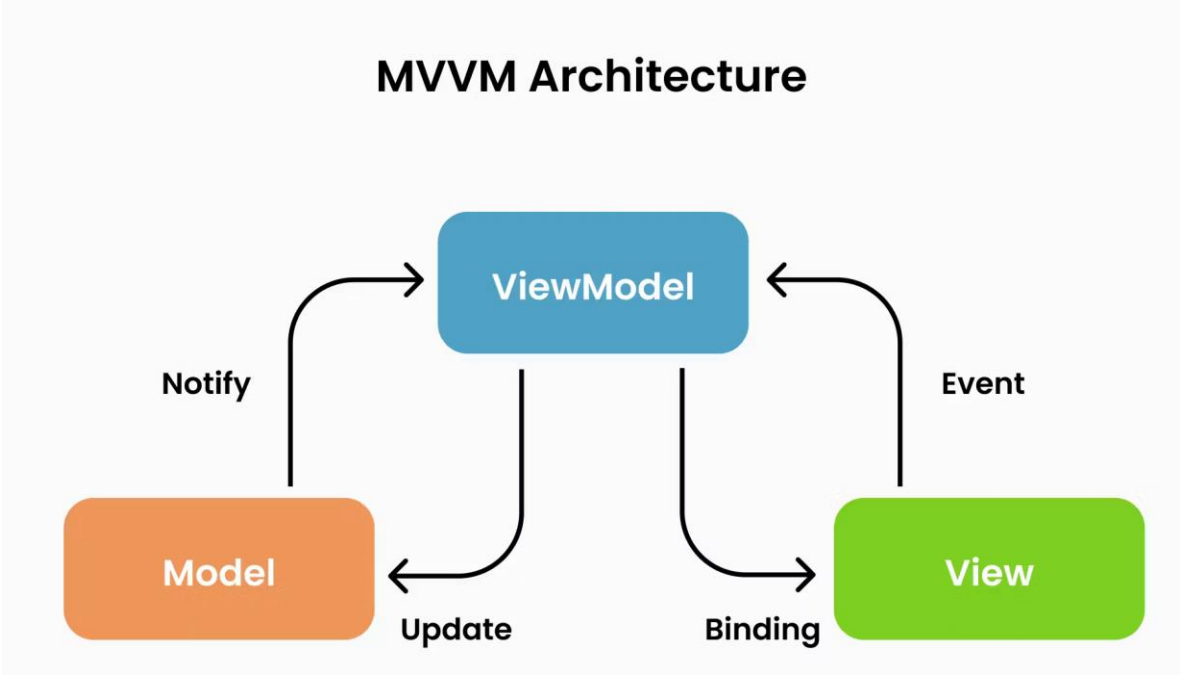


Рисунок 10 — Схема архітектурного патерну MVVM.

З погляду продуктивності й підтримуваності рішень у часі MVVM також підтверджує переваги. Оскільки відмальовування у SwiftUI відбувається диференційовано, ми спираємось на природний для фреймворка життєвий цикл перерисовок і уникаємо ручних анімацій/оновлень, що непрямо знижує overhead у головному потоці. При цьому кордони між шарами чіткі: заміна або оновлення Core ML-моделі (наприклад, перехід з YOLOv5n на YOLOv8n чи інший бекбон) торкається здебільшого Model і частково ViewModel (формат вихідних тензорів), але майже не зачіпає View. Така слабка зв’язаність спрощує ітерації, експерименти з архітектурами та подальшу оптимізацію.

Для узагальнення порівняння архітектур доречно подати зведену таблицю 8, яка відображає їхні властивості саме у реальночасовій задачі з ML-інференсом:

Таблиця 8 — Зведена таблиця порівняння архітектур

Підхід	**Переваги у наявному сценарії**	**Типові ризики/обмеження**
MVC	Низький поріг входу, добра інтеграція зі старим кодом	Massive VC, важко тримати чисті кордони відповідальностей при відеопотоці
MVP	Чіткіший розподіл між даними та відображенням	Імперативний менеджмент оновлень, більше клейового коду
VIPER / Clean	Висока модульність, тестованість великих систем	Значні накладні витрати, надлишкова складність для цієї задачі
Redux-подібні	Передбачуваність стану, відлагоджуваність	Налаштування продуктивності під high-FPS потоки, вербозність

MVVM	Природна реактивність, чисті кордони Model–ViewModel–View, легка заміна ML-моделі	Потреба дисципліни у потоках/стані, але добре вирішується через Concurrency/Actors

Підсумовуючи, саме MVVM у поєднанні зі SwiftUI та Core ML/Vision відповідає дослідницько-інженерній постановці: воно поєднує декларативний рендеринг, реактивну доставку результатів інференсу й модульність, необхідну для швидких ітерацій над моделлю та конфігураціями (пороги, NMS, розміри вхідного тензора). Обраний підхід дозволяє досягнути балансу між продуктивністю у реальному часі, стійкістю коду та чистотою архітектури, що є визначальним для мобільного застосунку з інтегрованим глибинним навчанням.

3.2.3 Вибір мобільного ML-фреймворку: Core ML проти альтернатив

Інтеграція моделі глибинного навчання у мобільний застосунок може бути реалізована різними способами. Для iOS доступні кілька варіантів:

- TensorFlow Lite — кросплатформний фреймворк, що дозволяє виконувати інференс моделей TensorFlow у форматі .tflite. Має широку підтримку оптимізацій (квантизація, прунінг), але його інтеграція у Swift вимагає використання додаткових бібліотек і шарів абстракції (C API, Swift обгортки). Це підвищує складність розробки та зменшує прозорість взаємодії з нативними API iOS.
- ONNX Runtime — універсальний інструмент для виконання моделей у форматі ONNX. Перевагою є можливість запускати одну й ту саму модель на різних платформах без зміни формату. Проте в iOS-контексті він працює через зовнішню бібліотеку, що створює додаткове навантаження на

розмір застосунку та ускладнює оптимізацію під конкретні апаратні можливості пристроїв Apple.

- Metal Performance Shaders (MPS) — низькорівневий API, який надає доступ до GPU для виконання обчислень. Це потужний, але складний у використанні варіант: розробник мусить самостійно описувати обчислювальний граф, керувати пам’яттю та оптимізувати конвеєр. Для академічних експериментів це може бути корисно, однак для повномасштабного застосунку та швидких ітерацій MPS є надмірно складним. На цьому фоні Core ML є найбільш природним і зручним вибором для iOS. Серед його ключових переваг:
- Глибока інтеграція в екосистему Apple: підтримка вбудованими API (Vision, AVFoundation, SwiftUI), автоматичне використання апаратних прискорювачів (Neural Engine, GPU, CPU).
- Зручна конвертація: завдяки інструменту coremltools моделі з PyTorch або TensorFlow можна відносно просто перенести у формат .mlmodel.
- Продуктивність у реальному часі: Core ML оптимізує виконання під конкретне залізо, забезпечуючи стабільний FPS навіть на середніх пристроях.
- Приватність: інференс виконується повністю локально, дані не залишають пристрій, що відповідає вимогам GDPR і політикам Apple.
- Простота інтеграції: модель використовується як звичайний Swift-клас, згенерований Xcode, без додаткових обгорток чи зовнішніх залежностей.

Таким чином, вибір Core ML не є лише питанням зручності для розробника, а випливає з системних вимог до продуктивності, приватності та стабільності. Саме тому у даній роботі Core ML розглядається як основний фреймворк для інтеграції моделі глибинного навчання у застосунок.

3.3 Опис реалізації

Після теоретичного аналізу архітектур і фреймворків у попередньому розділі, на практичному етапі було реалізовано повний цикл інтеграції моделі розпізнавання автомобілів у iOS-застосунок. Важливо підкреслити, що використана модель не була готовим продуктом, а була створена власноруч: навчена у середовищі Python, оптимізована й адаптована під мобільні обмеження, а згодом інтегрована в екосистему iOS за допомогою Core ML.

Базовою архітектурою для побудови системи стала YOLO (You Only Look Once), яка була обрана через оптимальне співвідношення між точністю та швидкістю. Навчання здійснювалось у середовищі PyTorch із використанням офіційної реалізації YOLOv5. У навчальний процес було включено два основні джерела даних: публічний датасет Stanford Cars та власний датасет колекційних моделей автомобілів, що дало змогу компенсувати доменний розрив між лабораторними умовами і реальними сценаріями використання.

У подальших підпунктах буде детально описано всі етапи цього процесу: від побудови та тренування моделі до інтеграції у застосунок iOS, роботи з камерою та відображення результатів у реальному часі.

3.3.1. Створення CoreML моделі

Практичне ядро системи сформоване довкола єдиної моделі, яка одночасно виконує детекцію та тонку класифікацію автомобілів. На відміну від підходів детектор + окремий класифікатор, було цілеспрямовано побудувано одноступеневий багатокласовий детектор на 196 класів (марка/модель/рік), аби отримати стабільний FPS і уникнути подвійного проходу по кадру. Критично важливо, що ця модель не є завантаженим артефактом з репозитарію: вона

підготовлена від нуля — від конвертації оригінальних анотацій Stanford Cars з додаванням власного датасету у формат YOLO до навчання на змішаному корпусі (публічний + власний) і подальшого експорту в Core ML з урахуванням вимог iOS-інференсу.

Початковим джерелом даних слугував Stanford Cars (Cars196). Девкіт містить каталоги зображень cars_train/, cars_test/ і набір .mat-файлів (cars_meta.mat, cars_train_annos.mat, cars_test_annos.mat) із назвами класів та рамками. Оскільки цільова модель — YOLOv8 у режимі detect, був потрібен канонічний формат YOLO: по одному .txt на зображення, де кожен рядок — це <class_id> <cx> <cy> <w> <h> у нормалізованих координатах (0..1). Для цього була написана утиліта cars_to_yolo.py, яка читає метадані з .mat, яка переносить рамки й формує єдину структуру: data/cars196/images/train|val і відповідні labels/train|val. У цьому ж кроці виконувався репрезентативний train/val-спліт, щоб утримувати валідацію в одному домені з тренуванням; тестовий піднабір не був використовуваний як валідаційний, щоб не порушувати первісну методологію. Запуск конвертера виглядав як звична офлайнова операція яка наведена у лістингу 1.

