–МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Олександр ЛИТВИНЕНКО
"_____"_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний модуль розпізнавання речей та обличчя пасажирів в аеропорту

Виконавець: _____ Софія КУПРІЄНКО

Керівник: _____ Володимир АРТЕМЧУК

Нормоконтролер:_____ Євгеній ТУПОТА

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук та технологій
Кафедра комп’ютеризованих систем управління

Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРЖУЮ
Завідувач кафедри
_____Олександр ЛИТВИНЕНКО
«____ »_____2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Купрієнко Софії Петрівни
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема кваліфікаційної роботи: «Програмний модуль розпізнавання речей та обличчя пасажирів в аеропорту» затверджена наказом ректора від «28» серпня 2023 р. № 1494/ст.

2. Термін виконання роботи (проєкту): з 02.10.2023 по 31.12.2023

3. Вихідні дані до роботи (проєкту): завдання на кваліфікаційну роботу, характеристики моделей нейронних мереж, документація мови програмування *Python*, інструмент *PyQt5*, фреймворк *DeepFace*, існуючі відеоматеріали з мережі інтернет, власні відеоматеріали.

4. Зміст пояснювальної записки:

1. Еволюція розвитку та впровадження комп’ютерного зору;

2. Аналіз та вибір програмних рішень для модулю розпізнавання образів;

3. Розробка програмного модулю.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1. Графік порівняння ефективності моделей нейронних мереж;

2. Схема алгоритму реалізації функціоналу програми;

3. Результати виявлення об’єктів з прогнозованими обмежувальними рамками.

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук та технологій
Кафедра комп’ютеризованих систем управління

Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРЖУЮ
Завідувач кафедри
Олександр ЛИТВИНЕНКО
«____ »_____2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Купрієнко Софії Петрівни
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема кваліфікаційної роботи: «Програмний модуль розпізнавання речей та обличчя пасажирів в аеропорту» затверджена наказом ректора від «28» серпня 2023 р. № 1494/ст.

2. Термін виконання роботи (проєкту): з 02.10.2023 по 31.12.2023

3. Вихідні дані до роботи (проєкту): завдання на кваліфікаційну роботу, характеристики моделей нейронних мереж, документація мови програмування *Python*, інструмент *PyQt5*, фреймворк *DeepFace*, існуючі відеоматеріали з мережі інтернет, власні відеоматеріали.

4. Зміст пояснювальної записки:

1. Еволюція розвитку та впровадження комп’ютерного зору;
2. Аналіз та вибір програмних рішень для модулю розпізнавання образів;
3. Розробка програмного модулю.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1. Графік порівняння ефективності моделей нейронних мереж;

2. Схема алгоритму реалізації функціоналу програми;

3. Результати виявлення об’єктів з прогнозованими обмежувальними рамками.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Аналіз літератури для дослідження еволюції технології комп’ютерного зору	02.10.2023 – 14.10.2023	
2	Проаналізувати еволюцію систем відеоспостереження і оцінити впровадження технології комп’ютерного зору в дані системи	15.10.2023 – 23.10.2023	
3	Характеристика нейронних мереж та вибір найбільш прийнятної	24.10.2023 – 29.10.2023	
4	Порівняння моделей згорткових нейронних мереж	30.10.2023 – 04.11.2023	
5	Вибір додаткових засобів розробки.	05.11.2023 – 12.11.2023	
6	Опис етапів роботи модулю	13.11.2023 – 17.11.2023	
7	Програмна реалізація роботи модулю	18.11.2023 – 07.12.2023	
8	Підбір даних для тестування побудованої моделі	08.12.2023 – 11.12.2023	
9	Тестування побудованої моделі на відібраних і власностворених відеозаписах	12.12.2023 – 14.12.2023	
10	Оформлення звітніх документів	15.12.2023 – 19.12.2023	
11	Оформлення графічного (ілюстрованого) матеріалу	20.12.2023	

7. Дата видачі завдання: "2" жовтня 2023 р.

Керівник кваліфікаційної роботи _____Володимир АРТЕМЧУК

(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____Софія КУПРІЄНКО

(підпис здобувача вищої освіти)

(П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Програмний модуль розпізнавання речей та обличчя пасажирів в аеропорту»: 87 сторінок, 20 рисунків, 1 таблиця, 27 використаних джерел.

КОМП'ЮТЕРНИЙ ЗІР, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, МОДЕЛЬ *YOLO*, МЕТРИКИ ОЦІНКИ ВИЯВЛЕННЯ, АЛГОРИТМИ ВИЯВЛЕННЯ ТА ВІДСТЕЖЕННЯ ОБ'ЄКТІВ, БАГАТОПОТОКОВІСТЬ.

Об'єкт дослідження – процес розпізнавання речей та обличчя пасажирів в аеропорту.

Предмет дослідження – модуль розпізнавання речей та обличчя пасажирів.

Метою кваліфікаційної роботи є розробка програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту.

Застосовано методи аналізу і порівняння, а саме, виконано теоретичне ознайомлення з історією розвитку комп'ютерного зору, систем відеоспостереження та існуючими результатами впровадження технології комп'ютерного зору в системах відеоспостереження. Виконано порівняння різних моделей нейронних мереж, і в ході порівняння було обрано найбільш прийнятну модель для реалізації. В ході розробки модулю використовувались методи об'єктно-орієнтованого програмування мови *Python*, додаткові фреймворки та набори інструментів такі, як наприклад *PyQt5*.

В результаті був змодельований програмний модуль, що може інтегруватися з наявними системами відеоспостереження. Основною задачею розробленого модулю є виявлення приналежності багажу людині та виявлення потенційних шахрайських чи терористичних дій пов'язаних з ним.

Дана система розроблена з метою застосування в людних громадських місцях, насамперед таких як аеропорти, вокзали, торгові центри тощо.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

SVM (Support Vector Machines) – алгоритм знаходження гіперплощини в *N*-мірному просторі, яка чітко класифікує точки даних.

ROI (Region of Interest) – частина зображення, яку необхідно відфільтрувати чи обробити.

NMS (Non Maximum Suppression) – метод комп'ютерного зору, що обирає один об'єкт з певної множини об'єктів, що перекриваються.

AP (Average Precision) – метод, що використовується для виміру точності детекторів об'єктів. Інтерпретується як площа нижче кривої точність-повнота.

mAP (mean Average Precision) – величина для оцінки продуктивності моделей машинного навчання і дорівнює середньому значенню показників середньої точності по всім класам моделі.

DeepSORT (Deep Simple Online Realtime Tracking) – сучасний алгоритм відстеження об'єктів, який містить в собі детектор об'єктів на основі глибокого навчання з алгоритмом відстеження.

IoU (Intersection over Union) – величина, що оцінює перекриття між передбачуваною областю дослідження і істинною.

Потоки програми – об'єкти всередині процесу, виконання яких можна запланувати, і в межах одного процесу використовують одну і ту ж пам'ять.

Процеси – екземпляри програми, що є незалежними і не мають доступу до однієї і тієї ж пам'яті.

Вбудовувач (*embedder*) – вбудовувані інструменти, що впроваджуються для роботи в код *Python* і надають додаткові можливості реалізації.

ВСТУП

Актуальність теми. Технології комп'ютерного зору дозволяють автоматизовано виділяти важливу інформацію з цифрових зображень чи відео. Комп'ютерний зір можна порівняти з людським зором, проте останній має переваги у вигляді контексту, тобто здатність відрізняти об'єкти один від одного, їх дальність розташування, рухомі характеристики тощо. Комп'ютерний зір навчає машини виконувати ці функції, після чого система може перевершувати людські можливості, оскільки зможе аналізувати тисячі об'єктів чи процесів за хвилину.

Комп'ютерний зір надає великі можливості для розробки додатків, метою яких є аналіз відео, візуальне спостереження, дистанційне зондування, доповнена реальність, візуальне роботизоване керування і автономні транспорті засоби – це лише декілька основних областей застосування.

Відеоспостереження почало розвиватися у вигляді аналогових систем відеоспостереження для збору інформації та спостереженнями за людьми, подіями та діяльністю. Більшість сучасних систем відеоспостереження забезпечують функціонал розпізнавання, зберігання та розповсюдження відео, тобто залишаючи завдання виявлення загроз операторам. Людський моніторинг є дуже трудомістким завданням, тому виникла необхідність в розробці і впровадженні інтелектуальних систем відеоспостереження. Проблеми безпеки та боротьби зі злочинністю є мотивуючим фактором для встановлення камер відеоспостереження.

Серед найбільш очікуваних розробок систем на основі технологій комп'ютерного зору – системи відеоспостереження для безпеки аеропорту. Великі аеропорти, як відомо, переповнені людьми і тому вразливі. Тому автоматичний аналіз і інтерпретація відеоматеріалів, отриманих сотнями камер, розміщених по всьому аеропорту, має вирішальне значення.

Серед найбільш затребуваних можливостей систем відеоспостереження на основі комп'ютерного зору є виявлення осіб, які демонструють незвичайну та

підозрілу поведінку, а також спостереження за різними об’єктами, які перевозяться в аеропорту. Ідентифікація та відстеження об’єктів, таких як валізи чи сумки, актуальні не лише для виявлення та запобігання крадіжкам, але й для боротьби з тероризмом.

Проблема стає більш складною, коли ми хочемо відстежувати взаємодію між людьми та об’єктами та виявляти незвичайні події, такі як здача об’єкта (багаж без нагляду в аеропортах), обмін сумками або вилучення об’єкта (крадіжка).

Протягом останнього десятиліття набуло розповсюдження технології відстеження кількох об’єктів (*Multiply Object Tracking*) що дозволяє знаходити та стежити за кількома об’єктами одночасно протягом деякого часу. Ця технологія використовується і в аеропортах. Наприклад для розпізнавання облич пасажирів що може застосовуватися при ідентифікації перед літаком, відстеження руху багажу що спрощує пошук втраченого чи залишеного багажу. Варіанти використання для спостереження можуть включати моніторинг потенційний загроз, виявлення покинутих предметів, автоматизоване виявлення вогню та диму, тощо.

Декілька популярних рішень практичного впровадження описаної технології включає:

– *DigiYatra* – децентралізована мобільна платформа для зберігання посвідчення особистості, де авіапасажири можуть зберігати свої посвідчення і проїзні документи. Обличчя пасажира стає його посадковим талоном чи посвідченням особи. Ця технологія також інтегрована з системою контролю авіакомпанії, тому тільки пасажири, що пройшли автентифікацію зможуть пройти контроль;

– Панорамна інфрачервона камера *Spynel* забезпечує безпеку аеропортів по всьому світу. В таких інфрачервоних системах використовується камера, що постійно обертається, внаслідок чого забезпечується детекція зображення на 360 градусів. Програмне забезпечення *Cyclope* виявляє і відстежує вторгнення в режимі реального часу, використовуючи зображення. Інфрачервоний датчик *Spynel*

здатний виявити людину на відстані кількох кілометрів у повній темряві, незважаючи на погодні умови;

– *K-Systems* – рішення, яке дозволяє безперервно відстежувати порушників або об'єкти, що становлять загрозу. Компанія дозволяє інтегрувати систему розпізнавання облич (для автентифікації), створювати віртуальну зону безпеки навколо вантажів, сповіщати про проникнення в приватну зону людей, що не мають до неї доступу та інші функції.

Актуальність розробки, що реалізується під час кваліфікаційної роботи, заключається в запропонованому механізмі підвищення безпеки з використанням нейронних мереж. Технологія буде реалізовувати виявлення відповідності особи і її особистих речей, при цьому реагуючи на зміну об’єму цих речей. Ця зміна розглядатиметься як можлива загроза безпеки аеропорту, тобто шахрайська чи терористична дія. Відповідне сповіщення надходитиме оператору, який буде розцінювати важливість цієї зміни об’єму і передавати дані службі безпеки аеропорту.

Мета і завдання кваліфікаційної роботи. Метою кваліфікаційної роботи є розробка програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту.

Для досягнення мети визначено наступні завдання:

1. Визначити актуальність та охарактеризувати застосування технологій комп’ютерного зору в безпеці аеропортів;
2. Підібрати технології для реалізації програмного модулю;
3. Здійснити програмну реалізацію модулю;
4. Підбір тренувальних даних, а саме відеопотоків та фото;
5. Аналіз результату роботи реалізованого модулю.

Об’єкт і предмет дослідження. Об’єкт дослідження – процес розпізнавання речей та обличчя пасажирів в аеропорту. Предмет – модуль розпізнавання речей та обличчя пасажирів.

Методи дослідження. Для реалізації обраного модулю спершу було застосовано методи аналізу і порівняння, а саме, виконано теоретичне

ознайомлення з історією розвитку комп'ютерного зору, систем відеоспостереження та існуючими результатами впровадження технології комп'ютерного зору в системах відеоспостереження. Окрім цього виконано порівняння різних моделей нейронних мереж, і в ході порівняння було обрано найбільш прийнятну модель для реалізації.

В ході розробки модулю використовувались методи об'єктно-орієнтованого програмування мови *Python*, додаткові фреймворки та набори інструментів такі, як наприклад *PyQt5*.

Наукова новизна отриманих результатів. В результаті виконання роботи вирішено актуальне науково-практичне завдання розроблення програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту. Отримано наступні наукові результати:

1. Запропоновано архітектуру програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту, що, на відміну від існуючих програмних рішень, забезпечує стеження не лише за пасажиром та його речами, а й встановлення приналежності (тобто визначення того, що окремий багаж належить саме цьому пасажиру), та реагування на зміну розміру чи типу багажу;
2. Розроблено нейронну мережу для встановлення приналежності багажу конкретній людині, що надає можливість реалізувати процес одночасного трекінгу пасажиру та його багажу, що, з однієї сторони, підвищує рівень безпеки, а, з іншої, зменшує навантаження на оператора системи;
3. Удосконалено розроблену нейронну мережу для відстеження зміни об'єму багажу, що належить конкретній людині, що надає можливість генерувати відповідні інформаційні повідомлення оператору у разі помітних відхилень такого об'єму з метою додаткової перевірки.

Практичне значення отриманих результатів. Розроблено програмний модуль розпізнавання речей та обличчя пасажирів. Дана система розроблена з метою застосування в людних громадських місцях, насамперед таких як аеропорти, вокзали, торгові центри тощо. Система ідентифікує особу, а після її багаж. Після цього відстежується переміщення особи та відбувається аналіз того

чи змінився її багаж. Якщо такі зміни відбуваються, то система сповіщає про це оператора, який аналізує ймовірний характер цієї зміни.

Особистий внесок випускника. Запропоновано архітектуру програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту, що, на відміну від існуючих програмних рішень, забезпечує стеження не лише за пасажиром та його речами, а й встановлення приналежності (тобто визначення того, що окремий багаж належить саме цьому пасажиру), та реагування на зміну розміру чи типу багажу.

Для цього визначено найбільш прийнятні моделі згорткових нейронних мереж що застосовуються в технологіях комп'ютерного зору. Обрано архітектуру згорткової нейронної мережі для реалізації модулю, а саме *YOLOv8*. Реалізовано систему, включаючи ініціалізацію моделі та налаштування гіперпараметрів, підбір найбільш прийнятного типу обраної моделі (за характеристиками швидкості і точності), створено користувацький інтерфейс, що представляє роботу оператора системи відеоспостереження. Окрім цього, підібрані відеозаписи з реальних веб-камер та інших джерел для тестування в умовах найбільш наближених до реальних. Додатково було проведено тестування на самостійно створеному відеозаписі.

Публікації.

Купрієнко С.П., Артемчук В.О. Роль комп'ютерного зору для забезпечення живучості автоматизованих систем відеоспостереження в авіаційній індустрії. Живучість та резильєнтність критичної інфраструктури – 2023 : збірник матеріалів міжнародної науково-практичної конференції, м. Київ, 19 жовтня 2023 р., ІПМЕ ім. Г.Є. Пухова НАН України. – 2023. – с. 101-103.

РОЗДІЛ 1
ЕВОЛЮЦІЯ РОЗВИТКУ КОМП'ЮТЕРНОГО ЗОРУ

1.1. Еволюція розвитку технологій комп'ютерного зору

Комп'ютерний зір – область штучного інтелекту, що застосовується для інтерпретації візуальної інформації. В свою чергу розвиток і впровадження штучного інтелекту, особливо в області глибокого навчання і нейронних мереж, дозволило комп'ютерному зору конкурувати з людським зором, але лише з точки зору розпізнавання об'єктів і образів.

Традиційний комп'ютерний зір використовував поверхневі методи машинного навчання і функції, створені вручну. В обмежених задачах і контрольованих ситуаціях ці методи працювали добре. Але у них були серйозні недоліки, такі як відсутність можливості узагальнювати свіжі, неперевірені дані.

Сучасний комп'ютерний зір навпаки заснований на моделях наскрізного навчання, таких як моделі глибокого навчання, які можуть вирішувати задачу, отримуючи зразки даних і керуючий сигнал. Дозволяючи створювати складні моделі, що здатні обробляти зображення і відеодані, моделі глибокого навчання змінили комп'ютерний зір.

В кінці 1950-х років перший цифровий сканер цифрових зображень перетворив зображення в числові сітки для комп'ютерної інтерпретації [1]. Після чого Лоренс Робертс, своєю доповіддю про виділення *3D*-даних із блочних *2D*-зображень, звернув увагу багатьох вчених на важливість розпізнавання зображень реального світу, що і призвело до фундаментальних досліджень в області сегментації і виявлення об'єктів.

Впровадження нейронних мереж для підвищення точності та ефективності в технології комп'ютерного зору розпочалися в 1970 роках. Комп'ютери змогли навчатися (тренуватися) на дуже великих наборах даних саме за допомогою нейронних мереж, імітуючи те, як людський мозок обробляє інформацію.

В 1980 році Куніхіко Фукусіма створив попередника сучасної згорткової нейронної мережі *(Convolutional neural Network – CNN)*, що назвав неокогнітроном (*neocognitron*) [2]. Після цього відкриття структури *CNN* стали ключовою архітектурою, завдяки якій відбувається розвиток комп'ютерного зору за допомогою нейронних мереж, зокрема структур *CNN*.

До появи глибокого навчання задачі, що могли виконати технології комп'ютерного зору, були дуже обмежені і вимагали великої кількості ручного кодування і втручань зі сторони розробників і операторів. Відповідно, і похибка була досить великою.

Машинне навчання надало інший підхід до вирішення проблем комп'ютерного зору. Завдяки машинному навчанню розробники більше не мали необхідності в великомасштабних ручних налаштувань в додатках з технологіями комп'ютерного зору. Замість цього розробники програмували певні функції, що представляли з себе невеликі додатки, які могли виявляти деякі закономірності в зображеннях. Потім вони використовували статичний алгоритм навчання, такий як лінійна регресія, логістична регресія, дерева рішень чи машини опорних векторів (*SVM*), для виявлення закономірностей, класифікації зображень і виявлення в них певних об'єктів. Глибоке навчання забезпечила принципово новий підхід до машинного навчання. Глибоке навчання спирається на нейронні мережі, які можна визначити як функцію загального призначення, яка може вирішити будь-яку проблему, яку можна представити прикладами. Коли розробники надають нейронній мережі множину помічених прикладів одного типу даних, вона зможе виділяти загальні закономірності між цими прикладами і перетворювати їх в математичне рівняння, яке допоможе класифікувати майбутні фрагменти інформації.

Глибоке навчання є дуже ефективним методом комп'ютерного зору, і в більшості випадків створення кісного алгоритму глибокого навчання зводиться до збору великої кількості навчальних даних і налаштуванні певних параметрів (наприклад, тип та кількість шарів нейронної мережі, епохи навчання тощо).

Сучасними знаковими моментами в комп'ютерному зорі є впровадження системи розпізнавання обличь *Facebook* в 2010 році, *TensorFlow* від *Google* в 2015 році (відкрита програмна бібліотека для машинного навчання, що використовується для задач автоматичного знаходження і класифікації образів).

Застосування комп'ютерного зору:

– комп'ютерний зір дозволяє безпілотним автомобілям розпізнавати навколишнє середовище. Камери розпізнають зображення під різними кутами навколо автомобіля і передають його в програмне забезпечення комп'ютерного зору, яка обробляє отримане зображення в режимі реального часу;

– комп'ютерний зір використовується для алгоритмів розпізнавання обличь, що реалізують задачі ідентифікації особи;

– комп'ютерний зір використовується в технологіях доповненої реальності, вирішуючи задачу накладання і вставки віртуальних об'єктів в зображення реального світу, визначаючи глибину і розміри об'єктів в фізичному світі;

– технології комп'ютерного зору значно покращили медичні технології, допомагаючи автоматизувати такі задачі, як виявлення ракових родимок на зображенні шкіри чи пошук симптомів на рентгенівських знімках чи МРТ.

1.2. Проблеми комп'ютерного зору

Технології комп'ютерного зору успішно використовуються в багатьох сучасних сферах, покращуючи та автоматизуючи безліч процесів, які раніше виконували люди. Проте дані технології мають деякі проблеми, які можна поділити на проблеми програмного забезпечення та апаратні проблеми.

1.2.1. Проблеми програмного забезпечення

Відомо, що моделі, орієнтовані на дані, і наскрізні моделі навчання походять від алгоритмів. З тих пір як були розроблені моделі глибокого навчання акцент

змінився з розробки алгоритмів на розробку, орієнтовану на дані. Замість того, щоб покладатися на функції, створені вручну, цей метод навчає моделі на великих наборах даних з анотаціями.

