НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НН Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

До захисту допущено:
Завідувач кафедри
_____ Оксана ТИМОЩУК
«___»_____2023 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системний аналіз і управління»

спеціальності 124 «Системний аналіз»

на тему: «Виявлення та відстежування об'єктів методами машинного навчання»

Виконав: студентка IV курсу, групи КА-93
Сухомлин Гліб Ігорович _____

Керівник: д.т.н., доцент
Недашківська Надія Іванівна _____

Консультант з нормоконтролю: к.ф-м.н.
Статкевич Віталій Михайлович _____

Консультант з економічного розділу: доцент, к.е.н.
Рощина Надія Василівна _____

Рецензент: професор, д.т.н.
Валерій Якович Данилов _____

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

Київ – 2023 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

НН Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 124 «Системний аналіз»

Освітня програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Оксана ТИМОЩУК

«__» травня 2023 р.

ЗАВДАННЯ

на дипломну роботу студенту

Сухомлину Глібу Ігоровичу

1. Тема роботи «Виявлення та відстежування об’єктів методами машинного навчання», керівник роботи доцент, д.т.н. Недашківська Надія Іванівна, затверджені наказом по університету від «30» травня 2023 р. № 2065-с.

2. Термін подання студентом роботи 12.06.2023.

3. Вихідні дані до роботи – COCO, Market 1501 та MOT17 датасети.

4. Зміст роботи: 1. Дослідження предметної області; 2. Математичні основи роботи; 3. Практичні результати; 4. Функціонально-вартісний аналіз.

5. Перелік ілюстративного матеріалу: представлення вихідних даних до роботи, архітектури нейромереж для виявлення об’єктів, структури алгоритмів, презентація.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В., доц., к.е.н.		

7. Дата видачі завдання 3.03.2023

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми БДР. Ознайомлення зі структурою БДР згідно з Положення про випускну атестацію студентів КПІ ім. Ігоря Сікорського та з державним стандартом України ДСТУ ГОСТ 3008-2015 та стандарти ЄСПД.	17.04.2023-23.04.2023	Виконав
2.	Аналіз сучасних джерел та досліджень з теми виявлення об’єктів шляхом нейронних мереж та алгоритму DeepSort. Освоєння необхідного мінімуму теоретичного матеріалу для розуміння роботи сучасних систем виявлення та відслідковування.	24.04.2023-30.04.2023	Виконав

3.	Дослідження та вибір практичних підходів для побудови моделей вилення та відслідковування об’єктів - прогрмні пакети, фреймворки, середовище, а також технічне обладнання. Пошук та вибір даних для тренування та тестування відповідних моделей**.**	1.05.2023-7.05.2023	Виконав
4.	Реалізація та тестування детекторів та трекерів побудованих на основі методів машинного навчання.	8.05.2023-14.05.2023	Виконав
5.	Було проведено удосконалення та документування програмного продукту.	15.05.2023-21.05.2023	Виконав
6.	Написання функціонального-вартісного аналізу розробленої практичної частини, аналіз результатів, оформлення дипломної роботи.	22.05.2023-28.05.2023	Виконав

Студент Гліб СУХОМЛИН

Керівник Надія НЕДАШКІВСЬКА

РЕФЕРАТ

Дипломна робота: 158 с., 83 рис., 6 табл., 2 додатки, 42 джерела.

НЕЙРОННІ МЕРЕЖІ, ВИЯВЛЕННЯ ТА ВІДСТЕЖУВАННЯ ОБ’ЄКТІВ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, ПРОГРАМА ДЛЯ ВІДСТЕЖУВАННЯ.

Об’єкт дослідження – виявлення та відстежування об’єктів з використанням периферійних пристроїв (різного роду відео-камер).

Часто коли люди працюють з відео-матеріалами, вони стикаються з проблемою виявлення та класифікації об’єктів, які знаходяться на поточному кадрі. Це є необхідним у багатьох сферах людської діяльності. Наприклад, для створення автономної системи керування автомобілем, оскільки перш ніж штучний інтелект буде приймати рішення щодо керування авто має бути чітке розуміння з якими об’єктами він стикається у поточний момент часу.

Однак може виникнути задача відстеження цілої історії об’єкта або об’єктів на відповідному відео матеріалі, точніше кажучи – траєкторії руху цілей, які були присутні на відео. Таким чином задача відстежування об’єктів є логічним продовженням попередньої задачі.

Мета роботи – розробити програму з використанням існуючих моделей для виявлення та відстежування об’єктів, причому зробити це з оптимальним використанням ресурсів.

Бажано щоб розроблена програма виконувала поставлену задачу в реальному часі, а також щоб модель була не занадто складною з точки зору часу опрацювання зображень аби можна було не витрачати додаткові ресурси на облаштування серверів, тобто щоб по суті уся робота виконувалася лише периферійними пристроями.

Програмний продукт розроблено на мові програмування Python. Було реалізовано модель yolo в комбінації з алгоритмом DeepSort. Практичним результатом роботи є система виявлення і відстеження об'єктів.

ABSTRACT

Bachelor thesis: 158 p., 83 fig., 6 tabl., 2 appendices, 42 sources.

NEURAL NETWORKS, OBJECT DETECTION AND TRACKING, CONVOLUTIONAL NEURAL NETWORKS, TRACKING SOFTWARE.

The object of research is the detection and tracking of objects using peripheral devices (various video cameras).

Often, when people work with video materials, they face the problem of identifying and classifying objects that are in the current frame. This is necessary in many areas of human activity. For example, to create an autonomous car control system, because before artificial intelligence can make decisions about driving a car, there must be a clear understanding of what objects it encounters at the current moment in time.

However, there may be a problem of tracking the entire history of an object or objects on the corresponding video material, more precisely, the trajectory of the targets that were present on the video. Thus, the object tracking task is a logical continuation of the previous task.

The goal of the work is to develop a program using existing models for object detection and tracking, and do it with the optimal use of resources.

It is desirable that the developed program performs the task in real time, and also that the model is not too complicated in terms of image processing time so that additional resources cannot be spent on setting up servers, i.e. that essentially all work is performed only by peripheral devices.

The software product is developed in the Python programming language. The yolo model was implemented in combination with the DeepSort algorithm. The practical result of the work is a system of detection and tracking of objects.

ЗМІСТ

ВСТУП

Більшість систем автономного керування потребують чіткої інформації про стан навколишнього середовища з яким вони взаємодіють. Відстежування об’єктів в реальному часі дозволяє системі швидко та оперативно реагувати на відповідно зміни щоб уникнути різного роду аварійних ситуацій.

Без програм, які можуть розпізнавати та відстежувати ключові образи на поточних відео-фрагментах у системи зайняло б занадто багато часу на врахування усіх можливих факторів і на поточний момент висновки цієї автономної системи керування могли б бути не актуальними. Тому варто шукати такі моделі виявлення об’єктів, які б забезпечували баланс між швидкістю обробки зображень та якістю зроблених висновків.

Схожа ситуація і з використанням систем спостереження. Вони відграють важливу роль в житті людей. Необхідні камери не є проблемою з фінансової точки зору, найдорожча частина даної системи – це моніторинг. Для моніторингу в режимі реального часу часто потрібно призначати цілу команду, тобто персонал служби безпеки. Однак методи машинного навчання (використання згорткових нейромереж) суттєво вирішують цю проблему, допомагаючи не тільки значно зекономити на утриманні цілого персоналу, а й зробити систему більш надійною та швидкодіючою.

Предметом дослідження є різні найбільш ефективні архітектури згорткових нейромереж які реалізують процес виявлення об’єктів та їх подальша взаємодія з алгоритмами Sort та DeepSort. Причому кожна з них не обов’язково є ефективною одночасно з точки зору якості класифікації і регресійної частини роботи та витраченого часу на оброблення зображень.

В даній роботі буде зроблено порівняльний аналіз різних детекторів (моделей виявлення об’єктів) з використанням алгоритмів Sort та DeepSort, як вдосконалення алгоритму Sort, які використовуються для відстежування об’єктів. Потім на основі отриманих оцінок виберемо найкращу модель, яка і буде кінцевим результатом нашої роботи.

Пояснювальна записка складається з чотирьох розділів. У першому описується актуальність поставленого завдання та уточняється, де саме можуть бути застосовані сучасні моделі виявлення та відслідковування об’єктів. У другому розглядається математичний апарат, який є необхідним для розуміння цих моделей. У третьому наводиться порівняльна характеристика апробованих моделей з точки зору їх ефективності і вибирається найкраща для вирішення поставленого завдання (можливо цих моделей буде декілька, тобто кожна буде найкраще задовольняти деякий критерій). У четвертому розділі проводиться функціонально-вартісний аналіз побудованої моделі.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність роботи

Дана тематика є актуальною оскільки задача виявлення та відстежування об’єктів виникає доволі часто в різних сферах людської діяльності. Окрім вище наведених систем автономного керування автомобілем та систем спостереження, які відповідають по суті за моніторинг рівня безпеки, це можуть бути й інші сфери людської діяльності. (Приклади наведені нижче)

Спорт: трекери також можна використовувати у спорті, наприклад для відстеження головних елементів гри – м’яч, шайба і т.д. або для відстеження гравців. Це допоможе виявляти фоли, правомірні голи у грі, скоєння дій, які суперечать правилам, та багато іншого.