Лістинг 1 — Запуск конвертера

```
python cars_to_yolo.py \
  --devkit dataset/car_devkit/devkit \
  --train  dataset/cars_train \
  --out    data/cars196 \
  --val-split 0.1
```

На виході формувався data.yaml (шляхи, кількість класів, імена категорій) і зручний для iOS-фази labels.json з повним переліком назв (саме цей файл згодом був використаний в застосунку для мапінгу індексу у звичний для людского ока

лейбл). Окремий невеликий скрипт-перевірка валідував, що всі bounding boxes лежать у [0,1] і не виходять за межі; ця дрібниця позбавила хвостових помилок у постобробці.

Навчання виконували на YOLOv8 (Ultralytics). Методологічно ми почали з димового прогона — швидкої перевірки конвеєра (аугментації, читання даних, базова збіжність) — і лише потім переходили до тривалого тренування. Попередній прогін на yolov8s.pt із помірним розміром зображення (640) дозволив оцінити коректність розмітки та профіль пам’яті, після чого ми перейшли на цільову конфігурацію YOLOv8m з 300 епохами. З практичного боку ми використовували дві апаратні платформи: Windows (CUDA) і macOS (MPS). На Apple Silicon навчання відбулося коректно, однак значно повільніше; основні тривалі прогонові сесії ми виконували на Windows-машині з сучасним GPU, де через нову архітектуру (SM_120) знадобилася nightly-збірка PyTorch із підтримкою CUDA 12.8. Стабільний запуск продемонстрований у лістингу 2 і 3.

Лістинг 2 — Sanity-check

```
yolo detect train \
  data=data/cars196/data.yaml \
  model=yolov8s.pt \
  epochs=10 \
  imgsz=640 \
  batch=16 \
  device=0 \
  workers=4 \
  cache=ram \
  amp=True \
  mosaic=0.5 translate=0.1 hsv_h=0.015 hsv_s=0.7 hsv_v=0.4 \
  name=cars196_smoketest
```

Лістинг 3 — Final run

```
yolo detect train ^
  data=data/cars196/data.yaml ^
  model=yolov8m.pt ^
  epochs=300 ^
  imgsz=640 ^
  batch=auto ^
  device=0 ^
  workers=2 ^
  cache=disk ^
  amp=True ^
  close_mosaic=10 ^
  name=cars196_yolov8m_e300
```

Режим batch=auto не завжди коректно оцінює VRAM: при нестабільності фіксувалось batch вручну 8 або 16 і зменшувалось workers до 2 чи 0. Після завершення навчання стандартні артефакти runs/detect/<exp>/weights/best.pt, графіки метрик, results.csv слугували базою для наступного етапу — експорту в Core ML. Конкретні числові результати якості mAP, precision/recall узагальнено в розділі 4, тут фокусується лише інженерна лінія постачання моделі в iOS.

Експорт у Core ML здійснювався на macOS, оскільки утиліта Ultralytics підтримує цей таргет нативно саме на платформі Apple. Важливе проектне рішення — відмовитися від вбудованого NMS: на етапі експорту, навмисно вимикаємо NMS, щоб виконувати його на пристрої це дає гнучкість у тюнінгу порогів під конкретний девайс і сцену. Команда має звичний вигляд:

Лістинг 4 — Код експорту моделі у CoreML форматі

```
yolo export \
  model="runs/detect/cars196_yolov8m_e300/weights/best.pt" \
  format=coreml \
```

```
imgsz=640 \
nms=False \
half=False \
device=cpu
```

На виході отримуємо .mlpackage. На практиці зустрічаються різні схеми вихідних тензорів та імен ключів: наприклад, масив координат [N, 4] або [4, N] і масив логітів/ймовірностей класів [N, 196] можуть бути позначені як coordinates/confidence або мати технічні імена. У iOS-коді закладений м'який механізм автодетекції ключів і нормалізації осей: при ініціалізації детектора застосунок читає словник виходів моделі, ідентифікує вектори рамок та класів за розмірами та вибудовує адаптер, щоб надалі працювати з уніфікованими структурами. Це дозволяє без перекомпіляції додатка підмінити .mlpackage, навіть якщо внутрішні назви зміняться при повторному експорті.

3.3.2. Підключення та інтеграція Core ML у застосунок

Після етапу побудови .mlpackage (див. 3.3.1) центральним завданням стає архітектурно коректне та відтворюване підключення моделі Core ML у мобільний конвеєр: від прийому кадру камери у вигляді CVPixelBuffer до повернення уніфікованого набору детекцій, придатного для негайного відображення в SwiftUI. У реалізації програмного застосунку цей шлях формалізовано трьома опорними модулями: фабрикою моделі ModelFactory, детектором Detector і завантажувачем метаданих LabelsLoader. Разом вони утворюють стабільний контракт, який не залежить від конкретного експорту .mlpackage і дозволяє замінювати модель без перекомпіляції UI-шару.

Ініціалізація моделі та вибір обчислювальних ресурсів

Фабрика інкапсулює створення Core ML-моделі й політику використання

апаратних прискорювачів. Ключове рішення — доручити Core ML вибір виконувальних одиниць через MLModelConfiguration.computeUnits = .all, що, за наявності Neural Engine, забезпечує найменшу латентність інференсу, з коректною деградацією до GPU/CPU на старіших пристроях. Додатково на цьому рівні фіксуємо, що інтеграція з Vision відбувається через VNCoreMLModel, адже фреймворк Vision бере на себе узгодження орієнтації кадру та режиму масштабування використовується параметр .scaleFit. Код фабрики наведений у лістингу 5. Завдяки фабриці централізується апаратна політика й звільняються вищі шари від знання деталей моделі:

Лістинг 5 — Ініціалізація Core ML і побудова VNCoreMLModel.

```
import CoreML
import Vision

final class ModelFactory {
    private let configuration: MLModelConfiguration

    init(useAllComputeUnits: Bool = true) {
        let cfg = MLModelConfiguration()
        cfg.computeUnits = useAllComputeUnits ? .all : .cpuAndGPU
        self.configuration = cfg
    }

    func makeVisionModel() throws -> VNCoreMLModel {
        let compiled = try CarDetector(configuration: configuration)
        let vn = try VNCoreMLModel(for: compiled.model)
        vn.inputImageFeatureName = compiled.model.modelDescription.inputDescriptionsByName.first?.key
        return vn
```

```
        }
    }
```

Уніфікація виходів моделі, мапінг класів і постобробка

Навіть у межах одного експорту Ultralytics назви й осі вихідних тензорів можуть відрізнятися (наприклад, координати [N,4] протиставлені [4,N], а матриця класів [N,C] — [C,N]; імена ключів — людяні або технічні). Щоб не прошивати в коді конкретні імена, у Detector закладено адаптер виходів, який на етапі ініціалізації читає modelDescription, знаходить потрібні багатовимірні масиви за формою і типом, і приводить їх до канонічного вигляду: координати як [N,4] у форматі cx, cy, w, h (нормалізовано до [0,1]), а класові оцінки — як [N,C], де C дорівнює довжині словника класів, який зчитує LabelsLoader.

Лістинг 6 — Код класу LabelsLoader.

```
    final class LabelsLoader {
        static func load(from url: URL) throws -> [String] {
            let data = try Data(contentsOf: url)
            return try JSONDecoder().decode([String].self, from:
data)
        }
    }
```

У детекторі будується VNCoreMLRequest з обробником, який дістає з VNRequest, через модель, доступну як VNCoreMLModel, сирі MLFeatureValue, виявляє координатний і класовий тензори, за потреби транспонує їх і лише тоді переходить до постобробки: порогова фільтрація та non-maximum suppression (NMS). Вихід — масив уніфікованих записів DetectionResult, де координати зберігаються в нормованій квадратній системі входу моделі, що зручно для подальшого мапінгу на екран.

Лістинг 7 — Код класу Detector.

```
import Vision
import CoreML

struct DetectionResult: Identifiable {
    let id = UUID()
    let bbox: CGRect      // нормовані [0,1] у квадраті входу
моделі
    let classIndex: Int
    let className: String
    let confidence: Float
}

final class Detector {
    private let vnModel: VNCoreMLModel
    private let labels: [String]
    private let confThreshold: Float
    private let iouThreshold: Float

    init(vnModel: VNCoreMLModel, labels: [String],
         confidence: Float, iou: Float) {
        self.vnModel = vnModel
        self.labels = labels
        self.confThreshold = confidence
        self.iouThreshold = iou
    }

    func makeRequest(completion: @escaping ([DetectionResult])
-> Void) -> VNCoreMLRequest {
        let req = VNCoreMLRequest(model: vnModel) { [labels,
confThreshold, iouThreshold] r, _ in
```

```
                // 1) Дістаємо сирі фічі
                guard let coreML = (r as? VNRequest)?.model as?
VNCoreMLModel else {
                    completion([]); return
                }
                // Отримуємо словник виходів у вигляді [назва:
MLFeatureValue]
                let raw: [String: MLFeatureValue] =
Self.extractOutputs(from: r, using: coreML)
                // 2) Знаходимо координати й клас-скори за формою
і приводимо до [N,4] / [N,C]
                guard let canon = Self.canonicalize(raw,
numClasses: labels.count) else {
                    completion([]); return
                }
                //  Перетворюємо рядки канонічних тензорів у
детекції
                var dets: [DetectionResult] = []
                let N = canon.coords.shape[0].intValue
                for i in 0..<N {
                    let cx = canon.coords[[i as NSNumber,
0]].floatValue
                    let cy = canon.coords[[i as NSNumber,
1]].floatValue
                    let w = canon.coords[[i as NSNumber,
2]].floatValue
                    let h = canon.coords[[i as NSNumber,
3]].floatValue

                    // argmax по класах
                    var best = -Float.greatestFiniteMagnitude
                    var idx = 0
                    for c in 0..<labels.count {
```

```
                        let s = canon.scores[[i as NSNumber, c as
NSNumber]].floatValue
                        if s > best { best = s; idx = c }
                    }
                    guard best >= confThreshold else { continue }
                    let rect = CGRect(x: Double(cx - w/2), y:
Double(cy - h/2),
                                      width: Double(w), height:
Double(h)).standardized
                    dets.append(.init(bbox: rect, classIndex: idx,
                                      className: labels[idx],
confidence: best))
                }
                // NMS
                completion(Self.nms(dets, iou: iouThreshold))
            }
            req.imageCropAndScaleOption = .scaleFit
            return req
        }

        // Збір сирих виходів із VNCoreMLRequest [назва:
MLFeatureValue]
        private static func extractOutputs(from request:
VNRequest,
                                           using model:
VNCoreMLModel) -> [String: MLFeatureValue] {
            var out: [String: MLFeatureValue] = [:]
    VNCoreMLFeatureValueObservation; звіряємо з modelDescription
            let desc =
model.model.modelDescription.outputDescriptionsByName
            (request.results ?? []).forEach { obs in
                if let fvObs = obs as?
VNCoreMLFeatureValueObservation,
```

```
                    let name = desc.first(where: { $0.value ==
fvObs.featureValue })?.key {
                        out[name] = fvObs.featureValue
                    }
                }
                return out
            }

            // Приведення осей: coords [N,4]; scores [N,C]
            private static func canonicalize(_ raw: [String:
MLFeatureValue],
                                             numClasses: Int) ->
(coords: MLMultiArray, scores: MLMultiArray)? {
                var coords: MLMultiArray?
                var scores: MLMultiArray?

                for (name, fv) in raw {
                    guard let ma = fv.multiArrayValue else { continue
}
                    let shape = ma.shape.map(\.intValue)
                    if shape.count == 2 && shape.contains(4) {
                        coords = (shape[0] == 4) ? try?
ma.transposed(permutation: [1,0]) : ma
                    } else if shape.count == 2 {
                        if shape[1] == numClasses { scores = ma }
                        else if shape[0] == numClasses { scores = try?
ma.transposed(permutation: [1,0]) }
                    }
                }
                if let c = coords, let s = scores { return (c, s) }
                return nil
            }
```

```
        // Класичний NMS за IoU
        private static func nms(_ xs: [DetectionResult], iou:
Float) -> [DetectionResult] {
            let sorted = xs.sorted { $0.confidence > $1.confidence
}
            var kept: [DetectionResult] = []
            var removed = Set<UUID>()
            for i in 0..<sorted.count where
!removed.contains(sorted[i].id) {
                let a = sorted[i]; kept.append(a)
                for j in (i+1)..<sorted.count where
!removed.contains(sorted[j].id) {
                    let b = sorted[j]
                    if iouRect(a.bbox, b.bbox) > iou {
removed.insert(sorted[j].id) }
                }
            }
            return kept
        }

        private static func iouRect(_ a: CGRect, _ b: CGRect) ->
Float {
            let inter = a.intersection(b)
            guard !inter.isNull else { return 0 }
            let i = inter.width * inter.height
            let u = a.width*a.height + b.width*b.height - i
            return u > 0 ? Float(i/u) : 0
        }
    }
```

