ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття ступеня вищої освіти «магістр»
(освітньо-кваліфікаційний рівень)

на тему Використання методів комп'ютерного зору для розпізнавання жестів з урухуванням характеристик апаратного середовища
The use of computer vision methods for gesture recognition considering hardware environment characteristics

Виконав: студент денної форми навчання
спеціальності 123 – Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
Освітня програма «Комп'ютерна інженерія»
(назва освітньої програми)
Осипов Артем Валентинович
(прізвище, ім'я, по-батькові)
Керівник к.ф.-м.н., доц. Шпінарева І. М.
(науковий ступінь, вчене звання, прізвище та ініціали, підпис)
Рецензент к.техн.н., доц. Пенко В.Г.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___» _____ 2024 р.

Завідувач кафедри
_____ Євгеній МАЛАХОВ
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___
протокол №__від «___» _____ 2024 р.
Оцінка _____/_____/_____
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК
_____ Алла КОБОЗЄВА
(підпис) (ім'я, прізвище)

Одеса – 2024

АНОТАЦІЯ

Кваліфікаційна робота присвячена процесу розпізнавання жестів у відео з урахуванням апаратних та програмних засобів.

Метою даної роботи є підвищення ефективності розпізнавання жестів у відео, шляхом вибору моделі глибокого навчання для конкретного пристрою на основі аналізу характеристик апаратного середовища та продуктивності моделей комп'ютерного зору.

В сучасних умовах методи комп'ютерного зору широко застосовуються у різних сферах. Використання цих технологій для розпізнавання жестів сприяє покращенню взаємодії, автоматизації управління та розвитку інклюзивних технологій. Важливу роль відіграє врахування характеристик апаратного середовища, що дозволяє створювати системи з високою точністю й швидкістю роботи, оптимізуючи їх для пристроїв з обмеженими ресурсами.

У роботі досліджено методи комп'ютерного зору, включаючи алгоритми виявлення об'єктів, такі як single shot detection. Для навчання моделей використано набір даних «HaGRID», один із найповніших для задач розпізнавання жестів. Моделі протестовані на системах з різним апаратним забезпеченням, результати порівняно. Розроблено клієнт-серверний додаток для підбору найкращої конфігурації та використання моделей розпізнавання жестів.

Аналіз цих методів є важливим для розвитку технологій взаємодії людини з машиною. Апаратно свідомі системи мають потенціал покращити якість життя, забезпечуючи ефективність, доступність та зручність у використанні навіть на пристроях із обмеженими ресурсами.

ABSTRACT

The qualification work is dedicated to the process of gesture recognition in video, taking into account hardware and software resources.

The aim of this work is to improve the efficiency of gesture recognition in video by selecting a deep learning model for a specific device based on an analysis of the hardware environment characteristics and the performance of computer vision models.

In the modern world, computer vision methods are widely used in various fields. Utilizing these technologies for gesture recognition enhances interaction, automates control, and fosters the development of inclusive technologies. Considering hardware characteristics plays a significant role in creating systems with high accuracy and speed, optimizing them for devices with limited resources.

The work explores computer vision methods, including object detection algorithms such as Single Shot Detection. The «HaGRID» dataset, one of the most comprehensive for gesture recognition tasks, was used for model training. The models were tested on systems with different hardware configurations, and the results were compared. A client-server application was developed to select the optimal configuration and to use gesture recognition models.

The analysis of these methods is essential for the development of human-machine interaction technologies. Hardware-aware systems have the potential to improve quality of life by ensuring efficiency, accessibility, and ease of use, even on devices with limited resources.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ 7

ВСТУП 8

1 ОГЛЯД ІСНУЮЧИХ ТЕХНОЛОГІЙ ТА МЕТОДІВ РОЗПІЗНАВАННЯ ЖЕСТІВ 11

1.1 Технології апаратно-залежного машинного навчання 11

1.2 Методи розпізнавання жестів 13

2 ПРОЕКТУВАННЯ СИСТЕМИ РОЗПІЗНАВАННЯ ЖЕСТІВ 19

2.1 Моделі MobileNetV2 19

2.1.1 MobileNetV2 SSD FPN-Lite 19

2.1.2 MobileNetV2 21

2.2 Моделі YOLOv10 та MediaPipe 22

2.3 Набір даних «HaGRID» 23

2.4 Загальна архітектура системи 27

2.5 Стек технологій 29

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ РОЗПІЗНАВАННЯ ЖЕСТІВ 31

3.1 Основна програмна частина 31

3.2 Реалізація веб-сервісу на базі FastAPI і Uvicorn 32

3.2.1 Архітектура системи розпізнавання жестів 33

4 ТЕСТУВАННЯ СИСТЕМИ 35

4.1 Тестування моделей розпізнавання жестів 35

4.2 Апаратно-залежний вибір моделі 37

4.3 Тестування апаратно-залежного вибору моделі 39

ВИСНОВКИ 41

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 42

ДОДАТОК А Класи моделей розпізнавання жестів 44

ДОДАТОК Б Сервер системи 49

ДОДАТОК Г Клас процесу розпізнавання жестів 54

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

ASGI (asynchronous server gateway interface) – інтерфейс шлюзу асинхронного сервера.

CNN (convolutional neural network) – згорткова нейронна мережа.

FPS (frames per second) – кадри у секнуду.

NNCF (neural network compression framework) – фреймворк стиснення нейронних мереж.

SSD – Single Shot Detection.

Бенчмарк – контрольне завдання, необхідне для визначення порівняльних характеристик продуктивності комп'ютерної системи.

ВСТУП

Технології розпізнавання об’єктів, які належать до галузі комп’ютерного зору, є одним із ключових напрямів розвитку штучного інтелекту. Вони дозволяють автоматизувати процеси, що раніше вимагали значних зусиль людини, такі як ідентифікація об’єктів, людей або патернів у зображеннях і відео. Завдяки їх широкому впровадженню, ці технології стали невід’ємною частиною багатьох індустрій, включаючи охорону здоров’я, транспорт, промисловість, роздрібну торгівлю та безпеку.

Популярність та актуальність цих технологій підтверджується статистикою. У 2023 році обсяг світового ринку розпізнавання зображень оцінювався в 53.3 мільярда доларів США і, за прогнозами, зростатиме зі середньорічним темпом зростання 12.8% у період з 2024 по 2030 рік [1].

Актуальність цих рішень обумовлена не лише їхньою здатністю автоматизувати рутинні процеси, але й тим, що вони підвищують рівень безпеки, оптимізують роботу та дозволяють ефективніше використовувати ресурси. Швидкий розвиток апаратного забезпечення, зокрема графічних процесорів, та доступ до великих обсягів даних для навчання моделей сприяють постійному вдосконаленню цих технологій. Завдяки цьому розпізнавання об’єктів залишається одним із найбільш динамічних напрямів у сфері штучного інтелекту.

Розпізнавання жестів є важливим напрямком у технології розпізнавання об’єктів, що сприяє створенню більш природної взаємодії між людиною та машиною. Завдяки можливості розуміти рухи користувача, такі системи дозволяють виконувати команди без фізичного контакту з пристроєм. Це стало особливо актуальним у період пандемії, коли попит на безконтактні рішення значно зріс. Наприклад, у автомобільній індустрії технологія розпізнавання жестів допомагає водіям керувати мультимедійними системами чи іншими функціями без потреби відривати руки від керма. Інновації у цій сфері активно

впроваджуються такими компаніями, як Cerence Inc., яка у 2020 році представила платформу для безкнопкових інтерфейсів [2]. У медицині розпізнавання жестів дозволяє лікарям взаємодіяти з обладнанням у стерильних умовах, а пацієнтам із фізичними обмеженнями – керувати пристроями за допомогою рухів рук. У роздрібній торгівлі ця технологія використовується для покращення користувацького досвіду через інтерактивні вітрини, а в ігровій індустрії створює більш занурюючий досвід у віртуальній реальності.

Економічна актуальність технології підтверджується її стрімким розвитком. У 2023 році ринок розпізнавання жестів оцінювався в 21.04 мільярда доларів США, і за прогнозами до 2030 року його обсяг зросте до 70.18 мільярда доларів із середньорічним темпом зростання 18.8%. Особливий попит спостерігається в регіонах Азії, які займають 36.7% світового ринку, що обумовлено зростанням індустріалізації та високою чисельністю населення [3]. При цьому основними напрямками розвитку залишаються автомобільна та медична галузі.

Однак технологія стикається з низкою викликів. По-перше, це необхідність високої обчислювальної потужності для обробки жестів у реальному часі. По-друге, на результативність впливають зовнішні умови, такі як освітлення та наявність перешкод. По-третє, розробка та впровадження систем вимагають значних фінансових інвестицій. Попри це, розпізнавання жестів продовжує залишатися перспективною технологією, що змінює підхід до взаємодії з цифровими пристроями та системами, роблячи його більш інтуїтивним і доступним для широкого кола користувачів.

Метою дипломної роботи є підвищення ефективності розпізнавання жестів у відео, шляхом вибору моделі глибокого навчання для конкретного пристрою на основі аналізу характеристик апаратного середовища та продуктивності моделей комп’ютерного зору.

Об’єкт дослідження – процес розпізнавання жестів у відео з

урахуванням апаратних та програмних засобів.

Предметом дослідження є методи та алгоритми інтелектуального аналізу при розпізнаванні жестів у відео.

Для досягнення мети необхідно виконати такі задачі:

1) дослідження сучасних методів та моделей розпізнавання жестів;

2) розробка різних архітектур моделей розпізнавання жестів;

3) реалізація модифікації обраних моделей розпізнавання жестів та аналіз роботи конфігурацій моделей на системах з різними апаратним забезпеченням;

4) тестування роботи моделей розпізнавання жестів в різних апаратних середовищах;

5) реалізація системи, яка на основі характеристик апаратного середовища і проведеного аналізу буде видавати рекомендовану конфігурацію моделей розпізнавання жестів.

1 ОГЛЯД ІСНУЮЧИХ ТЕХНОЛОГІЙ ТА МЕТОДІВ РОЗПІЗНАВАННЯ ЖЕСТІВ

1.1 Технології апаратно-залежного машинного навчання

Головним технічним викликом сучасних моделей машинного навчання пов'язані зі складністю адаптації моделей до обмежених ресурсів апаратного забезпечення. Це породжує попит на автоматизовані інструменти, здатні забезпечувати ефективність моделей без значної втрати продуктивності. У цьому контексті виникають рішення, такі як AutoQ, BootstrapNAS, та інші, спрямовані на оптимізацію та вибір нейронних мереж під специфічні апаратні обмеження.

Intel Labs активно досліджує методи оптимізації штучного інтелекту для апаратних платформ. Їхні проекти, як Hardware-Aware Automated Machine Learning (апаратно-орієнтоване автоматизоване машинне навчання), зосереджені на створенні автоматизованих інструментів для компресії моделей, архітектурного пошуку та ефективної адаптації до конкретних пристроїв. Наприклад, їхня NNCF надає комплексні інструменти для компресії моделей, зокрема через автоматизований пошук архітектур [4].

AutoQ – це автоматизований інструмент змішаної точності, інтегрований у платформу NNCF від Intel Labs [5]. Він використовує алгоритм Deep Deterministic Policy Gradient (DDPG) для навчання оптимальної політики змішаної точності через епізодичний процес. AutoQ здатен визначати точність для кожного шару нейронної мережі, щоб забезпечити компресію з мінімальною втратою точності. Система дозволяє користувачам налаштовувати цільові параметри, як співвідношення компресії, що є ключовим для оптимізації під конкретне апаратне забезпечення. Наприклад, при роботі з моделями для ImageNet система може зменшувати точність до 4-бітної з утриманням високої якості моделі. Також AutoQ є корисним для розробників, які потребують швидких рішень для апаратно-залежного

компресування.

BootstrapNAS є інструментом Neural Architecture Search (NAS), розробленим Intel Labs для автоматизації створення супермереж із розділеними вагами [6]. Основна ідея полягає у зменшенні обчислювальних витрат за допомогою спільного використання ваг між різними архітектурами. Система оптимізує архітектури нейронних мереж, враховуючи обмеження апаратного забезпечення, та забезпечує збереження високої продуктивності навіть у компресованих моделях. Використання BootstrapNAS дозволяє значно скоротити час і ресурси, необхідні для пошуку архітектур, забезпечуючи баланс між продуктивністю та апаратною ефективністю. Цей підхід особливо ефективний для великих мовних моделей і завдань комп'ютерного зору.