Моніторинг дорожнього руху: трекери можна використовувати для моніторингу дорожнього руху та відстеження транспортних засобів на дорозі. Також за їх допомогою можна оцінити завантаженість дорожнього руху, рівень небезпеки деяких ділянок дороги, виявляти порушення ПДД і хто саме скоїв конкретне порушення. створюючи такі системи, як система автоматичного розпізнавання номерних знаків.

Спостереження за допомогою кількох камер: у відстеженні можна застосувати спостереження за допомогою кількох камер. У цьому ключовою ідеєю є повторна ідентифікація. Якщо людину відслідковують в одній камері з ідентифікатором, а людина виходить з кадру і повертається в іншу камеру, то тоді людина збереже той самий ідентифікатор, який мав раніше. Ця програма може допомогти повторно ідентифікувати об’єкти, які знову з’являються в іншій камері, і може використовуватися для виявлення злочинців, постійних клієнтів, щоб краще зрозуміти їх вподобання, та для використання в багатьох інших прикладних сферах.

Використання у воєнній галузі: сучасні дрони базуються на гарних системах спостереження, які можуть точно та своєчасно, тобто в режимі реального часу ідентифікувати об’єкти, які їх оточують, щоб своєчасно були прийняти оптимальні рішення щодо усунення наявної небезпеки.

Також варто зазначити, що моделі виявлення та відслідковування об’єктів здатні неабияк прискорити та вдосконалити механізми навчання з підкріпленням. Вони можуть виявляти усі елементи, які необхідні нам на деякому відео-фрагменті і до того ж зберігати розташування цих елементів. Таким чином нам не потрібно запам’ятовувати все відео, а лише ту інформацію, яку дає нам на виході сама модель. Це суттєво спрощує задачу, зменшуючи ймовірність потрапити в локальні мінімуми при мінімізації (або максимізації) деякого функціоналу, при цьому залишаючи всю необхідну інформацію, щоб навчитися оптимальній стратегії.

1.2 Види трекінгових систем

Трекери (моделі для покадрового відслідковування об'єктів) можна класифікувати на основі багатьох категорій, таких як методи відстеження або кількість об’єктів, які слід відстежувати [37].

Трекери одного та кількох об’єктів.

Трекер одного об'єкта:

Ці типи трекерів відстежують лише один об’єкт, навіть якщо в кадрі багато інших об’єктів. Вони працюють, спочатку ініціалізуючи розташування об’єкта в першому кадрі, а потім відстежуючи його протягом усієї послідовності кадрів. Ці типи методів відстеження дуже швидкі. Багато з них створені за допомогою традиційного комп’ютерного бачення. Проте тепер доведено, що трекери на основі глибокого навчання набагато точніші, ніж традиційні трекери.

Трекер кількох об'єктів:

Ці типи трекерів можуть відстежувати кілька об’єктів, присутніх у кадрі. На відміну від традиційних трекерів, трекери кількох об’єктів або MOT навчаються на великій кількості даних. Таким чином, вони виявилися більш точними, оскільки можуть відстежувати декілька об’єктів і навіть різних класів одночасно, зберігаючи високу швидкість. Саме такий тип трекерів буде розглядатись в даній роботі, точніше це алгоритм Sort та DeepSort як його модифікація.

Відстеження шляхом виявлення або не використовуючи детектори.

Відстеження шляхом виявлення об’єктів:

Тип алгоритму відстеження, коли детектор об’єктів виявляє об’єкти в кадрах, а потім виконує об’єднання даних між кадрами для створення траєкторій, отже, відстеження об’єкта. Ці типи алгоритмів допомагають відстежувати кілька об’єктів і відстежувати нові об’єкти, введені в кадр. Найважливіше те, що вони допомагають відстежувати об’єкти, навіть якщо виявлення об’єкта не вдається.

Відстеження не використовуючи детектори:

Тип алгоритму відстеження, коли координати об’єкта ініціалізуються вручну, а потім об’єкт відстежується в наступних кадрах. Цей тип здебільшого використовується в традиційних алгоритмах комп’ютерного зору.

Варто ще зазначити, що перевагою моделей для відслідковування об’єктів є те, що вони можуть давати вірний результати навіть тоді коли детектор збивається і дає невірний результат (це стосується моделей відстеження шляхом виявлення об’єктів), так як у сучасних алгоритмах трекінгу використовуються різні прийоми згладжування, наприклад, фільтр Калмана.

1.3 Постановка задачі та її формалізація

На теоретичному етапі дослідження необхідно провести огляд сучасної літератури, наукових статей за темою «Виявлення та відслідковування об’єктів методами машинного навчання».

На практичному етапі спочатку вибираємо відео фрагменти для тренування та для тестування, на другому будемо заміряти якість побудованої моделі виявлення та відслідковування об’єктів. При цьому кожному кадру відповідає деяка мітка, яка містить координати обмежувальної рамки кожного з об’єктів, які є на відповідному кадрі, а також дана мітка містить назви категорій відповідних об’єктів. Ці мітки по суті являють собою кортеж різної довжини, однак якщо є така необхідність, то для зручності можна із якихось апріорних уявлень обрати максимальну кількість об’єктів, які можна зустріти на одному кадрі і таким чином вибирається максимально довжину кортежа, що дозволяє нам усі кортеж зробити в принципі однакової довжини, просто в останніх комірках буде присутня інформація про те, що дані об’єкти відсутні.

Далі використовуючи прийоми transfer learning вибираємо детектор, точніше завантажуємо вже натренований детектор. Вибираємо цей натренований детектор таким чином, щоб дані на яких він був натренований по своїй природі були схожі з тими даними, які зустрічаються на наших відео фрагментах, при необхідності дотреновуємо останні шари нашого детектора на даних, які по своїй природі схожі з кадрами взятих відео фрагментів. Важливо розуміти, що зображення, на яких тренується детектор не повинні співпадати з кадрами вибраних відео фрагментів, щоб по-перше, були досить точно встановлені параметри моделі відслідковування об’єктів, які встановлюються саме під час тренування, по-друге, кінцева оцінка якості моделі, яку ми отримуємо, була з маленькою дисперсією та зміщенням від реального показника якості цієї моделі.

В якості моделі, яка буде виконувати задачу відслідковування об’єктів, тобто надавати відповідні id кожному об’єкту на кожному кадрі таким чином

щоб однакові об’єкти мали однакові id, будемо використовувати алгоритм DeepSort. Цей алгоритми є розширенням звичайного алгоритму Sort, тому аналізуючи його при різних значеннях гіперпараметрів ми по суті так само досліджуємо і алгоритм Sort, якщо звичайний Sort є більше ефективний, то в ході аналітики ми це дізнаємось. Більш детально про метрики якості, які будуть вимірювати якісь зроблених висновків нашої побудованої модель виявлення та відсотковий об’єктів буде розказано в наступному розділі.

Тренування детектора буде здійснюватися шляхом в мінімізації деякого функціоналу - для задачі класифікації це може бути перехресна ентропія, для задачі регресії це може бути mse, для нашої задачі це буде по суті лінійна комбінація цих двох типів функцій втрат – регресійної та класифікаційної, про неї також більш детально буде розказано для кожної моделі детектора, якого будемо апробовувати, в наступному розділі.

З формалізованою точки зору у нас є відображення f: X->Y, область визначення X - це є множина відео фрагментів, тут можу сказати що це певна сім’я підмножина кадрів. Аналогічно область значень Y є сім’я підмножин міток. Це відображення працює таким чином, що кожному відео фрагменту ставиться у відповідність послідовність міток, які чітко описують об’єкти, які присутні на кожному кадрі. Якщо у нас є деякі відео фрагменти як описувались вище, то це означає, що нам доступні окремі значення цього відображення: $\{f(x_i) | \, i = 1, .., n\}$ $n -$ кількість відео фрагментів. На основі цих окремих значень ми будемо намагатися знайти такі ваги для нашої моделі детектора (тренуючи його як описувалося вище), знайти параметри для трекеру (теж тренуючи його) та підібрати такі гіперпараметри алгоритму DeepSort (та можливо ще й самого детектора), щоб кінцева натренована модель доволі непогано апроксимувала дане відображення на всій області визначення, його область визначення це не обов’язково є усі можливі підмножини кадрів - загалом тільки ті відео фрагменти, які нас безпосередньо цікавлять, ті явища за якими ми безпосередньо слідкуємо.

Дані для тренування або/і тестування детектора можна взяти з COCO датасету [38]. Дані для тестування трекеру візьмемо з MOT Challenge [39]. Дані для тренування нейромереж, які частиною виключно системи трекінгу візьмемо з Mars [40] або з Market1501 [41] датасетів. Більш детально про вибрані датасети викладено в розділі 3, який стосується практики.

Висновки до розділу 1

В цьому розділі був наведений приклад сфер людської діяльності в яких застосовуються моделі для виявлення та відслідковування об’єктів, починаючи від спортивної закінчуючи воєнною, таким чином було доведено актуальність даної роботи . Також було наведено класифікація моделей, які відповідають за покадрове відслідковування об’єктів, ця класифікація може виконуватись як з точки зору методів відстеження так і з точки зору кількості об’єктів, які слід відстежувати.