В нову епоху проблема ґрунтується на забезпеченні якості даних і здатності моделі узагальнювати нові, недосліджені дані. Доступність високоякісних даних є однією з найбільших перешкод, що постають перед сучасним комп'ютерним зором. Отримання високоякісних репрезентативних даних може виявитись найскладнішим етапом, і водночас він є дуже важливий для правильного функціонування моделі.

Якість даних можна вважати високою, якщо вони охоплюють широкий діапазон можливих сценаріїв і крайніх випадків. Окрім цього, дані, що використовуються для навчання повинні відповідати поставленим цілям

До проблем програмного забезпечення можна віднести погане планування створення моделі машинного навчання, що може проявлятися в тому, що бізнес модель:

- не відповідає бізнес-цілям;
- вимагає високої обчислювальної потужності;
- є високовартісною;
- забезпечує недостатню точність і продуктивність.

Окрім якості даних важливим є їх кількість. Об'єм даних, що необхідні для навчання варіюється в залежності від складності задачі і використовуваної моделі, адже складні задачі сегментації моделі глибокого навчання мають одні вимоги, тоді як проста задача класифікації – інші. В більшості варіантів використання краще використовувати великий об'єм навчальних даних, оскільки він дозволяє моделі вивчити більш надійні представлення задачі комп'ютерного зору.

1.2.2. Апаратні проблеми

Сучасний комп’ютерний зір має значні проблеми з проектуванням апаратного програмного забезпечення, що засновані на ціні, обчислювальній потужності і зручності використання. Розробка апаратного забезпечення для додатків комп’ютерного зору зіштовхується з серйозними проблемами з точки зору обчислювальної потужності. Обчислювальна потужність апаратної платформи повинна бути достатньою для ефективного виконання моделі глибокого навчання. Для багатьох додатків, включаючи виявлення об’єктів, розпізнавання облич, водіння автомобіля тощо є необхідність роботи з великими об’ємами даних і запуск складних алгоритмів в режимі реального часу.

Ще одна важлива проблема, яку необхідно враховувати при створенні обладнання для додатків комп’ютерного зору – зручність використання. Створення і впровадження засобів зв’язку, додатків і моделей повинно бути простим у використанні, здатним до адаптації і масштабування.

Підприємства мають значні перешкоди, пов’язані з витратами при інтеграції обладнання для додатків комп’ютерного зору. Слід ще враховувати, що технологія комп’ютерного зору реалізується за допомогою комбінації програмного і апаратного забезпечення. Щоб забезпечити ефективність системи, бізнесу зазвичай необхідно встановити камери і датчики високо розширення.

Кроки до оптимізації ціни можуть включати:

– зосередження на конфіденційності та безпеці: порушення цих характеристик може призвести до дуже високих витрат, хоча, з іншої сторони, впровадження кращих методів забезпечення безпеки також є високовартісним;

– використання попередньої обробки зображень: методи попередньої обробки зображень оптимізують і стандартизують зображення перед їх передачею в модель глибокого навчання, що підвищує точність моделі;

– використання комп’ютерного зору на периферії: даний варіант може бути більш рентабельним, особливо для масштабованих додатків з малою затримкою, в порівнянні з хмарними *API*;

– використання технології *low-code/no-code*: зменшення об’єму ручного написання коду прискорює доставку додатків, дозволяючи створювати, розгортати і оновлювати їх набагато швидше і з меншими ризиками;

– використання моделей глибокого навчання нового покоління: сучасні платформи роблять великий крок вперед в таких методах, як продуктивність виявлення об’єктів в реальному часі;

– інвестування в мультиплатформні рішення: для зниження витрат важливо мати можливість використовувати і обмінюватися міжплатформним обладнанням і програмних забезпеченням.

1.3. Еволюція систем відеоспостереження

Еволюцію систем відеоспостереження можна поділити на декілька фаз.

Першою фазою можна вважати перше задокументоване використання камери відеоспостереження в Німеччині в 1942 році у військових цілях, а саме за спостереженням за ракетами [3]. Використовувана архітектура складалася з монітору і камери і не мала можливості записувати зображення для подальшого перегляду.

Другою фазою можна вважати стрімкий розвиток 60-х років, що дозволив використання декількох камер, що підключалися до одного монітору, а впровадження розподіленої коробки дозволило користувачам перемикатися між декількома камерами на одному моніторі. Але одночасно можна було переглядати лише одну камеру. В 70-х роках з’явилися додаткові системи, такі як твердотілі камери, відеомагнітофони (*VCRs*) і мультиплексори [4]. Ці три системи вносять значний вклад в розвиток технології відеоспостереження. Твердотілі камери покращили загальну якість і надійність камер, мультиплексори дозволили користувачам спостерігати за декількома камерами одночасно на одному моніторі за допомогою роздільного екрану. Найбільш дороговартісним покращенням стала інтеграція технології відеомагнітофона, що дозволила записувати відзнятий

матеріал і контролювати його. Це означало зниження втручання людини від архітектури.

Вже створена екосистема технологій спостереження мала свої недоліки. Наприклад, якість записаних матеріалів було поганим, з зернистістю і низьким розширенням. Це зробило використання в цілях розпізнавання обличчя (наприклад, в правоохоронних органах) неефективним, оскільки з відзнятого матеріалу неможливо було зробити однозначні висновки через неякісний результат запису. Крім того, технологія відеомагнітофонів мала недолік однозадачності – оператори могли одночасно або переглядати запис або спостерігати в реальному часі. Для вирішення цих проблем з'явилась технологія цифрового відеозапису (*Digital Video Recording (DVR*)) [4]. Завдяки цифровому відеореєстратору якість відео стала набагато краща при більш високих показниках розширення, і це усунуло найбільшу перешкоду в архітектурі – поломка записуючих стрічок і необхідність їх постійної заміни. Технічно в системі *DVR* були вбудовані мультиплексори. Це дозволило скоротити кількість обладнання. Потім, сховище більше не будувалось на основі відеокасет, а використовувало цифрові накопичувачі, які пропонували більше місця для відзнятого матеріалу і автоматичне видалення через визначений час. Технологія відеореєстратору стала повністю автоматизована і не вимагала ручного втручання. Це дозволило операторам одночасно переглядати записані кадри і відслідковувати камери в реальному часі, отримати детальну інформацію про дату і час подій, тощо. Крім того, системи з підтримкою *IP* дозволили користувачам переглядати і керувати камерами віддалено.

Сучасне покоління технологій відеоспостереження набагато більш просунуті і поєднують різні вітки технологій:

– технології камери: *IP*-камери, камери високої чіткості, камери з функцією панорамування, нахилу і масштабування (*PTZ*), тепловізійні камери, камери на основі штучного інтелекту;

– технології зберігання: цифрові відеорегістратори (*DVR*), мережеві відеореєстратори (*NVR*), хмарне сховище;

– мережеві технології: інтернет, *Wi-Fi*, мобільний зв'язок;

– програмні технології: програмне забезпечення для керування відео (*VMS*), відеоаналітика, штучний інтелект і машинне навчання, інтернет речей (*Interner of Things* (*IoT*)), хмарне програмне забезпечення для спостереження [5].

1.4. Практичні результати впровадження комп'ютерного зору в системи відеоспостереження аеропортів

Безпека сучасних аеропортів залежить від ефективного моніторингу, аналізу і реагування на потенційні загрози. Інноваційні рішення дозволили аеропортам забезпечити цілодобовий нагляд за своєю територією. Використовуючи інтелектуальні камери високої чіткості і платформи керування відео, що підтримуються надійними серверами і рішеннями для зберігання даних, оператори аеропортів можуть отримувати детальне і багатостороннє представлення про свої операції, що дозволяє їм швидко діяти у випадку потенційних загроз безпеки.

Розглянемо компанії що впровадили комп'ютерний зір в системи відеоспостереження, що інтегруються в систему аеропортів.

Elbit Security System ("ELSEC") надає рішення для зовнішньої і внутрішньої безпеки і захисту об'єктів аеропортів і морських портів. Зовнішній периметр об'єктів контролюється за допомогою комбінації двох типів оптико-електронних систем спостереження: дальньої дії (*LORROSTM*) і денних і нічних систем ближнього/середнього радіусу дії (*SeROS*). Міри внутрішньої безпеки включають системи виявлення зловмисників всередині приміщень, системи контроля доступу, системи відеоспостережень, системи цифрового, відео- та
аудіозапису [6].

IndigoVision розробило систему відеоспостереження, шо уже впроваджена в більш чим 100 аеропортів. Їх камери працюють зі швидкістю 1 кадр в секунду, коли немає ніяких дій, і підвищують швидкість до 30 кадрів в секунду коли вони є. Система вирішує такі задачі як виявлення покинутих об'єктів чи тих що не пересувались підозріло довго, виявлення підвищеної завантаженості терміналу,

слідкує за несанкціонованим перетином обмежених зон в аеропортах, виявляє те, що хтось рухається через систему безпеки в зворотному напрямку [7].

Dallmeier пропонує широкий спектр рішень для багатьох складних процесів в аеропорту: від звичайного відеоспостереження в зоні безпеки до систем моніторингу перону і злітно-посадкової смуги. Це забезпечується за допомогою технології мультифокальних сенсорних камер *Panomera* компанії *Dallmeier*, що дозволяють захопити дуже великі просторові території зі значно меншими інфраструктурними і експлуатаційними витратами, чим це було раніше [8].

1.5. Висновки до розділу

В даному розділі охарактеризовано еволюцію розвитку комп'ютерного зору. Різні етапи еволюцію охарактеризовано за допомогою провідних досягнень відповідних років. Тобто застосування цифрового сканеру в 1950 році, впровадження нейронних мереж в 1970 році, створення попередника сучасних згорткових нейронних мереж в 1980 році. Після цього означено впровадження машинного навчання в технології комп'ютерного зору і оцінено ефективність цього впровадження.

Окрім вище наведених характеристик визначено і проблеми використання технологій комп'ютерного зору, що були поділені на проблеми програмного забезпечення і апаратні проблеми.

Нашою метою є розробка модулю розпізнавання необхідних нам образів для систем відеоспостереження, тому надалі охарактеризовано еволюцію систем відеоспостереження. Еволюцію цих систем описано також за допомогою провдіних досягнень, таких як використання камери відеоспостереження в 1942 році у військових цілях, впровадження розподіленої коробки, застосування твердотілих камер, відеомагнітофонів та мультиплексорів, поява технології цифрового відеозапису.

Останньою метою, що відведена цьому розділі є характеристика уже відомих практичних результатів впровадження технологій комп'ютерного зору в

системи відеоспостереження саме аеропортів. Ми розглянули такі системи та технології як *ELSEC, IndigoVision, Dallmeir*.

РОЗДІЛ 2
АНАЛІЗ РІШЕНЬ ДЛЯ РОЗПІЗНАВАННЯ ОБРАЗІВ

2.1. Моделі нейронних мереж для застосування в комп’ютерному зорі

Різні моделі нейронних мереж мають різні можливості і властивості, які роблять їх придатними для різних завдань у сфері комп'ютерного зору та обробки зображень. Ось кілька основних типів моделей та їхні можливості:

– згорткові нейронні мережі (*Convolutional Neural Networks* (*CNN*)) – клас нейронних мереж, що спеціалізується на обробці даних, що мають сітчасту топологію, тому вони часто використовуються для класифікації об'єктів на зображеннях, ефективні в розпізнаванні образів та фільтрації зображень та здатні впізнавати патерни та ознаки на зображеннях;

– рекурентні нейронні мережі (*Recurrent Neural Networks* (*RNN*)) – клас нейронних мереж, що використовує послідовні дані чи дані часових рядів. Використовуються для обробки послідовних даних, таких як текст або часові ряди. Окрім цього вони можуть запам'ятовувати попередні стани та використовувати їх для прийняття рішень та підходять для завдань генерації тексту, машинного перекладу та аналізу послідовностей;

– трансформери (*Transformers*) – новітня архітектура нейронних мереж метою яких є рішення послідовних задач. Даний тип нейронних мереж заснований на механізмах уваги. Вони призначені для обробки послідовних та не послідовних даних та широко використовуються у завданнях обробки природної мови (*NLP*) та машинного перекладу. Окрім цього вони здатні до паралельної обробки даних і виявлення складних зав’язків;

– мережі кодувальника-декодувальника (*Encoder-decoder networks*) – особлива форма мереж *CNN*, що широко використовуються для сегментації

зображень, сумісної реєстрації і зменшення артефактів. Архітектура включає в себе двоетапний процес, в якому вхідні дані спочатку кодуються в числове представлення фіксованої довжини, яке потім декодується для отримання вихідних даних, що відповідають бажаному формату.

Це лише кілька прикладів різних типів моделей, існує безліч інших архітектур і алгоритмів, які можуть бути використані для різних завдань обробки зображень та аналізу даних. Вибір моделі повинен враховувати конкретні потреби та характеристики нашої задачі для досягнення оптимальних результатів тому за основу звісно буде взято згорткову нейронну мережу.

Наразі на ринку існують різні моделі та аналоги для завдань визначення облич людини та моніторингу їхнього багажу. Кожна з цих моделей має свої унікальні особливості, метрики та якість їх використання.

Однією з популярних альтернатив є *SSD* (*Single Shot MultiBox Detector*), яка відома своєю швидкістю та точністю в виявленні об'єктів. Вона має високі метрики точності в реальному часі та відома своєю здатністю розпізнавати об'єкти на зображеннях.

Іншою альтернативою є *Faster R-CNN* (*Region-based Convolutional Neural Network*), що використовує регіони областей для виявлення об'єктів. Ця модель має високу точність та здатність визначати об'єкти з високою деталізацією.

Також варто відзначити *Mask R-CNN*, яка додає до *Faster R-CNN* можливість сегментації об'єктів. Це дозволяє визначати не тільки положення об'єктів, але і їхні контури, що є важливим для деяких завдань.

Щодо метрик і якості використання, важливо враховувати, що кожна модель має свої характеристики та метрики, такі як точність (*precision*), відгук (*recall*), *F1*-оцінка, а також швидкість роботи.

В області комп’ютерного зору останні роки відбувся значний прогрес, завдяки досягненням в області глибокого навчання, а саме згорткових нейронних мереж.

Клас нейронних мереж що спеціалізуються на обробці даних, що мають сітчасту топологію, такі як зображення – згорткові нейронні мережі (*CNN, ConvNet*) [10].

Переваги згорткових нейронних мереж:

– можуть вчитися на необроблених пікселях даних, не вимагаючи ручного проектування функцій чи попередньої обробки. Тобто вони можуть автоматично виявляти і адаптуватися до найбільш важливих характеристик зображень. Таким як краї, форми, кольори, текстури і об’єкти;

– зменшення розмірність і складність вхідних даних, роблячи навчання і вивід більш швидким і ефективним;

– можуть використовувати просторову і ієрархічну структуру зображення, використовуючи фільтри, що зберігають локальну прив’язаність і контекст пікселів, а також створюючи більш абстрактні і високорівневі представлення по мірі їх заглиблення в мережі. Це дозволяє даному типу мережі розпізнавати зміну і різноманіття зображень, а також гарно узагальнювати нові і раніше невідомі дані.

Недоліки використання згорткових нейронних мереж:

– для ефективного навчання їм потрібний великий об’єм даних, що може бути високовартісним і трудомістким процесом;

– схильні до перенавчання, тобто вони можуть запам’ятовувати шум і деталі навчальних даних і не можуть узагальнювати нові і різні дані. Щоб уникнути перенавчання, необхідно застосовувати різні методи регулювання, такі як нормалізація, фільтрація, збільшення даних, що може збільшити складність обчислювальних витрат мережі;

– часто їх вважають чорними ящиками, що означає що їх складно інтерпретувати і пояснювати.

Багато сучасних моделей, що використовуються для вирішення задач відеоспостереження, використовують архітектуру глибокого навчання для досягнення високоякісних результатів. Тим не менш правильний компроміс між швидкістю і точністю при побудові конкретної моделі для цільового варіанта

використання це одне з найважливіших рішень для побудови ефективного програмного забезпечення.

Існує два розповсюджених підходи до виявлення об’єктів: одноступеневі та двоступеневі методи.

Модель двохетапного виявлення складається з двох етапів: пропозиція регіону, а потім класифікація цих регіонів і уточнення прогнозу місцерозташування. Одноетапне виявлення пропускає етап пропозиції регіону і відразу дає кінцеву локалізацію і прогнозування контенту. *Faster-RCNN* є популярним вибором для двохетапних моделей, в той час як *SSD* і *YOLO* – популярні одноетапні підходи. Архітектура *YOLO* швидше *SSD*, але менш точна.

R-FCN – ще одна популярна двохетапна метаархітектура. В цьому підході мережа пропонованих регіонів (*Region Proposal Network – RPN*) пропонує можливі *ROIs* (регіони інтересів), які потім застосовуються на картах оцінок. Всі шари що навчаються є згортковими і обраховуються для всього зображення. Майже всі обрахунки різних пропонованих регіонів є загальними. Обрахункові витрати на кожен *ROI* незначні в порівнянні з *Faster-RCNN. R-FCN* представляє свого роду гібрид одноетапного і двохетапного підходу [11].

В той час як моделі двохетапного виявлення забезпечують більш високу продуктивність, одноетапне виявлення знаходиться в оптимальному положенні по продуктивності і швидкості/ресурсам. Крім того *SSD* навчається швидше і має більш швидкий вивід, чим двохетапний детектор. Більш швидке навчання дозволяє досліджувати ефективно створювати прототипи і експериментувати, не витрачаючи значних коштів на хмарні обрахунки. Що не менш важливо, властивість швидкого виводу зазвичай є вимогою, коли мова йде про додатки реального часу. Так як він вимагає менше обрахунків, він споживає набагато менше енергії на прогноз. Цей час і енергоефективність забезпечує широкий спектр застосувань, особливо на кінцевих приладах, і позиціонують *SSD* як кращий підхід до виявлення об’єктів в багатьох випадках.

SSD використовує мережу *VGG-16* в якості магістралі і модифікує її, замінюючи останні два повнозв’язних шари згортковими шарами, а також

додаючи ще чотири згорткових шари пізніше, щоб остаточно сформувати мережу виявлення ознак як *Conv4_3, Conv7, Conv8_2, Conv9_2, Conv10_2*, і *Conv11_2*, розміри яких складають (38, 38), (19, 19), (10, 10), (5, 5), (3, 3) і (1, 1) відповідно [12]. Схематично це позначено на рис. 2.1.

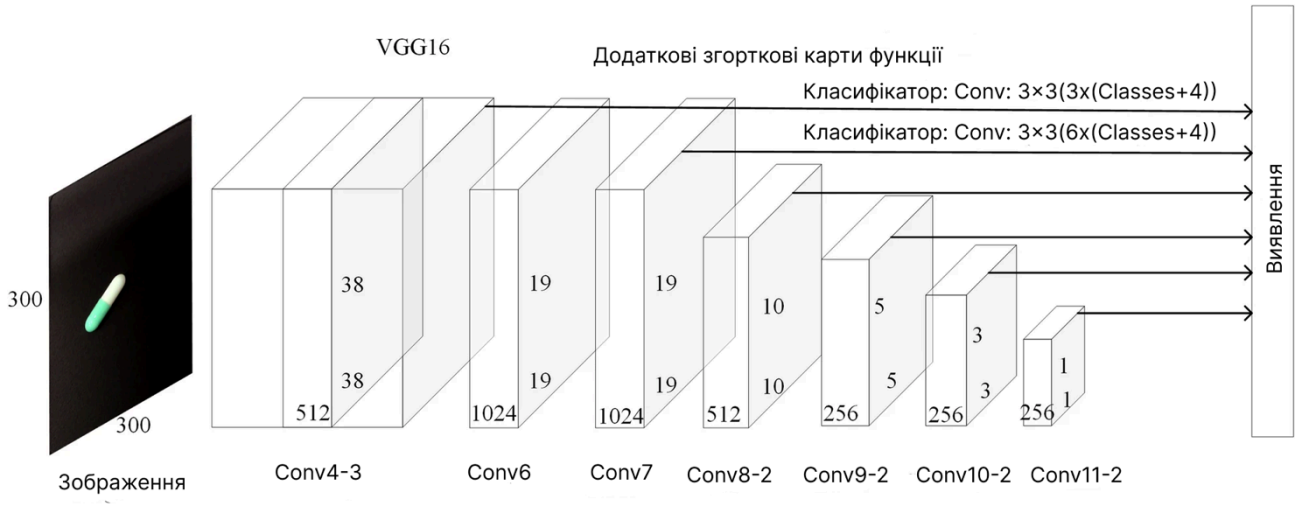

Рис. 2.1. Основна мережа алгоритму *SSD*

SSD навчений отримувати набір обмежуючих рамок фіксованого розміру і оцінки прогнозування класів цілей в обмежуючих рамках. Потім надлишкові обмежуючі рамки відфільтровуються, і остаточні результати виявлення генеруються за допомогою алгоритму немаксимального пригнічення (*NMS*), який дає гарні результати з точки зору швидкості та точності виявлення.

YOLO – набір популярних алгоритмів комп'ютерного зору. Він включає в себе декілька задач, таких як класифікація, сегментація зображень і виявлення об'єктів. *YOLO* використовує одноетапні моделі виявлення для обробки зображень за один прохід і виведення відповідних результатів. *YOLO* є частиною сімейства одноетапних моделей виявлення об'єктів, які обробляють зображення по шаблонам згорткової нейронної мережі.