На практиці використовується саме цей м’який шлях: жодних хардкодів на ключі/осі, канонізація за формою, уніфіковані DetectionResult нагору.

Взаємодія з кадровим конвеєром та орієнтацією

Під час виконання запиту Vision ми задаємо imageCropAndScaleOption = .scaleFit. Це означає, що вхід моделі завжди квадратний, а зображення з камери вписується зі збереженням аспекту додаються letterbox поля. Саме тому координати, які повертає модель, нормовані в квадраті, і для відображення на фактичному прев'ю потрібне коректне відображення координат. Перевага .scaleFit у тому, що не деформується зображення автомобіля у вхідних тензорах; натомість один раз акуратно трансформуємо детекцію назад у площину екрана.

Узгодження координат із площею прев'ю

Після постобробки DetectionResult.bbox зберігається у нормованій квадратній системі. На етапі відмальовування оверлею мапимо його в локальні координати CameraPreview/OverlayRepresentable. Це робиться на основі фактичного frame прев'ю (ширина/висота після layout SwiftUI) й обраної політики масштабування. Завдяки розширенню CGRect+Finite.swift використовується допоміжне перетворення, щоб отримати екранні координати без спотворень і відсікання.

Лістинг 8 — Фрагмент коду з файлу CGRect+Finite.

```
extension CGRect {
        func mapUnitRect(to frame: CGRect) -> CGRect {
            CGRect(x: frame.minX + self.minX * frame.width,
                   y: frame.minY + self.minY * frame.height,
                   width: self.width * frame.width,
                   height:           self.height           *
frame.height).standardized
        }
}
```

Допоміжні перетворення зображень

Хоча основний конвеєр працює zero-copy з CVPixelBuffer, у застосунку є допоміжні перетворення наприклад у UIImage+PixelBuffer.swift для діагностики, скріншотів чи зовнішньої валідації. Ці утиліти не використовуються на гарячому шляху інференсу, щоб не вносити зайві копії/алокації, але корисні для тестування і журналювання.

Підсумок

На цьому етапі було завершено інженерну інтеграцію Core ML у застосунок: модель створюється фабрикою з правильною конфігурацією обчислень; виходи моделі автоматично адаптуються за формою й перетворюються у канонічний вигляд; постобробка threshold + NMS виконується на пристрої, забезпечуючи керованість і відтворюваність; координати коректно ремапляться у площину UI, що гарантує геометрично правильне відображення. Цей дизайн ізолює всі мінливі деталі експорту усередині Detector, зберігаючи чистоту MVVM і дозволяючи безболісно оновлювати .mlpackage.

3.3.3. Робота з камерою

Жодна система комп’ютерного зору у реальному часі не може обійтися без стабільного й ефективного джерела кадрів. У середовищі iOS за цю задачу відповідає фреймворк AVFoundation, який надає низькорівневий контроль над камерою, форматами зображень, частотою кадрів та буферизацією. Головна мета — побудувати конвеєр, у якому кожен кадр із камери у вигляді CVPixelBuffer вчасно доходить до моделі Core ML, обробляється й повертається у вигляді детекцій, не накопичуючи затримок і не порушуючи плавність інтерфейсу.

Ініціалізація камери й вибір налаштувань

У класі CameraViewController формується AVCaptureSession. Було свідомо обрано профіль hd1280x720: це компроміс між достатньою деталізацією для виявлення дрібних деталей автомобілів і прийнятною швидкістю обробки на мобільних пристроях.

Лістинг 9 — Код класу CameraViewController.

```
    final class CameraViewController: UIViewController,
AVCaptureVideoDataOutputSampleBufferDelegate {

        // MARK: - UI

        private let overlayView = OverlayView()
        private var overlayItems: [OverlayView.Item] = [] {
            didSet {
                DispatchQueue.main.async {
                    self.overlayView.draw(items:
self.overlayItems)
                }
            }
        }

        // MARK: - Camera

        private let captureSession = AVCaptureSession()
        private let sessionQueue = DispatchQueue(label:
"camera.session")
        private let videoQueue = DispatchQueue(label:
"camera.video")
```

```
        private var frameIndex = 0
        private var videoDimensions = CMVideoDimensions(width: 0,
height: 0)

        // MARK: - Detector

        private var detector = Detector(configuration: .init())
        private var classLabels: [String] = []
        private var coordinateTransform: ((CGRect) -> CGRect) = {
$0 }

        // MARK: - Public API

        func     updateDetectorConfiguration(_     configuration:
Detector.Config) {
            detector.reconfigureIfNeeded(configuration)
        }

        // MARK: - Life Cycle

        override func viewDidLoad() {
            super.viewDidLoad()

            view.backgroundColor = .black
            overlayView.frame = view.bounds
            overlayView.autoresizingMask    =    [.flexibleWidth,
.flexibleHeight]
            view.addSubview(overlayView)

            classLabels = LabelsLoader.load()
            configureSessionAsync()
        }
```

```
        override func viewDidLayoutSubviews() {
            super.viewDidLayoutSubviews()
            if let previewLayer = view.layer.sublayers?.compactMap({ $0 as? AVCaptureVideoPreviewLayer }).first {
                previewLayer.frame = view.bounds
                buildCoordinateTransform(previewLayer: previewLayer)
            }
        }

        override func viewDidAppear(_ animated: Bool) {
            super.viewDidAppear(animated)
            sessionQueue.async { self.captureSession.startRunning() }
        }

        override func viewWillDisappear(_ animated: Bool) {
            super.viewWillDisappear(animated)
            sessionQueue.async { self.captureSession.stopRunning() }
        }

        // MARK: - Camera configuration

            private func configureSessionAsync() {
            AVCaptureDevice.requestAccess(for: .video) { granted in
                guard granted else { print("❌ Доступ до камери заборонений"); return }
                self.sessionQueue.async {
                    self.captureSession.beginConfiguration()
```

```
                    self.captureSession.sessionPreset = .high

                    guard
                        let                device                =
AVCaptureDevice.default(.builtInWideAngleCamera,    for:    .video,
position: .back),
                        let         input         =         try?
AVCaptureDeviceInput(device: device),
                        self.captureSession.canAddInput(input)
                    else {
                        self.captureSession.commitConfiguration()
                        return
                    }

                    self.captureSession.addInput(input)

                    let videoOutput = AVCaptureVideoDataOutput()
                    videoOutput.alwaysDiscardsLateVideoFrames    =
true
                    videoOutput.videoSettings                    =
[kCVPixelBufferPixelFormatTypeKey            as            String:
kCVPixelFormatType_32BGRA]
                    if
self.captureSession.canAddOutput(videoOutput)                    {
self.captureSession.addOutput(videoOutput) }
                    videoOutput.setSampleBufferDelegate(self,
queue: self.videoQueue)

                    if          let          connection          =
videoOutput.connection(with: .video),
                       connection.isVideoOrientationSupported {
                        connection.videoOrientation = .portrait
                    }
```

```
                    DispatchQueue.main.async {
                        let             previewLayer             =
AVCaptureVideoPreviewLayer(session: self.captureSession)
                        previewLayer.videoGravity                =
.resizeAspectFill
                        previewLayer.frame = self.view.bounds

self.view.layer.insertSublayer(previewLayer, at: 0)

self.buildCoordinateTransform(previewLayer: previewLayer)
                    }

                    self.captureSession.commitConfiguration()
                }
            }
        }

        private    func    buildCoordinateTransform(previewLayer:
AVCaptureVideoPreviewLayer) {
            let                    orientation                    =
previewLayer.connection?.videoOrientation ?? .portrait
            let             isMirrored                            =
previewLayer.connection?.isVideoMirrored ?? false

            coordinateTransform = { [weak self] rect in
                guard let self = self else { return rect }

                let           width           =           max(1,
CGFloat(self.videoDimensions.width))
                let           height           =           max(1,
CGFloat(self.videoDimensions.height))
```

```
                var normalizedRect: CGRect
                switch orientation {
                case .portrait:
                    normalizedRect = CGRect(x: rect.minY / height,
y: 1.0 - (rect.maxX / width), width: rect.height / height, height:
rect.width / width)
                case .portraitUpsideDown:
                    normalizedRect = CGRect(x: 1.0 - (rect.maxY /
height), y: rect.minX / width, width: rect.height / height, height:
rect.width / width)
                case .landscapeRight:
                    normalizedRect = CGRect(x: rect.minX / width,
y: rect.minY / height, width: rect.width / width, height:
rect.height / height)
                case .landscapeLeft:
                    normalizedRect = CGRect(x: 1.0 - (rect.maxX /
width), y: rect.minY / height, width: rect.width / width, height:
rect.height / height)
                @unknown default:
                    normalizedRect = CGRect(x: rect.minX / width,
y: rect.minY / height, width: rect.width / width, height:
rect.height / height)
                }

                if isMirrored { normalizedRect.origin.x = 1.0 -
normalizedRect.origin.x - normalizedRect.size.width }
                return
previewLayer.layerRectConverted(fromMetadataOutputRect:
normalizedRect)
            }
        }

        // MARK: - AVCaptureVideoDataOutputSampleBufferDelegate
```

```
            func captureOutput(_ output: AVCaptureOutput,
                               didOutput                sampleBuffer:
CMSampleBuffer,
                               from connection: AVCaptureConnection) {
                frameIndex &+= 1
                let            everyNth            =            max(1,
detector.configuration.everyNthFrame)
                if frameIndex % everyNth != 0 { return }

                guard           let           pixelBuffer           =
CMSampleBufferGetImageBuffer(sampleBuffer) else { return }

                videoDimensions       =       CMVideoDimensions(width:
Int32(CVPixelBufferGetWidth(pixelBuffer)),
                                                  height:
Int32(CVPixelBufferGetHeight(pixelBuffer)))

                let detections = detector.predict(
                    pixelBuffer: pixelBuffer,
                    toLayer: coordinateTransform,
                    labels: classLabels
                )

                let items = detections.map {
                    OverlayView.Item(
                        rect: $0.boundingBox,
                        title:    "\($0.className)    \(String(format:
"%.2f", $0.confidence))"
                    )
                }
                overlayItems = items
        }
```

```
}
```