До того ж була вказана корисність цих технологій при навчанні з підкріпленням, тобто як вони можуть суттєво прискорити цю процедуру і загалом зробити і більш ефективно. В кінці цього розділу було чітко сформована та формалізована поставлена задача.

2 МАТЕМАТИЧНІ ОСНОВИ РОБОТИ

2.1 Нейромережі. Загальні відомості

Використовуючи джерело [1], можна сказати, що в найбільш загальному випадку нейромережа це є деякий ациклічний орієнтовний граф, де вершина позначає зміну, в загальному випадку тензор. Операції, які відбуваються над цими змінними, щоб з деякої групи змінних отримати іншу змінну, позначаються на ребрах графу. Такі графи називаються обчислювальними (див., наприклад, рис. 2.1 [1]). Їх приклади наведений нижче.

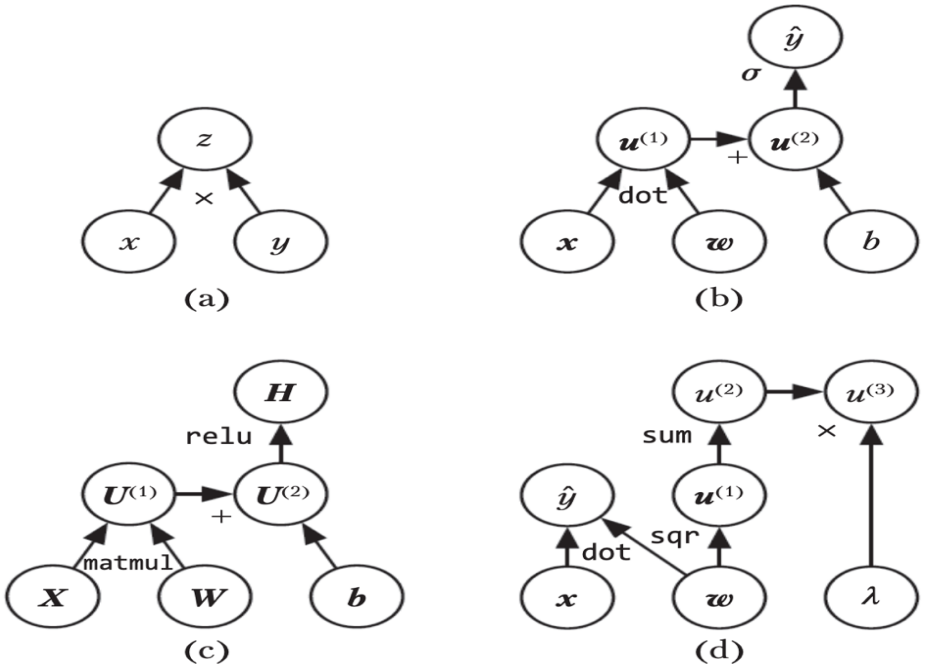


Рисунок 2.1

Особливе значення займають нейромережі прямого розповсюдження - це нейромережі, в яких початковий сигнал послідовно обробляється деякими функціями, які називаються шарами цієї нейромережі. Кількість цих функцій визначають довжину нейромережі прямого розповсюдження.

Таким чином будь-яка нейромережа – це функція $f(x, \omega)$, де x – вхідний сигнал, ω – тензор, який представляє параметри даної нейромережі.

Значення параметрів встановлюється шляхом навчання, тобто мінімізуючи деяку функцію втрат, який залежить тільки від параметрів (їх іноді ще називають вагами) нейромережі та тренувального набору даних, який нам відомий. Тренувальний набір – це така множина: $\{(x, y) | \, x -$ вхідний сигнал, $y -$ вихідний, $y = f(x)\}$, f – це є та функція, яку ми намагаємося апроксимувати за допомогою нейромережі.

2.2 Згорткові нейромережі. Загальні відомості

В теорії машинного навчання операція згортки вводиться таким чином:

$S(i, j) = (I * K)(i, j) = \sum_{m,n} I(i \times s + m, j \times s + n) K(m, n)$ (2.1) [1]

(В математиці її називають перехресного кореляцією). Отже в даному випадку операція згортки- це функція, яка діє на матрицю I використовуючи ядро K і на виході отримаємо матрицю S. Якщо замість I стоїть тривимірна матриця, то ядро K повинно мати такий самий розмір по третьому виміру як і I, тоді на виході отримуємо теж двовимірну матрицю S, яка визначається формулою вище, тільки сумування вже буде по всім трьом вимірам. Параметрами для цієї операції, які встановлюються під час навчання є елементи матриці (ядра) K, s – гіперпараметр, який встановлюється до навчання на етапі побудови згорткової нейромережі. Якщо матриця I має розмір k на l, а ядро K - k1 на l1, то вихід згортки буде мати вихід розміром [(k-k1)/s]+1 на [(l-l1)/s]+1. Іноді до матриці I дописують нулі по краям, щоб вихід згортки мав такий самий розмір як і вхід.

Операція pooling. Дана операція використовується для того щоб зменшити розмір зображення (тензора – в нашому випадку він буде завжди тривимірний), яке обробляється, зібравши з нього головну і при цьому

очистивши його від зайвих шумів. Ця операція не має параметрів, які піддаються навчанню, тому при навчанні воно дає деякі переваги стосовно швидкості навчання. Пулінг задається формулою (K: R^{m*n}->R)

$$S(i,j) = (I * K)(i,j) = K(I(i \times s : i \times s + m, j \times s : j \times s + n)) \quad (2.2)$$

[1]

Усі інші твердження стосовно параметра s, розмірів вхідного і вихідного тензорів такі самі як і для операції згортки. Найбільш розповсюдженні види пулінгу – це max pooling (в якості К береться функція максимуму по всім елементам) та avarage pooling (в якості К береться функція середнього по всім елементам).

Згортковий шар. Згортковий шар складається з таких елементів:

По-перше, на етапі побудови моделі задається кількість операцій згортки, які будуть виконуватися в даному шарі. Кожна з цих операцій згортки застосовується до відповідного вхідного тензора(зазвичай тривимірного), після кожної з тих операцій отримуємо вже двовимірні тензори, які при об’єднанні по останньому виміру утворюють тривимірний тензор.

По-друге, застосовується функція активації до кожної координати у вихідному тензорі, перед цим до цього вихідного тензора додається вектор зміщення (точніше кажучи до кожного двовимірного тензора по першим двом вимірам додається скаляр, ці скаляри і складають вектор зміщення), цей вектор так само є параметром, який встановлюється при навчанні. Функція активації частіше за все повинно бути нелінійною щоб з згорткова нейромережа могла імітувати складні нелінійні функції.

По-третє, зазвичай після цього іде операція пулінгу, яка вже застосовується до кожного двовимірного тензора окремо, двовимірний тензор виділяється по першим двом вимірам, таким чином в результаті отримаємо тривимірну матрицю, у якої розмір по 3-му виміру зберігається, а по першим двом змінюється за рахунок операції пулінгу.

Щільний шар.

На вхід до щільного шару подається деякий вектор, на виході

отримуємо теж вектор, але вже іншої довжини. Це один з найбільш перенавантажених з точки зору кількості параметрів шарів з усіх більш-менш відомих в нейромережах. Тому кількість щільних шарів в нейромережах зазвичай доволі мала порівняно з загальною глибиною нейромережі. Отже загальна формула для обчислення виходу щільного шару така: g(z), z=Wx+b; W,b – параметри, які встановлюються при навчанні, g – деяка функція активації, x - вхідний вектор.

Отже згорткова нейромережа - це така нейромережа, яка в своїх шарах використовує операцію згортки. На практиці ці нейромережі на початку використовують згорткові шари , потім перетворення до одновимірного тензора (тобто звичайного вектора) та декілька щільних шарів (кінцева функція активації залежить від апроксимуючої функції).

Одна з дуже важливих властивостей згорткової нейромережі - це її інваріантність відносно паралельних переміщення. Тобто якщо мережа навчилися розпізнавати деякі об’єкти в конкретних місцях на зображенні, то при паралельних приміщеннях цього об’єкта зображення воно так само буде їх правильно розпізнавати точніше це можна стверджувати з доволі високою імовірністю, оскільки за невдалої функції втрат, неадекватної побудови цієї нейромережі або доволі не репрезентативного тренувального набору даних цього звичайно виконуватись не буде. Отже, такі нейромережі можуть навчитися розпізнавати образи в деяких місцях на зображені навіть якщо не було відповідних тренувальних даних, саме головне, щоб просто було достатньо даних, де цей об’єкт в принципі зустрічається.

2.3 Функції активації

Про необхідність застосування функцій активації вже говорилося у минулому пункті (це є функції, які застосовуються до деякого вхідного

тензору,). Найбільш поширеними серед них є relu (див. рис. 2.2 [2]), leaky relu (див. рис. 2.3 [2]), гіперболічний тангенс (див. рис. 2.4 [2]), softmax, сігмоїда (див. рис. 2.4 [2]), silu(див. рис. 2.5 [2]), mish (див. рис. 2.6 [2]).