Ще один приклад – Azure AI, який надає рекомендації з вибору моделей для конкретних завдань, з урахуванням їхньої продуктивності, вартості та сумісності з інфраструктурою користувача [7]. Інструмент Azure AI Studio дозволяє тестувати та порівнювати моделі на різних апаратних платформах, що спрощує вибір оптимального рішення для додатків, таких як обробка мовлення, зображень чи текстів. Ця екосистема дозволяє швидко адаптуватися до різних обчислювальних платформ, включаючи хмарні сервери та IoT-пристрої, розширюючи можливості інтеграції оптимізованих моделей у виробничі середовища.

Існує також універсальна платформа, яка використовується для побудови високопродуктивних обчислювальних конвеєрів, що аналізують відео, аудіо або сенсорні дані – MediaPipe [8]. Її унікальна архітектура забезпечує ефективну обробку даних у реальному часі навіть на пристроях з обмеженими апаратними ресурсами. Одним із ключових досягнень MediaPipe є можливість створювати мультимодальні рішення за допомогою модульного підходу. Кожен компонент, або «калькулятор», виконує окрему функцію в рамках обчислювального графа, дозволяючи розробникам інтегрувати й оптимізувати рішення під конкретні задачі у рамках однієї системи.

MediaPipe демонструє високу гнучкість і масштабованість завдяки підтримці різних платформ, включно з мобільними пристроями та браузерами. Оптимізація обчислень і використання GPU робить її однією з найефективніших платформ для розробки систем комп’ютерного зору. Крім того, розробники мають доступ до вбудованих інструментів для візуалізації та налагодження, що сприяє швидкому створенню та тестуванню нових моделей. Усе це робить MediaPipe одним з найпопулярнішим інструментом у задачах аналізу людських рухів і жестів, забезпечуючи високу точність при мінімальних витратах обчислювальних ресурсів.

1.2 Методи розпізнавання жестів

Згорткові нейронні мережі є основою багатьох моделей комп'ютерного зору [9]. Основними елементами CNN є згорткові шари, які витягують просторові особливості зображень, і шари субдискретизації/пулінгу (pooling), що знижують розмірність, зберігаючи суттєві риси. Для задач розпізнавання жестів, вибір глибини моделі та кількості фільтрів у шарах CNN є критично важливим.

Для покращення роботи згорткових нейронних мереж застосовуються додаткові методи та шари, які значно оптимізують процес навчання моделей і підвищують їхню ефективність. Одним із таких ключових компонентів є нормалізація пакетів (batch normalization), який виконує нормалізацію активацій кожного шару нейронної мережі. Цей метод коригує середнє значення і стандартне відхилення в кожному міні-пакеті, забезпечуючи стабільний розподіл вхідних даних для кожного шару. Це суттєво зменшує чутливість до початкових значень ваг і забезпечує швидшу збіжність моделі під час навчання. Крім того, нормалізація пакетів знижує ймовірність переобчислень градієнтів, що дозволяє застосовувати вищі значення навчальної швидкості. У результаті модель стає стійкішою до коливань під час навчання і показує кращі результати на валідаційних даних.

Шар усередненого пулінгу 2D-даних (average pooling 2D), а саме його модифікація – усереднений глобальний пулінг (global average pooling 2D) – є важливим інструментом для зменшення кількості параметрів моделі, особливо у великих нейронних мережах, що працюють із зображеннями (рис. 1.1).

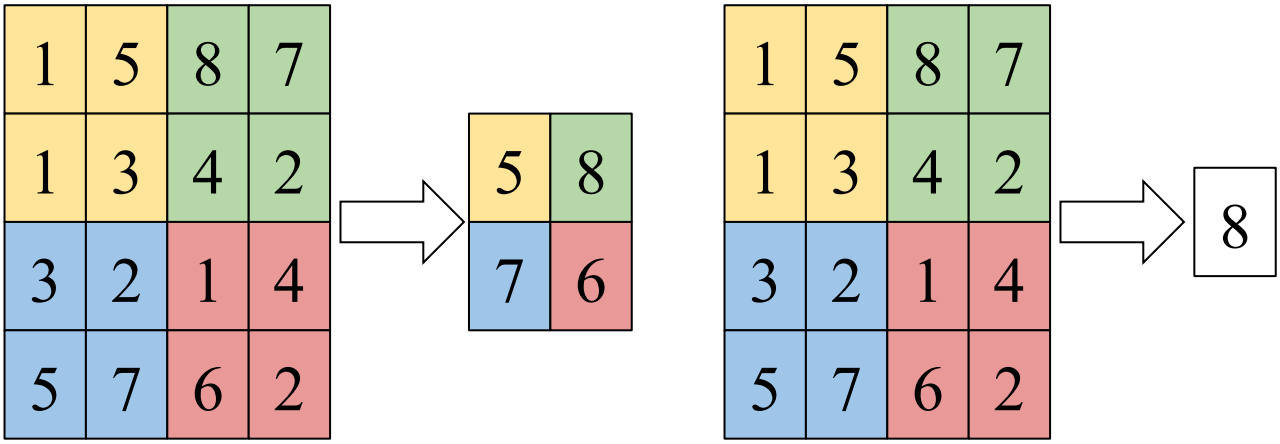

Рисунок 1.1 – а) усереднений пулінг, б) усереднений глобальний пулінг

GlobalAveragePooling2D обчислює середнє значення для кожного каналу вхідного зображення, повністю усуваючи просторові розміри (висоту та ширину), що дає змогу отримати компактний вектор ознак для класифікації. Це знижує кількість параметрів, запобігає перенавчанню та є стійким до змін у розташуванні об'єктів на зображенні. У свою чергу, шар звичайного усередненого пулінгу зменшує розмірність, обчислюючи середнє в межах вікна заданого розміру, яке переміщається по зображенню. Цей метод зберігає просторову інформацію у зменшеній карті ознак, що є корисним для завдань, де важливі локальні деталі.

Глибинна 2D згортка (depthwise convolution 2D) – це оптимізований тип згорткових шарів, який суттєво знижує обчислювальні витрати моделі. На відміну від стандартної згортки, яка застосовує фільтри одночасно до всіх каналів вхідного зображення, глибинна виконує окрему згортку для кожного каналу. Потім результати поєднуються через операцію згортки між каналами.

Такий підхід значно знижує кількість обчислень і розмір моделі, роблячи її придатною для використання на мобільних пристроях та вбудованих системах, де обчислювальні ресурси обмежені. Глибинна згортка забезпечує оптимальне співвідношення між швидкістю і точністю, що робить його популярним у легких нейронних мережах.

Шар «вузького місця» (bottleneck) (рис. 1.2) є невід'ємною частиною багатьох сучасних архітектур нейронних мереж. Вони спрямовані на зменшення розмірності простору ознак перед виконанням основних операцій, таких як згортка.

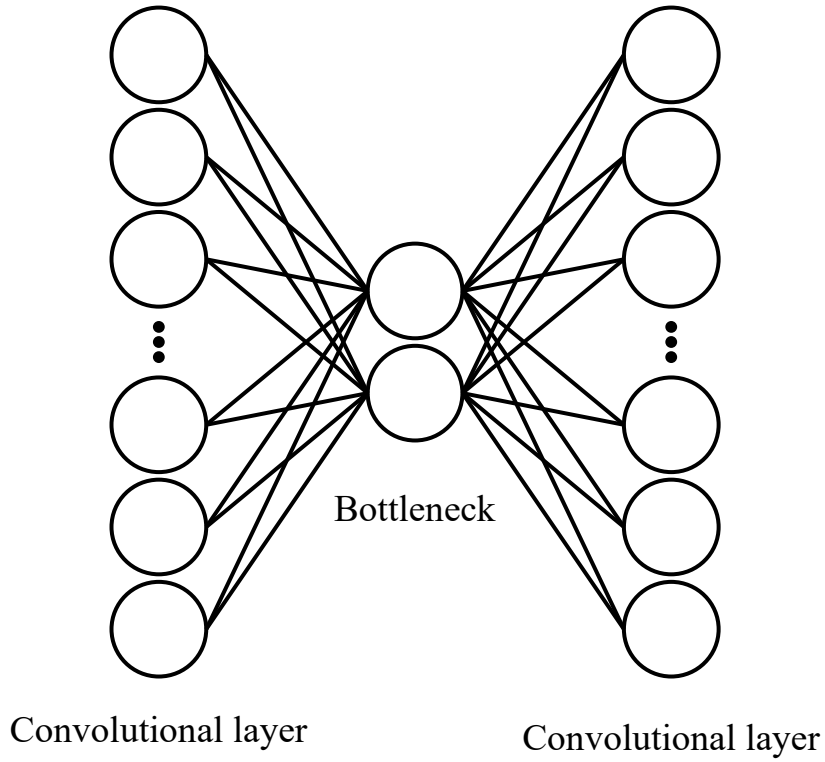

Рисунок 1.2 – Шар Bottleneck

Це дозволяє істотно скоротити кількість параметрів і обчислювальних витрат, особливо у випадках роботи з глибокими мережами. У bottleneck шарах основна увага приділяється стиску даних перед застосуванням фільтрів високої розмірності, що забезпечує баланс між продуктивністю моделі і її ефективністю.

Застосування таких шарів дозволяє будувати глибокі мережі з мінімальними втратами обчислювальної продуктивності, що є критичним для високоточних задач розпізнавання, включаючи розпізнавання жестів у реальному часі.

Для підвищення точності розпізнавання жестів, особливо у випадках з варіативним фоном, застосовуються методи виділення руки з зображення. Це дозволяє зосередити увагу на релевантних ділянках, значно знижуючи шумові впливи.

SSD (Single Shot Detector) є методом одноетапного виявлення об'єктів, який інтегрує всі необхідні етапи у єдиний нейронний мережевий процес. Основною ідеєю SSD є використання множинних шарів особливостей (feature maps) для передбачення об'єктів різного розміру, без необхідності генерувати попередні пропозиції (bounding box proposals), як це робиться у більш складних підходах, таких як Faster R-CNN.

Цей метод дискретизує простір вихідних координат рамки у фіксований набір «стандартних» рамок з різними співвідношеннями сторін та масштабами. У момент передбачення мережа генерує ймовірності для кожної категорії об'єкта в кожному з стандартних рамок, а також коригує їх положення та розмір для точнішого узгодження з формою об'єкта. Використання декількох карт особливостей з різною роздільною здатністю дозволяє SSD обробляти як великі, так і малі об'єкти.

На противагу SSD, двоетапні моделі, як Faster R-CNN, виконують два ключових етапи:

1) генерація регіональних пропозицій (Region Proposal): Використовуються регіональні пропозиції (RPN), які створюють попередні рамки-кандидати для кожного об'єкта в зображенні. Цей етап забезпечує точність шляхом виділення потенційно важливих ділянок;

2) класифікація та регресія: Кожна згенерована рамка передається через другу частину мережі для уточнення його координат і класифікації.

Одноетапна модель є швидшою за двоетапну, оскільки обходить етапи

повторного семплювання пікселів чи особливостей, зберігаючи високу якість виявлення завдяки оптимізації архітектури і використанню багаторівневого прогнозування.

Однак завдяки окремому етапу попередніх пропозицій двоетапна модель точніше ідентифікує межі об'єктів. Вона краще справляється зі складними сценами, де об'єкти можуть перекриватися або мати різну форму.

Для вирішення поставлених завдань розробки системи розпізнавання ключових точок та виявлення руки на зображеннях, обрано архітектури MobileNetV2 для локалізації ключових точок та SSDMobileNetV2 для детекції руки [10].

MobileNetV2 є легковажною архітектурою, розробленою для мобільних та вбудованих пристроїв (рис. 1.3).