Таким чином виконується прив’язуєка основної задньої камери і встановлено єдина точка входу для відеопотоку. Важливо, що beginConfiguration()/commitConfiguration() дають можливість атомарно оновлювати налаштування, уникаючи конфліктів.

Отримання кадрів через AVCaptureVideoDataOutput

Щоб отримати доступ до сирих кадрів, було додано вихід AVCaptureVideoDataOutput. Його делегат отримує CMSampleBuffer, із якого без копіювання дістаємо CVPixelBuffer. Саме цей формат очікує Core ML як вхідний для моделей зображень.

Лістинг 10 — Фрагмет коду з доступом до сирих даних.

```
videoOutput.alwaysDiscardsLateVideoFrames = true
videoOutput.videoSettings = [
    kCVPixelBufferPixelFormatTypeKey as String: kCVPixelFormatType_32BGRA
]
videoOutput.setSampleBufferDelegate(self, queue: queue)
session.addOutput(videoOutput)

if let conn = videoOutput.connection(with: .video),
   conn.isVideoOrientationSupported {
    conn.videoOrientation = .portrait
}
```

Тут ключова опція — alwaysDiscardsLateVideoFrames = true. Вона

запобігає накопиченню кадрів у черзі: якщо модель ще не встигла обробити попередній кадр, наступні будуть відкинуті. Це формує політику back-pressure: користувач бачить плавне відео і лише актуальні детекції, навіть якщо модель не встигає за максимальною швидкістю камери.

Контроль навантаження: прапор isInferenceRunning

У реалізації є простий, але ефективний механізм контролю: прапор isInferenceRunning. Коли кадр відправляється на інференс, прапор встановлюється, і поки результат не повернеться, нові кадри ігноруються. Як тільки модель відпрацювала — прапор скидається. Це усуває лавиноподібне накопичення задач і стабілізує використання CPU/GPU/ANE.

Лістинг 11 — Фрагмет коду з флагом isInferenceRunning.

```
    private var isInferenceRunning = false

    func captureOutput(_ output: AVCaptureOutput,
                       didOutput sampleBuffer: CMSampleBuffer,
                       from connection: AVCaptureConnection) {
        guard !isInferenceRunning,
              let pb = CMSampleBufferGetImageBuffer(sampleBuffer)
else { return }
        isInferenceRunning = true
        let orientation = CGImagePropertyOrientation.right
        delegate?.camera(self,   didOutput:   pb,   orientation:
orientation)
    }

    func unlock() {
        isInferenceRunning = false
    }
```

Ця стратегія дає контрольований FPS: застосунок ніколи не зависає, оскільки кадри не накопичуються, і ми завжди працюємо з найсвіжішим буфером.

Відображення прев'ю (CameraPreview)

Щоб користувач бачив живе відео, CameraPreview реалізовано як UIViewRepresentable у SwiftUI. Він інкапсулює AVCaptureVideoPreviewLayer і відображає його як підкладку. Таким чином забезпечується апаратно оптимізований рендеринг без додаткового копіювання буферів.

Лістинг 12 — Код класу CameraPreview.

```
struct CameraPreview: UIViewRepresentable {
    let session: AVCaptureSession

    func makeUIView(context: Context) -> UIView {
        let view = UIView(frame: .zero)
        let layer = AVCaptureVideoPreviewLayer(session: session)
        layer.videoGravity = .resizeAspectFill
        view.layer.addSublayer(layer)
        layer.frame = view.bounds
        return view
    }

    func updateUIView(_ uiView: UIView, context: Context) {
        if let layer = uiView.layer.sublayers?.first as? AVCaptureVideoPreviewLayer {
            layer.frame = uiView.bounds
        }
    }
}
```

У поєднанні з OverlayRepresentable цей підхід дозволяє малювати SwiftUI-графіку поверх живого відео з мінімальними затримками.

Підсумок

Таким чином, камера у розробленій системі реалізована як двоступеневий конвеєр:

- CameraViewController забезпечує захоплення буферів через AVFoundation і керує частотою подачі кадрів;
- CameraPreview відображає живе відео користувачу;
- політика back-pressure й прапор isInferenceRunning гарантують стабільний FPS і відсутність накопичення затримки.

Цей дизайн створює ідеальне джерело даних для подальшої інтеграції з Core ML і накладанням результатів у SwiftUI.

3.3.4. Відображення результатів (Overlay, SwiftUI UI)

Після того, як кадр із камери пройшов інференс Core ML і постобробку threshold + NMS, застосунок має розв’язати, здавалося б, просту задачу: надійно та читабельно відобразити детекції поверх живого відео. На практиці це вимагає акуратної координації кількох аспектів: відповідності координатних систем, вхід моделі — квадрат із letterbox, прев’ю — прямокутник екрана, енергоефективності мінімум перерахунків і перерисовок, стійкості UI відсутність мигання та накладення підписів. У розробленій реалізації ці питання вирішуються тандемом SwiftUI + UIViewRepresentable: система працює як реактивний шар, який слухає зміни стану у ViewModel і безпосередньо накладає геометрію поверх AVCaptureVideoPreviewLayer.

Узгодження координат: від нормалізованого квадрата до площини

прев’ю

Модель повертає рамки в нормованій квадратній системі: bbox = [cx, cy, w, h] у межах [0,1], причому Vision подає кадр із imageCropAndScaleOption = .scaleFit. Це означає, що вхідний квадрат коректно вписаний у фактичний прямокутник прев’ю з потенційними полями letterbox зверху/знизу або зліва/праворуч. Щоб рамки опинились рівно над об’єктами, треба мати єдину функцію мапінгу з одиничної системи в систему локальних координат в’юшки. Акуратніше тримати це в утилітарному розширенні CGRect+Finite.swift / допоміжний мапер, щоб логіка була централізована і тестована.

Лістинг 13 — Фрагмент коду з файлу CGRect+Finite.

```
import CoreGraphics

extension CGRect {
    func mapped(to frame: CGRect) -> CGRect {
        CGRect(
            x: frame.minX + self.minX * frame.width,
            y: frame.minY + self.minY * frame.height,
            width: self.width * frame.width,
            height: self.height * frame.height
        ).standardized
    }
}
```

Рендеринг оверлею: SwiftUI у ролі легкого полотна

OverlayView.swift відповідає за малювання рамок і підписів. Основна ідея — не торкатися буферів відео, а лише рендерити легкі векторні форми поверх живого AVCaptureVideoPreviewLayer. SwiftUI тут ідеальний: його декларативна модель чудово реагує на зміни стану у ViewModel списки детекцій, а відмальовка відбувається диференційовано, без зайвих перерисовок.

Лістинг 14 — Код файлу OverlayView.

```
import SwiftUI

struct OverlayView: View {
    let detections: [DetectionResult]  // bbox y [0,1]
    let frameRect: CGRect                  // фактичний фрейм
прев’ю

    var body: some View {
        ZStack {
            ForEach(detections) { det in
                let rect = det.bbox.mapped(to: frameRect)

                // Рамка
                Rectangle()
                    .strokeBorder(lineWidth: 2)
                    .foregroundStyle(.green)
                    .frame(width:     rect.width,     height:
rect.height)
                    .position(x: rect.midX, y: rect.midY)

                // Підпис (клас + %)
                Text("\(det.className) \(Int(det.confidence *
100))%")
                    .font(.system(size: 12, weight: .semibold,
design: .rounded))
                    .foregroundColor(.white)
                    .padding(.horizontal, 6)
                    .padding(.vertical, 3)
                    .background(.black.opacity(0.65),     in:
RoundedRectangle(cornerRadius: 6))
```

```
                        .position(x:    rect.minX    +    6,    y:
max(rect.minY - 12, 8))
                        .fixedSize()
                }
            }
            .allowsHitTesting(false)    //    щоб    оверлей    не
перехоплював жести
        }
    }
```

Ключові моменти: підпис приклеєний до верхнього лівого кута рамки, але з урахуванням меж max(..., 8), щоб не виходив за край екрану. Шрифт невеликий, контрастний; фон — напівпрозорий чорний, читабельний на будь-якому відео.