Relu. f: R->R; f(x) = max(0,x); x – скаляр. (2.3) [2]

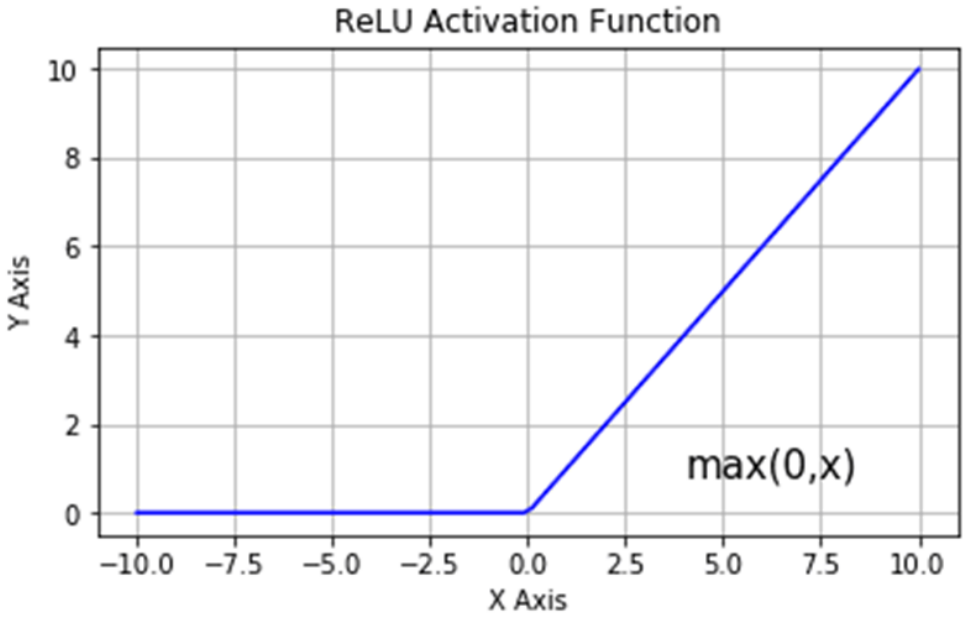


Рисунок 2.2

Leaky relu. f: R->R; f(x) = max(0.1*x, x); x - скаляр. (2.4) [2]

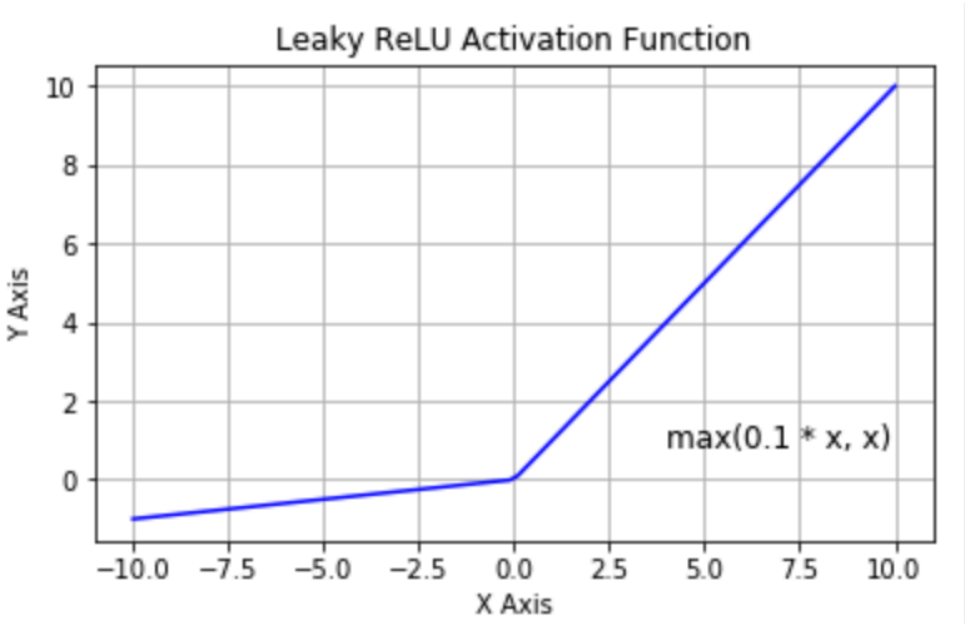


Рисунок 2.3

Softmax. f: R^n->R^n; $(f(x))_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$, x – вектор. (2.5) [1]

Її перевагою є те, що сума виходу по всім координатам дорівнює 1, таким чином softmax часто використовують для апроксимації ймовірності за умови певного x.

Сігмоїда. f: R->R; f(x) = $\frac{1}{1+\exp(-x)}$; x – скаляр. (2.6) [2]

Її часто використовують для апроксимації ймовірності за умови певного x для бінарної класифікації.

Гіперболічний тангенс. f: R->R; f(x) = $\frac{e^x-e^{-x}}{e^x+e^{-x}}$, x – скаляр.

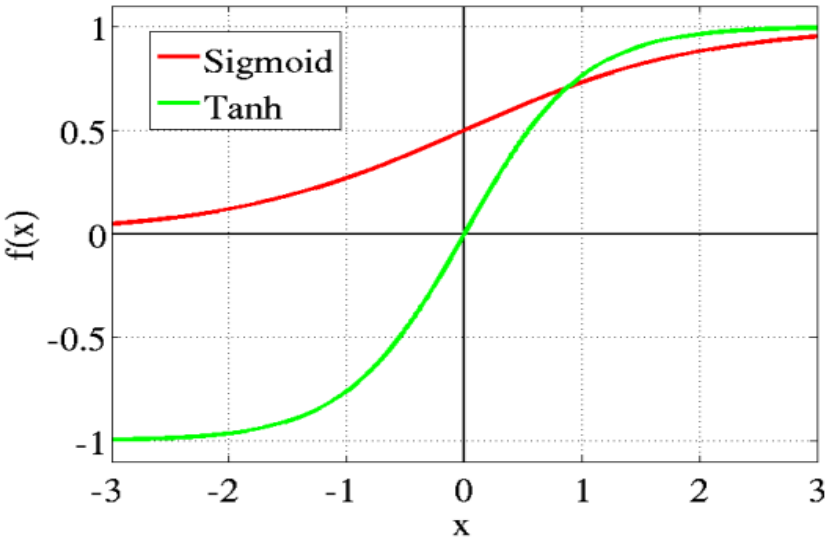


Рисунок 2.4

Silu. f: R->R; f(x) = x*sigmoid(x); x – скаляр. (2.7) [2]

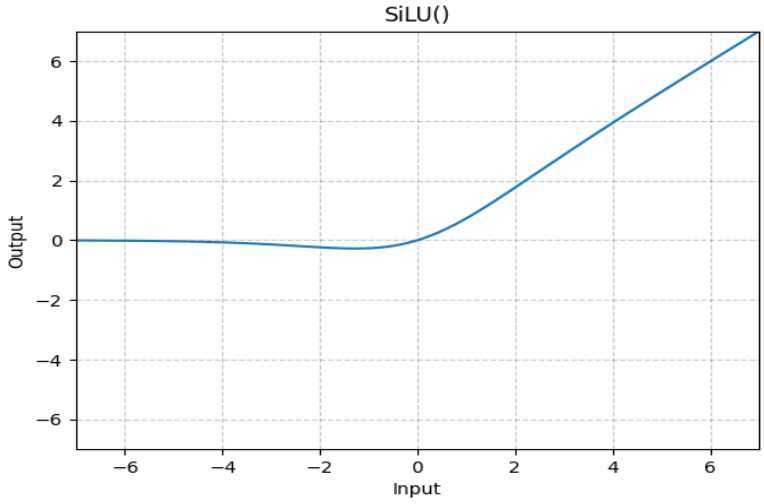


Рисунок 2.5

Mish. f: R->R; f(x) = x*tanh(ln(1+exp(x))); x – скаляр. (2.8) [2]

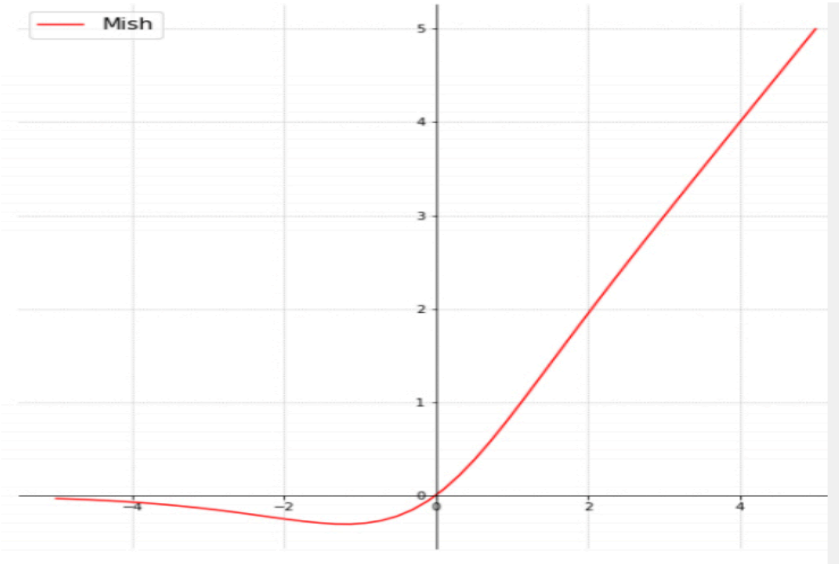


Рисунок 2.6

Спочатку широко використовувалися сігмоїда та гіперболічний тангенс, однак похідна від них з надто високою ймовірністю є близькою до нуля – таким чином ваги при деяких нейронів взагалі не покращуються (прояв проблеми зникнення градієнтів).