Архітектура *YOLO* заснована на моделі *GooLeNet* для класифікації зображень. Ця мережа має 24 згорткових шари, за якими слідує 2 повнозв'язних шари, що показано на рис. 2.2.

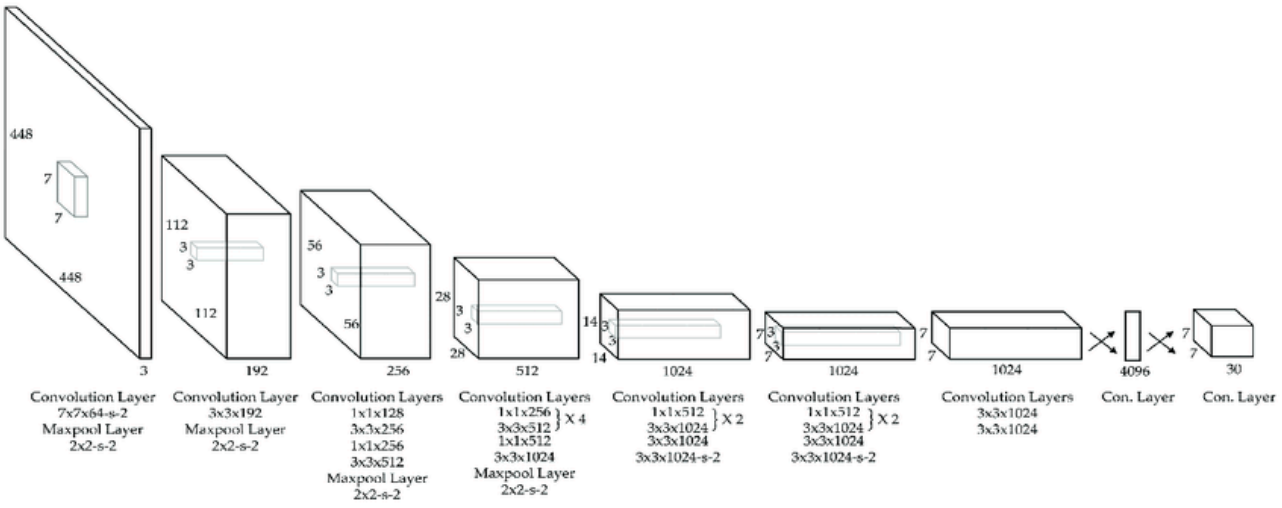

Рис. 2.2. Загальна архітектура мережі алгоритму *YOLO*

Спочатку розмір зображення змінюється до розміру *L* x *L* (на практиці 448 x 448) і ділиться на форму сітки *S* x *S* (*S* = 7, тому кожний фрагмент зображення 64 x 64 належить одній сітці). Кожна сітка пропонує *B* обмежуючих рамок (на практиці зазвичай *B* = 2). Кожна обмежуюча рамка відповідає чотирьом координатам (x_{center}, y_{center}, w, h) і 1 довіреному значенню. x_{center} і y_{center} – це координати центра обмежуючої рамки, а w і h – це ширина і висота обмежуючої рамки. Значення достовірності розраховується за формулою (2.1).

$$Confisence = Pr(Object)\, x\, IOU_{pre}^{truth}, \tag{2.1}$$

де IOU_{pre}^{truth} – це значення між прогнозованими обмежуючими прямокутниками і фактичним істинним блоком.

Якщо в комірці є істинний блок то $Pr(Object)$ дорівнює одиниці, інакше рівний нулю.

Крім того кожна сітка генерує *C* ймовірність умовного класу $Pr(Class_i Object)$ (на практиці *C* = 20). Тому зображення в кінцевому рахунку дає *S* x *S* x *(B* x *5 + C)* виходів (на практиці 7 x 7 x (2 x 5 + 20) = 1,470 виходів).

Під час тестування *YOLO* перемножує ймовірності умовних класів і прогнози ймовірності окремих блоків, відповідно до формули (2.2) що дає нам оцінки ймовірності для кожного обмежувального прямокутника [13].

$$Pr(Class_i Object)\ x\ Pr(Object)\ x\ IOU_{pre}^{truth} = Pr(Class_i)\ x\ IOU_{pre}^{truth} \quad (2.2)$$

Остання версія *YOLO – YOLOv8*. Її рекомендують використовувати так як продуктивність (*mAP*), швидкість (*FPS*) і точність вища, а обрахункові витрати нижчі

YOLOv8 – найновітніше сімейство моделей виявлення об’єктів на основі *YOLO* від *Ultralystic*, що забезпечує найефективніші характеристики. Ця модель представляє собою значний крок вперед у розробці систем виявлення об'єктів завдяки своїм передовим можливостям та оптимізації, тому ми зупинили свій вибір саме на цій моделі.

2.2. Особливості роботи *YOLOv8*

Архітектуру *YOLOv8* можна поділити на основні частини: *head, neck, backbone.* Розглянемо ці частини детальніше:

1. *Head YOLOv8* складається з декількох згорктових шарів за якими слідує ряд повністю зв’язаних шарів. Ці шари відповідають за прогнозування обмежувальних рамок, оцінок об’єктності і ймовірностей класів для об’єктів, виявлених на зображенні. Саме один із ключових аспектів *YOLOv8* є використання *Anchor-free Split Ultralytics Head*, що сприяє підвищенню точності і більш ефективному процесу виявлення в порівнянні з підходами з прив’язкою. Крім цього *head* характеризується механізмом самообслуговування. Цей механізм дозволяє моделі фокусуватися на різних частинах зображення і регулювати важливість різних функцій в залежності від їх відповідності задачі;

2. *Neck* – компонент що з’єднує *backbone* з *head.* Його роль заключається в зменшенні розміру карти об’єктів і збільшенні розширення об’єктів. *Neck* призначена для пошуку балансу між точністю і швидкістю, що робить *YOLOv8* швидким і ефективним. Це особливо важливо для задач знаходження об'єктів у реальному часі, коли ви хочете, щоб система була якнайшвидшою без втрати точності;

3. *Backbone* – основа *YOLOv8* в якості якої використовується модифікований *CSPDarknet53*. Ця архітектура складається з 53 згорткових шарів і використовує метод, що називається міжетапними частковими з'єднаннями, для покращення потоку інформації між різними рівнями мережі [14].

Використання передових архітектурних рішень для вище розглянутих компонентів дозволяє краще виділяти ознаки зображень та покращує точність виявлення об'єктів. Це означає, що модель може більш ефективно розпізнавати об'єкти навіть на зображеннях низької якості або в ускладнених умовах.

Ще однією важливою особливістю *YOLOv8* є його здатність виявляти великомасштабні об'єкти. Модель використовує мережу пірамідних об'єктів для виявлення об'єктів різних розмірів і масштабів на зображенні. Ця пірамідальна мережа об'єктів складається з декількох шарів, які виявляють об'єкти в різних масштабах, що дозволяє моделі виявляти великі і маленькі об'єкти на зображенні.

YOLOv8 поставляється з низкою попередньо навчених моделей, що включають в себе різні архітектури та резолюції. Це дає можливість вибрати найкращу модель для конкретного завдання.

YOLOv8 має певні особливості реалізації. Вона працює, спершу розділяючи вхідне зображення на сітку комірок. Для кожної комірки відбувається прогнозування набору обмежуючих рамок, а також ймовірності класів для кожної обмежуючої рамки. Потім *YOLOv8* використовує алгоритм *NMS* для фільтрації перекриваючих обмежуючих рамок і вибору найбільш ймовірних обмежувальних рамок для кожного об'єкта на зображення.

Ефективність обраної моделі можна оцінити за допомогою галузевого стандарту оцінки моделей виявлення об'єктів, що представляє собою масштабний набір даних для виявлення об'єктів, сегментації, виявлення ключових точок – *COCO (Common Object in Context)*. При порівнянні моделей на *COCO* необхідно дивитися на значення *mAP* і *FPS*. На момент написання даного розділу точність *COCO YOLOv8* є найбільш сучасною для моделей зі співставленими затримками виводу, що показано на рис. 2.3.

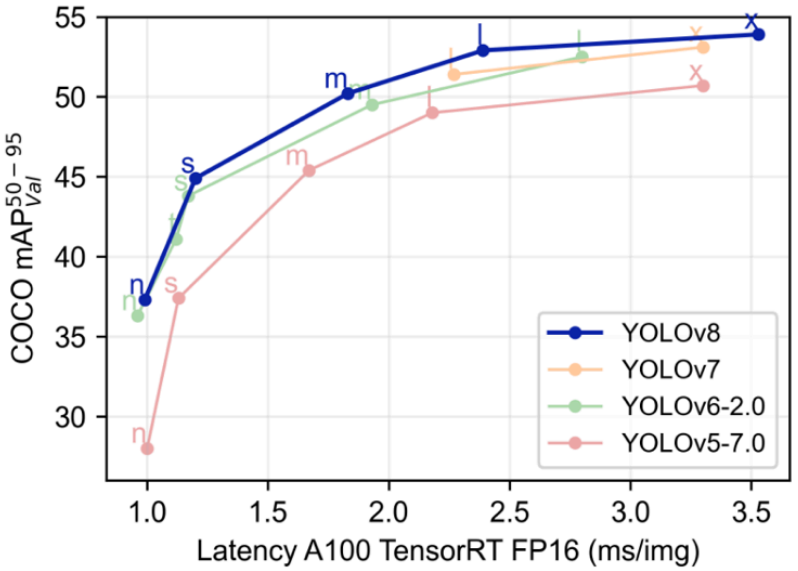

Рис. 2.3. Порівняння різних версій *YOLO*: вісь *x* – час який необхідний нейронній мережі для видачі прогнозу для однієї вихідної вибірки, що має кращі характеристики при менших значеннях; вісь *y* – середня точність, що має кращі характеристики при більших значеннях

Порівняно з іншими версіями моделі в продуктивність *YOLOv8* були внесені значні покращення з точки зору швидкості, точності і архітектури.

2.3. Додаткові засоби та методи розробки

Python – одна з самих популярних мов програмування в області штучного інтелекту та, зокрема, комп'ютерного зору завдяки своїй простоті, універсальності і великою екосистемою бібліотек і фреймворків.

Мова *Python* – інтерпретована мова програмування загального призначення. В мові реалізований об'єктно-орієнтований підхід, який дозволяє програмістам писати логічний і зрозумілий код. Він представляє масштабну екосистему бібліотек і платформ, таких як *TensorFlow*, *PyTorch*, які спрощують реалізацію складних моделей машинного навчання [16].

Переваги *Python:*

1. Призначений як для нових так і для досвідчених розробників, адже *Python* має простий синтаксис;

2. *Python* має вбудований інструмент для відладки *PDB*, який робить відладку коду на цій мові програмування більш зручною і доступною;

3. *Python* оснащений вбудованими бібліотеками, що забезпечують спрощений підхід до розробки. Це дозволяє розробникам зосередитися на створенні додатків на основі штучного інтелекту, а не витрачати час на вивчення і реалізацію базової структури.

Слід зазначити систему наслідування *Python,* що дозволяє створювати класи, які можуть успадковувати властивості та методи інших класів. Наслідування є однією з основних концепцій об'єктно-орієнтованого програмування (ООП) і дозволяє створювати ієрархії класів, де батьківські класи передають свої характеристики дочірнім класам.

Основні аспекти системи наслідування в *Python*:

1. Базовий та похідний класи, що представляють собою основу для технології успадкування властивостей та методів;

2. Спадкування властивостей і методів: похідний клас може успадковувати атрибути та методи з базового класу. Це означає, що об'єкти похідного класу матимуть доступ до всіх атрибутів і методів базового класу, які були оголошені як доступні для успадкування;

3. Розширення функціональності: похідний клас може додавати нові атрибути і методи, розширюючи функціональність базового класу;

4. Перевизначення методів: похідний клас може перевизначати (перевантажувати) методи базового класу, надаючи їм нову реалізацію. Це називається поліморфізмом і дозволяє викликати різні реалізації одного методу в залежності від типу об'єкта;

5. Ієрархія класів: можна створювати складні ієрархії класів, де декілька класів успадковують властивості та методи з одного або декількох батьківських класів;

6. Класи-міксіни: в *Python* можна використовувати класи-міксини, які містять функціональність, яка може бути додана до інших класів через успадкування.

Загалом, система наслідування дозволяє створювати гнучкі та повторно використовувані класи в програмах *Python*. Вона допомагає організовувати код у більш структуровані ієрархії та сприяє поліморфізму та абстракції, що є ключовими концепціями ООП.

В процесі реалізації модуля ми будемо оперувати поняттям потік, тому необхідно розглянути механізми багатопотоковості в *Python.*

Виконувана програма називається процесом. Процес може складатися з декількох потоків виконання. Потік – незалежний потік виконання, або ж екземпляр процесу.

І багатопроцесорність і багатопотоковість використовуються для збільшення обчислювальної потужності системи.

Багатопотоковість – процес при якому потоки процесу виконуються одночасно, а створення процесу в багатопотоковості здійснюється відповідно до економічності. Багатопотоковість *Python* полегшує спільне використання простору даних і ресурсів декількох потоків з основним потоком, що забезпечує ефективний просторовий зв’язок між потоками [16].

Окрім багатопотоковості існує процес багатопроцесорності – здатність процесора одночасно виконувати декілька незв’язаних процесів. Ці процеси не використовують спільні ресурси. Багатопроцесорінсть розбиває процеси на дрібні підпрограми, які виконуються незалежно. Багатопроцесорність гарантує що кожен процесор отримає своє власне ядро процесора і що виконання буде безперешкодним [16].

Таблиця 2.1

Порівняння багатопотоковості і багатопроцесорності

Багатопотоковість	Багатопроцесорність
Метод при якому процес одночасно відтворює декілька потоків	Метод, при якому декілька процесів одночасно виконуються на декількох процесорах/ядрах процесора
Реалізує паралелізм	Реалізує паралелізм в його істинній формі

Закінчення таблиці 2.1

Створення ілюзії, що потоки працюють паралельно, хоча насправді – одночасно.	Модуль багатопроцесорності полегшує паралельний запуск незалежних процесів за рахунок використання підпроцесів
При багатопотоковості *GIL* попереджує одночасну роботу потоків	При багатопроцесорній обробці кожен процес має власний інтерпретатор *Python,* що реалізує виконання

Використовуючи багатопотоковість, всі важливі аспекти такі як продуктивність, рендер, швидкість і витрата часу, будуть значно покращені за рахунок використання правильної багатопотоковості.

Розглянемо деякі додаткові матеріали та методи, які ми будемо використовувати в подальшій реалізації модулю розпізнавання людей і їх багажу.

2.3.1. *Qt*

Qt для *Python* або *PyQt* – набір прив'язок *Python* для платформи додатків *Qt* і набір інструментів для створення додатків з графічним інтерфейсом. Ці прив'язки і інструменти доступні для всіх платформ, тому можна розробляти додатки з графічним інтерфейсом для *Windows*, *MacOS* та *GNU/Linux* [17].

Засоби виведення даних та взаємодії з системою через фреймворк *PyQt5* відкривають широкі можливості для створення зручного та інтуїтивно зрозумілого інтерфейсу для користувача.

За допомогою *PyQt5* можна створити вікно програми, яке буде служити головним інтерфейсом для відображення даних та взаємодії з системою. В цьому вікні можуть бути розміщені віджети, такі як тексти для виведення інформації про обличчя та багаж, кнопки для виконання дій, та інші елементи, які забезпечать зручну роботу з системою.

Завдяки обробникам подій, можна реалізувати взаємодію користувача з системою. Наприклад, користувач може вибрати групу обличь людей та отримати детальну інформацію про їхній багаж, або запустити процес аналізу на нових зображеннях.

PyQt5 дозволяє також налаштовувати вигляд та розмір вікон, кольори та шрифти для створення естетично приємного інтерфейсу.

Завантаження та збереження даних можуть бути реалізовані через інтерфейс програми, що робить цей процес зручним та доступним для користувача.

За допомогою візуалізації даних можна створювати графіки та діаграми, які допоможуть користувачу краще розуміти і аналізувати інформацію.

PyQt5 – це потужна бібліотека для створення графічних інтерфейсів користувача (*GUI*) в програмах на мові програмування *Python*. Вона забезпечує широкі можливості для розробки відмінних інтерфейсів для десктопних додатків на платформі *Windows, macOS* і *Linux.* Наведемо детальний опис можливостей *PyQt5*:

– багатофункціональність: *PyQt5* включає в себе безліч готових елементів управління, таких як кнопки, поля для тексту, таблиці, списки, вкладки, вікна, діалоги і багато інших. Це дозволяє легко створювати різні види інтерфейсів;

– підтримка різних платформ: *PyQt5* надає можливість розробляти мульплатформні додатки, які працюють на *Windows, macOS* і *Linux* без змін у джерелах;

– розширені можливості кастомізації: можна вільно налаштовувати вигляд і поведінку елементів управління, задавати стилі, змінювати шрифти, кольори та багато інших параметрів;

– підтримка подій і сигналів: *PyQt5* має потужну систему обробки подій і сигналів, що дозволяє відстежувати взаємодію користувача з інтерфейсом і реагувати на неї;

– широкий набір модулів: бібліотека містить модулі для графіків, мультимедіа, мережевого програмування, роботи з базами даних, операційної системи і багатьох інших завдань;

– мови програмування: *PyQt5* підтримує використання мови програмування *Python* і *QML* (*Qt Meta-Object Language*) для опису інтерфейсів;

– підтримка мультиплатформних додатків: можна створювати інтерфейси, які адаптовані для мобільних пристроїв і планшетів, що робить *PyQt5* відмінним вибором для розробки мультиплатформних додатків;

– спрощене тестування: *PyQt5* надає зручні засоби для тестування інтерфейсу, що допомагає виявляти і виправляти помилки швидше;

– документація та спільнота: *PyQt5* має велику та активну спільноту користувачів і розробників, а також високоякісну документацію, що допомагає знайти відповіді на питання та вирішити проблеми.

PyQt5 володіє потужною сигнальною системою, яка дозволяє взаємодіяти з подіями і сигналами у *GUI* додатку. Ця система базується на патерні "спостерігач" (*Observer pattern*) і грає важливу роль у реакції на події в інтерфейсі [17].

Основні аспекти сигнальної системи в *PyQt5*:

– сигнали і слоти: можна визначити сигнали в об'єктах (наприклад, кнопка має сигнал "натиснута"), і можна підключити ці сигнали до слотів (функцій), які будуть викликані при виникненні сигналу. Наприклад, можна підключити сигнал "натиснута" кнопки до функції, яка виконує певну дію;

– події і обробники подій: *PyQt5* також підтримує обробку подій, де можна прив'язувати функції (слоти) до певних подій, таких як "клацання мишею", "натискання клавіші", "закриття вікна" і багато інших. Це дозволяє реагувати на взаємодію користувача з інтерфейсом;

– передача даних через сигнали: сигнали можуть передавати дані, тобто можна використовувати їх для передачі інформації між об'єктами в вашій програмі. Наприклад, при натисканні на кнопку можна передати певні дані іншому об'єкту для обробки;

– каскадні сигнали: *PyQt5* підтримує можливість каскадного спуску сигналів, де один сигнал може викликати інший сигнал або низку сигналів, що розширює можливості обробки подій;

– використання в *QML*: *PyQt5* дозволяє використовувати сигнали і слоти для взаємодії з *QML*, що робить його потужним інструментом для розробки мультиплатформних додатків, які об'єднують декларативне ім'я з імперативною логікою [17].

Ця сигнальна система робить *PyQt5* ідеальним вибором для розробників, які потребують динамічної взаємодії між елементами інтерфейсу та бажають легко керувати подіями та даними у своєму додатку.

Проекти зберігаються в файлах `.*ui*`, що представляють собою *XML*-файли, які створюються і використовуються в *Qt Designer* для опису графічного інтерфейсу користувача (*GUI*). Ці файли містять інформацію про вікна, віджети, їх розміщення і властивості. Вони відіграють важливу роль у розробці *GUI*-додатків з використанням *PyQt5*.

Основні аспекти файлів`.*ui*`:

– візуальний редактор: для створення `.*ui*-файлів можна використовувати візуальний редактор, такий як *Qt Designer*, що дозволяє перетягувати та розміщувати віджети на формі, налаштовувати їх властивості та з'єднувати сигнали та слоти;

– декларативний опис інтерфейсу: файли `.*ui*` містять декларативний опис елементів інтерфейсу. Можна визначати, які віджети будуть використовуватися, як вони розміщені на вікні, які властивості мають і багато іншого;

– роздільність і вигляд: можна визначити роздільність (розміри) вікна і його вигляд, включаючи фоновий колір і зображення, що використовуються в якості фону;

– робота з віджетами: можна додавати віджети, такі як кнопки, текстові поля, списки тощо, та налаштовувати їх властивості, такі як текст, розмір шрифту, кольори і тощо;

– сигнали та слоти: можна з'єднувати сигнали віджетів зі слотами (функціями), що будуть викликані при виникненні певних подій. Це дозволяє створювати інтерактивні додатки;

– мова програмування: файли `*.ui*` можуть бути інтегровані з *PyQt5*, і можна створювати функції в *Python* для обробки подій та взаємодії з інтерфейсом, який описаний у `*.ui*`-файлі;

– мова опису: синтаксис *XML* використовується для опису вмісту файлів `*.ui*`, що робить їх легкими для редагування і розуміння, навіть без використання візуального редактора;

– генерація *Python* коду: файли `*.ui*` можуть бути конвертовані в *Python*-код за допомогою утиліти `*pyuic5*`, що дозволяє імпортувати інтерфейс з `*.ui*`-файлів у *Python*-додаток [18].