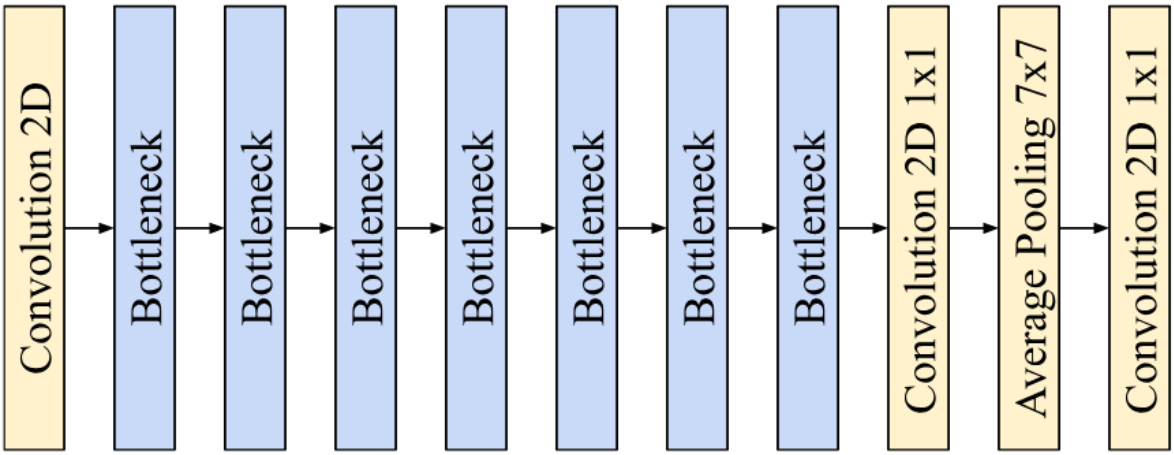

Рисунок 1.3 – Архітектура MobileNetV2

Основу її структури складають інвертовані залишкові блоки з лінійними «вузькими місцями», що забезпечують ефективне використання ресурсів. Архітектура включає глибокі згортки (depthwise convolutions), які знижують кількість обчислень, та полегшують навчання завдяки використанню shortcut-з'єднань між вузькими прошарками. Це дозволяє MobileNetV2 досягати високої точності при мінімальних обчислювальних затратах, роблячи її ідеальною для задач розпізнавання ключових точок на обмежених ресурсах.

SSDMobileNetV2 поєднує MobileNetV2 як екстрактор ознак із архітектурою Single Shot Detector (SSD) для одноетапного виявлення об’єктів. Використання SSDLite, оптимізованого варіанту SSD. Така архітектура працює швидко навіть на мобільних пристроях і демонструє високу ефективність у задачах детекції об’єктів, що підтверджено її продуктивністю на реальних наборах даних.

Ці моделі обрано через їхню готовність до інтеграції в систему та можливість налаштування під конкретні потреби. Їхня оптимізована структура дозволяє виконувати розпізнавання жестів із достатньо високою точністю, забезпечуючи при цьому низькі затрати обчислювальних ресурсів, що є ключовою вимогою для задач із використанням великої кількості різних пристроїв.

2 ПРОЕКТУВАННЯ СИСТЕМИ РОЗПІЗНАВАННЯ ЖЕСТІВ

2.1 Моделі MobileNetV2

2.1.1 MobileNetV2 SSD FPN-Lite

Як зазначено в Розділі 1, основною моделлю системи розпізнавання жестів обрана MobileNetV2. Для виділення руки за основу використана MobileNetV2 SSD FPN-Lite, а для знаходження ключових точок звичайна MobileNetV2.

Архітектура MobileNetV2 SSD FPN-Lite (рис. 2.1) використовує згортку, що розділяється за глибиною, зменшуючи кількість параметрів і обчислень порівняно з традиційними згортками.

FPN (Feature Pyramid Network – мережа піраміди ознак) удосконалює SSD, додаючи багатомасштабну екстракцію ознак, яка необхідна для виявлення об'єктів різного розміру.

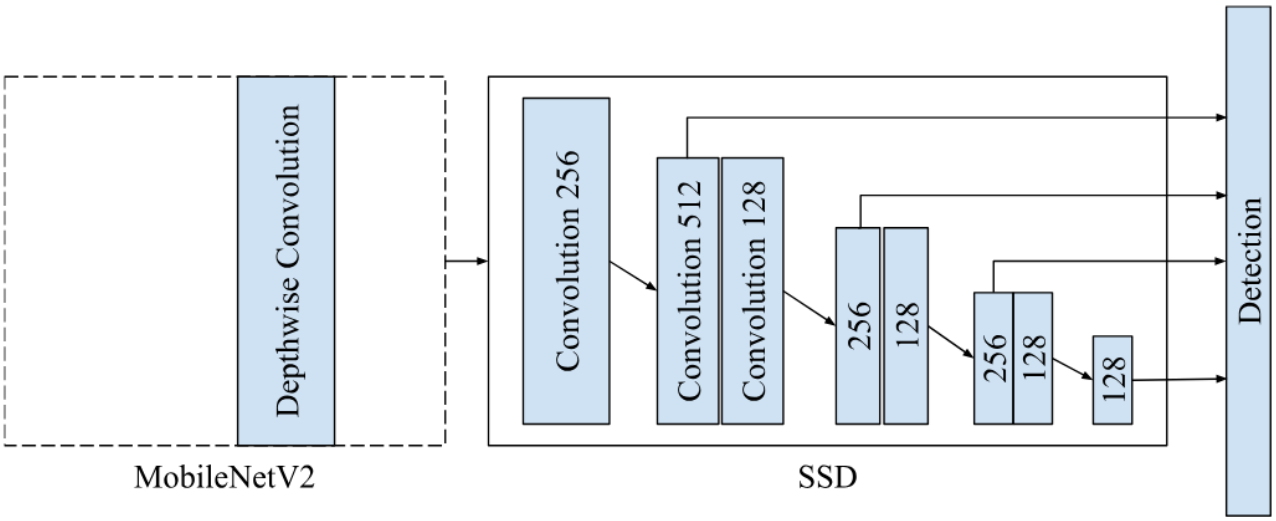

Рисунок 2.1 – Архітектура MobileNetV2 SSD

FPN-Lite – це більш оптимізована, легка версія, яка зберігає переваги FPN, але зменшує її складність, що робить її більш придатною для пристроїв з обмеженими ресурсами, таких як мобільні телефони або вбудовані системи.

Під час навчання модель вчиться передбачати розташування і клас об'єктів на заданому зображенні. Функція втрат, що зазвичай використовується, – це комбінація втрат локалізації (точності визначення рамок) і втрат класифікації (прогнозування класу об'єкта) (табл.2.1).

Таблиця 2.1 – Показники оцінювання MobileNetV2 SSD FPN-Lite

mAP (Mean Average Precision)	IoU (Intersection over Union)	Inference Time (ms)
0.80	0.77	40

Модель навчається на великому наборі даних мічених зображень, де для кожного об'єкта на зображенні вказані обмежувальні рамки та мітки класів.

Mean Average Precision (середня точність) – це стандартна метрика для оцінки моделей виявлення об'єктів.

Вона усереднює точність для різних порогових значень IoU. Значення mAP 0.78 свідчить про те, що модель добре розпізнає руки на різних рівнях точності.

IoU (Intersection over Union) вимірює перекриття між прогнозованою обмежувальною рамкою і реальною обмежувальною рамкою. Вона визначається як площа перекриття, поділена на площу об'єднання.

IoU 0.77 вказує на сильне перекриття, що свідчить про те, що прогнозовані рамки, як правило, дуже близькі до фактичних положень рук.

Inference time (час виконання) вимірює, скільки часу потрібно моделі, щоб зробити прогноз. Менший час має вирішальне значення для додатків, що працюють у реальному часі.

Час виведення 40 мс означає, що модель може швидко обробляти кадри, що робить її придатною для застосунків, які потребують прогнозування в реальному часі.

2.1.2 MobileNetV2

Без додаткового алгоритму SSD MobileNetV2 використана для виявлення ключових точок на руці, такі як кінчики пальців, центр долоні та суглоби.

Основною проблемою при виявленні орієнтирів руки є розпізнавання положення, орієнтації та повної детальної структури руки за різних умов освітлення та оклюзії у цілому зображені. Для прикладу навчено дві моделі:

1) модель для обробки цілого зображення для прогнозування 21 точки руки без допоміжних модулей;

2) модель, яка працює у парі з MobileNetV2 SSD FPN-Lite, для обробки зображення відокремленої руки.

Процес навчання обох версій MobileNetV2 був однаковим. Відрізнялась лише обробка використаного набору даних.

Спочатку модель навчена за допомогою стандартного навчання з перенесенням, використовуючи попередньо навчену модель MobileNetV2. Попередньо навчені ваги отримані з набору даних ImageNet, який забезпечив міцну основу для вилучення ознак для мережі. Початкові шари мережі (як правило, ранні згортки) заморожені під час навчання, щоб зберегти загальні риси, отримані з ImageNet.

Після початкового етапу навчання модель доопрацьована. Цей процес включав розморожування деяких раніше заморожених шарів і перенавчання їх на наборі даних орієнтирів рук. Під час точного налаштування оновлювалися лише верхні шари мережі, тоді як попередні шари, які фіксують низькорівневі особливості (наприклад, краї або текстури), залишалися «замороженими». Такий підхід дозволив моделі вивчити специфічні особливості руки, зберігаючи при цьому знання з оригінальної попередньо навченої моделі.

Завдяки такому підходу до навчання модель навчилася ефективно локалізувати орієнтири рук, навіть з обмеженими ресурсами. Хоча модель, яка працює сама, без кооперації з SSD моделею, показує значно гірші результати

(табл. 2.2).

На основі порівняльного аналізу між моделями MobileNetV2 та MobileNetV2 + MobileNetV2 SSD FPN-Lite за основними показниками оцінювання можна зробити висновки, що використання моделі SSD FPN-Lite та MobileNetV2 істотно покращує результати моделі, роблячи її значно кращою для задач розпізнавання жестів у середовищах з підвищеними вимогами до точності.

Таблиця 2.2 – Порівняння показників оцінювання між MobileNetV2 (basic) та MobileNetV2 + MobileNetV2 SSD FPN-Lite (ssd+)

Модель	MSE	MAE	R^2 Score	Detection Accuracy
basic	0.31	0.26	0.57	26.4%
ssd+	0.030	0.045	0.86	79.1%

2.2 Моделі YOLOv10 та MediaPipe

Для порівняння власноруч навчених моделей, використані готова YOLOv10 модель від авторів набору даних «HaGRID» для відокремлення руки на зображені, та MediaPipe для отримання ключових точок руки.

Для порівняння власноруч навчених моделей використано дві добре зарекомендовані моделі:

1) YOLOv10, яка застосовується для сегментації та визначення рук на зображенні;

2) MediaPipe, для отримання ключових точок руки.

YOLO (You Only Look Once) – це одна з найпопулярніших архітектур для задач детекції об'єктів [11]. У версії YOLOv10, яку адаптовано авторами набору даних «HaGRID», зроблено низку покращень, спрямованих на ефективну сегментацію рук на зображеннях. На відміну від MobileNetV2 SSD

FPN-Lite, вона є достатньо вимогливою до ресурсів.

MediaPipe – це програмний фреймворк від Google, який надає високоякісні інструменти для комп'ютерного зору. У випадку аналізу рук, MediaPipe Hand Landmarks пропонує точне визначення 21 ключової точки (landmark) на руці, що включає суглоби пальців та інші характерні точки.

Таким чином, використання цих моделей дозволяє порівняти продуктивність власноруч навчених моделей із сучасними рішеннями.

2.3 Набір даних «HaGRID»

HaGRID (Hand Gesture Recognition Image Dataset) – це комплексний набір зображень, призначений для систем розпізнавання жестів рук [12]. Набір даних зібраний за допомогою краудсорсингових платформ, що забезпечило різноманітність учасників, сцен, умов освітлення і відстані від об'єкта до камери, тим самим підвищивши його застосовність до реальних сценаріїв (рис.2.2).

Рисунок 2.2 – Приклад зображень датасету «HaGRID»

Набір даних «HaGRID» містить 552 992 зображення, розподілених на 18 класів жестів, включно з категорією «без жестів», що забезпечує повноцінну

основу для навчання та оцінки моделей розпізнавання жестів. Кожне зображення ретельно анотоване: вказані рамки жестів, класи, а також провідна рука, що дозволяє досягти високої точності у задачах локалізації та класифікації.