Вбудовування оверлею у прев'ю: OverlayRepresentable і GeometryReader

Щоб поверх нативного AVCaptureVideoPreviewLayer з'явився SwiftUI-оверлей, використовуємо UIViewRepresentable для прев'ю і накладаємо на нього OverlayView з правильним розміром через GeometryReader. Це дозволяє щофреймово підлаштовувати координати оверлею під фактичні розміри в'ю після автолейауту.

Лістинг 15 — Код файлу ContentView.

```
import SwiftUI

    struct ContentView: View {
        @StateObject var vm: AppViewModel
```

```
            var body: some View {
                ZStack {
                    CameraPreview(session: vm.captureSession)
                        .overlay {
                            GeometryReader { geo in
                                OverlayView(detections:
vm.displayDetections,
                                            frameRect:
geo.frame(in: .local))
                            }
                        }
                    SettingsPanel(vm:    vm)    //    поверх    у
правому/лівому куті
                }
                .ignoresSafeArea()
            }
}
```

Стабілізація відображення: гістерезис, top-k, запобігання дублювання лейблів

Мобільний відеопотік завжди шумний: score може коливатись на межі класів, рамка — трохи дихати. Щоб інтерфейс не мигав, застосоваго легкий гістерезис у ViewModel: якщо нова детекція дуже близька до старої за геометрією та неоднозначна за класами різниця score < δ, короткочасно тримаємо попередній підпис 2–3 кадри, зменшуючи стрибки. Додатково в місцях щільних об’єктів обмежується top-k наприклад, показуємо не більше N підписів у межах невеликої області, щоб уникнути накладання лейблів. Ці прийоми додають людяності інтерфейсу, зберігаючи точність.

Схема стабілізації проста: у VM тримаємо історичний набір lastDetections і перед оновленням зливаємо його з новим списком, вирішуючи конфлікти по

IoU і різниці score. Оскільки це логіка ViewModel, UI залишається чистим.

Продуктивність: мінімум роботи в рендері

Важлива особливість оверлею — він не торкається зображення, працює виключно з геометрією. Відео відмальовує системний AVCaptureVideoPreviewLayer апаратно оптимізований, а SwiftUI малює тонкі векторні елементи. Оскільки OverlayView декларативний, перерисовується лише за фактом зміни vm.displayDetections. Це означає, що навіть на середніх пристроях оверлей майже не впливає на FPS, основна латентність належить саме інференсу.

Коректність і тестованість: інваріанти координат

Нарешті, координатна арифметика має бути перевірена. Тести з LetterboxMathTests.swift закріплюють інваріанти:

- нормалізована рамка [0,1] × [0,1] після мапінгу ніколи не виходить за межі frameRect;
- перетворення з урахуванням .scaleFit зберігає пропорції;
- перехід між портретом/ландшафтом не змінює взаємного розташування боксу й об’єкта.

Лістинг 16 — Код тесту LetterboxMathTests.

```
import XCTest
@testable import CVCar
import CoreVideo
import UIKit

final class LetterboxMathTests: XCTestCase {

    private func letterboxInfo(srcW: CGFloat, srcH: CGFloat, dstW: CGFloat, dstH: CGFloat)
    -> (scale: CGFloat, dx: CGFloat, dy: CGFloat) {
```

```
            let scale = min(dstW / srcW, dstH / srcH)
            let rw = srcW * scale
            let rh = srcH * scale
            let dx = (dstW - rw) * 0.5
            let dy = (dstH - rh) * 0.5
            return (scale, dx, dy)
        }

        private func toLetterboxRect(xywh: CGRect, scale: CGFloat,
dx: CGFloat, dy: CGFloat) -> CGRect {
            return CGRect(x: xywh.minX * scale + dx,
                          y: xywh.minY * scale + dy,
                          width: xywh.width * scale,
                          height: xywh.height * scale)
        }

        private  func  fromLetterboxRect(_  r:  CGRect,  scale:
CGFloat, dx: CGFloat, dy: CGFloat) -> CGRect {
            CGRect(x: (r.minX - dx) / scale,
                   y: (r.minY - dy) / scale,
                   width: r.width / scale,
                   height: r.height / scale)
        }

        func testRoundtripLetterbox() {
            let srcW: CGFloat = 1920, srcH: CGFloat = 1080
            let dstW: CGFloat = 640,  dstH: CGFloat = 640

            let (scale, dx, dy) = letterboxInfo(srcW: srcW, srcH:
srcH, dstW: dstW, dstH: dstH)

            let rSrc = CGRect(x: 400, y: 200, width: 600, height:
300)
```

```
            let r640 = toLetterboxRect(xywh: rSrc, scale: scale,
dx: dx, dy: dy)
            let rBack = fromLetterboxRect(r640, scale: scale, dx:
dx, dy: dy)

            XCTAssertLessThan(abs(rBack.minX - rSrc.minX), 0.5)
            XCTAssertLessThan(abs(rBack.minY - rSrc.minY), 0.5)
            XCTAssertLessThan(abs(rBack.width - rSrc.width), 0.5)
            XCTAssertLessThan(abs(rBack.height   -   rSrc.height),
0.5)
        }
    }
```

Це дозволяє відловлювати регресії наприклад, якщо змінилася політика Vision або формат моделі, ще до релізу.

3.3.5. Панель налаштувань і параметрів

Функціонування системи розпізнавання у реальному часі завжди пов’язане з компромісами між точністю, швидкістю й читабельністю інтерфейсу. Тому параметри інференсу мають бути керованими на пристрої — без переекспорту моделі й без перезапуску відеосесії. У реалізації це забезпечено панеллю налаштувань на SwiftUI SettingsPanel.swift у поєднанні з легкими елементами керування LabeledSlider.swift, LabeledStepper.swift і реактивною логікою в AppViewModel. Параметри: поріг упевненості, поріг IoU для NMS, профіль моделі швидка/точна, видимість підписів змінюються в рантаймі і тут же впливають на роботу Detector, не торкаючись AVCaptureSession.

Лістинг 17 — Код класу SettingsPanel.

```
struct SettingsPanel: View {
     @Binding var configuration: Detector.Config
    var onClose: () -> Void

    private var confidenceBinding: Binding<Float> {
        Binding<Float>(
            get: { configuration.confidenceThreshold },
            set: { configuration.confidenceThreshold = $0 }
        )
    }
    private var iouBinding: Binding<Float> {
        Binding<Float>(
            get: { configuration.iouThreshold },
            set: { configuration.iouThreshold = $0 }
        )
    }
    private var maxDetectionsBinding: Binding<Int> {
        Binding<Int>(
            get: { configuration.maxDetections },
            set: { configuration.maxDetections = $0 }
        )
    }
    private var everyNthFrameBinding: Binding<Int> {
        Binding<Int>(
            get: { configuration.everyNthFrame },
            set: { configuration.everyNthFrame = $0 }
        )
    }
    private var inputSizeBinding: Binding<Detector.InputSize>
{
        Binding<Detector.InputSize>(
            get: { configuration.inputSize },
            set: { configuration.inputSize = $0 }
```

```
    )
}
private var computeUnitsBinding: Binding<MLComputeUnits> {
    Binding<MLComputeUnits>(
        get: { configuration.computeUnits },
        set: { configuration.computeUnits = $0 }
    )
}

var body: some View {
    VStack(spacing: 16) {
        HStack {
            Text("Settings").font(.headline)
            Spacer()
            Button("Close", action: onClose)
        }
        Group {
            LabeledSlider(
                title: "Confidence",
                value: confidenceBinding,
                valueRange: 0.01...0.9,
                valueStep: 0.01
            )
            LabeledSlider(
                title: "IoU",
                value: iouBinding,
                valueRange: 0.1...0.9,
                valueStep: 0.01
            )
            LabeledStepper(
                title: "MaxDet",
                value: maxDetectionsBinding,
                valueRange: 1...10
```

```
            )
            LabeledStepper(
                title: "Every N-th",
                value: everyNthFrameBinding,
                valueRange: 1...5,
                valueSuffix: "×1"
            )
        }
        HStack {
            Text("Input")
            Spacer()
            Picker("", selection: inputSizeBinding) {
                Text("320").tag(Detector.InputSize._320)
                Text("512").tag(Detector.InputSize._512)
                Text("640").tag(Detector.InputSize._640)
            }
            .pickerStyle(.segmented)
            .frame(width: 300)
        }
        HStack {
            Text("Compute")
            Spacer()
            Picker("", selection: computeUnitsBinding) {
                Text("CPU").tag(MLComputeUnits.cpuOnly)
                Text("GPU").tag(MLComputeUnits.cpuAndGPU)
                Text("ALL").tag(MLComputeUnits.all)
            }
            .pickerStyle(.segmented)
            .frame(width: 300)
        }
    }
    .padding(16)
    .background(.ultraThinMaterial,                    in:
```

```
RoundedRectangle(cornerRadius: 20))
            .padding(.horizontal, 16)
            .padding(.bottom, 12)
            .frame(maxHeight: 260, alignment: .bottom)
            .frame(maxWidth:  .infinity,  maxHeight:  .infinity,
alignment: .bottom)
        }
    }
```

Лістинг 18 — Код класу LabeledSlider.

```
    struct LabeledSlider: View {
        let title: String
        @Binding var value: Float
        let valueRange: ClosedRange<Float>
        let valueStep: Float

        var body: some View {
            VStack(alignment: .leading, spacing: 6) {
                HStack {
                    Text(title)
                    Spacer()
                    Text(String(format: "%.2f", value))
                        .foregroundStyle(.secondary)
                }
                Slider(value:  $value,  in:  valueRange,  step:
valueStep)
            }
        }
    }
```

Лістинг 19 — Код класу LabeledStepper.

```
    struct LabeledStepper: View {
```

```
        let title: String
        @Binding var value: Int
        let valueRange: ClosedRange<Int>
       var valueSuffix: String = ""

    var body: some View {
        HStack {
            Text(title)
            Spacer()
            Stepper(value: $value, in: valueRange) {
                Text("\(value)\(valueSuffix)")
                    .monospacedDigit()
                    .foregroundStyle(.secondary)
            }
        }
    }
}
```

Архітектурний контракт: View, ViewModel, Detector

Дані від контролів не зв’язуються напряму з Core ML: вони потрапляють у AppViewModel через @Published властивості, і вже ViewModel перестворює або переконфігурує детектор передаючи нові confidence/iou. не чіпаючи камери. Така декомпозиція відповідає MVVM: UI чистий, вся логіка — у VM, ML-модуль отримує стабільний контракт на вході.