Файли `*.ui*` є важливою складовою для створення та редагування *GUI* в *PyQt5*, і вони спрощують процес розробки інтерфейсу для додатків.

Усі ці можливості роблять *PyQt5* потужним інструментом для розробки десктопних додатків з вдосконаленими графічними інтерфейсами для широкого спектра застосувань. З її допомогою можна створювати як прості віконні додатки, так і складні програми зі складними інтерфейсами.

2.3.2. Алгоритм *DeepSort*

DeepSort – це потужний алгоритм відстеження об'єктів, вже вбудований в системи комп'ютерного зору та обробки відео. Цей алгоритм відіграє важливу роль у відеоспостереженні, аналізі відео та розпізнаванні рухомих об'єктів.

Особливості роботи *DeepSort*:

– відстеження об'єктів: *DeepSort* використовує нейронні мережі для розпізнавання та відстеження об'єктів на відеопотоці. Алгоритм визначає рухомі об'єкти, виділяє їх на відео та встановлює їхні унікальні ідентифікатори для подальшого відстеження;

– асоціація об'єктів: *DeepSort* вирішує проблему асоціації об'єктів у різних кадрах відеопотоку. Він використовує інформацію про положення та ознаки об'єктів, щоб встановити зв'язок між ними в різних кадрах та визначити, приналежність об'єктів;

– сортування об'єктів: *DeepSort* використовує метод сортування для призначення унікального ідентифікатора кожному об'єкту та відстеження їх руху в часі. Це допомагає відстежувати об'єкти навіть у випадках, коли вони змінюють своє положення, розмір або вигляд;

– прогнозування руху: *DeepSort* може прогнозувати майбутнє положення об'єктів на відео, що є корисною функцією для передбачення траєкторії руху об'єктів.

Переваги *DeepSort*:

– висока точність: *DeepSort* використовує нейронні мережі для розпізнавання об'єктів, що робить його дуже точним у відстеженні рухомих об'єктів навіть у складних умовах;

– можливість відстеження багатьох об'єктів: *DeepSort* може відстежувати одночасно багато об'єктів на великій області відео, що робить його придатним для великих систем відеоспостереження;

– спроможність розрізняти об'єкти: алгоритм може розрізняти різні об'єкти та надавати їм унікальні ідентифікатори, навіть якщо вони мають схожий вигляд або рухаються близько один до одного;

– можливість використання в реальному часі: *DeepSort* може працювати в режимі реального часу, що дозволяє використовувати його для автоматичної системи навігації та інших застосувань, де час важливий параметр;

– легка інтеграція: *DeepSort* вже вбудований у системи комп'ютерного зору та обробки відео, що робить його доступним інструментом для розробників, які працюють над системами відеоспостереження та аналізу відео [19].

Ці особливості визначають *DeepSort* незамінним інструментом для відстеження об'єктів у відеопотоці. Він відіграє важливу роль у багатьох застосуваннях, де потрібно аналізувати рух та взаємодію об'єктів на відео. Його

висока точність та ефективність роблять його незамінним інструментом для багатьох завдань у сфері комп'ютерного зору.

DeepSort можна адаптувати модулями для отримання представлення об'єктів у вигляді векторів, які потім використовуються для асоціації та відстеження об'єктів на відеопотоці. Ці модулі можуть включати наступні надбудови (*embedder*):

1. *MobileNetv2* – полегшена глибока нейронна мережа, що використовує згорткові блоки глибиною 53 шари. Мережа має два типи блоків, а саме один залишковий блок з кроком 1, і інший блок з кроком 2 для зменшення розміру. Вона підтримує опцію половинного обчислення (*half-precision*) для поліпшення швидкодії на підтримуваних пристроях та може використовувати *GPU* для прискорення обчислень;

2. *TorchReID* – бібліотека повторної ідентифікації людини, яка особливо корисна для вилучення особливостей людей. Можна обрати модель за іменем (*model_name*) та використовувати *GPU* для прискорення обчислень;

3. *CLIP* (*Contrastive Language-Image-Pre-Training*) – нейронна мережа, навчена на різних парах (зображення, текст), що використовується для обробки відео та зображень з бібліотеки *CLIP* [20]. Модель *CLIP* приймає текст в якості вхідних даних і реалізує вбудовування (вектор з *N* чисел). Крім того, вона може приймати зображення в якості вхідних даних і створювати ті ж вбудови. Вбудови дозволяють нам порівнювати показники схожості між текстами та/або зображення. З математичної точки зору модель *CLIP* просто відображає (перетворює) тексти і зображення в точки *N*-мірного простору. Таким чином ми можемо обрахувати відстань між ними – наскільки близько ці точки розташовані один до одного. Часто цю відстань називають показником релевантності. Надбудова з цією нейронною мережею також може використовувати *GPU* для прискорення обчислень [21].

Використання *MobileNet* в контексті *DeepSort* є важливим аспектом для досягнення ефективності та швидкодії в алгоритмі відстеження об'єктів.

MobileNet – це легка глибока нейронна мережа, спеціально розроблена для вбудовування в мобільні та вбудовані пристрої з обмеженими ресурсами, і вона грає ключову роль в оптимізації обчислень для *DeepSort*. Розглянемо, як *MobileNet* використовується в *DeepSort* та які це забезпечує переваги:

1. Легкість та швидкодія: *MobileNet* відома своєю легкістю та здатністю працювати на пристроях з обмеженими обчислювальними ресурсами, таких як мобільні телефони, вбудовані системи та одноплатні комп'ютери. Ця легкість дозволяє запускати *DeepSort* на пристроях реального часу, які мають обмежену потужність обчислень;

2. Оптимізація для відстеження: *MobileNet* може бути оптимізований для завдань відстеження об'єктів. Використовуючи *MobileNet* у якості базової моделі для розпізнавання об'єктів, *DeepSort* може ефективно виділяти та аналізувати об'єкти на відеопотоці. Оптимізована модель *MobileNet* забезпечує швидке відстеження, що важливо в режимі реального часу;

3. Точність та швидкодія: використання *MobileNet* в поєднанні з *DeepSort* дозволяє зберігати баланс між точністю та швидкодією. Є можливість налаштувати модель *MobileNet* так, щоб вона надавала задовільну точність в розпізнаванні об'єктів, при цьому забезпечуючи досить високу швидкодію для відстеження багатьох об'єктів одночасно;

4. Можливість вбудовування: оскільки *MobileNet* розроблено для вбудовуваних систем, його можна легко інтегрувати в різноманітні пристрої та додатки. Це робить його ідеальним вибором для *DeepSort*, коли потрібно використовувати відстеження об'єктів на мобільних пристроях або вбудованих системах [22].

2.3.3. *DeepFace*

DeepFace – це легковаговий фреймворк для розпізнавання обличчя та аналізу атрибутів обличчя (вік, стать, емоції та раса) у *Python*. Він комбінує в собі кілька передових моделей розпізнавання обличчя, включаючи *VGG-Face, Google*

FaceNet, OpenFace, Facebook DeepFace, DeepID, ArcFace, Dlib та *SFace*. Основні можливості DeepFace включають:

1. Розпізнавання обличчя: *DeepFace* надає можливість розпізнавати обличчя, визначаючи, чи належать два обличчя одній і тій же особі;
2. Пошук обличчя: функція *find* дозволяє знаходити обличчя на зображенні та повертати інформацію про розпізнані особи;
3. Аналіз атрибутів обличчя: *DeepFace* може визначати різні атрибути обличчя, такі як вік, стать, емоції та раса, на зображеннях обличчя;
4. Представлення векторів: *DeepFace* дозволяє отримувати векторні представлення обличчя, які можна використовувати для подальшої обробки;
5. Підтримка різних моделей: *DeepFace* підтримує кілька моделей розпізнавання обличчя, таких як *VGG-Face, FaceNet, ArcFace, Dlib* та інші;
6. Можливість роботи в реальному часі: *DeepFace* може бути використаний для аналізу обличчя в реальному часі за допомогою вебкамери;
7. Підтримка різних методів схожості: Можливість вибору різних метрик схожості, таких як косинусна схожість, Евклідова відстань тощо;
8. Підтримка різних детекторів обличчя: *DeepFace* підтримує різні алгоритми детекції обличчя, такі як *OpenCV, SSD, Dlib, MTCNN, RetinaFace, MediaPipe, YOLOv8 Face* і *YuNet*;
9. *API* і командний рядок: *DeepFace* надає *API* для інтеграції з іншими додатками та можливість використовувати командний рядок для запуску функцій [23].

Цей фреймворк дозволяє виконувати завдання розпізнавання обличчя та аналізу атрибутів обличчя з легкістю та високою точністю, використовуючи різні моделі та методи.

2.3.4. *SQLAlchemy*

Додатки на мові *Python* можуть використовувати різні базі даних, наприклад *SQLite*, *PosrgreSQL* тощо, недоліком використання яких є необхідність писати

запити на мові *SQL*. Крім того, якщо ми захочемо перейти з однієї бази даних до іншої, то нам необхідно буде внести чимало змін. Для вирішення цих проблем застосовуються *ORM*-бібліотеки (*Object Relational Mapper*), що дозволяють абстрагуватися від будови конкретної бази даних і працювати з даними як з об'єктами стандартних класів *Python.* Однією з найпопулярніших таких бібліотек є *SQLAlchemy.*

Перевагами використання подібних бібліотек можна визначити наступне:

1. Абстракція – забезпечення рівня абстракції між базою даних і кодом додатку, дозволяючи розробникам зосередитися на бізнес логіці свого додатку, а не на деталях операції з базою даних;

2. Простота використання – наявність більш інтуїтивно зрозумілого і простого у використанні інтерфейсу для взаємодії з базами даних, що має переваги для розробників, що не мають досвіду роботи з *SQL*;

3. Цілісність і узгодженість даних – *SQLAlchemy* надає такі функції як управління транзакціями, забезпечуючи цілісність даних під час критичних операцій. Розробники можуть виконувати декілька операцій з базою даних в рамках однієї транзакції, що дозволяю виконувати відкат у випадку виникнення помилки;

4. Побудова записів і зв'язку – реалізація потужного *API* для побудови запитів, який дозволяє розробникам створювати складні запити з використанням синтаксису *Python.* Він також обробляє зв'язки між таблицями, спрощуючи навігацію і вилучення пов'язаних даних;

5. Переносимість – спрощений механізм переносу додатку на інший механізм бази даних, так як *ORM*-бібліотеки абстрагують базовий синтаксис *SQL*, специфічний для бази даних;

6. Продуктивність – частіша більш ефективна генерація, що призводить до підвищення продуктивності;

7. Безпека – можливість попередження атак з використанням *SQL*-ін'єкцій, за допомогою автоматичного екранування вхідних значень.

2.4. Метрики оцінки виявлення об’єктів

Середня точність (*AP*) і середньоарифметична точність (*mAP*) є найбільш популярними показниками, що використовуються для оцінки моделей виявлення об’єктів таких як *Faster R-CNN, Mask RCNN, YOLO* та інші.

На низькому рівні оцінка продуктивності детектора об’єктів зводиться до визначення правильності виявлення.

Значення деяких термінів:

1. *True Positive (TP)* – правильне виявлення моделі;
2. *False Positive (FP)* – неправильне виявлення детектором;
3. *False Negative (FN)* – основна істина пропущена (не виявлена) детектором об’єкта;
4. *True Negative (TN)* – фонова область, яка правильно не виявляється моделлю. Ця метрика не використовується при виявленні об’єктів, так як такі області не анотуються явно при підготовці анотацій.

Метрика *IoU* при виявленні об’єктів оцінює ступінь перекриття між істиною (*gt*) і передбаченням (*pd*). Основна істина і передбачення можуть мати будь-яку форму: прямокутну рамку, круг чи неправильну форму. Вона розраховується відповідно до формули (2.3).

$$IoU = \frac{area\,(gt \cap pd)}{area(gt \cup pd)} \tag{2.3}$$

Схематично *IoU* визначається як область перетину, розділена на область об’єднання між основною істиною і блоком що передбачається). Математично це зображено за допомогою формули (2.4).

$$IoU = \frac{area\ of\ overlap}{area\ of\ union} = — \tag{2.4}$$

IoU знаходить в діапазоні від нуля до одиниці, де нуль означає відсутність перекриття, а одиниця означає ідеальне перекриття між *gt* і *pd*. Метрика *IoU*

корисна при встановленні порогу, тобто нам потрібний поріг (наприклад α), щоб визначити, чи правильне виявлення.

AP@α (середня точність) – це площа під кривою точного відгуку, що оцінена при порозі α *IoU*. Математично дана метрика обраховується за допомогою формули (2.5).

$$AP@\alpha = \int_0^1 p(r)dr \tag{2.5}$$

AP@α чи *APα* означає, що точність *AP* оцінюється з порогом *α IoU*. Тобто якщо ми бачимо такі показники, як *AP50* чи *A75*, вони означають, що *AP* розрахована при *IoU* = 0,5 і *IoU* = 0,75 відповідно.

Велика площа під кривою *PR* означає високу точність.

Середньоарифметична точність (*mAP*) – середньоарифметичне значення всіх значень *AP* по всім класам, що математично можна записати відповідно до формули (2.6).

$$mAP@\alpha = \frac{1}{n}\sum_{i=1}^{n} AP_i \tag{2.6}$$

AP розраховується при заданому порозі *α IoU*. Виходячи з цього, *AP* можна розраховувати в діапазоні порогових значень [24].

Наприклад, якщо *AP* розраховується в діапазоні від 50 % до 95 % з кроком 5 %, позначення буде *AP@[.50:.5:.95]*.

2.5. Висновки до розділу

В даному розділі розглянуто моделі нейронних мереж, що придатні до застосування у сфері комп'ютерного зору. Визначено, що значний прогрес у сфері комп'ютерного зору було досягнуто за допомогою згорткових нейронних мереж,

тому для подальшої розробки було обрано саме цей тип мереж. Окрім цього визначено переваги та недоліки даного типу мереж.

Після цього є необхідність у вибору моделі. Тому було здійснено характеристику моделей на двохетапні та одноетапні моделі виявлення об’єктів, і визначено, що для нашої мети більш ефективною є одноетапна модель виявлення об’єктів.

Після порівняння таких одноетапних моделей як *SDD*, *YOLO*, що є найпопулярнішими, було обрано подальше використання саме моделі *YOLO*. Враховуючи різні фактори, *YOLOv8* стає важливим інструментом для багатьох сфер застосування, включаючи відеоспостереження, автономні автомобілі, медичну діагностику та багато інших. Його універсальність, точність та швидкість роблять його першочерговим вибором для реалізації системи знаходження об'єктів у реальному часі. Аналіз боксів (областей на зображенні, які обведені прямокутниками) в системі для визначення обличь людей та моніторингу їхнього багажу – це важлива складова завдання, яка передбачає формування груп людей на зображенні, подальшу роботу з цими групами та використання констант.

Ми визначили, що користувацький інтерфейс ми будемо розробляти за допомогою *PyQt5*. Ми створюватимемо вікно програми, яке буде служити головним інтерфейсом. В ньому будуть віджети, такі як тексти для виведення інформації про обличчя та багаж, кнопки для виконання різних дій, та інші елементи, які забезпечать зручну роботу з системою.

Окрім цього задетектовані дані необхідно заносити в базу даних. Для роботи з нею було обрано *SQLAlchemy*. Вона дозволяє абстрагуватися від будови конкретної бази даних і працювати з даними як з об’єктами стандартних класів *Python.*

Також, обрано легковаговий фреймворк *DeepFace*, що використовуватиметься для розпізнавання обличчя. В свою чергу відстеження об’єктів будемо реалізувати на основі алгоритму відстеження об’єктів *DeepSort.*

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАМНОГО МОДУЛЮ

3.1. Опис роботи модулю

Завдання, що стоїть перед запропонованою системою, полягає у розвиненні інтелектуальних здібностей аналізу візуальної інформації з веб-камери та виявленні різних аспектів взаємодії людини з оточуючим середовищем.

Основною метою системи є визначення обличь людей і встановлення їхньої ідентифікації. Після цього система повинна вміти відстежувати цих осіб за допомогою веб-камери, реєструвати їх рухи та визначати, чи мають вони при собі багаж. Інформація про багаж, включаючи його тип, розмір і характеристики, повинна зберігатися та оновлюватися в системі для подальшого аналізу.

Система має постійно слідкувати за змінами у багажі та реагувати на них, тобто мати засіб сповіщення або сигналізації, який активується в разі виявлення змін у багажі особи. Це дозволить оператору або відповідному персоналу вчасно реагувати на ситуації, коли багаж може бути втрачений, викрадений або змінений.

Така система виконує важливу роль у забезпеченні безпеки та контролю на публічних місцях, допомагаючи виявляти та вирішувати потенційні проблеми, які можуть виникнути у великих людних потоках. Вона забезпечує ідентифікацію та моніторинг осіб, а також виявлення змін у їхньому багажі, що робить її цінним інструментом для забезпечення безпеки та ефективного управління публічними подіями та просторами.

Користувачем визначимо оператора системи відеоспостереження, який аналізуватиме необхідні дані. Визначено, що програма реалізуватиме режими роботи, що зображена на рис. 3.1 у вигляді *Use Case* діаграми.

Окрім цього для кращого розуміння роботи системи побудовано діаграму прецедентів на якій показано різні варіанти використання, що доступні користувачу (рис. 3.2). Після відкриття програми оператор має змогу переглянути

ідентифікованих людей, переглянути результати детектування («Логи», що є групою (комбінацією) людини і її багажу), обрати окрему камеру та переглянути результати виявлення з неї та завантажити відео з файлу, наприклад для тестування справності системи.

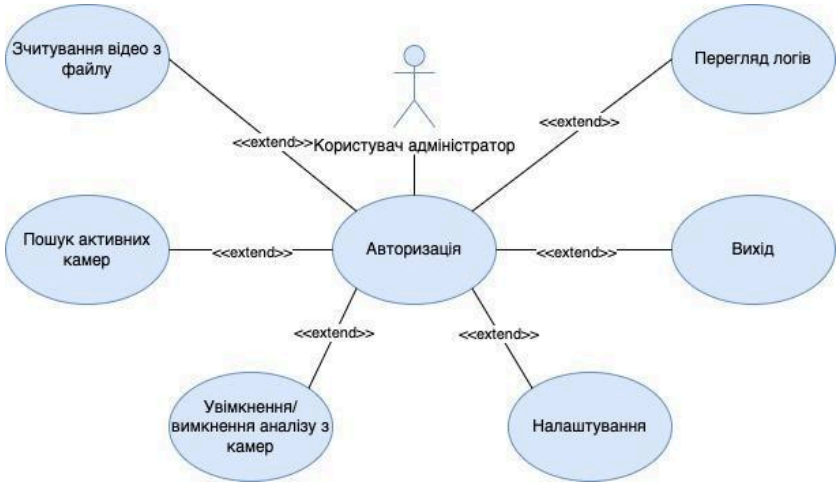

Рис. 3.1. *Use Case* діаграма роботи програми

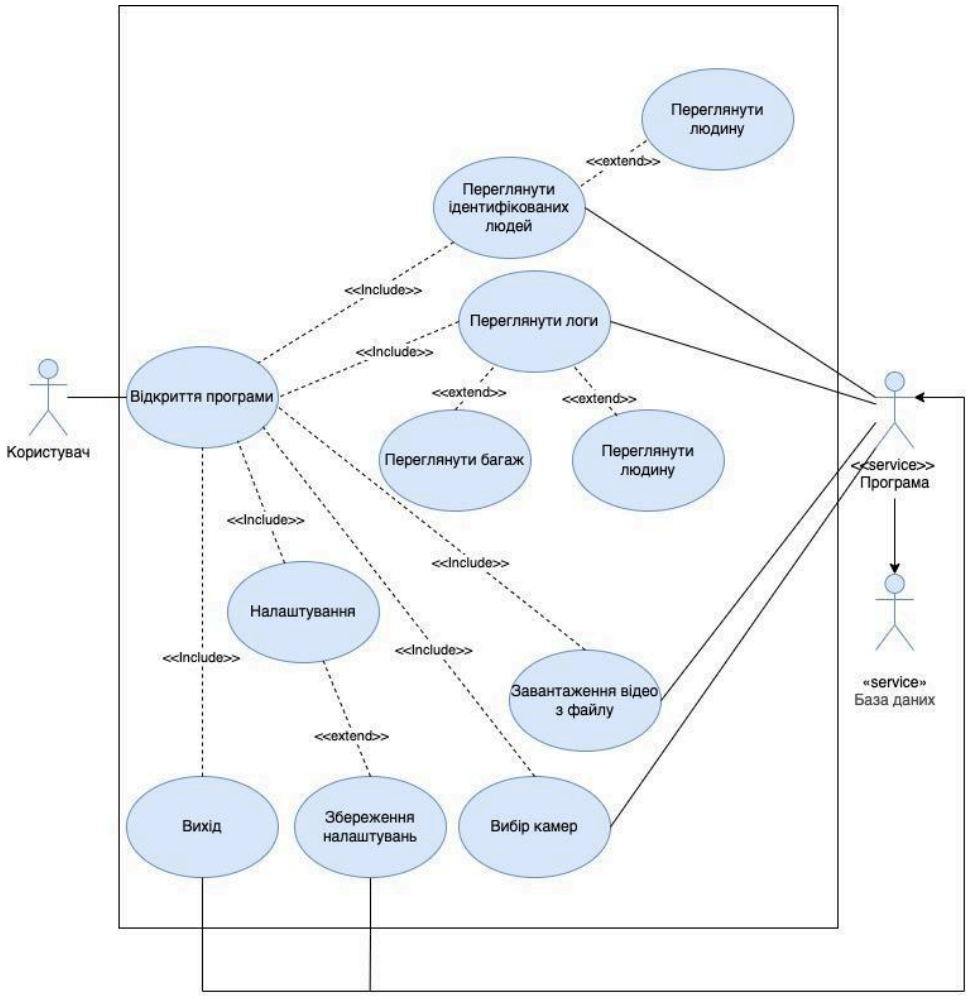

Рис. 3.2. Діаграма прецедентів

Система для сканування обличь людини та моніторингу їхнього багажу, повинна обирати моделі для розпізнавання обличь та визначення багажу з усіх існуючих моделей. Важливо враховувати різні метрики точності та продуктивності цих моделей для досягнення оптимального результату.