З практичних міркувань у цьому дослідженні застосовано скорочену версію набору даних розміром 15 ГБ. Він був створений шляхом відбору репрезентативної підмножини, яка зберегла основну різноманітність оригінального набору, що забезпечує надійність результатів при суттєвому зменшенні обсягу обчислювальних ресурсів.

Перед використанням набір даних зазнав кількох етапів попередньої обробки для поліпшення якості та релевантності:

– видалені зображення без обмежувальних рамок і орієнтирів рук, адже такі анотації є критично важливими для навчання моделей;

– виключені зображення з кількома руками, що дозволило спростити задачу розпізнавання та оптимізувати процес навчання;

– зображення були масштабовані до розміру 256x256 пікселів для досягнення однорідності та сумісності з обраними архітектурами нейронних мереж.

Очищений і оброблений набір даних розділено на дві основні підмножини:

1) підмножина з обмежувальними рамками – використовується для навчання моделі SSD, яка виконує завдання виявлення та локалізації об'єктів;

2) підмножина з ключовими точками руки – використовується для навчання звичайної згорткової нейронної мережі, орієнтованої на точкову локалізацію ключових частин руки.

Проведено аналіз різноманітності набору даних «HaGRID». Для початку отримаємо щільність розміщення рук (рис. 2.3) для одного з найважливіших критеріїв якісного виділення руки на зображені.

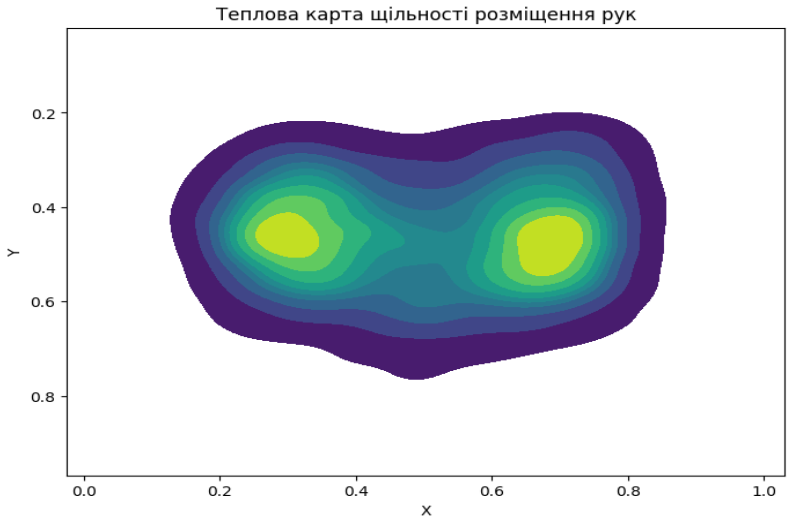

Рисунок 2.3 – Теплова карта щільності розміщення рук у «HaGRID»

На тепловій карті щільності розміщення рук можна побачити, що зона з руками знаходиться ближче до центру, що трохи обмежує варіативність. Крім цього, також чітко видно дві особливо яскраві зони, кожна з яких відповідає за одну з рук.

Не менш важливим за щільність розміщення рук, є їх розмір, або дальність від камери (рис. 2.4), що особливо важливо для SSD моделей виділення.

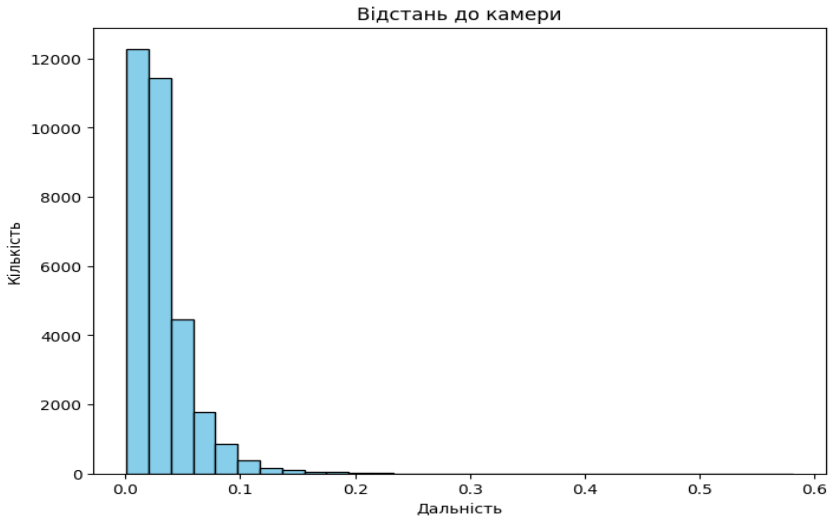

Рисунок 2.4 – Діаграма відстані рук до камери у наборі даних «HaGRID»

Як видно з діаграми, понад дві третини зображень містять руки, розташовані досить близько до камери. Це може створювати певні обмеження для універсальності кінцевої моделі, проте для задач, спрямованих на розпізнавання жестів за допомогою веб-камери ноутбука або мобільного пристрою, такий набір є цілком достатнім і відповідає практичним вимогам.

Ще одним важливим параметром є яркість зображення (рис. 2.5). Збалансованість яскравості є важливим аспектом, оскільки моделі розпізнавання жестів повинні бути стійкими до змін освітлення, які можуть варіюватися залежно від середовища, часу доби, джерел світла чи тіней.

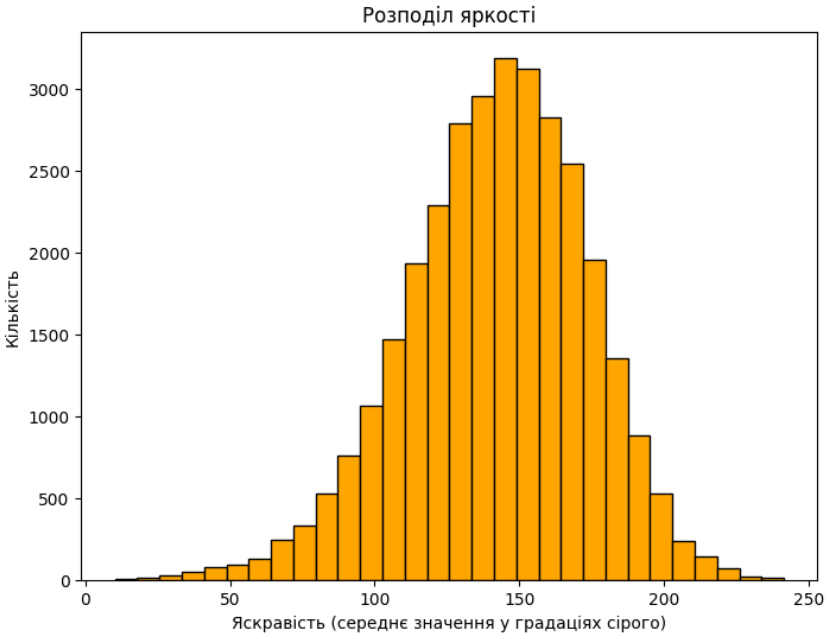

Рисунок 2.5 – Діаграма відстані рук до камери у наборі даних «HaGRID»

За отриманою діаграмою можна зробити висновок, що набір даних «HaGRID» демонструє добре збалансоване різноманіття рівнів яскравості зображень. Це свідчить про те, що дані були ретельно відібрані та підготовлені таким чином, щоб забезпечити їх придатність для навчання моделей комп’ютерного зору у реальних умовах.

2.4 Загальна архітектура системи

Завдання кваліфікаційної роботи передбачає створення системи, яка повина рекомендувати моделі розпізнавання жестів в залежності від апаратних ресурсів. Отже, загальна архітектура системи наведена на рисунку 2.6.

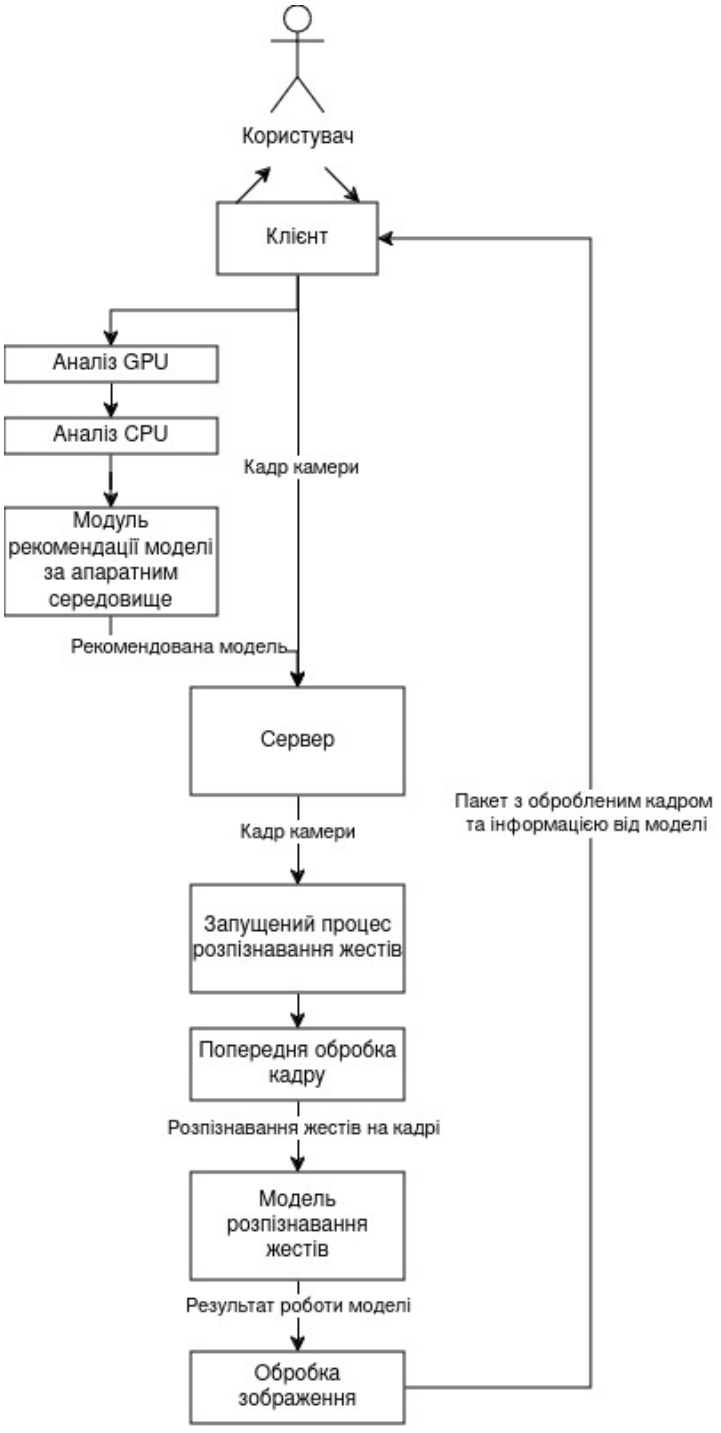

Рисунок 2.6 – Загальна архітектура системи.

Загальна архітектура системи складається з кількох основних компонентів, взаємодія між якими забезпечує ефективну роботу системи. Нижче наведено опис етапів роботи:

1) користувач – взаємодіє із системою через клієнтський додаток, який отримує відеопотік з камери;

2) клієнт:

a) приймає кадри з камери для подальшої обробки;

b) виконує аналіз апаратного середовища (CPU та GPU);

c) використовує модуль рекомендацій для вибору оптимальної моделі розпізнавання жестів залежно від характеристик апаратного забезпечення;

d) надсилає рекомендовану модель і кадри камери на сервер;

3) сервер:

a) отримує кадри з камери та інформацію про рекомендовану модель;

b) запускає процес розпізнавання жестів;

4) попередня обробка кадру – на сервері виконується попередня обробка зображення для приведення його до формата використання у моделях розпізнавання жестів;

5) модель розпізнавання жестів – визначає жести, які зображені на кадрі;

6) обробка результатів – зображення оброблюється на основі результата роботи моделі;

7) відправка результатів користувачу:

a) сервер формує пакет для відправки, що включає оброблений кадр із розпізнаними жестами та додаткову інформацію про результати роботи моделі;

b) пакет передається назад клієнту;

c) оброблений кадр і результати роботи моделі демонструються користувачу через клієнтський інтерфейс.