Relu дає значно кращі результати, однак через те що вона дорівнює 0 при x<0 – вона частково має ту ж саму проблему, до того ж вона ніколи не може бути менше нуля, що в свою чергу іноді обмежує нас в пошуках гарної апроксимації для конкретної функції. Її негладкість теж іноді є незначною проблемою.

Для вирішення усіх перелічених проблем були введені функції Silu та Mish. За своєю поведінкою вони схожі на Relu (тому як мінімум здатні давати не гірші результати), однак є гладкими, здатні приймати негативні значення і загалом за результатами роботи [2] показують кращі результати для різних архітектур нейромереж. Також згідно цієї роботи Mish показує трохи кращі результати ніж Silu і до того ж є більш стійкою до шуму.

2.4 Функції втрат

Функції втрат – це такі функції, які залежать від реальних даних і даних виданих моделлю (нейромережею). Якщо похибка між реальними даними та даними моделі прямує до нуля, то сама функція втрати теж прямує до нуля. Мінімізуючи функцію втрат – отримуємо параметри (ваги), за яких модель повинна в теорії добре апроксимувати шукану функціональну залежність. Розглянемо деякі основні приклади функції втрати для задач класифікації та регресії (далі будуть наведені функції втрат для одного екземпляра даних – щоб отримати для цілого набору даних потрібно додатково просумувати по всім екземплярам в наборі).

Задача багатокласової класифікації. Для цієї задачі вводиться перехресна ентропія:

$L(y_{real}, y_{model}) = -\sum_i y_i ln((y_{model})_i)$; y_{real} – це вектор, де усі елементи нулі, окрім єдиного, який дорівнює 1 – показує до якого класу належить об’єкт; y_{model} – це вектор такої ж розмірності як і y_{real} , і-та координата дорівнює оцінки ймовірності належності до і-того класу. (2.9) [1]

В [1] наведена теорема про те, що якщо модель точно апроксимує реальний розподіл при єдиному наборі значень параметрів, то оцінка, яку отримаємо шляхом мінімізації перехресної ентропії є консистентною. Отже, ця теорема в принципі аргументує вибір цієї функції як функції втрат.

Задача бінарна класифікації. Більш зручний спосіб запису перехресної ентропії для бінарного випадку:

$L(y_{real}, y_{model}) = -y_{real} ln(y_{model}) - (1 - y_{real} ln(1 - y_{model}))$; в даному випадку y_{real} – скаляр (1 та 0 відповідають за два різні класи), y_{model} – оцінка ймовірності належності до класу, який відповідає y=1. (2.10) [1]

Важливою умовою вищеописаних задач є те що кожен об’єкт може належати тільки до одного класу.

Задача регресії.

Середньо квадратичне відхилення (MSE):

$L(y_{real}, y_{model}) = (y_{real} - y_{model})^2$, y_{real} – скаляр (реальне значення), y_{model} – скаляр (регресійна оцінка моделі для y_{real}). (2.11) [1]

Її використовують коли великим відхиленням надають більшого значення, тобто при її мінімізації модель буде мінімізувати великі похибки значно частіше та сильніше ніж маленькі.

Відхилення по модулю:

$L(y_{real}, y_{model}) = |y_{real} - y_{model}|$, y_{real} – скаляр (реальне значення), y_{model} – скаляр (регресійна оцінка моделі для y_{real}). (2.12) [1]

В цьому випадку не надається переваги до жодної з похибок – ні великим, ні маленьким.

Функція втрат Губера: $L(y_{real}, y_{model}) = L_\delta(y_{real} - y_{model})$, де

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta \cdot \left(|a| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases}$$

(2.13) [1]

MSE використовують доволі часто, однак в ній є недолік, який полягає у тому що вона занадто чутлива до викидів, функція втрат Губера вирішує цю проблему роблячи її лінійною для досить великих похибок.

Для нашої задачі регресійна частина виводу буде мати чотири координати для однієї обмежувальної рамки, тому якщо використовувати вищевказані функції втрат, то потрібно зробити усереднення по цим чотирьом координатам. Однак, тоді їх значення будуть збільшуватись пропорційно того як збільшуються площі обмежувальних рамок, в результаті для великих обмежувальних рамок алгоритм мінімізації буде працювати краще для маленьких, тобто нам потрібні функції втрат інваріантні відносно масштабу площ обмежувальних рамок.

Такі функції втрат можна отримати з використанням операції IoU (Intersection over union). Нехай A і B – дві обмежувальні рамки на певному зображенні, тоді IoU(A, B) = (площа перетину A та B)/(площа об’єднання A та B). Функції втрат мають такий вигляд:

1) 1- IoU(A, B) (2.14) [3]

A – апроксимація для справжньої обмежувальної рамки B.

2) $$DIoU = 1 - \text{IoU} + \frac{d^2}{c^2}$$ (2.15) [3]

d – відстань між центрами обмежувальних рамок A і B, c – максимальна відстань між кінцями двох обмежувальних рамок A і B. Дає кращі результати якщо початкове апроксимація ініціалізується далеко від реальної (з точки зору відстані між центрами).(робота [3])

3)CIoU (Complete IoU loss):

$$loss_3 = 1 - IoU\,(A, B) + \frac{\rho^2\,(b, b^{gt})}{c^2} + \alpha v$$
$$v = \frac{4}{\pi^2}\left(\arctan\frac{w^{gt}}{h^{gt}} - \arctan\frac{w}{h}\right)^2$$
$$\alpha = \frac{v}{(1 - IoU) + v}$$

(2.16) [3]

Усе означає теж саме, що і в попередньому випадку, окрім останнього члену – там ще окремо явно враховано, що відношення ширини до довжини апроксимуючої обмежувальної рамки при мінімізації має прямувати до відповідного відношення справжньої обмежувальної рамки – в роботі [3] показано, що такі нововведення прискорює алгоритм навчання.

2.5 Метрики якості

Метрики якості детекторів. Розглянемо для початку бінарну класифікацію. Для детекторів вводяться граничні значення для ймовірностей, кожну з яких вони оцінюють (Тобто якщо відповідна імовірність більше свого граничного значення то вважаємо що об’єкт належить до відповідного класу, у противному випадку - не належить), також вводиться граничне значення для регресійної складової IoU_min: A - обмежувальна рамка як вихід моделі, B -

справжня обмежувальна рамка; якщо IoU(A, B)>= IoU_min та клас відповідного об‘єкта в рамці визначений правильно, то приймаємо що в цьому випадку класифікація зроблена правильно (*). Таким чином можемо використовувати лише метрики, які оцінюють якість класифікації і при цьому з урахуванням (*) буде враховуватись і якість регресивної складової нашої моделі.

Введемо такі позначення для бінарної класифікації:

TP (True positives) – кількість позитивних екземплярів (y=1), які визначені детектором правильно;

TN (True negatives) - кількість негативних екземплярів (y=0), які визначені детектором правильно;

FP (False positives) - кількість негативних екземплярів, які визначені детектором неправильно;

FN (False negatives) - кількість позитивних екземплярів, які визначені детектором неправильно;

Опишемо формулами типові метрики якості:

Правильність:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (2.17)\ [1]$$

Точність (Precision), повнота (Recall), F-міра (F1 score):

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (2.18)\ [1]$$

F-міра буде достатньо великою (близькою до 1) тільки для моделей у яких точність та повнота теж близькі до 1.

AP (Average precision) – середня точність. Змінюючи граничне

значення для ймовірності (від 0 до 1) – отримуємо графік залежності точності від повноти (крива p(r)). За означенням AP = площа фігури під цим графіком. Однак прийнято згладжувати криву p(r) таким чином: $p_{interp}(r) = \max_{\tilde{r} \geq r} p(\tilde{r})$. Тоді AP рахується усередненням по деякій кількості точок, однаково розташованих одна від одної, або підрахунком реальної площі (перший варіант використовується наприклад в COCO dataset зі 101 точкою). AP_{75} означає IoU_min=0.75 для даного детектора. $AP_{50:95}$ означає підрахунок AP для кожного IoU_min від 0.50 до 0.95 з кроком зазвичай 0.05 та подальше усереднення цих показників. Метрику AP іноді називають mAP – mean AP, оскільки для багатокласової класифікації ми рахуємо цю метрику для кожного класу окремо та беремо середнє.

Метрики якості трекерів. До метрик якості трекерів (моделі для відслідковування) відносять такі: MOTA, MOTP, MT, ML, ID, FM. В контексті трекерів FN – це кількість екземплярів, які модель не змогла виявити; FP – це кількість областей, які модель виявила, однак там не міститься шуканих об’єктів.