Лістинг 20 — Код класу AppViewModel.

```
final class AppViewModel: ObservableObject {
    @Published var confidenceThreshold: Float = 0.25
    @Published var iouThreshold: Float = 0.45
    @Published var showLabels: Bool = true
    @Published var selectedModel: ModelPreset = .fast
```

```
        let captureSession: AVCaptureSession
        @Published    private(set)    var    displayDetections:
[DetectionResult] = []

        // ML-ланцюжок
        private var detector: Detector?
        private var visionRequest: VNCoreMLRequest?
        private var labels: [String] = []

        init(session: AVCaptureSession,
             labelsURL: URL,
             factory: ModelFactory) {
            self.captureSession = session
            self.labels    =    (try?    LabelsLoader.load(from:
labelsURL)) ?? []
            // Ініціалізація детектора за поточним пресетом
            reconfigureDetector(factory: factory)
            // Підписка на зміни налаштувань
            setupBindings(factory: factory)
        }

        private func setupBindings(factory: ModelFactory) {
            // На зміну порогів — легка реконфігурація (без заміни
моделі)
            Publishers.CombineLatest($confidenceThreshold,
$iouThreshold)
                .dropFirst()
                .sink    {    [weak    self]    _,    _    in
self?.updateDetectorThresholds() }
                .store(in: &cancellables)

            // На зміну профілю моделі — повна переініціалізація
VNCoreMLModel
```

```
            $selectedModel
                .dropFirst()
                .sink      {      [weak      self]      _      in
self?.reconfigureDetector(factory: factory) }
                .store(in: &cancellables)
        }

        private func reconfigureDetector(factory: ModelFactory) {
            do {
                // Завантажуємо відповідний .mlpackage із бандла
за пресетом
                let vn = try factory.makeVisionModel() // у фабриці
можна враховувати selectedModel
                let det = Detector(vnModel: vn,
                                   labels: labels,
                                   confidence:
confidenceThreshold,
                                   iou: iouThreshold)
                self.detector = det
                self.visionRequest = det.makeRequest { [weak self]
results in
                    guard let self else { return }
                    DispatchQueue.main.async {
                        self.displayDetections = self.showLabels ?
results : results.map {
                            //  опційно  можемо  ховати  лейбли,
залишаючи рамки
                            DetectionResult(bbox:        $0.bbox,
classIndex: $0.classIndex,
                                            className:
self.showLabels ? $0.className : "",
                                            confidence:
$0.confidence)
```

```
                        }
                    }
                }
            } catch {
                print("Reconfigure error: \(error)")
            }
        }

        private func updateDetectorThresholds() {
            // Простий шлях: пересобрати Detector з тими ж
VNCoreMLModel і новими порогами
            guard let det = detector, let vn = det.value(forKey:
"vnModel") as? VNCoreMLModel else { return }
            self.detector = Detector(vnModel: vn,
                                     labels: labels,
                                     confidence:
confidenceThreshold,
                                     iou: iouThreshold)
            self.visionRequest = self.detector?.makeRequest {
[weak self] results in
                DispatchQueue.main.async                    {
self?.displayDetections = results }
            }
        }

        // Виклик з конвеєра камери:
        func performRequest(on pixelBuffer: CVPixelBuffer,
orientation: CGImagePropertyOrientation) {
            guard let req = visionRequest else { return }
            let handler = VNImageRequestHandler(cvPixelBuffer:
pixelBuffer,
                                                orientation:
orientation, options: [:])
```

```
            try? handler.perform([req])
        }

        private var cancellables: Set<AnyCancellable> = []
    }

    enum ModelPreset: String, CaseIterable { case fast, accurate }
```

Панель налаштувань завершує практичну інтеграцію: параметри інференсу стали керованими на пристрої, зміни застосовуються без перезапуску AVCaptureSession і без переекспорту моделі. Через MVVM-посередництво зберігається чистота архітектури: UI лишається декларативним і тонким, ділова логіка — у ViewModel, а ML-модуль ізолює всі обчислювальні деталі. Завдяки цьому застосунок готовий до польового використання на різних пристроях і у різних умовах зйомки, що критично для систем розпізнавання у реальному часі.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ МОБІЛЬНОЇ СИСТЕМИ IOS-ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ АВТОМОБІЛІВ У РЕАЛЬНОМУ ЧАСІ

4.1 Результати навчання та оцінка якості моделі розпізнавання

У цьому підрозділі наведено результати експериментальної перевірки якості роботи моделі детекції автомобілів, навченої в середовищі Ultralytics YOLOv8 та інтегрованої в iOS-застосунок. Оцінювання проводилось у два етапи:

1. кількісна оцінка на валідаційній вибірці (offline-режим), що дозволяє порівнювати якість моделі за формальними метриками;
2. прикладна оцінка на кадрах, наближених до реального сценарію використання, зйомка камерою мобільного пристрою, що дає уявлення про поведінку моделі в умовах доменного зсуву, інше освітлення, інший фон, відблиски, часткові перекриття об’єкта тощо.

4.1.1 Результати навчання та оцінка якості моделі розпізнавання

Базовим джерелом даних для навчання та первинної валідації слугував набір Stanford Cars (Cars196), який містить зображення автомобілів із розміченими обмежувальними рамками bounding boxes та назвами класів. Для сумісності з постановкою задачі object detection у YOLO-форматі дані було приведено до стандартного подання розмітки: для кожного зображення сформовано .txt-файл із записами виду <class_id> <cx> <cy> <w> <h> у нормалізованих координатах. Валідаційна вибірка формувалась як репрезентативний спліт train/val з часткою валідації 0.1, що забезпечує

однорідність домену між тренуванням і перевіркою, щоб вимірювати саме якість навчання, а не ефект зміни домену.

Окремо, з огляду на цільовий сценарій використання, розпізнавання у реальному часі з камери iPhone, було використано додатковий набір тестових прикладів, отриманих у “польових” умовах. На відміну від академічних датасетів, такі дані характеризуються:

- наявністю шумів (озмиття від руху, цифровий шум при низькому освітленні;
- складним фоном, який може містити об’єкти схожої форми/кольору;
- відблисками на лакофарбовому покритті або пластику;
- частковими перекриттями авто іншими предметами;
- варіативністю масштабу об’єкт займає відносно малу частину кадру.

Саме тому offline-валідація та прикладна перевірка на реальних кадрах розглядаються як взаємодоповнюючі підходи: перша дає формальну якість на контрольованих даних, друга — очікувану стабільність у застосунку.

4.1.2 Метрики оцінювання та критерії якості прогнозів

Для оцінювання якості моделі використовувались метрики, стандартні для задачі детекції об’єктів:

- mAP@0.50 (IoU=0.50) — узагальнена точність детекції за порогом перетину рамок 0.5;
- mAP@0.50:0.95 — більш “жорсткий” показник, що усереднюється по кількох порогах IoU з кроком 0.05 і краще відображає точність локалізації

об’єкта;

- Precision — частка коректних спрацювань серед усіх спрацювань;
- Recall — частка знайдених об’єктів серед усіх наявних у кадрі;
- F1-score — гармонійне середнє між Precision та Recall, корисне для підбору компромісу між пропусками та хибними спрацьовуваннями.

Оскільки цільова система працює у реальному часі, практично важливим є не лише факт наявності детекції, а й стабільність прогнозів у відеопотоці. Тому додатково аналізувались:

- чутливість до порогу confidence, як змінюються хибні спрацювання та пропуски при зміні threshold;
- стабільність рамки в сусідніх кадрах візуально та за непрямими ознаками: “миготіння” класу/відсотка.

4.1.3 Кількісні результати навчання

Після завершення навчання було отримано стандартні артефакти Ultralytics графіки метрик та results.csv, на основі яких сформовано підсумкові числові показники якості. Результати наведено на рисунку 11 та таблиці 9.

Таблиця 9 — Підсумкові метрики якості на валідаційній вибірці

Епоха	Precision	Recall	mAP@0.50	mAP@0.50:0.95	val/box_loss	val/cls_loss	val/dfl_loss
10	0.5327 (53.27 %)	0.6436 (64.36 %)	0.6665 (66.65 %)	0.6290 (62.90%)	0.3562	1.3131	0.9515
50	0.9441 (94.41 %)	0.9527 (95.27 %)	0.9847 (98.47 %)	0.9484 (94.84%)	0.2872	0.2404	0.8889
100	0.9779 (97.79 %)	0.9768 (97.68 %)	0.9924 (99.24 %)	0.9703 (97.03%)	0.2520	0.1532	0.8558

150	0.9861 (98.61 %)	0.9860 (98.60 %)	0.9940 (99.40 %)	0.9770 (97.70%)	0.2351	0.1347	0.8420
200	0.9881 (98.81 %)	0.9868 (98.68 %)	0.9941 (99.41 %)	0.9819 (98.19%)	0.2149	0.1250	0.8284
250	0.9888 (98.88 %)	0.9886 (98.86 %)	0.9939 (99.39 %)	0.9859 (98.59%)	0.1945	0.1184	0.8167
300	0.9884 (98.84 %)	0.9898 (98.98 %)	0.9939 (99.39 %)	0.9874 (98.74%)	0.1823	0.1114	0.8103

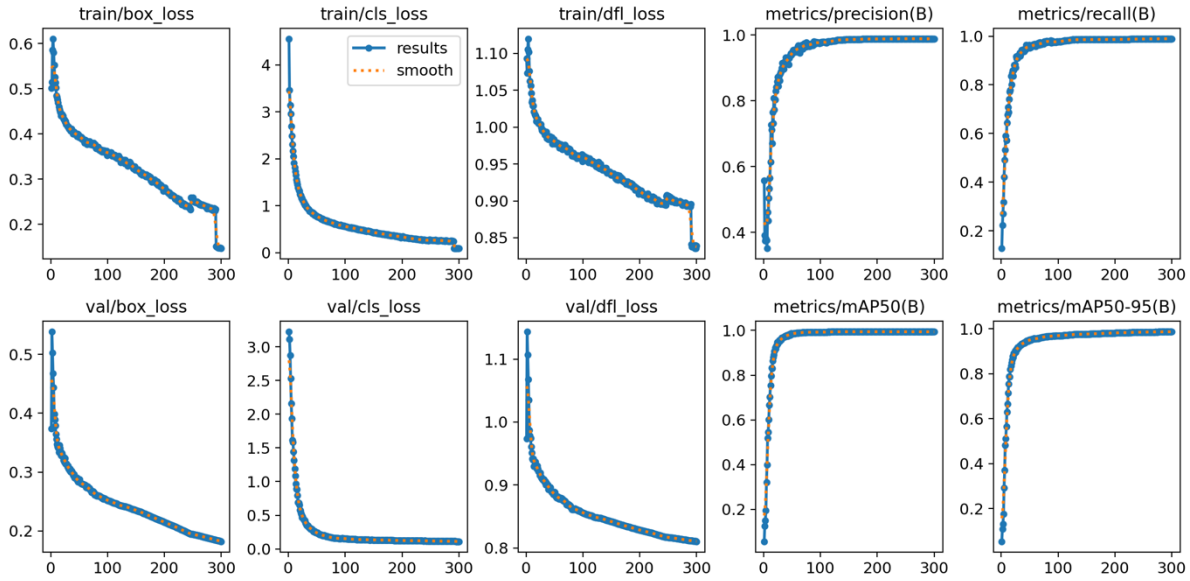


Рисунок 11 — Динаміка навчання та валідації (loss і метрики).