2.5 Стек технологій

Для реалізації системи потрібно обрати середовище, яке б дозволило працювати із великими моделями та мала велику обчислювальну спроможність, обрати програмні реалізації моделей наведених вище та допоміжні інструменти розробки.

Python – обраний мовою програмування через велику кількість реалізованих бібліотек для машинного навчання, в тому числі імплементацій описаних вище моделей, що дозволяє зручно модифікувати їх та експериментувати [13].

Бібліотеки TensorFlow [14] і PyTorch [15] дозволяют інтегрувати попередньо навчені моделі. Однак ці бібліотеки мають низку специфічних вимог, таких як необхідність встановлення CUDA [16] для роботи з GPU. Це створює певні труднощі, оскільки:

– налаштування CUDA потребує технічних знань і часу, що може бути складним для кінцевого користувача;

– на Windows доступ до CUDA можливий лише через WSL2 (Windows Subsystem for Linux), що також ускладнює встановлення та використання.

Для спрощення використання програми, забезпечення її доступності на будь-якому апаратному забезпеченні та зменшення потреби у попередньому налаштуванні середовища прийнято рішення про використання контейнеризації за допомогою Docker [17].

Docker дозволяє створити ізольоване середовище із заздалегідь налаштованими компонентами, необхідними для роботи серверної частини програми. Це значно знижує вимоги до системи користувача, забезпечує миттєву готовність програми до роботи після завантаження контейнера та мінімізує конфлікти, пов’язані з версіями бібліотек чи операційних систем.

Docker – це платформа для створення, доставки та запуску додатків у контейнерах. Контейнер – це ізольоване середовище, яке містить усе необхідне для роботи програми: код, залежності, системні бібліотеки тощо.

Використання контейнерів дозволяє уникнути проблем сумісності між різними операційними системами, версіями бібліотек або апаратним забезпеченням.

За основу контейнеру взят образ nvidia/cuda створений спеціально для додатків, які використовують обчислення на GPU. Він забезпечує сумісність із CUDA та cuDNN, що дозволяє ефективно використовувати GPU-ресурси для навчання та розгортання моделей машинного навчання.

Розглянемо Dockerfile для створення образу:

```
# Використання базового образу NVIDIA CUDA
FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu22.04

# Запобігає інтерактивним запитам під час встановлення пакетів
ENV DEBIAN_FRONTEND=noninteractive
# Миттєвий вивід Python-логів у консоль
ENV PYTHONUNBUFFERED=1

# Встановлення системних залежностей
RUN apt-get update && apt-get install -y \
     python3.10 \
     python3-pip \
     libgl1-mesa-glx \
     libglib2.0-0 \
     && rm -rf /var/lib/apt/lists/*
WORKDIR /app # Встановлення робочої директорії
COPY requirements.txt . # Копіювання файлу залежностей
RUN pip3 install --default-timeout=100 --no-cache-dir -r
requirements.txt # Встановлення Python-залежностей
COPY app/ . # Копіювання коду програми
COPY app/models/ ./models/ # Копіювання моделей розпізнавання

EXPOSE 8000 # Відкриття порту
# Команда для запуску програми
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port",
"8000"]
```

Лістинг 2.1 – Dockerfile для створення образу

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ РОЗПІЗНАВАННЯ ЖЕСТІВ

3.1 Основна програмна частина

У рамках розробки програмного забезпечення для розпізнавання жестів реалізовано три класи (додаток А), що забезпечують роботу з різними моделями обробки жестів:

1) LandmarksOnlyTracker (додаток А, лістинг А.2) – цей клас використовує бібліотеку TensorFlow для прогнозування ключових точок (landmarks) на зображеннях. Він дозволяє ефективно аналізувати положення пальців і долонь;

2) SSDHandTracker (додаток А, лістинг А.3) – ця реалізація передбачає використання двох окремих моделей: одна відповідає за виявлення рук за допомогою алгоритму SSD, інша – за визначення ключових точок для виявленої області;

3) YOLOMediaPipeTracker (додаток А, лістинг А.4) – у цьому класі інтегровано алгоритм YOLOv10 для виявлення областей, що містять руки, а також бібліотеку MediaPipe для подальшого трекінгу жестів. Такий підхід дозволяє поєднати переваги точного виявлення об’єктів із високою швидкістю обробки жестів.

Кожен із зазначених класів включає методи для: обробки кадрів з вхідного відеопотоку; візуалізації результатів розпізнавання; ефективного управління апаратними ресурсами, зокрема використання GPU та завантаження моделей.

Ці моделі інтегруються у серверну частину програмного забезпечення, забезпечуючи реалізацію функції розпізнавання жестів.

3.2 Реалізація веб-сервісу на базі FastAPI і Uvicorn

Для забезпечення гнучкого й кросплатформенного доступу до системи розпізнавання жестів реалізовано серверний додаток у Docker-контейнері з використанням FastAPI [18] та Uvicorn [19]. Основна мета цієї архітектури – дозволити взаємодію з програмою через веб-сокети або HTTP-запити, уникаючи прив’язки апаратних ресурсів (таких як, камери) безпосередньо до контейнера.

FastAPI забезпечує легку й ефективну розробку веб-додатків завдяки підтримці асинхронних операцій і вбудованій інтеграції OpenAPI для генерації документації API. Сервер Uvicorn працює як високопродуктивний ASGI-сервер, що ідеально підходить для асинхронних додатків.

У серверній частині (додаток Б) передбачено два основних режими взаємодії: передача відео через веб-сокети та робота через стандартні HTTP-запити. Такий підхід робить програму універсальною й придатною для інтеграції в різноманітні системи. Використання веб-сокетів дозволяє передавати кадри з камери клієнта в реальному часі, уникаючи прив’язки камери до контейнера. Це особливо важливо, оскільки на системах Windows неможливо безпосередньо підключити фізичну камеру до контейнера через Docker.

Клієнтська частина програми забезпечує отримання відеопотоку з локальної камери і відправку його на сервер через веб-сокет. Серверна частина отримує ці кадри, обробляє їх (використовуючи відповідну модель розпізнавання жестів), і повертає результати клієнту.

Реалізація сервісу на базі FastAPI в Docker забезпечує високу продуктивність, простоту розгортання і можливість роботи на будь-якій платформі. Рішення з веб-сокетами знімає обмеження, пов’язані з апаратною підтримкою, а API дозволяє масштабувати й інтегрувати програму в інші екосистеми.

3.2.1 Архітектура системи розпізнавання жестів

Архітектура системи розпізнавання жестів побудована на принципі паралельної обробки даних, що дозволяє ефективно використовувати ресурси і забезпечити високу швидкість обробки відеопотоку. Сервер створює окремі процеси для моделей коли клієнт робить запит, що дозволяє ізолювати різні методи обробки (рис. 3.1).

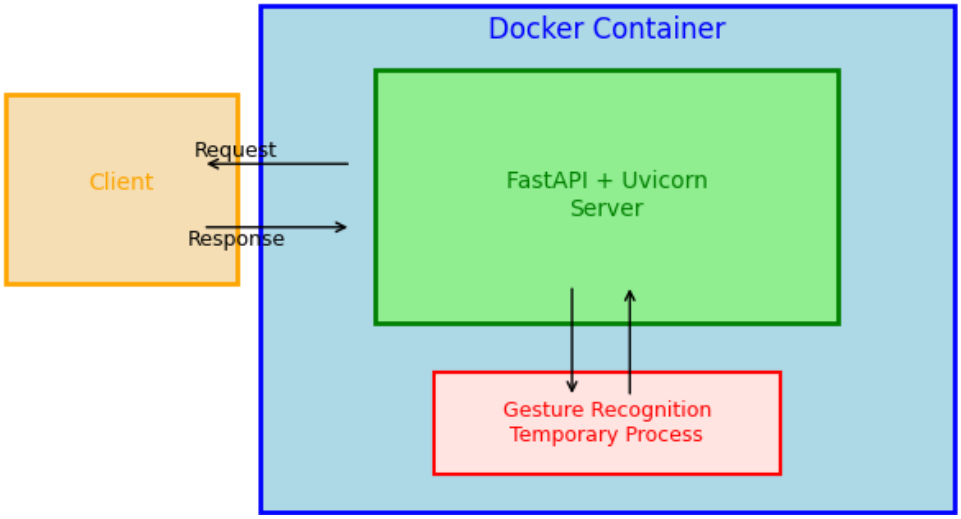

Рисунок 3.1 – Архітектура системи розпізнавання жестів

Клас TrackerProcess (додаток Г) є центральним елементом архітектури для обробки відеопотоку. Він відповідає за створення процесів для різних моделей розпізнавання жестів і їхню ініціалізацію. Всі операції, пов'язані з обробкою кадрів та розпізнаванням жестів, відбуваються в дочірньому від серверу процесу, що дозволяє моделям працювати незалежно, оптимізуючи використання ресурсів.

Моделі, такі як LandmarksOnlyTracker, SSDHandTracker і YOLOMediaPipeTracker, інкапсулюються в процеси за запитом клієнта. Вибір класу для кожного процесу визначається на основі переданого параметра mode, що дозволяє динамічно змінювати алгоритм розпізнавання.

Процес взаємодії між сервером і окремими процесами здійснюється через черги (queues) для обміну даними. Клієнтська частина передає кадри в процеси для обробки, а після завершення обробки процес повертає результати назад через іншу чергу. Це забезпечує асинхронну обробку кадрів, де кожен процес працює незалежно, але при цьому результати обробки передаються у загальний потік для подальшої обробки чи відправки клієнту.

Такий підхід до побудови системи дозволяє знизити навантаження на головний процес, забезпечити масштабованість і стабільність роботи системи в умовах високого навантаження.

Кожен процес може бути запущений або зупинений незалежно, що дає змогу оптимізувати використання ресурсів. Крім того, ізоляція процесів дозволяє уникнути помилок, пов'язаних із взаємодією різних моделей або з ресурсами, такими як GPU, оскільки кожен процес має свою власну ініціалізацію контексту для роботи з графічним процесором.

Загалом, архітектура на основі мультипроцесорності дозволяє системі бути більш адаптивною, ефективною та стійкою до помилок, що робить її придатною для використання в реальних умовах із високим рівнем вимог до продуктивності та масштабованості.

4 ТЕСТУВАННЯ СИСТЕМИ

4.1 Тестування моделей розпізнавання жестів

Для оцінки продуктивності навчених моделей розпізнавання жестів було проведено тестування на різних апаратних платформах. Використані пристрої представляють типові конфігурації обладнання з різним рівнем апаратних ресурсів, що дозволяє оцінити ефективність моделей у різних умовах. Характеристики пристроїв, залучених до тестування, наведено у таблиці 4.1.

Таблиця 4.1 – Апаратне забезпечення пристроїв

Модель	GPU memory (MB)	CPU cores	CPU frequency (GHz)
Пристрій №1	4096	8	3.401
Пристрій №2	2048	4	3.9

Рисунки 4.1 і 4.2 демонструють приклади функціонування моделей на двух пристроях.