MOTA (правильність відслідковування за багатьма об’єктами) – максимум = 1 – тільки тоді коли не було неправильних перенесень id з деякого кадру в наступний, а також не було помилок при роботі детектора на кожному кадрі. IDS_t - кількість неправильних перенесень id з t-1 - го кадру в наступний.

$$MOTA = 1 - \frac{\sum_t FN_t + FP_t + IDS_t}{\sum_t GT_t} \quad (2.19)\ [31]$$

MOTP (точність відслідковування за багатьма об’єктами) – враховує лише якість регресійної складової моделі. c_t – кількість реальних об’єктів, які були помічені детектором на t-ому кадрі, d^i_t – відстань між i-тою обмежувальною рамкою та її апроксимацією детктором (за відстань беремо 1-IoU).

$$\text{MOTP} = \frac{\sum_{i,t} d_t^i}{\sum_t c_t}.$$

(2.20) [31]

MT – відсоток тих траєкторій об’єктів, слід яких не втрачався більш ніж на 80% кадрів, на яких був присутній даний об’єкт.

ML - відсоток тих траєкторій об’єктів, слід яких не втрачався не більше ніж на 20% кадрів, на яких був присутній даний об’єкт.

ID (або IDSW) – кількість разів коли змінювався id одного і того самого об’єкта.

FM – кількість разів коли втрачався слід через те що детектор не зміг знайти об’єкт.

Метрика MOTA є найбільш репрезентативною з точки зору якості, так як висока оцінка за цією метрикою можлива лише тоді коли немає проблем ні з детектором ні з процесом відслідковування. Однак якщо вона низька, то не зрозуміло в чому саме причина і тоді можливо доцільно буде звернутися додатково до інших метрик згаданих вище.

2.6 Алгоритми навчання

Для вирішення задач мінімізації частіше за все використовують метод градієнтного спуску. Його суть полягає у наступній формулі (правило оновлення):

$w := w - \eta \nabla J(w)$, J – деяка функція втрат, w – тренувальні параметри (тензор), η – додатній скаляр, який позначає швидкість навчання. (2.21) [1]

η вибирається достатньо малим - існує теорема, яка каже про те що для опуклої задачі мінімізації цей алгоритм збігається до глобального мінімуму, якщо η вибрано спеціальним чином достатньо малим.

Однак у цього алгоритму є ряд проблем. По-перше, зазвичай ми маємо справу з дуже великими навчальному наборами даних і тому підрахунок такої

суми градієнтів є надзвичайно виснажливим з точки зору кількості операцій для комп'ютера. По-друге, навіть за маленькою швидкості навчання ми будемо робити дуже великі кроки і навіть якщо знаходимось поблизу до деякої оптимальної точки то будемо часто проскакувати і таким чином швидкість збіжності може бути дуже малою. Таким чином робляться деякі модифікації методу градієнтного спуску.

Стохастичний метод градієнтного спуску. Випадковим чином розіб'ємо наш тренувальний набір даних на менші тренувальні набори даних фіксованого розміру (пакети). Використовуємо формулу для градієнтного спуску в якості оновлення тренувальних параметрів (тепер суму беремо тільки по даному пакету) і таким чином проходимось по всьому навчальному набору даних. Так завершилась 1-а епоха тренування, Аналогічно проводиться 2-а, 3-я і так далі до певного заданого обмеження. Важливим параметром в даному алгоритми є розмір пакетів, на які ми розбиваємо наш тренування набір. Він повинен бути досить маленьким, щоб операції підрахунку градієнтів не було виснажливими, але якщо він буде занадто маленьким, то маленькі кроки, які ми будемо робити при цьому алгоритмі можуть бути занадто незначними, що провокує низьку швидкість збіжності.

Усі інші модифікації методу градієнтного спуску включають ту ж саму стохастичну стратегію, однак відрізняються вже самим методом вибору параметру швидкості навчання. Ці модифікації зазвичай краще тим, що в них швидкість навчання вже не відіграє такої надто важливої ролі, тобто навіть за наявності такого параметра як швидкість навчання вони є вже більш автоматичними. Опишемо деякі з них.

Nesterov Accelerated Gradient.

Даний метод описується такими формулами:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t \qquad (2.22)\ [1]$$

Суть його в тому що по суті ми враховуємо історію градієнтів і навіть

частково заглядаємо наперед, тобто якщо деякий проміжок часу ми рухаємося у правильному напрямку (градієнти набувають великого значення) Це означає що ми або повинні рухатись в тому ж напрямку або як мінімум не сильно відхилятися від цього напрямку. Чим більше параметр γ тим більше враховується історія градієнтів і тим менше буде відхилятися від гарних напрямків, які були незадовго у минулих кроках.

Adagrad.

Даний Алгоритм описується такими формулами :

$$g_t \equiv \nabla_\theta J(\theta_t)$$

$$G_t = G_t + g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

(2.23) [1]

В даному випадку якщо ми занадто часто оновлюємо деякі ваги то згодом ці ваги будуть дуже повільно змінюватись і, навпаки, сильніше стануть змінюватись ті тренувальні параметри, які рідко змінювалась до цього. Таким чином частково вирішується проблема нерівномірного розподілу даних (коли деяких образів в тренувальних зображеннях дуже мало для рівномірного навчання).

RMSProp.

Цей алгоритм описується такими формулами:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

(2.24) [1]

Недоліком минулого алгоритму є те, що у знаменнику згодом накопичується надто велика сума, що призведе до незначних змін вагів і таким чином алгоритм по суті може зупинитися ще до того як досяг оптимальних значень. Для цього використаємо прийом ковзного середнього і тоді цей алгоритм описуватиметься формулами вище.

Adam. Недоліком минулого алгоритму є те що в залежності від початкових значень ковзні середні можуть занадто довго накопичувати градієнти щоб працювати по такому принципу як це задумалось авторами, тобто сильніше оновлювати ті ваги, які до цього оновлювали занадто рідко. Цю проблему дещо вирішує алгоритм Adam використовуючи такі формули [1]:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.25}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t},\ \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

2.7 Регуляризація нейромереж

Регуляризація - це методи, які дозволяють знизити ефект перенавчання моделі, тобто коли оцінка якості моделі на тренувальному наборі даних значно перевищує оцінку якості моделі на тестовому наборі даних.

L1 регуляризація. Цей метод полягає у тому що до функції втрат додається L1 норма від тренувальних параметрів нейромережі помноженої на деякий додатній коефіцієнт (гіперпараметр): $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$, . Таким чином ті ваги, які підлаштовуються під шуми, будуть прямувати до 0 тобто модель буде по суті спрощуватись – таким чином не використовуючи зайві

структури.

L2 регуляризація. Вводиться аналогічно до L1 регуляризації, однак замість норми L1 використовується норма L2 в квадраті: $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = (\alpha/2)\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$

Може також використовуватись суперпозиція цих 2 методів регулярзації. Такі методи зазвичай використовуються коли моделі є занадто складними та ми не знаємо, які саме структури в цій моделі потрібно прибрати, щоб краще апроксимувалися цільові функціональні залежності. Звичайно ці два метода мають дещо різний вплив на процес мінімізації, у випадку L1 ваги будуть більш активно рухатись до нуля. Однак в обох цих методах між вагами, які несуть зайву ускладненість і тими які не несуть, буде утворюватися деякий розрив (перші будуть менш значущими).

Dropout Block. Перенавчання також часто з’являється тоді коли у моделі велика дисперсія, тобто в залежності від тому на якому саме наборі даних буде відбуватися тренування модель буде дуже сильно змінювати свій вихід. Для боротьби з цим можна використовувати техніку ансамблювання, тобто деяка комбінація декількох більш простих моделей. Для нейромереж дуже популярним в якості ансамблювання може бути використання шарів Dropout. Цей шар занулює координати вхідного тензора незалежним чином, ймовірність занулення кожної координати однакова і задається як гіперпараметр. В результаті в модель таким чином вводится випадковість, під виходом цієї нейромережі розуміється математичне сподівання (усереднення) від випадкового виходу, який формується як було показано вище. Це один з найбільш ефективних методів боротьби з перенавчанням в нейромережах, оскільки є ефективні технології імплементації такого інструменту на практиці, наприклад в пакеті Tensorflow (описаний детальніше наприклад в [1]).

2.8 Шари нормалізації

Розподіл виходів деяких шарів нейромережі можуть надто сильно змінюватись під час тренування. Як показано в роботі [4], це дає негативний вплив на тренування і щоб його нівелювати використовують шари нормалізації, які по суті нормалізують виходи деяких шарів нейромережі. В цій же роботі показано, що шари нормалізації можуть виконувати роль регуляризації (замість блоків Dropout).

Шари нормалізації також здатні вирішувати інші проблеми в нейромережах: такі як наприклад - зникаючі градієнти. Суть цієї проблеми в наступному: дуже часто похідні функції активації, які ми використовуємо, лежать в [0;1), коли ми рахуємо похідну композиції таких функцій то, використовуючи ланцюгове правило, навіть якщо швидкість навчання встановлена доволі високою самі градієнти набувають дуже маленького значення. Таким чином фактично процес навчання може припинитися зарано для деяких початкових шарів (Так як процес навчання іде у зворотному напрямку - від останніх шарів то перших) - про це сказано в роботі [5] .