Аналіз кривих навчання дозволяє зробити такі узагальнення:

- зменшення train/val loss у процесі навчання свідчить про коректну збіжність;
- стабілізація mAP на пізніх епохах означає вихід моделі на плато якості;
- відсутність різкого розходження між train та val метриками є ознакою того, що перенавчання не домінує (або перебуває в межах допустимого

для задачі).

Окремо варто підкреслити, що для задачі тонкої диференціації коли класи можуть відрізнятися не типом кузова, а моделлю/роком mAP@0.50:0.95 часто є більш показовим, оскільки вимагає точнішої локалізації та відображає складність сцен із невеликими об’єктами та дрібними деталями.

4.1.4 Кількісні результати навчання

Для практичного використання у реальному часі важливо обґрунтувати робочий поріг confidence. Для цього аналізуються PR-крива та залежність F1 від confidence, оскільки F1 відображає компроміс між Precision і Recall. Графіки наведено на рисунку 12 та 13.

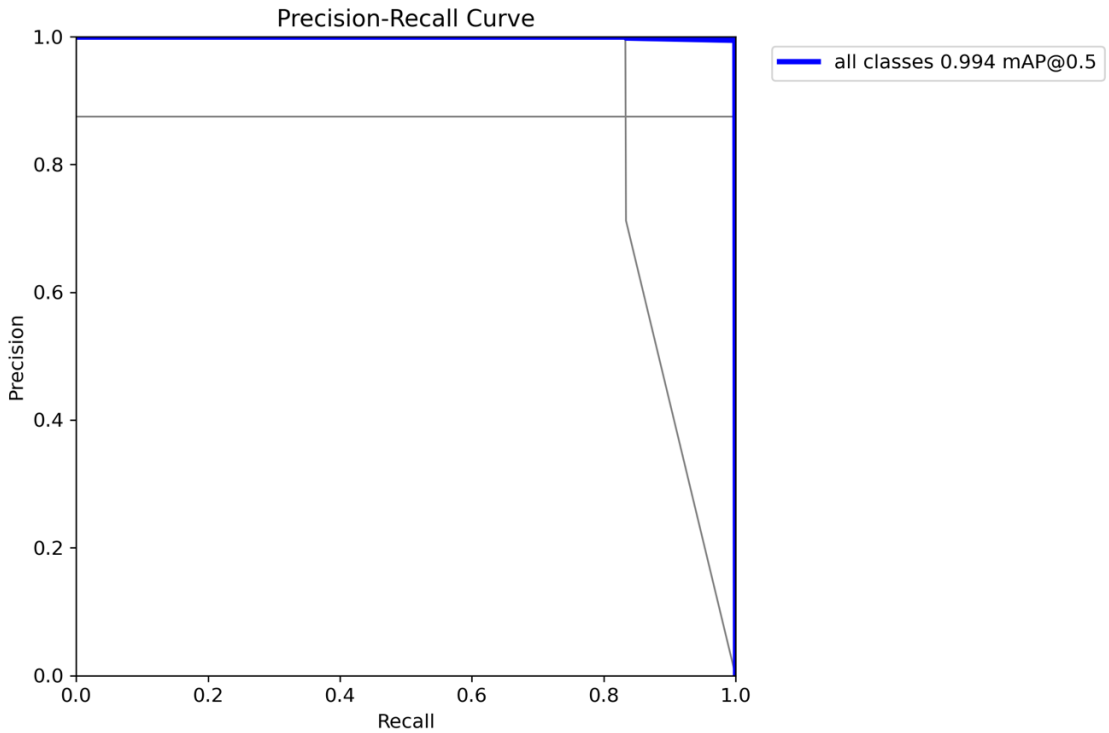


Рисунок 12 — PR-крива для моделі (оцінка компромісу Precision/Recall).

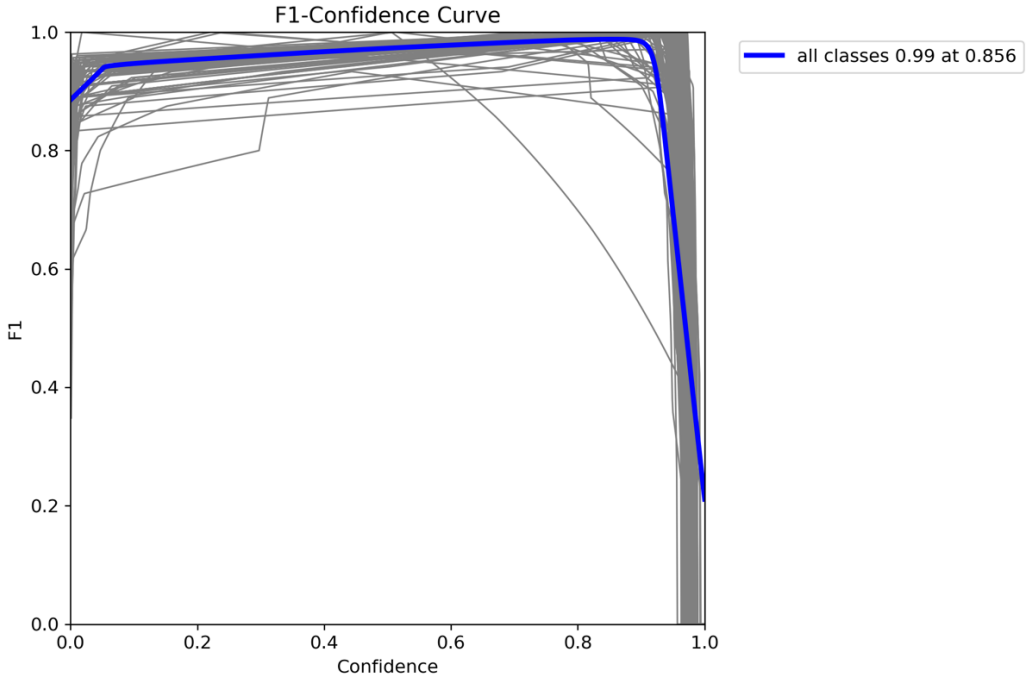


Рисунок 13 — *Залежність F1 від порогу* confidence *вибір робочого порог.*.

Окремо оцінювалась структура помилок за допомогою нормалізованої матриці невідповідностей. Вона дозволяє побачити, які саме класи автомобілів найчастіше плутаються між собою, типово це відбувається для візуально схожих моделей, за складного освітлення, при частковій оклюзії або на великих дистанціях. Графік наведено на рисунку 14.

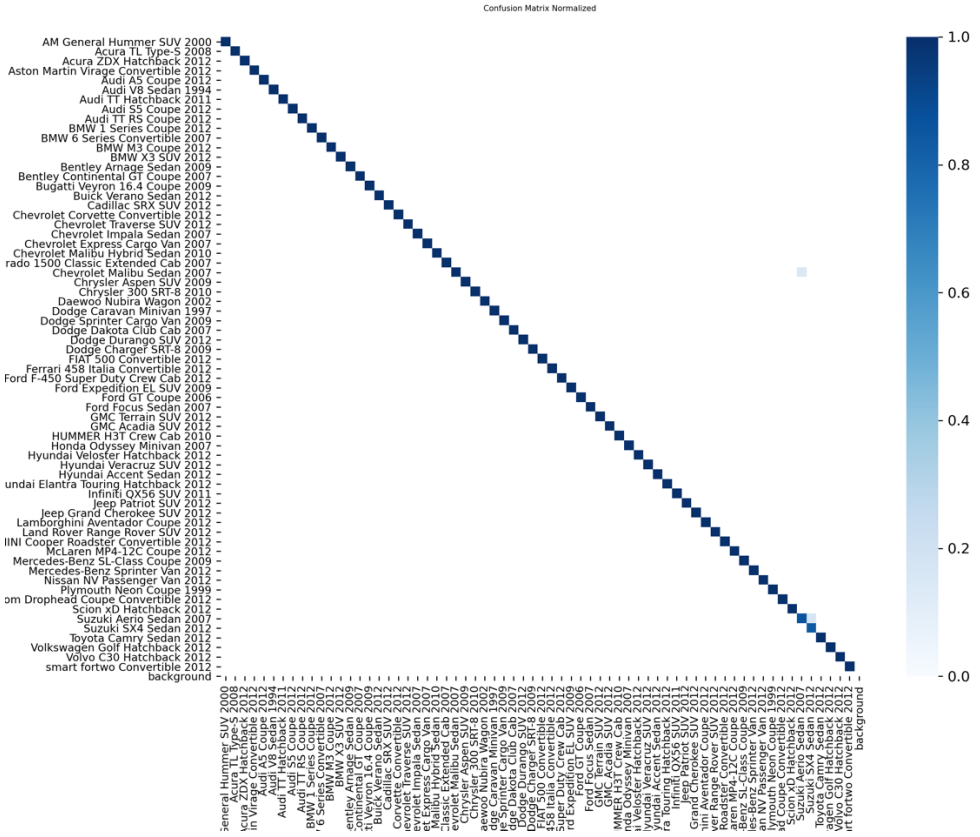


Рисунок 14 — *Нормалізована confusion matrix*.