Початково, вибір моделі є ключовим етапом. Модель повинна бути досить точною у визначенні обличь та моніторингу багажу. Було прийнято рішення використовувати *YOLOv8*, яка є відомою своєю високою швидкістю та точністю у завданнях об'єктного визначення на зображеннях. Ця модель була вибрана завдяки своїм характеристикам, що відповідають вимогам проекту.

Крім того, модель *YOLOv8* пропонує розширені можливості, такі як сегментація екземплярів, оцінка пози/ключових точок та класифікація, що може бути корисним для більш складних завдань.

Інтеграція *YOLOv8* в код була виконана через використання відповідного фреймворку та бібліотек для роботи з нейронними мережами в *Python*. Це дозволило ініціалізувати модель та забезпечити її коректну роботу в системі.

Запуск роботи програми і подальша послідовність дій для реалізації роботи системи наведена на рис. 3.3.

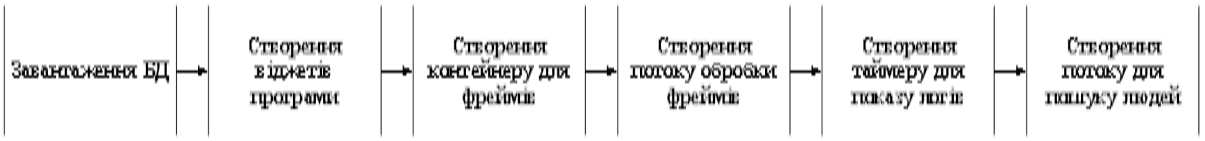

Рис. 3.3. Послідовність дій для реалізації роботи системи

Після вибору та інтеграції моделі, система проходить процес ініціалізації. Це включає в себе завантаження вагових коефіцієнтів моделі, встановлення параметрів, необхідних для її роботи, та підготовку до аналізу зображень.

3.2. Особливості формування даних

Розглянемо особливості формування груп та їх використання.

Формування груп розпочинається з аналізу боксів, що представляють обличчя людей. Для формування груп важливо визначити, які обличчя належать одній і тій же людині. Це можна зробити за допомогою алгоритмів розпізнавання особливих ознак, таких як орієнтація та розмір обличчя. Також можна використовувати техніки асоціації обличчя між кадрами за допомогою унікальних ідентифікаторів або характеристик, що зберігаються в базі даних.

Після формування груп людей, система може проводити аналіз та моніторинг кожної групи окремо. Це включає в себе визначення, чи має кожна група багаж, і виконання спеціальних алгоритмів для визначення характеристик багажу, таких як розмір, форма, вага тощо.

Константи в системі важливі для встановлення порогових значень та параметрів для аналізу та прийняття рішень. Наприклад, встановлення порогу для визначення, коли багаж змінився або коли відбулася зміна на зображенні, може допомогти сповіщати про можливі небезпеки або аномалії.

Система повинна працювати саме таким чином, оскільки це дозволяє вирішувати завдання визначення обличь людей та моніторингу багажу ефективно та надійно. Формування груп, робота з ними та використання констант допомагають підтримувати безпеку та контроль на публічних місцях, а також реагувати на можливі зміни в реальному часі.

Однією з ключових задач є налаштування гіперпараметрів моделі та системи в цілому. Гіперпараметри визначають різні аспекти роботи системи, такі як швидкість, точність та обсяг ресурсів, які вона вимагає. Основні гіперпараметри включають:

1. Шлях до моделі детектування: *yolov8_detect_model_path*;
2. Кількість фреймів, які будуть об’єднуватися і оброблятись в основному потоці програми: *main_process_count_frames* = 3;

3. Час очікування, якщо фреймів для обробки не виявиться для обробки в основному потоці програми: *main_process_timer_timestep* = 300;

4. Час очікування, якщо фрейми для обробки є в основному потоці програми: *main_process_timer_small_timestep* = 30;

5. Ширина і висота при виведенні на основній програмі: *standart_width* = 600, *standart_height* = 400;

6. Часові обмеження обробки на віджетах: *timeout_is_file_widget* = 30, *timeout_widget* = 30. (Кожен віджет самостійно обробляє свої фрейми із свого потоку відео або безпосередньо камери. Тому потік має певні часові обмеження для усунення перевантаження системи і загалом спрощення паралельного аналізу багатьох потоків);

7. Ширина зміни розміру, відображених в фрейм даних від моделі детекції, для виводу: *resize_with_width* = 600;

8. Шлях бази даних де будуть зберігатись фото людей: *database_url = 'sqlite:///people_logs_photos.db'*.

Гіперпараметри налаштовуються з урахуванням балансу між точністю та продуктивністю системи.

Ініціалізація системи для виконання завдань з визначення обличь людей та моніторингу багажу є важливим етапом, що реалізується за допомогою фрагмента коду наведеного нижче.

```
import torch
from ultralytics import YOLO
from config import yolov8_detect_model_path
@torch.no_grad()
def init_yolov8_bags(path_model, device='cpu'):
    model = YOLO(path_model)
    return model
yolov8_model_bags = init_yolov8_bags(yolov8_detect_model_path)
```

3.3. Програмна реалізація моделі

Початковий екран роботи системи зображено на рис. 3.4.

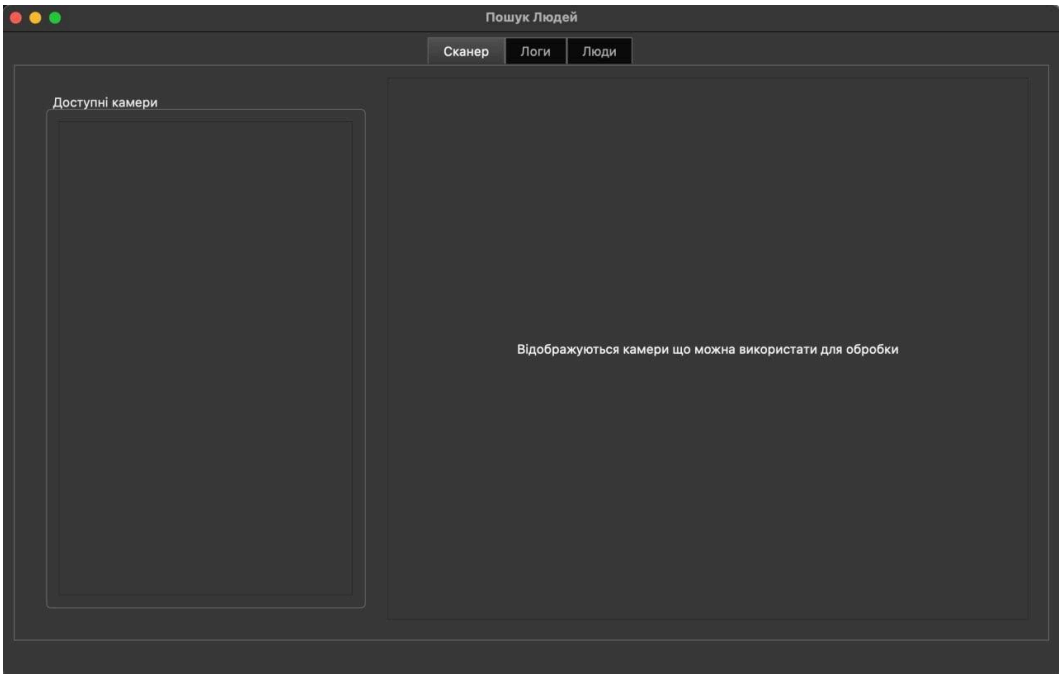

Рис. 3.4. Початковий екран системи

Основна програма включає ключові функції аналізу фрейму. Кожен потік пошуку веде аналіз відеоданих на своєму фактичному потоці, а не спільному для всіх, що дає можливість одночасної обробки інформації з декількох камер. Окрім цього кожному потоку відповідає конкретний віджет системи, який має свій компонент обробки фрейму.

В Додатку А наведено реалізацію функції, що представляє з себе потік, метою якого є обробка фреймів в кожному віджеті.

Надалі реалізовано обробку фрейму, який передається через спеціальний клас *FrameData*, який формує всі дані які потрібно передати. Реалізацію даного класу реалізовано відповідно до наведеного нижче фрагменту.

```
class FrameData:
    def __init__(self) -> None:
        self.id_worker = -1
        self.name = ""
```

```
        self.data = {}
        self.other_data = None
        self.timestamp = None
        self.yolov8 = None
        self.index_name_yolov8 = None
        self.retinaface = None
        self.real_frame = None
        self.dataframe_uid = 0
        self.logs = None
        self.humans_and_bags = None
        self.array_all_humans_and_bags = None
        self.entities = None
    def set_frame(self, data):
        self.data=data
        now = datetime.now()
        self.timestamp = datetime.timestamp(now)
    def get_frame(self):
        return self.data
```

Зручний механізм потоків, який використовується в системі подібний таймеру, так як має схожий функціонал і перекриває потреби системи в таких потоках. Клас основного потоку програми реалізується за допомогою наступного фрагменту коду.

```
class MainFunctionCallerThread(QThread):
    def __init__(self, func):
        super().__init__()
        self.func = func
        self.interval_ms = 1.0
        self.running = True
    def run(self):
        while self.running:
```

```
            self.func()
            time.sleep(self.interval_ms / 1000.0)
        def set_time_interval(self, time_interval):
            self.interval_ms = time_interval
        def stop(self):
            self.running = False
```

Він являє собою наслідування від *Qthread*, який дозволяє створити будь-яку форму власного потоку із своїми даними під свою задачу, а що головне цей потік буде реальним і може працювати паралельно іншим реальним потокам на машині користувача.

Далі переходимо до реалізацію власного потоку, що походить від *MainFunctionCallerThread*, і має свою подобу таймеру, адже від одного параметру може змінити свою внутрішню форму і систему роботи на таймер або на потік.

```
    class FunctionCallerThread():
        def __init__(self, func, rtt=function_caller_thread_state):
            self.rtt_variants = ["timer", "thread"]
            self.rtt = rtt
            self.func = func
            self.interval_ms = 1
            self.running = False
            if self.rtt == "thread":
                self.th = MainFunctionCallerThread(self.func)
                self.th.set_time_interval(self.interval_ms)
            elif self.rtt == "timer":
                self.timer = QTimer()
                self.timer.timeout.connect(self.func)
                self.timer.setInterval(self.interval_ms)
        def set_time_interval(self, time_interval):
            self.interval_ms = time_interval
            if self.rtt == "thread":
```

```
            self.th.set_time_interval(self.interval_ms)
        elif self.rtt == "timer":
            self.timer.setInterval(self.interval_ms)
    def start(self):
        if self.rtt == "thread":
            self.th.start()
            self.running = True
        elif self.rtt == "timer":
            self.timer.start(self.interval_ms)
            self.running = True
    def stop(self):
        if self.rtt == "thread":
            self.th.stop()
            self.th.quit()
            self.th.wait()
            self.running = False
        elif self.rtt == "timer":
            self.timer.stop()
            self.running = False
    def in_function(self):
        if self.rtt == "timer":
            if self.running is True:
                self.timer.stop()
    def out_function(self):
        if self.rtt == "timer":
            if self.running is True:
                self.timer.start(self.interval_ms)
```

Ключова функція такого потоку – викликати певну функцію за допомогою первинної ініціалізації де вказується чим вона буде, а далі передана функція

викликається в вигляді таймеру або потоку. Це формально дозволяє йому бути одним потоком із таймерами або деякою кількістю потоків.

Така реалізація є формою спрощеної відладки багатопотокової програми, так як таймери фізично працюють на одному потоці із викликами, і дозволяють перевіряти потрібні дані. Самі потоки не дають такої можливості і тому вони мають недоліки при відладці, але так як ключова основа може виконуватись як подоба таймеру в вигляді потоку то і її робота еквівалентна таймеру, які зручні для розробки але не зручні для використання. Головна особливість, що таймер викликає всі функції тоді коли вийшов час, але суть в тому що є необхідність зупиняти його в програмі, поки працює якась функція, щоб дані були структуровані і працювали досить гарно у визначеному змісті, щоб уникнути перевантаження системи і забезпечення почерговості обробки фреймів.

Таймери та потоки легко можна привести до еквівалентної форми роботи і проводити просту і зручну відладку, навіть за необхідності перемикати програму на режим в якому вони працюватимуть, тобто таймер або потік.

Основна система потоку працює із класу вікон для зручності і викликає через ці реалізації потоків головний модуль. Клас вікон це ті віджети що працюють паралельно, беруть фрейми і детектують їх та передають в основний клас програми (рис. 3.5).

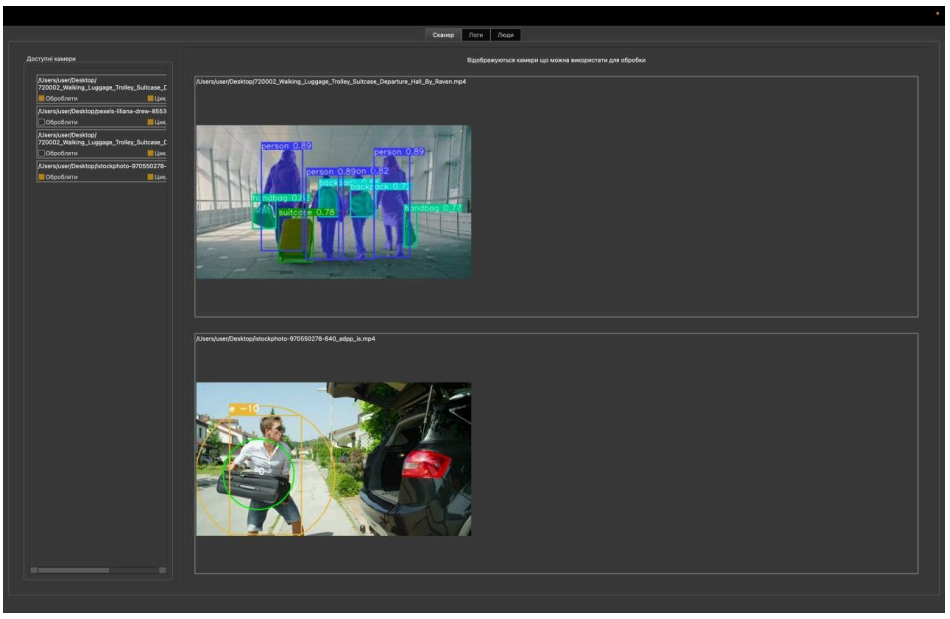

Рис. 3.5. Паралельна робота потоків програми

Відповідно до налаштованих гіперпараметрів видно, що ми описуємо латентність виклику, щоб отримати потрібні затримки. Загалом можна було б працювати і без них, але вони дозволяють системі працювати стабільніше та менше навантажувати саму операційну систему, а, як результат, комп’ютер користувача. Розглянемо ці аспекти більш докладно.

Перше, що варто відзначити, це те, що затримки та латентність є важливою частиною роботи багатьох програм і систем. Вони визначають, коли і які операції повинні виконуватися, регулюють часові інтервали між подіями та забезпечують правильну синхронізацію функцій і задач.

Іноді здається, що можна обійтися без налаштування латентності і затримок, але це не завжди є правильним рішенням. Ось декілька аргументів на користь використання модулю конфігурації для регулювання латентності:

– стабільність роботи – правильно налаштовані латентності дозволяють системі працювати стабільніше, а саме, допомагають уникнути надто швидких або повільних викликів, які можуть призвести до непередбачуваних помилок чи зависань;

– зменшення навантаження – встановлення оптимальних затримок допомагає зменшити навантаження на операційну систему та ресурси комп'ютера, що особливо важливо в великих проектах та серверних програмах;

– забезпечення вчасності – в деяких випадках, особливо в реальному часі, або в задачах з обробкою потоків даних, важлива точність виконання, а встановлення латентності допомагає забезпечити вчасне виконання завдань;

– керована робота – за допомогою конфігураційних параметрів затримки можна дозволити користувачам налаштовувати латентність під свої потреби, що робить програму більш гнучкою.

Для запуску процесу обробки фреймів необхідно створити новий потік за допомогою *FunctionCallerThread*, що показана нижче.

```
self.pt_time = FunctionCallerThread(self.update_frame)
        self.pt_time.set_time_interval(timeout_is_file_widget)
        self.pt_time.start()
```

Спершу ми створюємо потік і передаємо функції та латентність. Після цього запускаємо роботу цього потоку.

```
def process_frames(self):
    with self.pt_time:
        if (self.fdc.empty() is not True and
            self.fdc.limit_size (self.process_count_frames)):
            arrays_frames = []
            try:
                self.lock_mutex.lock()
                arrays_frames = self.fdc.get(max_count=self.process_count_frames)
            finally:
                self.lock_mutex.unlock()
            self.work_frames(arrays_frames)
            if self.fdc.limit_size(self.process_count_frames):
                self.pt_time.set_time_interval(self.process_timer_small_timestep)
            else: self.pt_time.set_time_interval(self.process_timer_timestep)
        else:
            self.pt_time.set_time_interval(self.process_timer_timestep)
```

Спочатку функція отримує фрейми в тій мінімальній кількості яка прописана в налаштуваннях, а потім передає їх на функцію обробки яка в свою чергу оброблює фрейми і створює пари результату виявлення, які ми називатимемо групами або «Логами».

Детальніше роботу можна описати так: в основному класі відбувається виклик *process_frames()*, що формує масив фреймів. Після цього передає його в *work_frames()*, що обробляє зображення і формує сутності. Виконується первинна детекція людей, що є просто об’єктами класу з точками, і завдяки об’єднанню перетинів формуються сутності. Сутність містить людей і багаж, що представляють собою великий набір об’єктів. Для аналізу обирається та сутність, що має найдовший шлях по визначеним точкам, що означає що задетектована

сутність є реальною, а не помилково визначеною, адже відомо, що реальні задетектовані об’єкти мають найдовший трек в кількості точок.

Після того як сутності готові, вони аналізуються групами об’єднань, які описують фактичну людину з врахуванням всіх перетинів що сформовані в результаті.

Результат виявлених груп об’єктів (людина та її багаж) показуватиметься на вкладці «Логи».

Розглянемо принцип обробки одного фрейму через модель. Первинна операція яка відбувається – це підготовка фрейму для опрацювання моделлю, що відбувається до наведеного нижче коду.

```
real_frame = data_frame.copy()
    real_frame = resize_with_aspect_ratio(real_frame, new_width=700)
    real_frame = cv2.cvtColor(real_frame, cv2.COLOR_BGR2RGB)
```

Спочатку зображення пропорційно зменшується по ширині, висота обирається автоматично. Далі іде зміна кольорового простору на *RGB* так як модель приймає на вхід саме цей формат зображення.

Після підготовки іде обробка і в модель передається підготовлене зображення.

```
with torch.no_grad():
    results = get_yolov8_model_bags()(real_frame, verbose=False)
```

Слід зауважити що обробка іде із вказуванням не зберігати градієнти а також не виводити службову інформацію із системи обробки. Після того як модель відпрацювала, із результату іде вилучення потрібних даних, що показано відбувається в наведемо нижче фрагменті коду.

```
results = get_yolov8_model_bags()(real_frame, verbose=False)
    result = results[0]
    boxes = result.boxes
    names2 = get_yolov8_model_bags().names
    cls_boxes = boxes.cls
    ci = get_object_names_with_indices(cls_boxes, names2)
```

```
annotated_frame = result.plot()
annotated_frame = resize_with_aspect_ratio(
    annotated_frame,
    new_width=resize_with_width)
```

В змінній *results* міститься результат розпізнавання з фрейму. Так як результат має в собі обробку для тої кількості фреймів яку ми передаємо, то беремо результат за індексом нуль так як ми подаємо один фрейм. Далі ми отримуємо бокси (*boxes*) і потрібні класові індекси (*ci*) за допомогою відповідної функції.