Нехай x - це вихід деякого шару нейронної мережі (вважаємо, що вихід скалярний, якщо це не так, то просто проробляється нижчеописана процедура для кожної координати окремо). Тоді робота шару нормалізації під час тренування, який йде одразу після x, описується такими формулами:

$$
\begin{aligned}
\mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i && \text{// mini-batch mean} \\
\sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_{\mathcal{B}})^2 && \text{// mini-batch variance} \\
\widehat{x}_i &\leftarrow \frac{x_i-\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} && \text{// normalize} \\
y_i &\leftarrow \gamma\widehat{x}_i+\beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) && \text{// scale and shift}
\end{aligned}
$$

(2.26) [5]

Параметри γ і β є тренованими, тобто визначаються в процесі тренування. При початковий ініціалізації часто беруть γ близька до 1, а β - до 0. Після закінчення тренування оцінюється середні та дисперсія виходу того шару нейронної мережі після якого йде шар нормалізації, причому оцінка відбувається за усім тренувальним набором. Тоді вихід шару нормалізації буде описуватись вже такою формулою (де β та γ - це ті параметри, які ми отримали в результаті тренування): $\frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}}\cdot x+\left(\beta-\frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$ (2.27) [5]

Це був загальний алгоритм за яким реалізується шар нормалізації, його називають пакетною нормалізацією (batch normalisation). Коротко опишемо деякі 2 його модифікації.

Крос-ітераційна пакетна нормалізація (CBN). Все так само як і в минулому алгоритмі, однак середнє та дисперсію пропонується оцінювати (під час тренування) використовуючи додатково декілька останніх міні-пакетів з тренувального набору, які ми вже пройшли, за допомогою таких формул ($x_{t,i}$ – вихід деякого шару для і-того екземпляра t-го міні-пакету; $x^l{}_{t,i}$ – теж саме, тільки конкретно вказується l-ий шар):

$$\mu_t(\theta_t) = \frac{1}{m}\sum_{i=1}^{m} x_{t,i}(\theta_t),$$
$$\sigma_t(\theta_t) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_{t,i}(\theta_t) - \mu_t(\theta_t))^2}$$
$$= \sqrt{\nu_t(\theta_t) - \mu_t(\theta_t)^2}, \qquad (2.28)\ [6]$$

$$\nu_t(\theta_t) = \tfrac{1}{m}\textstyle\sum_{i=1}^{m} x_{t,i}(\theta_t)^2, \qquad (2.29)\ [6]$$

$$\mu^l_{t-\tau}(\theta_t) \approx \mu^l_{t-\tau}(\theta_{t-\tau}) + \frac{\partial \mu^l_{t-\tau}(\theta_{t-\tau})}{\partial \theta^l_{t-\tau}}(\theta^l_t - \theta^l_{t-\tau}),$$
$$\nu^l_{t-\tau}(\theta_t) \approx \nu^l_{t-\tau}(\theta_{t-\tau}) + \frac{\partial \nu^l_{t-\tau}(\theta_{t-\tau})}{\partial \theta^l_{t-\tau}}(\theta^l_t - \theta^l_{t-\tau}). \qquad (2.30)\ [6]$$

$$\bar{\mu}^l_{t,k}(\theta_t) = \frac{1}{k}\sum_{\tau=0}^{k-1} \mu^l_{t-\tau}(\theta_t),$$
$$\bar{\nu}^l_{t,k}(\theta_t) = \frac{1}{k}\sum_{\tau=0}^{k-1} \max\left[\nu^l_{t-\tau}(\theta_t), \mu^l_{t-\tau}(\theta_t)^2\right],$$

$$\bar{\sigma}^l_{t,k}(\theta_t) = \sqrt{\bar{\nu}^l_{t,k}(\theta_t) - \bar{\mu}^l_{t,k}(\theta_t)^2}, \qquad (2.31)\ [6]$$

В роботі [6] показано, що такий підхід вирішує проблему минулого алгоритму, якщо розмір міні-пакетів є занадто малим (в такому разі цей один міні-пакет занадто не репрезентативний для оцінки середнього та дисперсії). Також в цій роботі показано як можна ефективно виконати обчислення часткових похідних, тобто щоб складність обчислень була $O(C_{out} \times C_{in} \times K)$, де C_{out}, C_{in} та К - це вихідна кількість каналів(розмір останнього виміру у тривимірному тензорі), вхідна кількість каналів та кількість каналів ядра. Формули для статистик на минулих міні-пакетах були отримані з використанням рядів Тейлора, а також з припущення, що похідні на минулих

шарах не будуть вносити сильний вплив, що було емпірично доведено в роботі [6].

Перехресна міні-пакетна нормалізація (CmBN). Такий самий алгоритм як і крос-ітераційна пакетна нормалізація, однак всі оцінки середнього та дисперсії виконується в межах певного пакету, який в свою чергу включає деяку фіксовану кількість міні-пакетів. Цей шар був вперше введений в роботі [21].

2.9 Skip connections

В глибоких нейромережах окрім проблем з перенавчанням та проблеми зі зникаючими градієнтами є також проблема деградації точності, як на тренувальній так і на тестовій виборці, зі збільшенням глибини мережі, причому причиною цьому не є ні зникаючі градієнти ні очевидно перенавчання. Ця проблема пов’язана з поняттям симетрії простору параметрів нейронної мережі. Нехай для деякого лінійного відображення T виконується : $f(x,w)=f(x,T(w))$ для будь-яких вхідних сигналів x та тренувальних параметрів w. Таким чином кожному такому лінійному відображення буде відповідати гіперплощина ($T(w)=w$), в якій будуть лежати деякий локальні екстремуми функції втрат, в загальному випадку далекі від глобальних. Якщо серед цих локальних екстремумів буде локальні мінімуми, то майже напевно використовуючи градієнтні методи ми прийдемо до одного з цих локальних мінімумів за умови що якимось чином опинимось в цій гіперплощині. Більш детально це описано в роботах [7, 8] .

Щоб боротися з цією проблемою нейромережу потрібно переробити таким чином щоб з одного боку вона могла так само імітувати складні нелінійності залишаючись такою ж глибокою, а з іншого боку була так би мовити менш симетричною. Для цього і вводяться зв’язки типу skip

connections, які також допомагають боротися з проблемою зникаючих градієнтів. До того ж за допомогою skip connections можна легше навчити функцію ідентичності (f(x)=x) і це дає підстави вважати що отримана таким чином глибша нейромережа як мінімум буде не гірше попередньої (робота [5]).

Residual block. Цей блок Реалізує skip connections у такий спосіб, описаний в роботах [5, 9] (див. рис. 2.7 [9]):

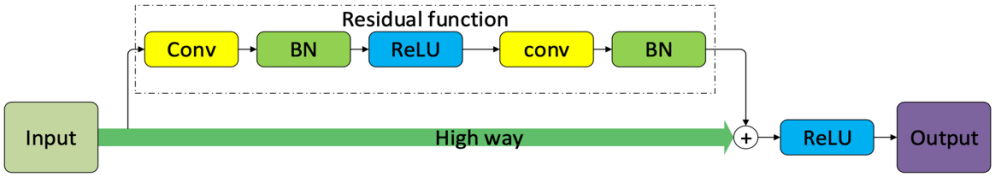


Рисунок 2.7

У нього є проблема пов'язана з особливістю функції Relu - магістральний сигнал (high way) з набагато більшою імовірністю буде підсилюватися ніж навпаки. Це може значно уповільнити пошук шуканої нелінійності. Для боротьби з цим водиться наступний тип skip connections.

Weighted residual block. Цей блок Реалізує skip connections у такий спосіб, описаний в роботі [9] (див. рис. 2.8 [9]):

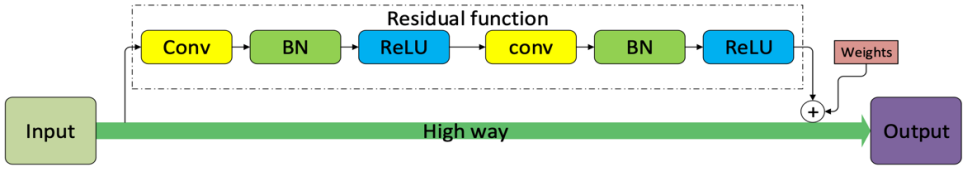


Рисунок 2.8

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i \Delta L_i(\mathbf{x}_i, \theta_i), \lambda_i \in (-1, 1), \quad (2.32)\ [9]$$

Щоб тренувальний параметр λ_i залишався у відповідних межах його ініціалізують нулем з дуже маленькою швидкістю навчання.

Dense residual block. Цей блок Реалізує skip connections у такий спосіб, описаний в роботі [10] (див. рис. 2.9 [10]):

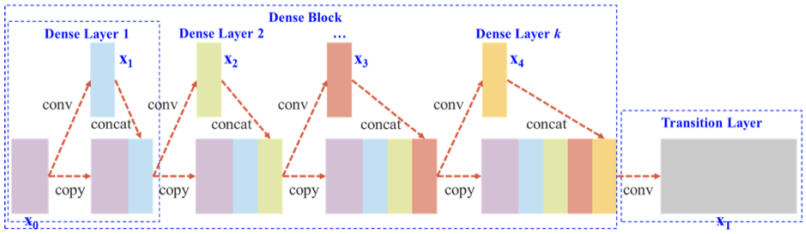


Рисунок 2.9

Цей блок є доволі ефективним з точки зору показників якості, однак не з точки зору витрачених ресурсів. Для знаходження балансу між якістю та використаними ресурсами вводиться наступний блок.