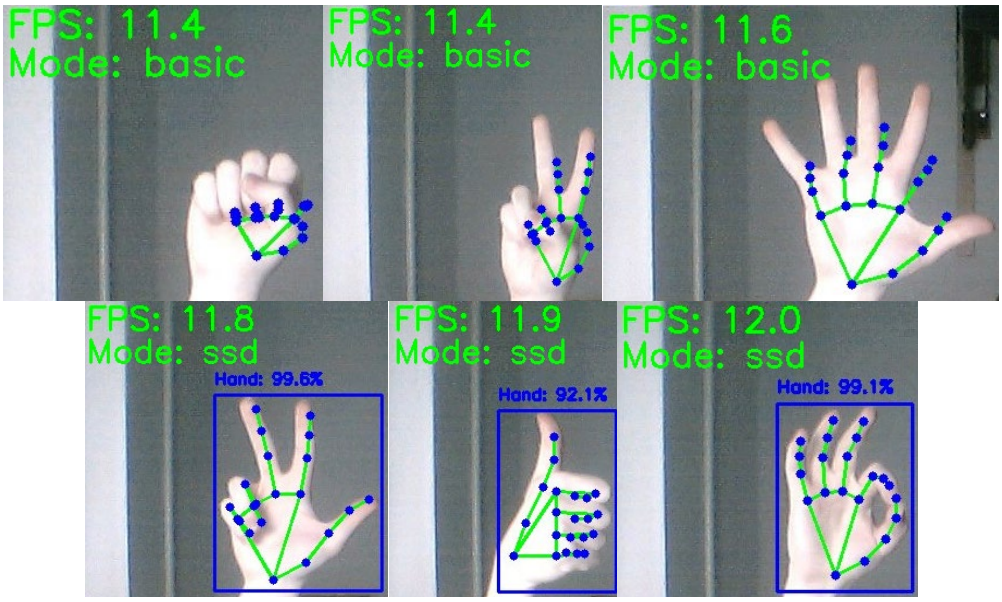

Рисунок 4.1 – Тестування моделей на пристрою №1

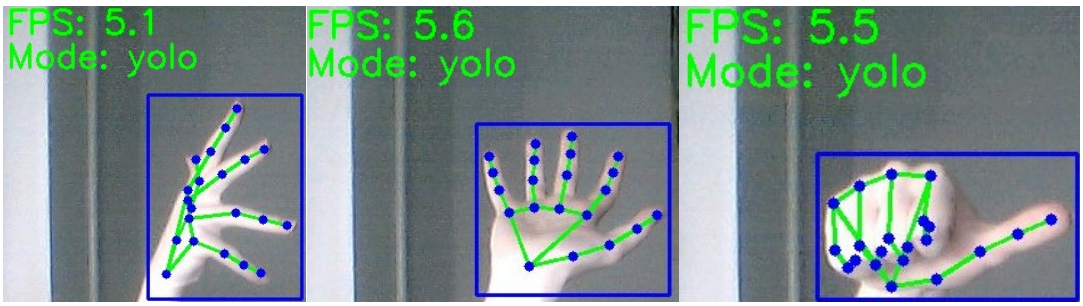

Рисунок 4.1, аркуш 2

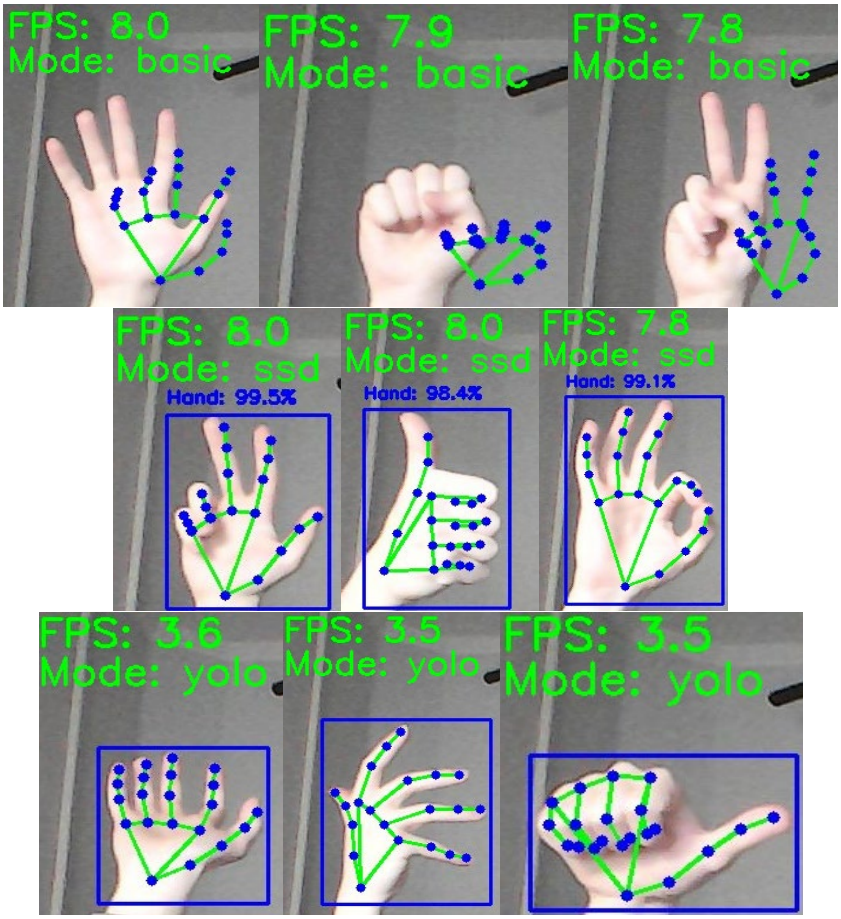

Рисунок 4.2 – Тестування моделей на пристрою №2

За результатами проведеного тестування отримано оцінку швидкості виконання відео алгоритмів у реальному часі – FPS, яка вказує на кількість оброблених кадрів за секунду різних моделей на двох пристроях, що демонструють їх продуктивність залежно від апаратного середовища та складності моделі (табл. 4.2).

Таблиця 4.2 – Порівняння роботи моделей розпізнавання жестів

Модель	Кількість параметрів (M)	Розмір файлу (MB)	Швидкість роботи на пристрою №1 (FPS)	Швидкість роботи на пристрою №2 (FPS)
MobileNetV2	~3.4	38.3	11–22	7-11
MobileNetV2 + MobileNetV2 SSD FPN-Lite	~8	59.9	11-13	6-8
YOLOv10 + Mediapipe	30+	61.2	4-6	1-4

4.2 Апаратно-залежний вибір моделі

Після проведення тестування розробленого програмного забезпечення на різних системах було створено бенчмарк, який демонструє залежність продуктивності від характеристик апаратного забезпечення. Для кожної системи наведено конфігурацію обладнання, включаючи об'єм відеопам'яті GPU, кількість ядер процесора, тактову частоту CPU, а також усереднені результати тестування для трьох моделей розпізнавання жестів: Basic, SSD та YOLO. Результати характеризуються показниками частоти кадрів (FPS), завантаження GPU (%), та завантаження CPU (%). Детальний опис кожної системи та відповідні результати відображено у таблиці (лістинг 4.1), що дозволяє оцінити ефективність алгоритмів залежно від ресурсів обчислювального середовища.

```
[{
    "gpu_mem": 2048,
    "cpu_cores": 4,
    "cpu_freq": 3.9,
    "basic": {"fps": 10, "gpu_util": 10, "cpu_util": 60},
    "ssd": {"fps": 7, "gpu_util": 27, "cpu_util": 50},
    "yolo": {"fps": 3, "gpu_util": 95, "cpu_util": 80}
  },
  {
```

Лістинг 4.1 – Бенчмарк систем

```
    "gpu_mem": 4096,
    "cpu_cores": 8,
    "cpu_freq": 4.6,
    "basic": {"fps": 20, "gpu_util": 5, "cpu_util": 15},
    "ssd": {"fps": 12, "gpu_util": 23, "cpu_util": 17},
    "yolo": {"fps": 6, "gpu_util": 80, "cpu_util": 40}
   },{
    "gpu_mem": 8192,
    "cpu_cores": 4,
    "cpu_freq": 3.4,
    "basic": {"fps": 30, "gpu_util": 5, "cpu_util": 5},
    "ssd": {"fps": 20, "gpu_util": 20, "cpu_util": 5},
    "yolo": {"fps": 15, "gpu_util": 85, "cpu_util": 10}}]
```

Лістинг 4.1, аркуш 2

Використовуючи отриману інформацію зіставлена формула для отримання найсхожого бенчмарка, що відповідає характеристикам поточної системи. Функція find_closest_benchmark визначає найбільш подібний еталон обладнання на основі порівняння специфікацій поточного обладнання з набором існуючих бенчмарків (лістинг 4.2).

```
def find_closest_benchmark(hardware, benchmarks):
   def similarity_score(benchmark):
     # Weighted sum of differences (normalize by max values in
benchmarks)
       gpu_score = abs(hardware["gpu_mem"] -
benchmark["gpu_mem"]) / max(b["gpu_mem"] for b in benchmarks)
       cpu_score = abs(hardware["cpu_cores"] -
benchmark["cpu_cores"]) / max(b["cpu_cores"] for b in
benchmarks)
       freq_score = abs(hardware["cpu_freq"] -
benchmark["cpu_freq"]) / max(b["cpu_freq"] for b in benchmarks)
       return gpu_score * 0.7 + (cpu_score + freq_score) * 0.3
# GPU weighted more
   return min(benchmarks, key=similarity_score)
```

Лістинг 4.2 – Функція find_closest_benchmark

Ця функція приймає два аргументи: hardware, який представляє конфігурацію поточного обладнання (із ключами, такими як "gpu_mem", "cpu_cores", "cpu_freq"), та список benchmarks, де кожний елемент описує бенчмарк з аналогічними ключами. Це дозволяє порівнювати характеристики

поточного пристрою із попередньо протестованими конфігураціями для вибору найбільш схожого бенчмарку.

Вбудована функція similarity_score обчислює рівень схожості між поточним обладнанням і кожним бенчмарком, використовуючи нормалізовані різниці характеристик. Різниця значення обсягу пам'яті GPU нормалізується через максимальне значення цієї характеристики серед усіх бенчмарків, що дозволяє уникнути впливу різних масштабів параметрів. Аналогічним чином обчислюються різниці за кількістю ядер і частотою процесора. Потім ці результати комбінуються у зважену суму: відмінність у пам'яті GPU отримує вагу 0.7, оскільки вона є ключовим фактором для обробки графічних задач, а різниці в ядрах і частоті процесора разом – вагу 0.3.

Результат обчислення повертає бенчмарк, який має найменшу різницю між поточним обладнанням і бенчмарками, тобто найбільш відповідає заданим характеристикам пристрою. Це забезпечує адаптивність системи до різноманітних апаратних платформ.

Після отримання найбільш схожого варіанту можна вирішувати, яку модель підключати. Наприклад:

```
def select_best_model(benchmark):
   # Prioritize better quality models if FPS is above a
threshold (e.g., 8 FPS)
   candidates = ["basic", "ssd", "yolo"]
   for model in reversed(candidates):  # Check higher-quality
models first
      if benchmark[model]["fps"] >= 8:
          return model
   return "basic"  # Fallback to the simplest model
```

Лістинг 4.3 – Функція select_best_model

4.3 Тестування апаратно-залежного вибору моделі

З використанням описаних функцій протестований модуль рекомендації конфігурації моделей розпізнавання жестів на двох

різних пристроях (рис. 4.3).

```
(hand) PS F:\Other\University\degree\hand-tracking-service> python .\client.py
GPU memory: 4096.0 MB
CPU cores: 8, CPU frequency: 3.401 GHz
Auto-selected mode based on hardware: ssd
{"status":"success","message":"Initialized ssd tracker"}

(testenv) godlam@artem-laptop:~/Other/hand-tracking-service$ python client.py
GPU Memory: 2048.0 MB
CPU Cores: 4, CPU Frequency: 3.9 GHz
Auto-selected mode based on hardware: basic
{"status":"success","message":"Initialized basic tracker"}
```

Рисунок 4.3 – Тест апаратно-залежного вибору моделі

Під час тестування модуль повернув конфігурацію моделі, яка, згідно з даними попереднього тестування, дійсно мала найкращі показники продуктивності для відповідного апаратного середовища. Це підтверджує коректність роботи алгоритму вибору моделей.

ВИСНОВКИ

У даній кваліфікаційній роботі підвищена ефективність розпізнавання жестів у відео, шляхом вибору моделі глибокого навчання для конкретного пристрою на основі аналізу характеристик апаратного середовища та продуктивності моделей комп’ютерного зору.

Для розробки системи кроссплатформенного розпізнавання жестів використані Docker, як стабільне середовище, та FastAPI з Uvicorn, як сервер до якого можна звертатися для взаємодії з навченими моделями, що дає можливість дуже гибкого використання можливостей системи.

У роботі досліджені сучасні методи та моделі розпізнавання жестів, такі як Single Shot Detection, MobileNetV2, YOLOv10 та MediaPipe. Розроблені три різних архітектур моделей розпізнавання жестів з використання MobileNetV2. Використано набір даних «HaGRID» для навчання алгоритмів.