Для перевірки практичної адекватності результатів також використовувався візуальний контроль прогнозів на прикладах валідаційної вибірки: порівнювались еталонні розмітки та детекції моделі, дивитись рисунок 15-16.



Рисунок 15 — *Нормалізована confusion matrix*.



Рисунок 16 — *Нормалізована confusion matrix*.

Загалом результати демонструють, що модель здатна стабільно виявляти автомобілі та коректно локалізувати їх у кадрі, що є достатньою основою для інтеграції у мобільний застосунок із обробкою відеопотоку в реальному часі.

4.2 Демонстрація роботи iOS-застосунку

Розроблений iOS-застосунок реалізує повний цикл роботи системи: отримання відеопотоку з камери, попередню обробку кадрів, виконання інференсу нейромережі та відображення результатів користувачу у режимі реального часу. Принципово важливо, що обробка виконується локально на пристрої, без надсилання зображень на сервер, що знижує затримки та підвищує конфіденційність даних.

4.2.1 Основний сценарій використання

Користувацький сценарій роботи застосунку складається з таких кроків:

1. запуск застосунку та ініціалізація модуля камери;
2. отримання дозволу на доступ до камери під час першого запуску;
3. безперервне зчитування кадрів та виконання інференсу моделі;
4. накладення рамок детекції на зображення та виведення підписів класу/впевненості;
5. можливість змінювати ключові параметри confidence threshold тощо для стабільної роботи в різних умовах.



Рисунок 17 — *Головний екран застосунку з активною камерою*.

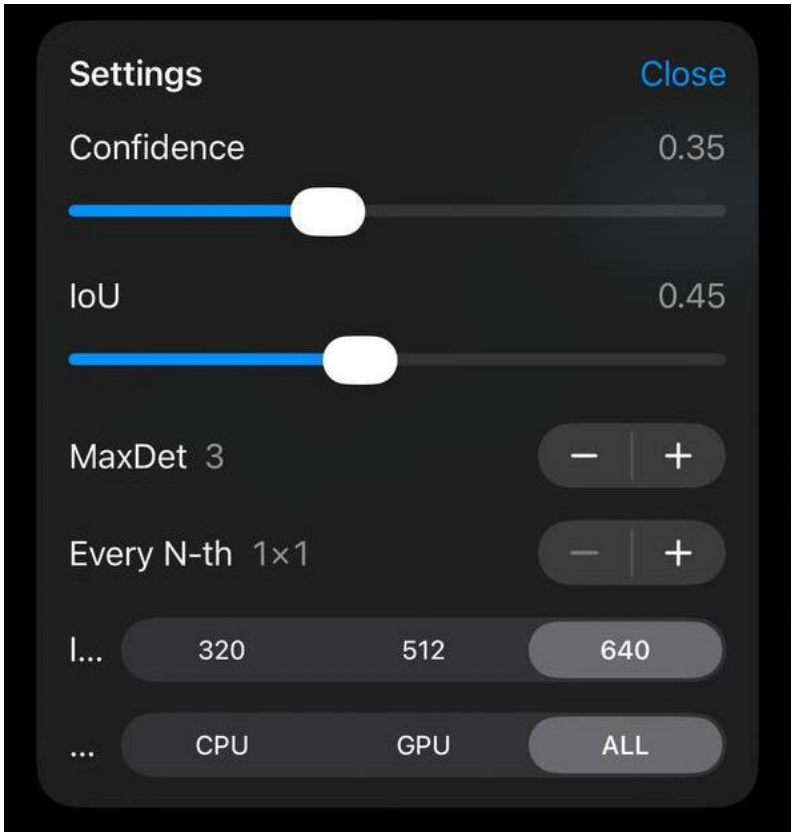


Рисунок 18 — *Екран застосунку з панеллю налаштувань*.

4.2.2 Можливості подальшого розширення та універсальність рішення

Запропоноване рішення є універсальним у тому сенсі, що при збереженні архітектури застосунку можна розширювати систему без повної переробки:

- розширення датасету та класів (нові марки/моделі/типи кузова);
- покращення стабільності у відеопотоці, трекинг між кадрами;
- адаптація до складних умов нічні сцени, дощ, розмиття руху за рахунок цільових аугментацій і донавчання;
- оптимізація продуктивності, квантування, інша роздільна здатність, вибір формату моделі під Neural Engine;
- розширення UX збереження кадру/результату, історія розпізнавань,

експорт, AR-накладення тощо.

Таким чином, навіть у поточному вигляді застосунок демонструє працездатну основу, яку можна масштабувати під ширший перелік сценаріїв та класів об’єктів.

Висновки до розділу 4

У розділі виконано оцінювання якості прогнозів моделі детекції автомобілів за допомогою стандартних метрик (Precision, Recall, mAP@0.50 та mAP@0.50:0.95), проаналізовано динаміку навчання та візуальні приклади детекцій. Показано, що модель досягає високих значень узагальненої точності та демонструє стабільну локалізацію об’єктів, що є критично важливим для реального часу. Також продемонстровано роботу iOS-застосунку, який інтегрує модель та забезпечує локальне розпізнавання з камери в реальному часі, підтверджуючи практичну придатність запропонованого підходу.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи досягнуто поставленої мети — розроблено автономний iOS-застосунок для розпізнавання автомобілів у реальному часі на основі нейронної мережі з інтеграцією через Core ML та Vision і використанням відеопотоку з камери пристрою.

1. Обґрунтовано актуальність задачі мобільного розпізнавання об’єктів у реальному часі та доцільність локального (on-device) виконання моделі з точки зору автономності та конфіденційності даних користувача.

2. Реалізовано повний цикл підготовки даних для навчання: вихідний набір Cars196 було перетворено у канонічний формат YOLO структура train/val + текстові анотації рамок, сформовано train/val-спліт, а також підготовлено labels.json для коректного мапінгу індексів класів у назви категорій у мобільному застосунку.

3. Навчено модель детекції на базі Ultralytics YOLOv8 конфігурація зображення 640 та 300 епох, що забезпечило отримання стабільних артефактів навчання ваги, графіки, results.csv та підготовило модель до подальшого розгортання на iOS.

4. Виконано експорт моделі у формат Core ML (.mlpackage) та прийнято інженерно обґрунтоване рішення вимкнути вбудований NMS під час експорту, щоб виконувати постобробку на пристрої та гнучко налаштовувати пороги під конкретний сценарій і продуктивність девайса.

5. Реалізовано механізм адаптації до різних схем виходів Core ML-моделі, різні назви ключів та форми тензорів, що підвищує універсальність рішення: застосунок може працювати з перевивантаженою/переекспортованою моделлю без потреби переписувати код під “технічні” імена виходів.

6. Для оцінювання якості моделі застосовано стандартні метрики

детекції (Precision, Recall, mAP@0.50, mAP@0.50:0.95, F1) та окремо враховано вимоги реального часу, чутливість до confidence-порога та стабільність прогнозів у відеопотоці.

7. За підсумками навчання на валідаційній вибірці отримано високі кількісні показники якості: на 300-й епосі Precision = 0.9884, Recall = 0.9898, mAP@0.50 = 0.9939, mAP@0.50:0.95 = 0.9874; при цьому значення валідаційних втрат зменшуються, що в сукупності вказує на коректну збіжність і вихід моделі на плато якості.

8. Практичний результат роботи — iOS-застосунок, який виконує розпізнавання автомобілів локально, без зовнішнього сервера, та може бути використаний як основа для спеціалізованих систем класифікації/детекції в прикладних доменах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Климов Д.В., магістрант, Міхайлуца О. М., доцент — науковий керівник. Розпізнавання об’єктів в реальному часі в iOS-застосунках з використанням Core ML. *Молода наука-2025* : зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2025. Т. 5. С. 475–476.

2. Apple Inc. Core ML Documentation. URL: https://developer.apple.com/documentation/coreml (дата звернення: 15.04.2025).

3. Apple Inc. Vision Framework. URL: https://developer.apple.com/documentation/vision (дата звернення: 15.04.2025).

4. Apple Inc. Create ML Documentation. URL: https://developer.apple.com/documentation/createml (дата звернення: 15.04.2025).

5. Krizhevsky A., Sutskever I., Hinton G. ImageNet Classification with Deep Convolutional Neural Networks // Advances in Neural Information Processing Systems. 2012. Vol. 25. P. 1097–1105.

6. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition // International Conf. on Learning Representations. San Diego, 2015.

7. Redmon J., Farhadi A. YOLOv3: An Incremental Improvement. 2018. URL: https://arxiv.org/pdf/1804.02767.pdf (дата звернення: 16.04.2025).

8. Liu W., Anguelov D., Erhan D. SSD: Single Shot MultiBox Detector // European Conference on Computer Vision. Amsterdam, 2016. P. 21–37.

9. Howard A. et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. URL: https://arxiv.org/pdf/1704.04861.pdf (дата звернення: 16.04.2025).

10. Iandola F., Han S. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters. 2016. URL: https://arxiv.org/pdf/1602.07360.pdf (дата звернення:

16.04.2025).

11. Hollance M. Core ML Survival Guide: How to use machine learning in your iOS apps. 2022. URL: https://machinethink.net/blog/coreml-survival-guide/ (дата звернення: 16.04.2025).

12. Turing M. Converting YOLOv5 to Core ML using ONNX and coremltools. Medium. 2021. URL: https://medium.com/@turing/convert-yolov5-coreml (дата звернення: 17.04.2025).

13. Cao Z., Hidalgo G., Simon T. OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields // IEEE Transactions on Pattern Analysis and Machine Intelligence. 2019. Vol. 43, No 1. P. 172–186.

14. Apple Inc. Machine Learning with Core ML 3. WWDC 2019 Session. URL: https://developer.apple.com/videos/play/wwdc2019/703/ (дата звернення: 17.04.2025).

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Климов Данило Віталійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-203@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Розпізнавання об'єктів в реальному часі в iOS-застосунках з використанням Core ML**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 01.12.2025 Підпис _____ Климов Данило Віталійович

(студент)

Дата 01.12.2025 Підпис _____ Міхайлуца Олена Миколаївна

(науковий керівник)