Після отримання іде накладання того що має фрейм в обробці на результуюче зображення, але зауважимо що це попередня обробка.

```
def get_object_names_with_indices(class_indices, class_names):
    result = []
    for i, index in enumerate(class_indices):
        class_name = class_names.get(int(index), 'Unknown')
        result.append((int(index), class_name))
return result
```

Після отримання всіх ключових даних іде обробка трекером менеджером пар. Ключове отримання треків іде за допомогою спеціальної бібліотеки під назвою *deep_sort_realtime* в якій для нас основним є *DeepSort,* який виконує трекінг об’єктів. В нашій системі він ініціалізується так як показано на фрагменті нижче.

```
tracker = DeepSort(
    max_age=create_basic_tracker_deepsort_max_age,
    embedder=create_basic_tracker_deepsort_embedder,
    max_iou_distance=create_basic_tracker_deepsort_max_iou_distance,
    embedder_gpu=create_basic_tracker_deepsort_embedder_gpu)
```

DeepSort підтримує кілька різних моделей для вбудовування що були розглянуті в Розділі 2. В нашому випадку *DeepSort* використовує в собі модель *MobileNet* для порівняння об’єктів. Використання *MobileNet* в контексті *DeepSort*

допомагає досягти високої ефективності та швидкодії відстеження об'єктів на пристроях з обмеженими обчислювальними ресурсами. Це важливо для застосувань, де необхідно здійснювати відстеження об'єктів в реальному часі, тобто системи відеоспостереження та автоматична система навігації.

Для нашої задачі обрано модель *MobileNetV2*, адже вона має оптимальні для нашої задачі характеристики по швидкості і якості. Для забезпечення цієї оптимальної якості роботи необхідно налаштувати цю модель, тому нижче наведено параметри які вона має.

DeepSort має численні параметри для налаштування обраного відстежувача об'єктів та вбудовувача об'єктів. Ці параметри допомагають налаштувати та керувати різними аспектами алгоритму відстеження. Короткий опис цих параметрів:

– *max_iou_distance*: поріг *IoU* для фільтрації асоціацій;

– *max_age*: максимальна кількість пропущених кадрів перед видаленням треку;

– *n_init*: кількість кадрів, протягом яких трек знаходиться у фазі ініціалізації;

– *nms_max_overlap*: поріг перекриття для *non-maxima suppression*;

– *nn_budget*: максимальний розмір дескрипторів зовнішнього вигляду;

– *gating_only_position*: використовується під час відсіювання, для порівняння передбачуваних та виміряних станів фільтра Калмана;

– *override_track_class*: клас, який перевизначає клас за замовчуванням для треків;

– *embedder*: вибір вбудовувача;

– *half*: використовувати половинне значення для глибокого вбудовувача (застосовується лише для *MobileNet*);

– *bgr*: очікувати *BGR* або *RGB* формат вхідного кадру;

– *embedder_gpu*: використовувати *GPU* для вбудовувача;

– *embedder_wts*: шлях до ваг моделі вбудовувача;

– *polygon*: показує, чи об'єкти є полігонами;

– *today*: поточна дата для іменування треків.

Всі ці параметри мають дуже важливе значення для роботи системи та при правильному налаштуванні покажуть значно якісніші результати та ефективність виконання поставленої задачі.

Далі продовжимо розгляд фреймворку і функцій які він має, а саме метод *push_tracks* із *PairsManager* що формує сутності. Але щоб краще зрозуміти суть потрібно розуміти що таке сутність і це буде описано далі. Основу сутності описує фрагмент зображений нижче.

```
class Entity:
  def __init__(self) -> None:
    self.current_id = trackers_capacitor.get_uid_entity()
    self.humans = []
    self.bags = []
    self.pairs = []
    self.bag_group = []
    self.images_with_camera_id = []
    self.max_len_deque_images_camera = entity_max_len_deque_images_camera
```

Основа сутності складається із людей, сумок (багажу), пар та груп сумок (декілька одиниць багажу). Із ключових функцій це сутність, що описуватиме одну людину і її багаж в вигляді груп. Групою є декілька сумок, що мають перетин, який має значне значення згідно налаштованих параметрів. Що найголовніше, він виправляє проблеми трекеру, що може втрачати людину і при знаходженні вважає її вже іншою особою. Параметрично для нього не можливо вказати якийсь параметр, який би це визначав і, технічно, він не в стані визначити це і тому групи є основою обробки після трекеру.

Human представляє з себе клас людини, що містить в собі бокси де була зафіксована людина, інформацію про камери, і надає доступ до цих даних для подальшого використання.

```
class Human:
```

```
    def __init__(self, human_id=-1, det_class=None, max_len_deque_points_id =
300) -> None:
        self.human_id = human_id
        self.det_class = det_class
        self.max_len_deque_points_id = max_len_deque_points_id
        self.last_camera_id = -1
        self.enable_last_camera_id = -1
        self.last_cam = deque(maxlen=1000
    def append(self, id_camera=-1, data=None):
        if id_camera != -1:
            while len(self.pts_bbox_ltrb) - 1 < id_camera:
                dq = deque(maxlen=self.max_len_deque_points_id)
                self.pts_bbox_ltrb.append(dq)
            current_timestamp = time.time()
            self.pts_bbox_ltrb[id_camera].append((current_timestamp, data))
            self.last_camera_id = id_camera
            self.last_cam.append(id_camera)
    def get_points(self, id_camera=-1):
        if self.exist_camera_track(id_camera) is True:
            pts_m = self.pts_bbox_ltrb[id_camera]
            if pts_m is not None: pts_m = list(pts_m)
                if len(pts_m) > 0: return pts_m
        return None
    def exist_camera_track(self, id_camera=-1):
        if id_camera != -1 and id_camera >= 0:
            if len(self.pts_bbox_ltrb) - 1 < id_camera:
                return False
            else: return True
        return False
```

Клас *Bag* це одиниця багажу. Він містить методи, аналогічні класу *Human*, і природньо що клас *Bag* мав змогу унаслідуватися від класу *Human*, проте було вирішено реалізувати їх окремо, адже ці сутності мають різну природу.

Основна задача класу *Pair* це об’єднати сутність людини і її багажу, вказати чи вони поряд чи ні, використовуючи методи які можуть це визначити, наприклад, за допомогою згладжування і іншої інтерполяції. В нашому випадку це визначення реалізується за допомогою функції на основі експоненційно згладжуваного середнього (*exponential_moving_average*), реалізація якої наведена на фрагменті нижче.

```
def exponential_moving_average(data, alpha):
    ema = [data[0]]
    for value in data[1:]:
        smoothed_value = alpha * value + (1 - alpha) * ema[-1]
        ema.append(smoothed_value)
    return ema
```

Застосування цієї функції до контексту пари дозволяє згладжувати стани короткого виходу із зони перетину, тобто уникнути помилкового розформування пари і чітко визначати приналежність багажу людини чи втрату цього ж багажу визначеною особою. Такий варіант пошуку гарно справляється з похибками, що виникають коли багаж може виходити із зони перетину з відповідною людиною.

Масиви станів в парі мають фіксований розмір що автоматично дозволяє не турбуватися про звільнення пам’яті, адже вона не може переповнитися пустими значеннями.

Окрім цього, пари мають прив’язку до камери на якій знайдені, а в масивах багажу та людей чітко визначено до якої камери належить та чи інша точка. Це реалізовано для випадку коли трекером визначено, що одна людина розпізнана декількома камерами. Тому, щоб не псувати дані змішуванням потрібно чітко знати яка камера зафіксувала людину вперше.

Розглянемо детальніше сутності і роботу з ними.

Сутність має в собі групи із багажу, при цьому кожна група це умовна одна одиниця багажу. Через помилки трекеру один і той самий багаж може постійно визначатися повторно як новий. Проте, через перетини такі події вдається визначати, і зв’язувати повторно задетектовані багажі в одну групу, так щоб один багаж відповідав одній групі. Декілька груп це, відповідно, декілька багажів.

Метод *get_unique_identifiers_active_status* повертає тільки тих в кого статус перетину рівний *True* і це означає, що наразі ця людина і її багаж поряд. Такий аналіз виконується по парам і ці пари постійно оновлюються що дозволяє оновити статус і чітко виявляти відповідність людини і багажу.

Також сутність включає функції зв’язування груп по перетину і також функції додавання сумок і інших компонентів.

EntityManager містить в собі *EBag* і *EHuman* та *ELog* та створює логи, при цьому *EHuman* містить в собі *EBag*. Функція оновлення що є всередині в результаті створює логи реєстрації прив’язки сумки (багажу) та людини та її вилучення. Всі такі зміни описуються через *ELog*.

Попередній клас досить простий але наступний має найбільший із всіх метод для аналізу даних *PairsManager*.

```
class PairsManager:   def __init__(self) -> None:
        self.human_ids = white_list_person
        self.bag_ids = white_list_handbag
        self.humans = []
        self.bags = []
        self.pairs = []
        self.unions_bags = {}
        self.unions_humans = {}
        self.entities = []
        self.max_len_deque_points_id = pairs_manager_max_len_deque_points_id
        self.human_groups = []
        self.bags_groups = []
        self.locker = QMutexContextManager()
```

Вище наведений конструктор і параметри що в ньому ініціалізуються. Ключова суть цього класу в формуванні сутностей (*Entity*), які ми описували раніше, і робота з ними.

Метод *push_tracks* використовується для оновлення стану об'єктів у класі *PairsManager* на основі вхідних треків, які представляють об'єкти, які відстежуються на камерах спостереження. Цей метод дозволяє додавати інформацію про рух людей і сумок на камерах до внутрішньої структури даних і використовувати її для подальшого виявлення взаємодії між цими об'єктами.
Схема алгоритму даного методу зображена на рис. 3.6.

Основні дії, які виконуються методом *push_tracks*:

– метод отримує список треків, які представляють собою об'єкти, що відстежуються на камерах спостереження;

– для кожного треку, який не є підтвердженим або кількість кадрів з моменту останнього оновлення більше одиниці, метод ігнорує цей трек і переходить до наступного;

– для кожного підтвердженого треку, метод визначає інформацію про його положення (*bbox* – обмежуючий прямокутник) і оригінальні дані про розташування (*original_ltwh*), а також клас треку, який визначає, чи це людина чи багаж;

– метод перевіряє, чи об'єкт, відстежуваний треком, є людиною чи багажем, на основі ідентифікаторів класів, які зберігаються в *human_ids* та *bag_ids*;

– якщо об'єкт є людиною, метод перевіряє, чи існує вже така людина в списку *humans*. Якщо такої людини немає, вона створюється як об'єкт типу *Human*, і інформація про її рух додається до неї. Якщо об'єкт вже існує, інформація також додається до нього;

– якщо об'єкт є сумкою, метод аналогічно перевіряє, чи існує вже така сумка в списку *bags*. Якщо такої сумки немає, вона створюється як об'єкт типу *Bag*, і інформація про її рух додається до неї. Якщо об'єкт вже існує, інформація також додається до нього;

– метод активує відстеження об'єктів на конкретній камері за допомогою виклику *enable_track_camera_id*;

– після обробки всіх треків метод виконує обчислення перетину між об'єктами, використовуючи методи *intersection_human* та *intersection_bag*. Це дозволяє визначити, чи взаємодіють об'єкти на камерах і визначити ступінь цієї взаємодії;

– метод оновлює інформацію про об'єкти і їх взаємодію, додаючи нові пари людей і сумок до списку пар *pairs*. Також метод додає інформацію про з'єднані об'єкти (сумки і людей) до словників *unions_bags* та *unions_humans*;

– нарешті, метод оновлює інформацію про групи об'єктів та сутностей, формуючи об'єднані групи для людей та їх багажу на основі їх взаємодії.

Всі ці дії допомагають відстежувати та аналізувати взаємодію між людьми та сумками на камерах спостереження і формувати відповідні групи та об'єднані сутності для подальшого аналізу.

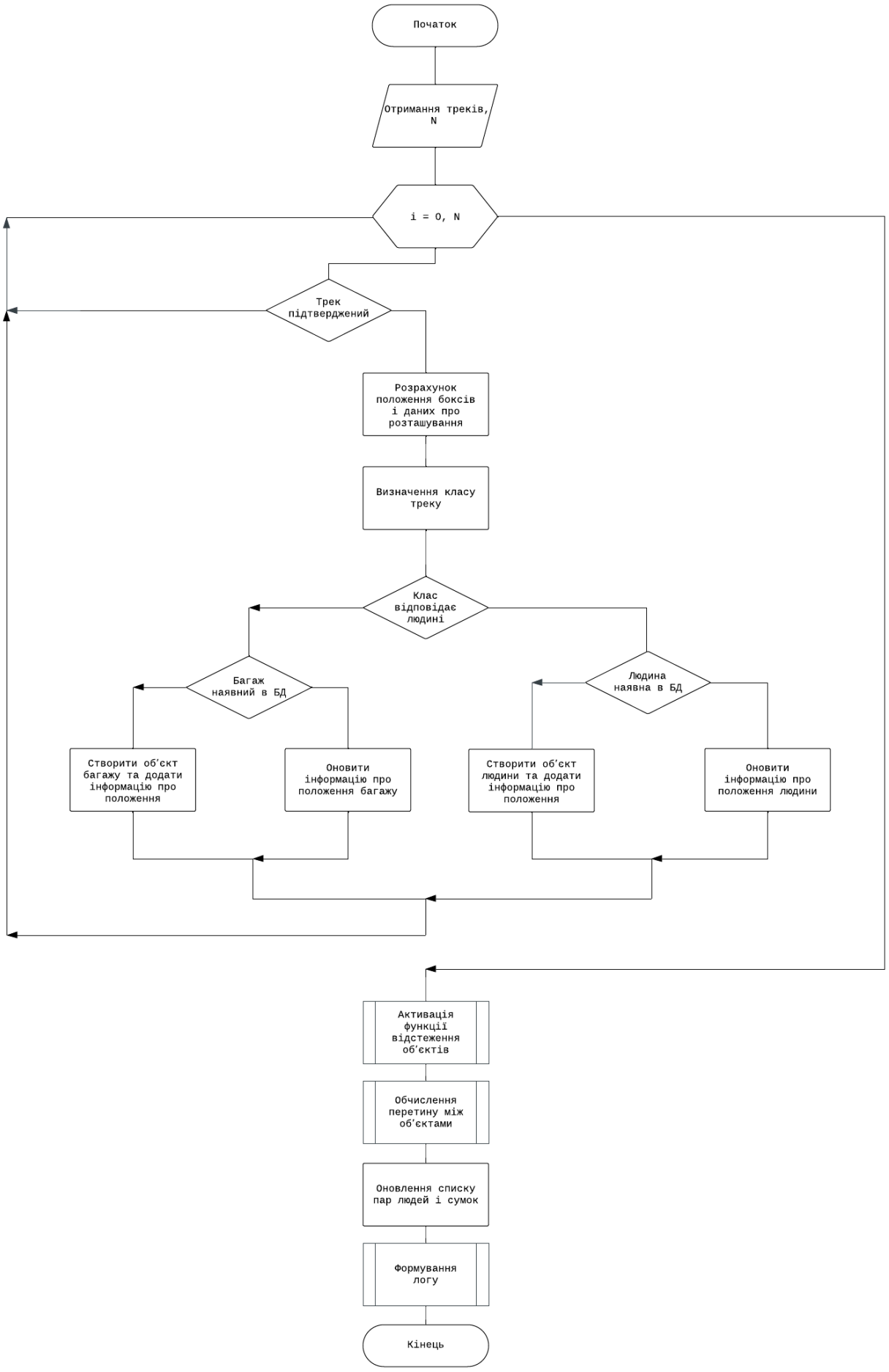
Початок
Отримання треків, N
i = 0, N
Трек підтверджений
Розрахунок положення боксів і даних про розташування
Визначення класу треку
Клас відповідає людині
Багаж наявний в БД
Людина наявна в БД
Створити об'єкт багажу та додати інформацію про положення
Оновити інформацію про положення багажу
Створити об'єкт людини та додати інформацію про положення
Оновити інформацію про положення людини
Активація функції відстеження об'єктів
Обчислення перетину між об'єктами
Оновлення списку пар людей і сумок
Формування логу
Кінець

Рис. 3.6. Схема алгоритму методу *push_track*

Кінцевою функцією є важлива функція *compare_arrays,* яка виконує формування даних для виведення Логу та масивів людей в контексті і загалом людей. Дана функція реалізована відповідно до представленого нижче фрагменту коду.

```
def compare_arrays(camera_id, pman, eman, array_current, array_all):
    if isinstance(eman, EntityManager) and isinstance(pman, PairsManager):
        logs = eman.update(camera_id, array_current, array_all, pman, eman)
        humans_and_bags = pman.get_bags_and_humans_by_logs(logs)
        array_all_humans_and_bags = []
        for elem in array_all:
            human_idx = elem[0]
            humans = pman.get_humans_by_array_idx(human_idx)
            bags_groups = []
            for elem_g in elem[1]:
                bags = pman.get_bags_by_array_idx(elem_g)
                bags_groups.append(bags)
            array_all_humans_and_bags.append((humans, bags_groups))
        return logs, humans_and_bags, array_all_humans_and_bags
    return None, None
```

В *EntityManager* наявний масив *logs[],* який містить в собі об’єкти, що є екземплярами класу *ELog*, що представляють потрібні пари виявлення. Саме цей масив ми передаємо для відображення необхідних нам візуальних даних (рис. 3.7).

Рис. 3.7. Екземпляри класу *ELog*

Ми маємо потік що відповідає за відображення фреймів з обмежувальними рамками. Зміна фреймів (кожного кадру) тригерить оновлення цих обмежувальних рамках.

Для реалізації цього функціоналу спершу використовуємо об’єкт *mutex*, що є блокуючим об’єктом, що призначений для сигналізації того, що критичним розділам коду необхідно попередити одночасне виконання інших потоків і доступ до одних і тих самих комірок пам’яті. Тобто ми блокуємо решту потоків, поміщуємо окремий фрейм у вказаний контейнер, що відповідає наведеному нижче фрагменту.

```
def do_work_frame_data(self, frame_data_loc:FrameData):
    try:
        self.lock_mutex.lock()
        self.fdc.put(frame_data_loc)
    finally:
        self.lock_mutex.unlock()
```

Зауважимо, що *self.fdc = FrameDataContainer()*. Тобто це не просто масив, а цілий клас, який має відповідний вигляд і функції.

FrameDataContainer – це контейнер для зберігання та управління об'єктами *FrameData*, які представляють кадри. Цей контейнер має такі можливості:

– додавання кадрів – кадри можна додавати до контейнера та пов'язувати їх з ідентифікаторами робочих одиниць;

– отримання кадрів – можливість отримувати кадри за ідентифікатором робочої одиниці або автоматично з будь-якої доступної робочої одиниці;

– обмеження розміру – контейнер може обмежувати кількість кадрів для кожної робочої одиниці до заданої максимальної кількості;

– перевірка на порожнечу – здатність перевіряти, чи є контейнер порожнім.

Цей контейнер допомагає у керуванні та організації потоку даних, особливо при аналізі відео чи інших послідовностей кадрів.

В контексті використання цей контейнер виконує роль системи зберігання фреймів, що надає доступ до отримання їх в потрібний момент по заданій кількості для подальшої обробки. Його роль, як системи яка зберігає фрейми, є важливою тому що зберігати їх в масиві не є оптимальним, адже буде необхідність дуже розширити основний клас, що значно збільшить код і ускладнить його розуміння.

Реалізація цього контейнеру відбувається відповідно до представленого нижче коду.

```
class FrameDataContainer:
    def __init__(self) -> None:
        self.containers = {}
    def find(self, inid):
        if inid in self.containers.keys():
            return True
        return False
    def get(self, inid=None, max_count=1):
        arr_frames = []
        if inid is not None:
```

```
            if self.find(inid):
                if isinstance(self.containers[inid], list):
                    if len(self.containers[inid]) >= max_count:
                        while len(arr_frames) < max_count:
                            arr_frames.append(self.containers[inid].pop(0))
            else:
                state, key = self.limit_size(max_count, return_tuple=True)
                if state is True:
                    while len(arr_frames) < max_count:
                        arr_frames.append(self.containers[key].pop(0))
            return arr_frames
        def put(self, frame_data_loc: FrameData):
            id_worker = frame_data_loc.id_worker
            if self.find(id_worker):
                if isinstance(self.containers[id_worker], list):
                    self.containers[id_worker].append(frame_data_loc)
                    return True
                return False
            else:
                self.containers[id_worker] = []
                if isinstance(self.containers[id_worker], list):
                    self.containers[id_worker].append(frame_data_loc)
                    return True
                return False
        def empty(self):
            return len(self.containers) == 0
```

Після отримання фреймів іде їх обробка та формуються відповідні масиви, що ідуть для відображення Логів, а також відбувається накладання боксів на фрейми для візуальної інтерпретації.

Для пошуку обличчя використовується *DeepFace*, який може виконувати багато операцій з обличчями та їх пошуком. В нашій системі він відіграє практично ключову роль для аналізу людей і об'єднання їх доданими через візуальний інтерфейс, що дозволяє проводити об'єднання структурних одиниць людей і в результаті, після знайдення, отримувати фактичну приналежність до якогось обличчя тієї чи іншої людини на відео.

Окремо слід розглянути програмну реалізацію функціоналу, що відповідає за зміну розміру об'єму багажу людини. Цій задачі в основному відповідатиме функція *update(),* що має наступні вхідні параметри:

– *camera_id* – ідентифікатор камери;

– *array_update* – масив що складається зі структури пари людина-багаж. При цьому ці сутності представляються окремими масивами. Об'єднавши дані масивів ми визначаємо єдину людину і багаж що їй належить;

– *pman – PairManager()*;

– *eman – EntityManager()*.

На основі даних з *array_update[]* формується масив Логів, які використовуються для відображення. Оновлення даних в Логах відбувається за допомогою порівняння двох масивів *array_update[]* та *old_array_update[]*.