Реалізовані модифікації обраних MobileNetV2 моделей розпізнавання жестів та проведено тестування роботи конфігурацій моделей на системах з різними апаратним забезпеченням.

Реалізована система, яка на основі характеристик апаратного середовища і проведеного аналізу видаває рекомендовану конфігурацію моделей розпізнавання жестів.

Робота представлена на XXI всеукраїнської конференції студентів і молодих науковції в міста Одеса 26 квітня 2024 р [9].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Image Recognition Market Size, Share & Trends Analysis Report By Technique, By Component (Hardware, Software, Service), By Deployment Mode, By Vertical, By Application, By Region, And Segment Forecasts, 2024 – 2030. [Електронний ресурс] – Режим доступу: https://www.grandviewresearch.com/industry-analysis/image-recognition-market.
2. The Future of the In-Car Experience is Here: Cerence Introduces Gesture-Based Interaction with the Button-Free Car of the Future. [Електронний ресурс] – Режим доступу: https://www.cerence.com/ja/news-releases/news-release-details/future-car-experience-here-cerence-introduces-gesture-based.
3. Gesture Recognition Market Size, Share & Trends Analysis Report By Technology (Touch-based, Touchless), By Industry (Automotive, Consumer Electronics, Healthcare), By Region, And Segment Forecasts, 2023 - 2030. [Електронний ресурс] – Режим доступу: https://www.grandviewresearch.com/industry-analysis/gesture-recognition-market.
4. Neural Network Compression Framework for enhanced OpenVINO™ inference. [Електронний ресурс] – Режим доступу: https://github.com/openvinotoolkit/nncf.
5. Automated Mixed-Precision Quantization for Next-Generation Inference Hardware. [Електронний ресурс] – Режим доступу: https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Automated-Mixed-Precision-Quantization-for-Next-Generation/post/1335745.
6. Automated Super-Network Generation for Scalable Neural Architecture Search. / Munoz J.P., Lyalyushkin N., Lacewell C.W. та ін. // Proceedings of the First International Conference on Automated Machine Learning: Proceedings of Machine Learning Research. – PMLR, 2022. – 15 с.
7. Azure AI. [Електронний ресурс] – Режим доступу: https://azure.microsoft.com/en-us/solutions/ai.

8. Cross-platform, customizable ML solutions for live and streaming media. [Електронний ресурс] – Режим доступу: https://github.com/google-ai-edge/mediapipe
9. Осипов А.В., Розпізнавання жестів з використанням глибокого навчання та комп'ютерного зору / Осипов А.В., Шпинарева І.М. // Інформатика, інформаційні системи та технології: тези доповідей двадцять першої всеукраїнської конференції студентів і молодих науковців. Одеса, 26 квітня 2024 р. – Одеса, 2024. – с.144-145
10. MobileNetV2: The Next Generation of On-Device Computer Vision Networks. [Електронний ресурс] – Режим доступу: https://research.google/blog/mobilenetv2-the-next-generation-of-on-device-computer-vision-networks/.
11. YOLOv10: Real-Time End-to-End Object Detection. [Електронний ресурс] – Режим доступу: https://docs.ultralytics.com/models/yolov10/.
12. HaGRID (Hand Gesture Recognition Image Dataset) [Електронний ресурс] – Режим доступу: https://www.kaggle.com/datasets/kapitanov/hagrid
13. Python [Електронний ресурс] – Режим доступу: https://www.python.org/
14. TensorFlow. [Електронний ресурс] – Режим доступу: https://www.tensorflow.org/.
15. PyTorch. [Електронний ресурс] – Режим доступу: https://pytorch.org/.
16. CUDA Toolkit Documentation. [Електронний ресурс] – Режим доступу: https://docs.nvidia.com/cuda/.
17. Docker Docs. [Електронний ресурс] – Режим доступу: https://docs.docker.com.
18. FastAPI. [Електронний ресурс] – Режим доступу: https://fastapi.tiangolo.com/.
19. Uvicorn. [Електронний ресурс] – Режим доступу: https://www.uvicorn.org/.

ДОДАТОК А

Класи моделей розпізнавання жестів

```
import cv2
import tensorflow as tf
import numpy as np
import os
from collections import deque
import torch
import mediapipe as mp
from ultralytics import YOLO
import argparse

class BaseHandTracker:
     def __init__(self):
     self.timer = ProcessTimer()

     self.CONNECTIONS = [
        # Thumb
        (0, 1), (1, 2), (2, 3), (3, 4),
        # Index finger
        (0, 5), (5, 6), (6, 7), (7, 8),
        # Middle finger
        (9, 10), (10, 11), (11, 12),
        # Ring finger
        (13, 14), (14, 15), (15, 16),
        # Pinky
        (0, 17), (17, 18), (18, 19), (19, 20),
        # Palm (connecting finger bases)
        (5, 9), (9, 13), (13, 17)
     ]

     def process_frame(self, frame):
     raise NotImplementedError
     def draw_landmarks(self, frame, landmarks, bbox=None,
score=None):
     if landmarks is not None:
        # Draw skeleton connections
        for connection in self.CONNECTIONS:
            start_idx, end_idx = connection
            start_point =
tuple(landmarks[start_idx].astype(int))
                end_point = tuple(landmarks[end_idx].astype(int))
          cv2.line(frame, start_point, end_point, (0, 255, 0), 2)
        # Draw landmark points
        for (x, y) in landmarks.astype(int):
           cv2.circle(frame, (x, y), 4, (255, 0, 0), -1)
        # Draw bounding box if provided
        if bbox is not None:
```

Лістинг А.1 – Базовий клас моделі розпізнавання жестів

```
            ymin, xmin, ymax, xmax = bbox
                 cv2.rectangle(frame, (xmin, ymin), (xmax, ymax),
(255, 0, 0), 2)
            if score is not None:
                cv2.putText(frame, f'Hand: {score*100:.1f}%',
                            (xmin, ymin - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
     cv2.putText(frame, f'FPS: {self.timer.get_fps():.1f}',
           (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
     return frame
     def cleanup(self):
     """Base cleanup method to be implemented by child
classes"""
     pass
```

Лістинг А.1, аркуш 2

```
class LandmarksOnlyTracker(BaseHandTracker):
     def __init__(self, landmarks_model_path):
     super().__init__()
     self.landmarks_model =
tf.keras.models.load_model(landmarks_model_path)

     def preprocess_frame(self, frame):
     self.timer.start("Frame Preprocessing")
     input_frame = cv2.resize(frame, (256, 256))
     input_frame = input_frame / 255.0
     input_frame = np.expand_dims(input_frame, axis=0)
     self.timer.stop("Frame Preprocessing")
     return input_frame

     def predict_landmarks(self, hand_input, frame):
     if hand_input is None:
        return None
       self.timer.start("Landmarks Prediction")
     landmarks = self.landmarks_model.predict(hand_input)[0]
     self.timer.stop("Landmarks Prediction")
     landmarks = landmarks.reshape(21, 2)
     landmarks[:, 0] *= frame.shape[1]
     landmarks[:, 1] *= frame.shape[0]
     return landmarks
     def process_frame(self, frame):
     self.timer.update_fps()
     input_frame = self.preprocess_frame(frame)
     landmarks = self.predict_landmarks(input_frame, frame)
     return self.draw_landmarks(frame, landmarks)
     def cleanup(self):
     """Clean up TensorFlow model resources"""
     if hasattr(self, 'landmarks_model'):
        self.landmarks_model = None
        tf.keras.backend.clear_session()
```

Лістинг А.2 – Клас моделі MobileNetV2 без SSD

```
class SSDHandTracker(BaseHandTracker):
     def __init__(self, ssd_model_path, landmarks_model_path):
     super().__init__()
     self.ssd_model = tf.saved_model.load(ssd_model_path)
     self.infer = self.ssd_model.signatures['serving_default']
     self.landmarks_model =
tf.keras.models.load_model(landmarks_model_path)
     def preprocess_frame(self, frame):
     self.timer.start("Frame Preprocessing")
     input_frame = cv2.resize(frame, (512, 512))
     input_frame = input_frame.astype(np.uint8)
     input_frame = np.expand_dims(input_frame, axis=0)
     self.timer.stop("Frame Preprocessing")
     return input_frame

     def process_hand_crop(self, frame, box):
     self.timer.start("Bounding Box Preprocessing")
     ymin, xmin, ymax, xmax = (box * [frame.shape[0],
frame.shape[1],
                         frame.shape[0],
frame.shape[1]]).astype(int)
     hand_crop = frame[ymin:ymax, xmin:xmax]
     if hand_crop.size == 0:
        return None, (ymin, xmin, ymax, xmax)
     hand_crop_resized = cv2.resize(hand_crop, (256, 256))
     hand_input = hand_crop_resized / 255.0
     hand_input = np.expand_dims(hand_input, axis=0)
     self.timer.stop("Bounding Box Preprocessing")
     return hand_input, (ymin, xmin, ymax, xmax)
     def predict_landmarks(self, hand_input, bbox):
     if hand_input is None:
        return None
     self.timer.start("Landmarks Prediction")
     landmarks = self.landmarks_model.predict(hand_input)[0]
     self.timer.stop("Landmarks Prediction")
     landmarks = landmarks.reshape(21, 2)
     ymin, xmin, ymax, xmax = bbox
     landmarks[:, 0] = landmarks[:, 0] * (xmax - xmin) + xmin
     landmarks[:, 1] = landmarks[:, 1] * (ymax - ymin) + ymin
     return landmarks
     def process_frame(self, frame):
     self.timer.update_fps()
     input_frame = self.preprocess_frame(frame)
     self.timer.start("SSD Inference")
     predictions = self.infer(tf.convert_to_tensor(input_frame))
     self.timer.stop("SSD Inference")
     detection_boxes = predictions['detection_boxes'][0].numpy()
     detection_scores =
predictions['detection_scores'][0].numpy()
     if detection_scores[0] <= 0.7:
     return frame
```

Лістинг А.3 – Клас моделі MobileNetV2 з SSD

```
     hand_input, bbox = self.process_hand_crop(frame,
detection_boxes[0])
     landmarks = self.predict_landmarks(hand_input, bbox)
     return self.draw_landmarks(frame, landmarks, bbox,
detection_scores[0])
     def cleanup(self):
     """Clean up TensorFlow model resources"""
     if hasattr(self, 'landmarks_model'):
        self.landmarks_model = None
        tf.keras.backend.clear_session()
     if hasattr(self, 'ssd_model'):
        self.ssd_model = None
        self.infer = None
        tf.keras.backend.clear_session()
```

Лістинг А.3, аркуш 2

```
class YOLOMediaPipeTracker(BaseHandTracker):
     def __init__(self, yolo_model_path, conf_threshold=0.5):
     super().__init__()
     self.device = 'cuda' if torch.cuda.is_available() else
'cpu'
     self.model = YOLO(yolo_model_path)
     self.conf_threshold = conf_threshold
     self.mp_hands = mp.solutions.hands
     self.hands = self.mp_hands.Hands(
        static_image_mode=False,
        max_num_hands=2,
        min_detection_confidence=0.5,
        min_tracking_confidence=0.5
     )
     def process_frame(self, frame):
     self.timer.update_fps()
     image_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
     self.timer.start("YOLO Detection")
     results = self.model(image_rgb)[0]
     self.timer.stop("YOLO Detection")
     hand_boxes = []
     for r in results.boxes.data.tolist():
        x1, y1, x2, y2, score, class_id = r
        if score >= self.conf_threshold:
            hand_boxes.append((int(x1), int(y1), int(x2),
int(y2)))
     annotated_image = frame.copy()
     for box in hand_boxes:

     x1, y1, x2, y2 = box
     hand_img = image_rgb[y1:y2, x1:x2]

     if hand_img.size == 0:
     continue
```