Далі наведена реалізація циклу, що допомагає відображати інформацію про первинно задетектований багаж людини в Логах.

```
else:
        for elem in array_update:
            human = EntityManager.EHuman(camera_id)
            human.push_idx(elem[0])
            for bag_s in to_list(elem[1]):
                human.push_bag(bag_s, camera_id)
                pre_logs.append((human, bag_s, "create"))
                print("C3")
            self.humans.append(human)
        pass
```

Якщо певна одиниця багажу міститься в *old_array_update[]*, проте вона відсутня в *array_update[],* то цю інформацію потрібно обновити, шляхом видалення інформації про раніше задетектований багаж. Дані кроки мають програмну реалізацію відповідно до наведеного нижче фрагменту коду.

```
if exist_human is True and exist_elem is not None:
        sub_elem_groups = exist_elem[1]
        removed_bags = []
        for seg_new in elem_groups:
            rt = False
            for seg_old in sub_elem_groups:
                if test_multy_issubset(seg_new, seg_old):
                    rt = True
                    break
            if rt is False:
                removed_bags.append(seg_new)
        pass
        for rm_elem in removed_bags:
            pass
            for i_human, human in enumerate(self.humans):
                if isinstance(human, EntityManager.EHuman):
                    if human.camera_id == camera_id:
                        if test_multy_issubset(elem[0], human.get_idx()):
                            human.remove_bag(rm_elem, camera_id)
                            pre_logs.append((human, rm_elem, "remove"))
                            print("R1")
                            break
```

Для виявлення оновлення стану одиниці багажу (спершу людина була виявлена лише з валізою, через деякий час – з ранцем) використовуємо наступний функціонал. Маючи дані про раніше задетектованих людей, ми можемо отримати інформацію про окрему людину та її багаж (весь багаж людини відповідає масиву

sub_elements_group[]). Об’єкт *human* містить, при наявності, інформацію про свій багаж. Якщо багаж окремої людини оновлюється, то в об’єкт *human,* за допомогою унікального ідентифікатора багажу, заноситься ця інформація. Якщо людина задетектована вперше, то створюється сутність людини разом з багажем. Реалізація даного алгоритму представлена фрагментом нижче.

```
for i, sub_elem in enumerate(array_update):  sub_elem_groups = sub_elem[1]
        exist_human = False
        for j, elem in enumerate(self.old_array_update):
            elem_groups = elem[1]
            if test_multy_issubset(sub_elem[0], elem[0]):
                exist_human = True
                break
        if exist_human is True:
            human = self.get_human(sub_elem[0], camera_id)
            if human is not None:
                human.push_idx(sub_elem[0])
                for bag_s in to_list(sub_elem_groups):
                    if human.exist_bag(bag_s, camera_id) is not True:
                        human.push_bag(bag_s, camera_id)
                        pre_logs.append((human, bag_s, "create"))
                        print("C1")
                    else: human.update_bag(bag_s, camera_id)
            pass
        elif exist_human is False:
            human = EntityManager.EHuman(camera_id)
            human.push_idx(sub_elem[0])
            for bag_s in to_list(sub_elem_groups):
                human.push_bag(bag_s, camera_id)
                pre_logs.append((human, bag_s, "create"))
                print("C2")
```

self.humans.append(human)

pass

Для того щоб задетектувати зміну розміру багажу людини ми маємо два масиви: перший (*bags_and_human_by_old_logs[]*) – містить в собі уже сформовані пари, інший (*new_logs[]*) – містить лише дані поточного фрейму (кадру).

Для того щоб виявити зміну розмір багажу окремої людини, ми беремо уже існуючі інформацію про її багаж зі старого масиву (при наявності такої). Якщо зміна багажу відбулася інформація оновлюється в *new_logs[]*.

Така система і її функціональні особливості є вірним кроком до того щоб розробити більш модернізовану систему пошуку із виконанням обчислень на прискорювачах. Основними особливостями поточної реалізації є розділена структура на класи функції і параметри а також, що важливо, це використання в ключових точках бібліотек що мають офіційне оновлення наприклад така як *from ultralytics import YOLO,* що оновлюється офіційно і має стабільні версії та багато навчених моделей. Система аналізу має в собі прості і ефективні прийому аналізу даних і обробки боксів людей. Трекер також містить прості і ефективні рішення, що гарно показують себе в обробці та аналізі. Ця бібліотека підтримується іншими людьми, що працюють над її покращенням. Загалом бібліотеки мають відкритий код що є дуже значним плюсом для них а також дозволяє при потребі правити їх так як буде вигідно тому хто їх буде використовувати.

3.4. Тестування системи

Після успішної реалізації системи та налаштування гіперпараметрів переходимо до тестування системи. У цьому розділі докладно розглянемо план тестування, об’єкти тестування і покажемо результати тестування а різних відео та потоках даних.

Розглянемо об’єкти тестування. Першим об’єктом тестування обрано реальні відеозаписи з веб–камер та інших джерел. Ці відео будуть містити різні сцени з людьми та їхнім багажем, що дозволить нам оцінити роботу системи в

реальних умовах. Під час тестування ми подаємо на вхід системи відеозаписи та відеопотоки, що містять обличчя людей та багаж.

Тестування системи є критично важливим етапом у розробці, оскільки воно дозволяє переконатися, що система працює на відповідному рівні точності та ефективності. Результати тестування дозволять нам підтвердити, що система відповідає поставленим вимогам та може ефективно виконувати завдання з визначення обличь та аналізу багажу в реальному часі.

Під час тестування ми подали на вхід системи різні відеозаписи та відеопотоки, які містили сцени з людьми та їхнім багажем. Під час тестування ми також записували метрики продуктивності системи, такі як час обробки кадру та час відповіді.

За допомогою меню «Операції» та «Завантаження файлу з камери» обираємо відео із файлової системи та запускаємо його аналіз в програмі. Цей вибір показано на рис 3.8.

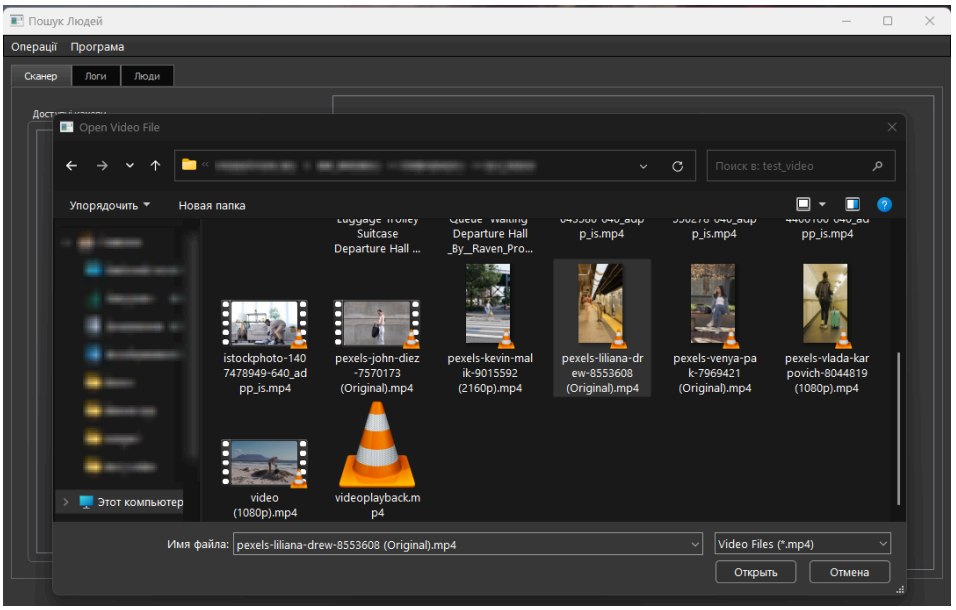

Рис. 3.8. Початковий вибір відео для аналізу

На рис. 3.9 ми бачимо роботу програми і відображення завантаженого відео. Автоматично відбувся запуск аналізу відео.

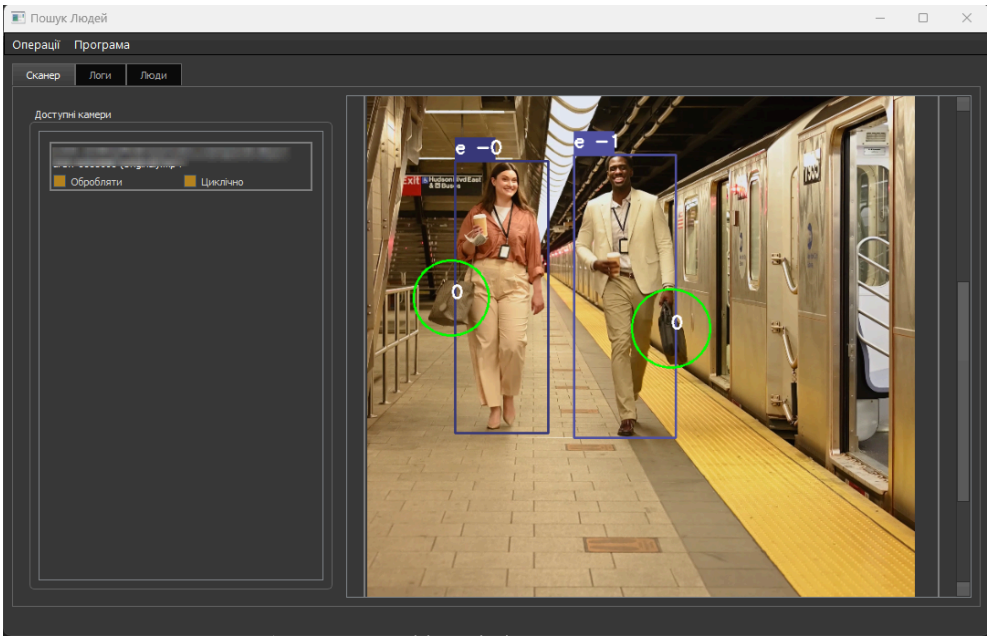

Рис. 3.9. Запуск аналізу відео та його попередні результати

Уже на рисунку вище можна спостерігати успішне виявлення людей та їх багажу.

Далі тестуємо перевірку знаходження людей та багажу в кадрі і відображення цієї інформації в інтерфейсі програми.

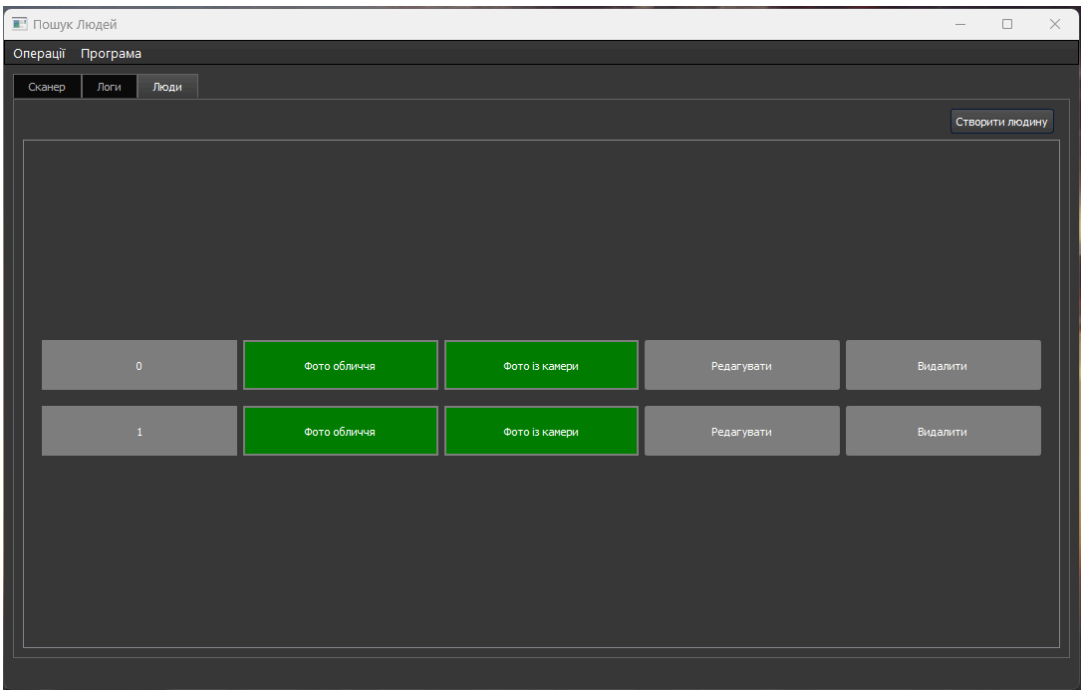

Рис. 3.10. Відображення виявленої інформації в інтерфейсі програми

Користувацький інтерфейс демонструє рис. 3.10, а саме результат того, що алгоритм успішно виявив необхідні нам образи і відповідно їх згрупував. Натиснувши на мітку, яка підписана «Фото із камери», маємо результат наведений на рис. 3.11.

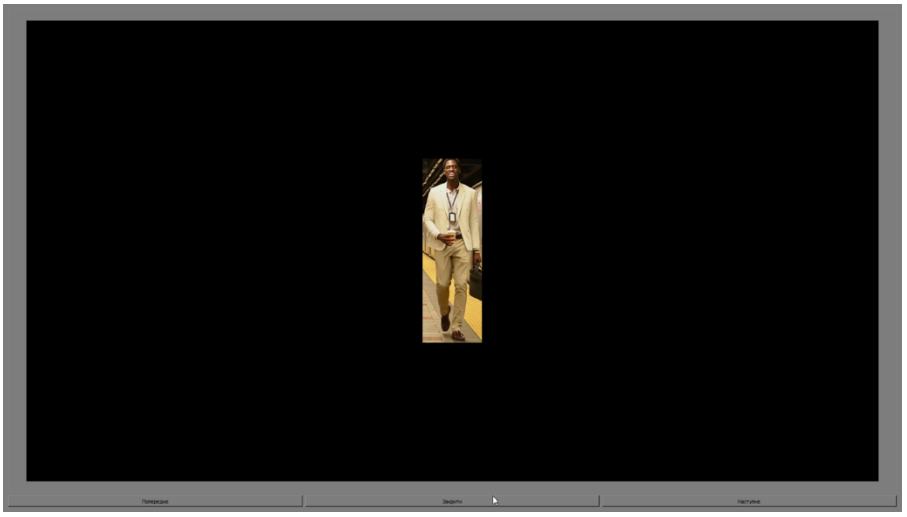

Рис. 3.11. Результат відображення інформації з віджету «Фото з камери»

Тобто ми бачимо відокремлення (вирізання) з кадру людини, що нас (оператора) цікавить.

Розглянемо детальніше вкладку Логи та їх вигляд, а також склад того, що буде виводитися в результаті. Проаналізуємо те саме відео, і розглянемо один із Логів, що сформувався і показаний на рис. 3.12.

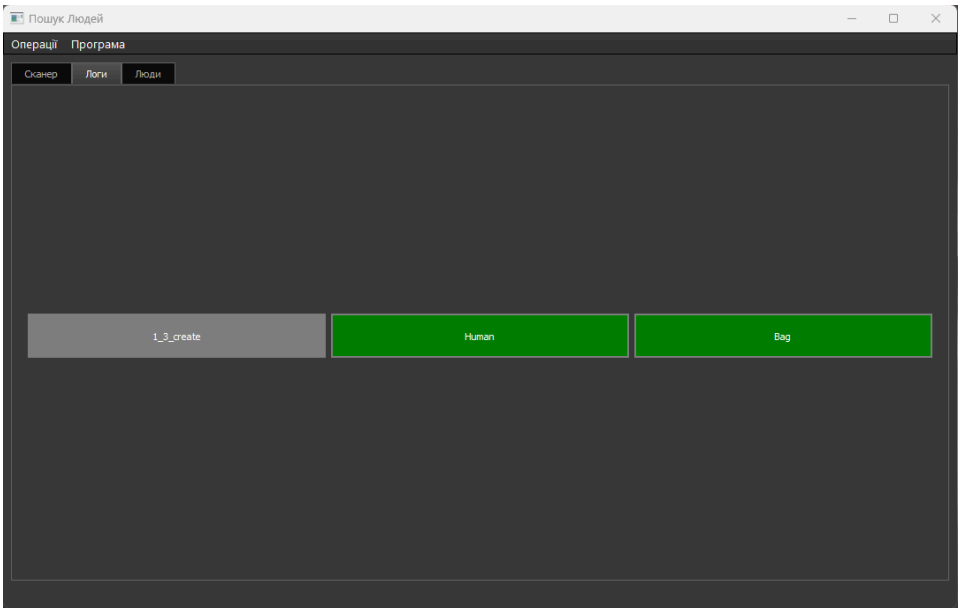

Рис. 3.12. Результат детекції з відео

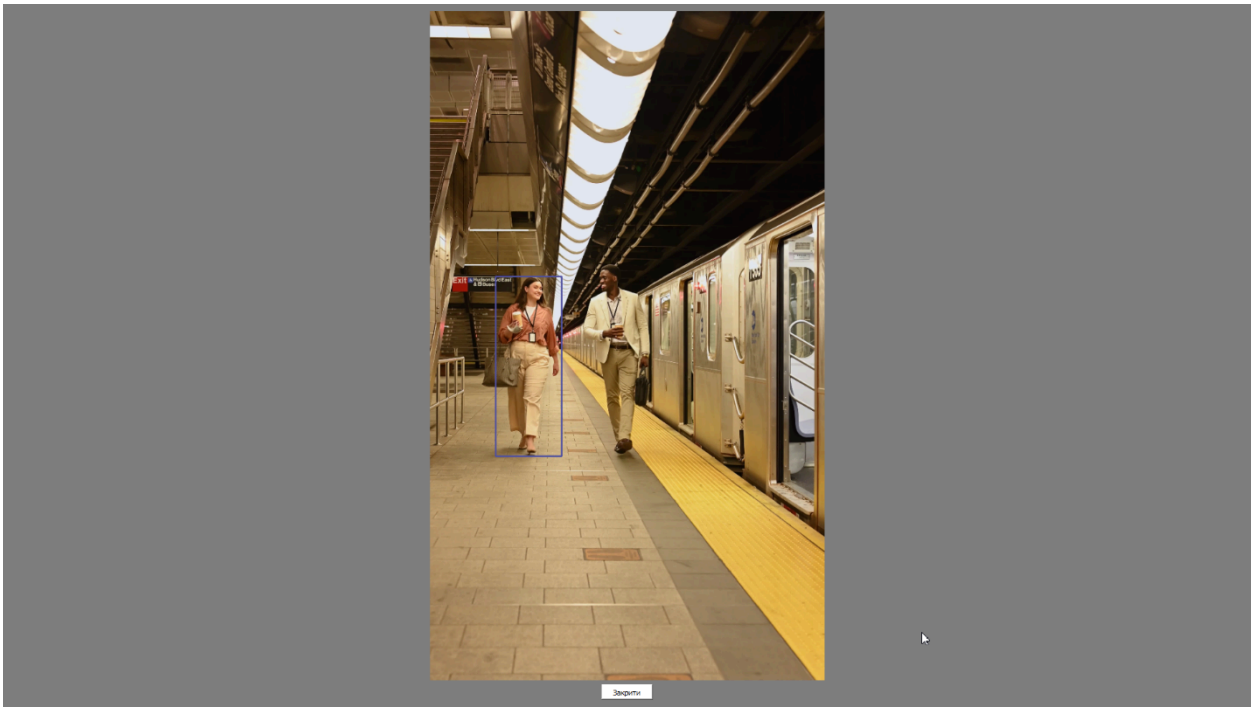

Рис. 3.13. Результат відображення інформації з віджету *Human*

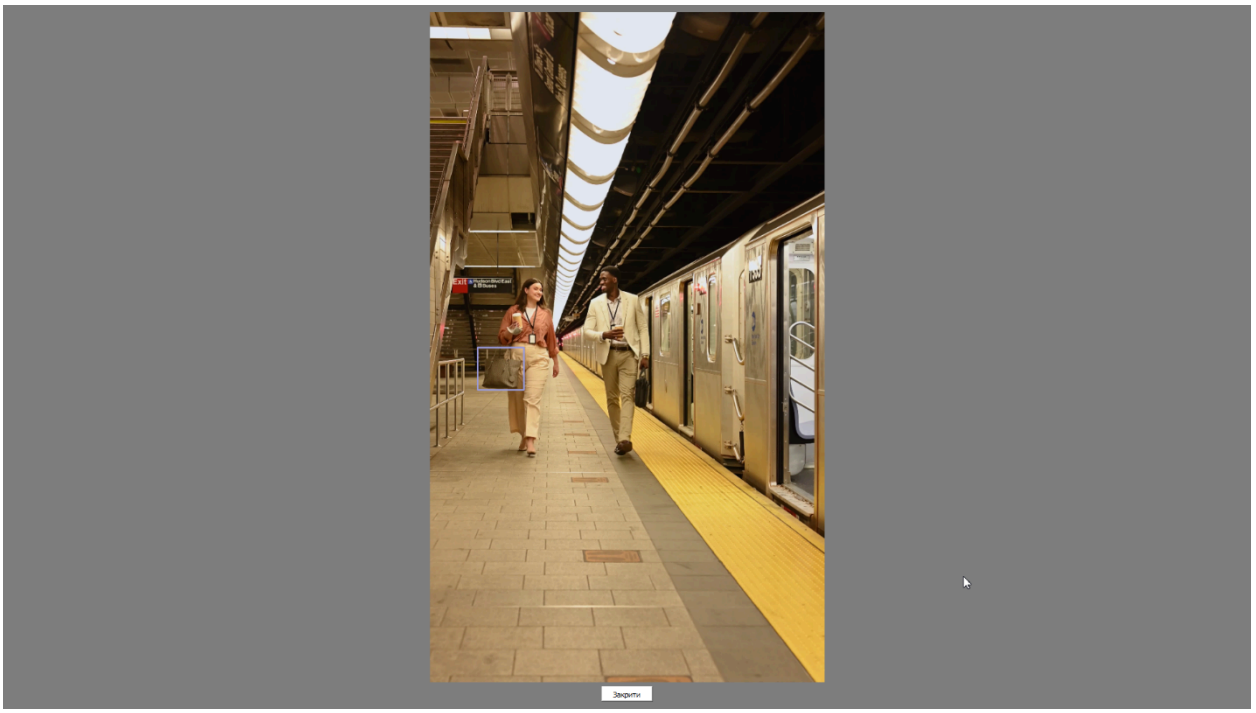

Рис. 3.14. Результат відображення інформації з віджету *Bag*

На рис. 3.13 та рис. 3.14 бачимо успішне виявлення багажу людини, і безпосередньо самої людини.

Важливою функцією реалізації є виявлення зміни багажу людини. Для візуалізації цього функціоналу записано власне відео. Спершу людина мала один тип багажу, а саме валіза (рис. 3.15).

Рис. 3.15. Первинна детекція людини з першим типом багажу

Після людина виходить виходить з кадру, і змінює тип багажу на інший, менший за об’ємом (рис. 3.16).

Рис. 3.16. Детекція людини з іншим типом багажу

В результаті на вкладці ми отримуємо візуалізацію, зображену на рис. 3.17, де червоним виділено важливу інформацію, що показує зміну розміру багажу однієї і тієї ж людини.

Рис. 3.17. Виділення інформації про зміну об’єму багажу однієї і тієї ж людини

3.5. Висновки до розділу

Даний розділ націлений на програмну реалізація програмного модулю розпізнавання облич і речей пасажирів аеропорту. Користувачем системи визначено оператора, що діє в системі і має певні повноваження, що було показано за допомогою *Use Case* діаграми та діаграми прецедентів.

Після цього ми визначили алгоритм, якого будемо притримуватися при реалізації функціоналу системи. Однією з першочергових задач є налаштування гіперпараметрів, що визначають різні аспекти роботи системи. Визначивши їх ми ініціалізували обрану модель та перейшли до реалізації функціоналу. Основними функціями визначено створення віджетів програми, створення контейнеру для фреймів, створення потоку обробки фреймів, створення таймеру для показу логів та створення потоку для пошуку людей.

Використовуючи принцип багатопотоковості реалізовано одночасну обробку декількох камер. Для цього була створена функція, що реалізує запуск окремого потоку. Отримавши дані з камери, а саме відеопотік, що ділиться на окремі кадри, функція передає ці дані для подальшої обробки. Ця обробка заключається в попередній обробці зображення, формуванні пар людина-багаж, подальше відстеження цих пар і відстеження їх зміни.

Для спрощеної відладки багатопотокової програми реалізовану функцію, за допомогою якої можна викликати інші функції, вказуючи тип функції (таймер чи потік). Це фактично дозволяє викликати будь яку функцію або у вигляді таймеру або у вигляді потоку. Це необхідно так як таймери фізично працюють на одному потоці із викликами і дозволяють перевіряти потрібні дані. Самі потоки не дають такої можливості і тому мають недоліки при відладці.

Окрім цього, налаштовувалися і затримки вхідного сигналу, що дозволяють системі працювати стабільніше та менше навантажувати саму операційну систему.

Для безпосереднього початку аналізу фреймів створюємо потік і запускаємо його роботу з відповідними затримками вхідного сигналу.

Для подальшого аналізу отриманих результатів визначено такі сутності, як людина, сумка (одиниця багажу), пара та група сумок (декілька одиниць багажу).

Об’єднання сутності людини і її багажу реалізується за допомогою функції на основі експоненційно згладжуваного середнього, що дозволяє згладжувати стани короткого виходу із зони перетину, тобто уникнути помилкового розформування пари.

Для тестування відповідності функціоналу поставленим задачам було вирішено спершу дати модулю відеопотік з мережі інтернет, в результаті чого виявлено успішне розпізнавання людини, та багажу що їй належить. Для перевірки того, чи система розпізнає зникнення багажу чи зміну його об’єму було використано власний відеопотік. В результаті чого, при зміні багажу з одно типу на інший (спочатку людина мала валізу, а після – ранець), система надала необхідні повідомлення.

ВИСНОВКИ

В результаті виконання роботи вирішено актуальне науково–практичне завдання розроблення програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту. Отримано наступні наукові результати:

1. Запропоновано архітектуру програмного модулю розпізнавання речей та обличчя пасажирів в аеропорту, що, на відміну від існуючих програмних рішень, забезпечує стеження не лише за пасажиром та його речами, а й встановлення приналежності (тобто визначення того, що окремий багаж належить саме цьому пасажиру), та реагування на зміну розміру чи типу багажу;
2. Розроблено нейронну мережу для встановлення приналежності багажу конкретній людині, що надає можливість реалізувати процес одночасного трекінгу пасажиру та його багажу, що, з однієї сторони, підвищує рівень безпеки, а, з іншої, зменшує навантаження на оператора системи;
3. Удосконалено розроблену нейронну мережу для відстеження зміни об’єму багажу, що належить конкретній людині, що надає можливість генерувати відповідні інформаційні повідомлення оператору у разі помітних відхилень такого об’єму з метою додаткової перевірки.

В ході виконання кваліфікаційної роботи виконана визначена кількість задач що допомогли досягнути мети.

Для досягнення поставленої мети спершу було вивчено еволюцію розвитку та впровадження комп’ютерного зору в різні суспільні сфери, визначені особливості технологій комп’ютерного зору від поверхневих методів машинного навчання до наскрізного навчання. Подальше визначення успішного використання комп’ютерного зору призвело до необхідності визначення можливих недоліків, що були поділені на дві групи – проблеми програмного забезпечення та апаратні проблеми. Після цього було охарактеризовано еволюцію впровадження технологій комп’ютерного зору безпосередньо в системи відеоспостереження, а також розглянуті уже відомі практичні результати впровадження.

Задача, що була висвітлена в другому розділі націлена на аналіз існуючих програмних рішень, їх порівняння, та визначення тих, що будуть застосовуватися безпосередньо при моделюванні. Спершу було розглянуто моделі нейронних мереж, їх поділ і визначено, що саме згорткові нейронні мережі значно вплинули на розвиток області комп'ютерного зору. Далі охарактеризовано підходи до виявлення об'єктів, і визначено, що саме моделі одноетапного виявлення надають оптимальне значення продуктивності. Вибір моделі зупинився на *YOLOv8*, що має найкращі характеристики продуктивності, швидкості і точності, при цьому маючи найнижчі обрахункові витрати.

В цьому розділі визначено і охарактеризовано бібліотеку *PyQt5*, яку ми будемо використовувати для побудови графічних інтерфейсів користувача. Окрім цього запропоновано використання алгоритму *DeepSort*, що застосовується в багатьох системах відеоспостереження, аналізі відео та розпізнаванні рухових об'єктів. В якості вбудови для цього алгоритму обрано *MobileNetv2*. Також необхідне використання легковагового фреймворку *DeepFace* для розпізнавання обличчя, що комбінує в собі кілька передових моделей розпізнавання обличчя.

Визначивши всі необхідні матеріали і методи ми перейшли до програмної реалізації. Спершу проведено аналіз системи та функцій які вона повинна виконувати. Визначено, що спершу має виявляти та ідентифікувати особу. Протягом стеження за цією особою повинна аналізуватись і інформація про її багаж. Система виконує ці функції безперервно, тобто моніторинг іде безперебійно. Якщо зміни приналежного багажу виявляються то оператор повинен бути сповіщений про це.

Для реаліазції такої системи побудовано алгоритм. Спершу іде підключення бази даних, налаштування гіперпараметрів що необхідні для ініціалізації моделі. Після цього можна переходити безпосередньо до реалізації програмних функцій.

Для роботи системи визначено особливості сутностей і робота з ними. Основні сутності визначені як людина, сумка (одиниця багажу), пара та група сумок (декілька одиниць багажу). Тобто сутність (*class Entity*) описуватиме одну людину і її багаж у вигляді груп.

Для пошуку обличчя використовується *DeepFace,* що в розроблюваній системі грає ключову роль для аналізу облич людей.

Використовуючи принцип багатопотоковості реалізовано модуль що має змогу слідкувати за декількома камерами одночасно, при цьому виявляючи приналежність багажу людини та зміну стану цієї взаємодії.

Основною особливістю розробленого модулю є визначення приналежності сутності людини та сутності одиниці багажу. Математично це відповідає перетину обмежувальних рамок сутності людини і сутності сумки. Функція, що знаходить перетин обмежувальних рамок цих сутностей побудована на основі експоненційно згаджуваного середнього. Таке застосування згладжує стани короткого виходу з зони перетину, тим самим уникаючи помилкового розформування пари.

Окрім цього, при реалізації функціоналу постала необхідність використання об’єкту *mutex* для оновлення відображення результатів побудови обмежувальних рамок кожного фрейму. З його допомогою ми блокуємо всі потоки обробки фреймів окрім поточного, після цього поміщуємо фрейм у контейнер для зберігання та подальше управління ним.

Розроблена система виконує важливу роль у безпеці аеропортів. Проте, слід зауважити, що система може використовуватися в усіх публічних місця та великих скупченнях людей, де є необхідність моніторингу людини і її багажу. Система допомагає виявляти і вирішувати потенційні загрози безпеки, що можуть включати терористичні чи шахрайські дії, тобто привласнення чужого багажу, чи різка зміна об’єму багажу. Необхідне сповіщення надаватиметься оператору і це, відповідно, зменшить навантаження на даних працівників.

Отже, в результаті розробки програмного модулю, розроблено функціонал що відіграє важливу роль в безпеці аеропортів, зменшує навантаження на операторів систем відеоспостереження, і загалом має тенденцію до підвищення ефективності управління публічними подіями та їх просторами.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бойченко С. В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету / С. В. Бойченко, О. В. Іванченко; Національний авіаційний університет. – Київ: – 63 с.

2. ДСТУ 3008–95. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. – Введ. 1995-23-02. – ІПрІн, УкрІНТЕЇ, Головний відділ стандартизації Технічного центру НАН України, 1995. – №58, 88 с.

3. ДСТУ *ISO* 5807:2016. Обробляння інформації. Символи та угоди щодо документації стосовно даних, програм та системних блок-схем, схем мережевих програм та схем системних ресурсів (*ISO* 5807:1985, *IDT*). – Введ. 2106-10-10. – ДП «УкрНДНЦ», 2016.

4. *Kirsch R. A. Experiments in Processing Pictorial Information with a Digital Computer/ Kirsch R. A., Cahn L., Ray C. // Proceedings of the eastern computer conference.* – 1957. – *P.* 221–229.

5. *Fukushima K. Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position / Fukushima K. // Biological Cubernetics*. – 1980. – № 36. – *P*. 193–202.

6. *Johnston B. A Brief History of Surveillance Cameras [Electronic resource]. – Access mode: https://www.deepsentinel.com/blogs/home-security/history-of-surveillance-cameras* (*lastaccess*: 02.10.2023)

7. *Santiago D. History and Evolution of Video Surveillance Technology | 3Sixty Integrated [Electronic resource]. – Access mode: https://www.3sixtyintegrated.com/blog/2023/07/26/history-video-surveillance/* (*lastaccess:* 03.10.2023).

8. *Devasia A. The Expansive Reach: 7 Diverse Applications of Video Surveillance [Electronic resource]. – Access mode: https://thenetworkinstallers.com/blog/applications-of-video-surveillance/* (*lastaccess*: 03.10.2023).

9. *Coastal Surveillance Systems [Electronic resource]. – Access mode: https://elbitsystems.com/products/homeland-security-systems/coastal-surveillance-systems/* (*lastaccess*: 04.10.2023).

10. *IndigoVision Intergation [Electronic resource]. – Access mode: https://www.indigovision.com/products/integration/* (*lastaccess*: 05.10.2023).

11. *Dallmeier sets another record for resolution and dynamic range with the new Panomera® S8 Ultraline2 [Electronic resource]. – Access mode: https://www.dallmeier.com/solutions/airport* (*lastaccess*: 06.10.2023).

12. *Types of Deep Neural Networks [Electronic resource]. – Access mode: https://mriquestions.com/deep-network-types.html* (*lastaccess*: 06.10.2023).

13. *CS231n: Convolutional Neural Networks for Visual Recognition [Electronic resource]. – Access mode: https://cs231n.github.io/convolutional-networks/* (*lastaccess*: 07.10.2023).

14. *R-FCN: Object Detection via Region-based Fully Convolutional Networks [Electronic resource]. – Access mode: https://github.com/daijifeng001/R-FCN#r-fcn-object-detection-via-region-based-fully-convolutional-networks* (*lastaccess*: 07.10.2023).

15. *SSD: Single Shot MultiBox Detector / W. Liu et al. // European Conference on Computer Vision. – Amsterdam,* 2016. – *P*. 21–37.

16. *You Only Look Once: Unified, Real-Time Object Detection / S. Redmond et al. // IEEE Conference on Computer Vision and Pattern Recognition (CVPR). – Las Vegas, NV,* 2016. – *P*. 779-788.

17. *Exploring the Power of YOLOv8: Comprehensive Analysis of Object Detection, Classification, and Segmentstion [Electronic resource]. – Access mode: https://medium.com/@photon4273/revolutionizing-the-ability-of-yolov8-a-depth-insight-of-object-detection-classification-and-8c3801524176* (*lastaccess*: 08.10.2023).

18. *The New Frontier of Vision AI [Electronic resource]. – Access mode: https://github.com/ultralytics/ultralytics* (*lastaccess*: 10.10.2023).

19. *Difference Between Multithreading vs Multiprocessing in Python [Electronic resource]. – Access mode:*

https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/ (*lastaccess*: 12.10.2023).

20. *PyQt Documentation v5.15.4. Introduction [Electronic resource]. – Access mode: https://www.riverbankcomputing.com* (*lastaccess*: 13.10.2023).

21. *Using a Designer UI File in Your C++ Application [Electronic resource]. – Access mode: https://doc.qt.io/qt-6/designer-using-a-ui-file.html* (*lastaccess*: 17.10.2023).

22. *Sanyam. Understanding Multiple Object Tracking using DeepSORT [Electronic resource]. – Access mode: https://learnopencv.com/understanding-multiple-object-tracking-using-deepsort/* (*lastaccess*: 19.10.2023).

23. *DeepSORT [Electronic resource]. – Access mode: https://github.com/levan92/deep_sort_realtime#appearance-embedding-network* (*lastaccess*: 18.10.2023).

24. *Chablani M. CLIP (Contrastive Language-Image-Pre-Training) – Summary and intuition [Electronic resource]. – Access mode: https://medium.com/@ManishChablani/clip-contrastive-language-image-pretraining-summary-and-intuition-52e329a67377* (*lastaccess*: 18.10.2023).

25. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications / Andrew G. et al. // arXiv: Computer Vision and Pattern Recognition*. – 2017. – *P*. 1-9.

26. *Deep Face Recognition [Electronic resource]. – Access mode: https://www.geeksforgeeks.org/deep-face-recognition/* (*lastaccess*: 19.10.2023).

27. *Koech K.E. Object Detection Metrics With Worked Example [Electronic resource]. – Access mode: https://towardsdatascience.com/on-object-detection-metrics-with-worked-example-216f173ed31e* (*lastaccess*: 20.10.2023).

ДОДАТОК А

Функція обробки фреймів в кожному окремому віджеті

```
def find(frame: FrameData, device='cpu'):
    """Основна функція детектування через модель yolov8"""
    data_frame = frame.get_frame()
    if data_frame is not None:
        frame_height, frame_width, channels = data_frame.shape
        real_frame = data_frame.copy()
        real_frame = resize_with_aspect_ratio(real_frame, new_width=700)
        real_frame = cv2.cvtColor(real_frame, cv2.COLOR_BGR2RGB)
        plot_frame = real_frame.copy()
        with torch.no_grad():
            results = get_yolov8_model_bags()(real_frame, verbose=False)
            result = results[0]
            boxes = result.boxes
            names2 = get_yolov8_model_bags().names
            cls_boxes = boxes.cls
            ci = get_object_names_with_indices(cls_boxes, names2)
            annotated_frame = result.plot()
            annotated_frame = resize_with_aspect_ratio(
                annotated_frame,
                new_width=resize_with_width)
        frame.real_frame = real_frame
        frame.yolov8 = result
        frame.index_name_yolov8 = ci
        dataframe_uid  = frame.dataframe_uid
        yolov8_data = result
```

```
        boxes = yolov8_data.boxes
        names2 = get_yolov8_model_bags().names
        cls_boxes = boxes.cls
        conf = boxes.conf
        masks = yolov8_data.masks    # noqa: F841
        keypoints = yolov8_data.keypoints    # noqa: F841
        probs = yolov8_data.probs  # noqa: F841
        xywh = boxes.xywh
        dp = []
        conf_shape = conf.shape[0]
        for i in range(conf_shape):
            if int(cls_boxes[i].item()) in all_white_list:
                el = ( [xywh[i][0] - int(xywh[i][2] / 2), xywh[i][1] - int(xywh[i][3] / 2),
xywh[i][2], xywh[i][3]], conf[i].item(),  int(cls_boxes[i].item()))
                dp.append(el)
        tracker = trackers_capacitor.get(dataframe_uid)
        tracks = []
        if isinstance(tracker, DeepSort): tracks = tracker.update_tracks(dp,
frame=real_frame)
        else:   if tracker is None:
                tracker = trackers_capacitor.create_basic_tracker(dataframe_uid)
                if isinstance(tracker, DeepSort):
                    tracks = tracker.update_tracks(dp, frame=real_frame)
        entities = pman.push_tracks(tracks, dataframe_uid, trackers_capacitor)
        if len(entities) > 0:
            pass
        cmap = plt.get_cmap('tab20b')
        colors = [cmap(i)[:3] for i in np.linspace(0, 1, 20)]
        for single_entity in entities:
            if isinstance(single_entity, Entity):
```

```
            color = colors[int(single_entity.current_id) % len(colors)]
            color = [i * 255 for i in color]
            se_info = single_entity.get_last_info(dataframe_uid)
            bbox = se_info["bbox_human"]
            original_ltwh = se_info["original_ltwh_human"]
            array_pts_group = se_info["array_pts_group_bag"]
            if original_ltwh is None:
                color = [255, 0, 0]
            if bbox is not None: #and original_ltwh is not None
                (circle_ph_last_x, circle_ph_last_y), circle_ph_last_radius =
box_to_circle(bbox)  thickness = 2
                cv2.circle(plot_frame, (circle_ph_last_x, circle_ph_last_y),
circle_ph_last_radius, color, thickness)
                cv2.rectangle(plot_frame, (int(bbox[0]),int(bbox[1])),
(int(bbox[2]),int(bbox[3])), color, 2)
                cv2.rectangle( plot_frame, (int(bbox[0]), int(bbox[1]-30)),
                    (int(bbox[0])+(len("cs") +len(str(single_entity.current_id)))*17,
                    int(bbox[1])), color, -1)
                cv2.putText( plot_frame, "e "+"-"+str(single_entity.current_id),
(int(bbox[0]), int(bbox[1]-10)),  0,  0.75, (255, 255, 255), 2)
            if len(single_entity.pairs) > 0: pass
            if len(array_pts_group) > 0:     pass
            for i_si, single_element in enumerate(array_pts_group):
                if len(single_element) > 0:
                    (center_x, center_y), radius = box_to_circle(single_element[0])
                    color = (0, 255, 0)
                    if single_element[1] is None: color = (255, 0, 0)
                    thickness = 2
                    cv2.circle(plot_frame, (center_x, center_y), radius, color, thickness)
                    cv2.putText(plot_frame,  ""+str(i_si), (int(center_x), int(center_y)),0, 0.75,
```

```
(255, 255, 255), 2 )
        global counter_f
        if counter_f % 2 == 0: frame.set_frame(plot_frame)
        else: frame.set_frame(annotated_frame)
        counter_f += 1
        logs, humans_and_bags = None, None
        array_keys_entity = []
        array_keys_entity_all = []
        array_all_humans_and_bags = []
        for single_entity in entities:
            if isinstance(single_entity, Entity):
                hk_all, bk_all = single_entity.get_unique_identifiers()
                array_keys_entity_all.append((hk_all, bk_all))
                hk, bk = single_entity.get_unique_identifiers_active_status(dataframe_uid)
                array_keys_entity.append((hk, bk))
        if len(array_keys_entity) > 0:
            logs, humans_and_bags, array_all_humans_and_bags =
compare_arrays(dataframe_uid, pman, eman, array_keys_entity, array_keys_entity_all)
            if len(logs) > 0:     pass
            if humans_and_bags is not None: if len(humans_and_bags[0]) > 0: pass
        if logs is not None:
            for single_log in logs:
                if isinstance(single_log, EntityManager.ELog):
                    single_log.camera_id = dataframe_uid
        frame.logs = logs
        frame.humans_and_bags = humans_and_bags
        frame.array_all_humans_and_bags = array_all_humans_and_bags
        frame.entities = entities
    return frame
```