Лістинг А.4 – Клас моделі YOLOv10 з Mediapipe

```
h, w, _ = hand_img.shape
self.timer.start("MediaPipe Processing")
   results = self.hands.process(hand_img)
   self.timer.stop("MediaPipe Processing")
   if results.multi_hand_landmarks:
         for hand_landmarks in results.multi_hand_landmarks:
           landmarks = []
           for landmark in hand_landmarks.landmark:
               cx = int(landmark.x * w) + x1
               cy = int(landmark.y * h) + y1
               landmarks.append([cx, cy])
           landmarks = np.array(landmarks)
           annotated_image = self.draw_landmarks(
               annotated_image,
               landmarks,
               (y1, x1, y2, x2))
return annotated_image
def cleanup(self):
"""Clean up PyTorch model and MediaPipe resources"""
if hasattr(self, 'model'):
   # Clean up YOLO model
   if isinstance(self.model, torch.nn.Module):
       self.model.cpu()
       del self.model
   else:
       self.model = None
   # Clean up CUDA cache if it was used
   if torch.cuda.is_available():
       torch.cuda.empty_cache()
# Clean up MediaPipe resources
if hasattr(self, 'hands'):
   self.hands.close()
   self.hands = None
```

Лістинг А.4, аркуш 2

ДОДАТОК Б

Сервер системи

```
from fastapi import FastAPI, WebSocket
from fastapi.responses import JSONResponse
import numpy as np
import cv2
from typing import Optional, Dict
from pydantic import BaseModel
import tensorflow as tf
import torch
import base64
from starlette.websockets import WebSocketDisconnect
import atexit
import gc
import asyncio
import signal
from tracker_process import TrackerProcess
class ProcessResponse(BaseModel):
     landmarks: Optional[list] = None
     bbox: Optional[list] = None
     fps: float
     timings: dict

app = FastAPI(title="Hand Tracking API")
# Global tracker process management
current_tracker: Dict[str, Optional[TrackerProcess]] =
{"instance": None}
active_connections = set()

# Define model paths based on mode
model_paths = {
     'basic': ['models/only-
landmarks/mobilenetv2_non_ssd_hand_landmarks.h5'],
     'ssd': [
     'models/ssd-plus-landmarks/ssd_mobilenetv2_hand',
     'models/ssd-plus-
landmarks/mobilenetv2_for_ssd_hand_landmarks.h5'
     ],
     'yolo': ['models/ssd-hagrid-plus-landmarks-
mediapipe/hagrid_yolo_hand_detection.pt']
}

def thorough_cleanup_gpu_memory():
     try:
     print("Starting thorough GPU memory cleanup")
     # Stop and clean up tracker process if exists
     if current_tracker["instance"] is not None:
                              current_tracker["instance"].stop()
```

Лістинг Б.1 – Сервер системи

```
        current_tracker["instance"] = None
        print("Tracker process stopped and cleaned up")
        print("Clearing TensorFlow session and resetting states")
        # Clear TensorFlow session and reset states
        tf.keras.backend.clear_session()
        for device in tf.config.list_physical_devices('GPU'):
                                                                try:
                    tf.config.experimental.reset_memory_stats(device)
                                                              except:
                                                                 pass

        # Clear PyTorch CUDA cache and reset states
        if torch.cuda.is_available():
           torch.cuda.empty_cache()
           torch.cuda.reset_peak_memory_stats()
           torch.cuda.reset_accumulated_memory_stats()

        # Force garbage collection multiple times
        for _ in range(3):
           gc.collect()

        print("Thorough GPU memory cleanup completed")
        except Exception as e:
        print(f"Error during cleanup: {e}")

# Register cleanup for different termination scenarios
atexit.register(thorough_cleanup_gpu_memory)
signal.signal(signal.SIGTERM, lambda signo, frame:
thorough_cleanup_gpu_memory())
signal.signal(signal.SIGINT, lambda signo, frame:
thorough_cleanup_gpu_memory())

@app.on_event("shutdown")
async def shutdown_event():
        # Clean up all active connections
        for websocket in active_connections.copy():
        try:
                                    await websocket.close(code=1000)
        except:
           pass
        thorough_cleanup_gpu_memory()

async def cleanup_connection(websocket: WebSocket):
        # Clean up ONE active connection:
        if websocket in active_connections:
        active_connections.remove(websocket)
        try:
           await websocket.close(code=1000)
        except:
           pass
```

Лістинг Б.1, аркуш 2

```
def initialize_tracker(mode: str) -> TrackerProcess:
     thorough_cleanup_gpu_memory()

     if mode not in model_paths:
     raise ValueError(f"Invalid mode: {mode}")

     tracker_process = TrackerProcess(mode, model_paths[mode])
     tracker_process.start()

     return tracker_process

@app.post("/initialize_tracker/{mode}")
async def initialize_model(mode: str):
     try:
     current_tracker["instance"] = initialize_tracker(mode)
     return {"status": "success", "message": f"Initialized
{mode} tracker"}
     except Exception as e:
     return JSONResponse(
                                                  status_code=500,
                content={"error": f"Failed to initialize tracker:
{str(e)}"}
     )

@app.get("/clear_tracker")
async def clear_model():
     thorough_cleanup_gpu_memory()
     return {"status": "success", "message": "Tracker cleared
and GPU memory cleaned"}

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
     if current_tracker["instance"] is None:
     await websocket.close(
                                                      code=4000,
                              reason="No tracker initialized. Call
/initialize_tracker/{mode} first"
     )
     return

     await websocket.accept()
     active_connections.add(websocket)

     try:
     while True:
        # Receive base64 encoded image
        try:
            data = await
asyncio.wait_for(websocket.receive_json(), timeout=5.0)
        except asyncio.TimeoutError:
            print("WebSocket timeout - cleaning up")
```

Лістинг Б.1, аркуш 3

```
            await cleanup_connection(websocket)
            return

        current_tracker["instance"].send_frame(data)
                                                        result =
current_tracker["instance"].get_result(block=True, timeout=5.0)
        if result and 'image' in result:
            await websocket.send_json(result)
     else:
            await websocket.send_json({
                'error': 'Failed to process frame',
            })
     except WebSocketDisconnect:
     print("WebSocket disconnected")
     except Exception as e:
     print(f"WebSocket error: {e}")
     finally:
     print(f"Cleaning up")
     await cleanup_connection(websocket)

@app.websocket("/ws/echo/0")
async def websocket_echo(websocket: WebSocket):
     await websocket.accept()

     try:
     while True:
        # Receive base64 encoded image
        data = await websocket.receive_json()
        resize_factor = data.get('resize_factor', 1.0)

        # Decode image
         img_data = base64.b64decode(data['image'].split(',')[1])
        nparr = np.frombuffer(img_data, np.uint8)
        frame = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
        if resize_factor != 1.0:
                frame = cv2.resize(frame, None, fx=resize_factor,
fy=resize_factor)

        # Encode processed frame
        encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), 85]
        _, img_encoded = cv2.imencode('.jpg', frame,
encode_param)
                                                    img_base64 =
base64.b64encode(img_encoded.tobytes()).decode('utf-8')

        await websocket.send_json({
                 'image': f'{img_base64}',
            'echo': True
        })
```

Лістинг Б.1, аркуш 4

```
     except WebSocketDisconnect:
     print("Client disconnected")

@app.get("/health")
async def health_check():
     try:
     # Verify GPU access
     gpu_available = len(tf.config.list_physical_devices('GPU'))
> 0
     if torch.cuda.is_available():
        torch_device = torch.device("cuda")
        torch.randn(1).to(torch_device)

        torch_available = True
     else:
        torch_available = False

     return {
        "status": "healthy",
        "gpu_available": gpu_available,
        "torch_available": torch_available,
        "tracker_active": current_tracker["instance"] is not
None
     }
     except Exception as e:
     return {"status": "unhealthy", "error": str(e)}


if __name__ == "__main__":
     import uvicorn
     uvicorn.run("main:app", host="0.0.0.0", port=8000,
reload=True)
```

Лістинг Б.1, аркуш 5

ДОДАТОК Г

Клас процесу розпізнавання жестів

```
import multiprocessing
import numpy as np
import cv2
import base64
import queue
import signal
import os
# Set CUDA initialization flags before importing frameworks
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow as tf
import torch
# Tracker classes
from hand_tracker import (
     BaseHandTracker,
     LandmarksOnlyTracker,
     SSDHandTracker,
     YOLOMediaPipeTracker
)

class TrackerProcess:
     def __init__(self, mode, model_paths):
     # Use a separate context for CUDA initialization in child
process
     multiprocessing.set_start_method('spawn', force=True)

     self.mode = mode
     self.model_paths = model_paths
     self.input_queue = multiprocessing.Queue()
     self.output_queue = multiprocessing.Queue()
     self.stop_event = multiprocessing.Event()
     self.process = None
     self.tracker = None

     def _initialize_gpu_context(self):
     # TensorFlow GPU initialization
     gpus = tf.config.list_physical_devices('GPU')
     if gpus:
        try:
        # Restrict TensorFlow to only use the first GPU
        tf.config.set_visible_devices(gpus[0], 'GPU')

        tf.config.experimental.set_memory_growth(gpus[0], True)
        except RuntimeError as e:
         print(f"GPU initialization error: {e}")

     # PyTorch CUDA initialization
```

Лістинг Г.1 – Клас TrackerProcess

```
    if torch.cuda.is_available():
                                                torch.cuda.init()
                        torch.cuda.set_device(0)  # Use first GPU

    def _tracker_worker(self):
    # Reinitialize GPU context in child process
    self._initialize_gpu_context()

    # Set up signal handling for clean termination
    def handle_sigterm(signum, frame):
                                               self.stop_event.set()
    signal.signal(signal.SIGTERM, handle_sigterm)
    signal.signal(signal.SIGINT, handle_sigterm)
    # Initialize tracker
    if self.mode == 'basic':
        self.tracker = LandmarksOnlyTracker(self.model_paths[0])
    elif self.mode == 'ssd':
       self.tracker = SSDHandTracker(
           self.model_paths[0],
           self.model_paths[1]
       )
    elif self.mode == 'yolo':
        self.tracker = YOLOMediaPipeTracker(self.model_paths[0])
    else:
       raise ValueError(f"Invalid mode: {self.mode}")
    while not self.stop_event.is_set():
       try:
           # Wait for input
           try:
                  input_data = self.input_queue.get(timeout=1.0)
           except queue.Empty:
               continue
           # Decode frame
           img_data =
base64.b64decode(input_data['image'].split(',')[1])
           nparr = np.frombuffer(img_data, np.uint8)
           frame = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
            resize_factor = input_data.get('resize_factor', 1.0)
           if resize_factor != 1.0:
               frame = cv2.resize(frame, None,
fx=resize_factor, fy=resize_factor)
             processed_frame = self.tracker.process_frame(frame)
           # Encode processed frame
              encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), 85]
           _, img_encoded = cv2.imencode('.jpg',
processed_frame, encode_param)
                                                    img_base64 =
base64.b64encode(img_encoded.tobytes()).decode('utf-8')
           # Put result in output queue
           output = {
                'image': f'{img_base64}',
```

Лістинг Г.1, аркуш 2

```
                'fps': self.tracker.timer.get_fps(),
                'timings': {name: timing['duration']
                           for name, timing in
self.tracker.timer.timings.items()}}
            self.output_queue.put(output)
        except Exception as e:
            print(f"Error in tracker process: {e}")
            self.output_queue.put({'error': str(e)})
     if hasattr(self.tracker, 'cleanup'):
        self.tracker.cleanup()
     print("Tracker process terminated")
     def start(self):
     self.process = multiprocessing.Process(
        target=self._tracker_worker,
        daemon=True)
     self.process.start()
     def stop(self):
     if self.process:
        # Signal the process to stop
        self.stop_event.set()
        # Clear queues
        while not self.input_queue.empty():
            try:
                self.input_queue.get_nowait()
            except queue.Empty:
                break
        while not self.output_queue.empty():
            try:
                self.output_queue.get_nowait()
            except queue.Empty:
                break
        # Wait for process to terminate
        self.process.terminate()
        self.process.join(timeout=5)
        # Force kill if not terminated
        if self.process.is_alive():
            self.process.kill()
        # Reset stop event
        self.stop_event.clear()
        print("Tracker process stopped")
     def send_frame(self, frame_data):
     """Send a frame to be processed"""
     self.input_queue.put(frame_data)
     def get_result(self, block=False, timeout=None):
     try:
        return self.output_queue.get(block=block,
timeout=timeout)
     except queue.Empty:
        return None
```

Лістинг Г.1, аркуш 3