CSP residual block. Цей блок Реалізує skip connections у такий спосіб, описаний в роботі [10] (див. рис. 2.10 та 2.11 [10]):

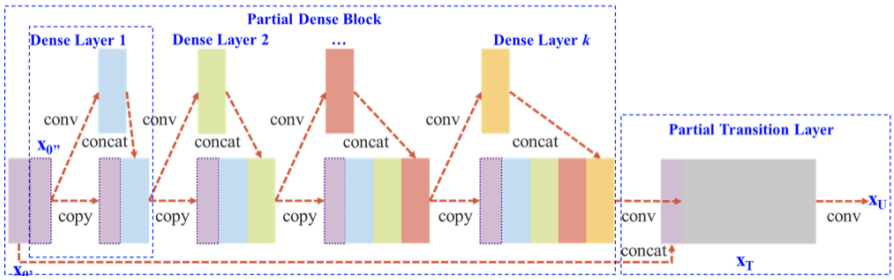


Рисунок 2.10

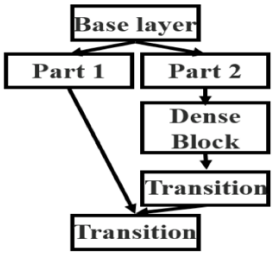


Рисунок 2.10

Спочатку іде розділення вхідного тензору по канальному виміру - з першою частиною відбуваються ті ж самі операцію що її в Dense residual block, а друга В кінці об’єднується з 1-ю по канальному виміру (зазвичай це третій - останній вимір) - стратегія CSPNet.

2.10 Сімейство моделей R-CNN

2.10.1 Вибірковий пошук (Selective search)

Більшість моделей із сімейства R-CNN беруть на вхід не тільки саме зображення а й запропоновані області, які сумарно мають охоплювати усі об’єкти на цьому зображенні. Для того щоб згенерувати такі області здебільшого використовують алгоритм вибіркового пошуку (точніше його ієрархічну версію запропоновану в роботі [11]).

Алгоритм використовує так звану міру схожості між 2 областями на зображенні. Вона змінюється від 0 до 1, 1 – якщо області ідентичні за деяким критерієм, 0 -якщо області різні за цим критерієм. Цей алгоритм працює таким чином:

1. Формуються початкові області.
2. Поєднуються ті області, які мають найбільшу міру схожості.
3. Таким чином утворюється новий рівень ієрархії де 2 області з минулого пункту замінюється 1 областю, в яку об’єдналися.
4. Повторюється крок 2 та 3 поки не залишиться рівень ієрархії з 1 елементом - усім зображенням.

Далі будемо вважати що 1-м рівнем ієрархії є те, що ми отримали на останньому кроці алгоритму (все зображення), другим - на передостанньому і так далі.

На кроці 1 ми використовуємо алгоритм з роботи [12] для генерування початкових областей. В [12] використовується графова модель сегментації, об’єднуються сусідні пікселі, які є найбільш схожими за значенням кольору (мається на увазі просто числове значення відповідного елемента матриці при використання деякої фіксованої колірної моделі), далі деякі групи пікселів можливо ще об’єднуються в інші групи пікселів якщо є достатньо схожими по своїй кольоровій гамі і не вичерпано кількість кроків даного алгоритму - алгоритм здійснює максимум m об’єднань, де m це кількість ребр, тобто кількість сусідніх пар пікселів. В даній моделі 2 області є достатньо схожими

для того щоб об’єднатися якщо мінімальний модуль різниці значень двох пікселів (перший з першої області, другий - з другої) <= $\min(Int(C_1)+\tau(C_1),\ Int(C_2)+\tau(C_2))$. C_1, C_2 – перша та друга область відповідно, Int(C) = максимум з модуля різниць значень двох сусідніх пікселів (обидва пікселі з області C), $\tau(C) = (k/|C|)$, k – параметр даного алгоритму.

Існують 4 різновиди мір схожості:

1. Міра схожості за кольором: Для кожного з 3 каналів у нас формується гістограма із 25 розбиттів, таким чином для певної області отримується вектор розмірністю 75, який є показником його кольорової гами в даній колірній моделі. Цей вектор нормується за нормою L1. Тоді міра схожості між 2 областями рахується таким чином:

$$s_{colour}(r_i, r_j) = \sum_{k=1}^{n} \min(c_i^k, c_j^k).$$ (c_i – вектор кольору для і-тої області-r_i) (2.33) [11]

2. Міра схожості за текстурою: Використовуючи прийоми алгоритму SIFT можна отримати векторне представлення текстури для кожної з областей. Якщо коротко, то робимо операцію згортки нашого зображення з Гаусовим ядром, точніше області, яку представляємо у вигляді зображення. При цьому послідовно збільшуємо параметр Sigma в Гаусовому ядрі (починаючи з Sigma=1), далі рахуємо різницю між парами послідовно зашумлених таким чином зображень. Можна задати кількість зображень, які ми отримуємо таким чином, нехай отримуємо 8 зображень враховуючи ще 3 кольорові канали отримуємо в кінці 24 зашумлених двовимірних зображення. Потім формуємо гістограму для кожного з цих зображень із 10 розбиттів, таким чином для нашої області отримаємо вектор, який описує її текстуру, розмірністю 240 (теж нормується за нормою L1). Формула для міри схожості за текстурою буде мати вигляд:

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} \min(t_i^k, t_j^k).$$ (t_i – вектор текстури для і-тої області-r_i) (2.34) [11]

3. Міра схожості за розміром: Ми бажаємо щоб маленькі області, які

ще не об’єдналися мали більше шансів на об’єднання ніж великі області. Таким чином приходимо до наступної формули для міри схожості за розміром

для 2 областей : $s_{size}(r_i, r_j) = 1 - \frac{\text{size}(r_i) + \text{size}(r_j)}{\text{size}(im)}$, (im-все зображення) (2.35) [11]

4. Міра схожості за наповненням: Головна ідея, що якщо певна область по суті огортає іншу область, то вони повинні бути об’єднані, для цього використовуємо таку формулу для міри схожості (BB_{ij}-найменша обмежувальна рамка, яка включає r_i та r_j):

$$s_{fill}(r_i, r_j) = 1 - \frac{\text{size}(BB_{ij}) - \text{size}(r_i) - \text{size}(r_i)}{\text{size}(im)} \quad (2.36) [11]$$

Загальна міра схожості буде мати вигляд:

$$s(r_i, r_j) = a_1 s_{colour}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{size}(r_i, r_j) + a_4 s_{fill}(r_i, r_j), \quad a_i \in \{0,1\} \quad (2.37) [11]$$

Отже, у алгоритмі вибіркового пошуку є такі параметри як: колірна модель, комбінації мір схожості, які ми будемо застосовувати для загальної міри схожості, також можна змінювати параметр k в алгоритми для генерування початкових областей (цей параметр відіграє важливу роль при генерування максимально допустимих по розміру областей, тобто чим більше цей параметр тим більшого розміру області буде генерувати нам даний алгоритм в якості початкових). Автори алгоритму вибіркового пошуку пропонують комбінувати результати алгоритмів отриманих при різних комбінаціях параметрів таким чином: кожній області, яка стоїть на і-тому ієрархічному рівні присвоюється число (рейтинг) випадковим чином із рівномірного розподілу на [0, і).

Це робиться для всіх варіантів алгоритму вибіркового пошуку, які ми використовуємо, після цього впорядковуємо усі отримані області по рейтингу та видаляємо дублікати з найнижчими рейтингами, за необхідності видаляємо ще деякі області з найнижчим рейтингом (якщо загальна кількість областей виходить занадто великою).

Оцінити якість зробленої сегментації можна за допомогою метрики MABO:

$$\text{ABO} = \frac{1}{|G^c|} \sum_{g_i^c \in G^c} \max_{l_j \in L} \text{Overlap}(g_i^c, l_j).$$

(2.38) [11]

G^c – множина справжніх обмежувальних рамок для класу c, L – запропоновані вибірковим пошуком, Overlap – теж саме що і IoU, MABO = усереднення ABO по всім класам.

Нижче наведені результати алгоритму вибіркового пошуку у порівнянні з іншими алгоритмами (див. рис. 2.12 - 2.14 [11]) - як бачимо це є один з найефективніших алгоритмів якщо не використовувати глибокі нейронні мережі (для оцінок використовувався VOC datasets):

threshold k in [13]	MABO	# windows
Flat [13] $k = 50, 150, \cdots, 950$	0.659	387
Hierarchical (this paper) $k = 50$	0.676	395
Flat [13] $k = 50, 100, \cdots, 1000$	0.673	597
Hierarchical (this paper) $k = 50, 100$	0.719	625

Рисунок 2.12

Version	Diversification Strategies	MABO	# win	# strategies	time (s)
Single Strategy	HSV C+T+S+F $k = 100$	0.693	362	1	0.71
Selective Search Fast	HSV, Lab C+T+S+F, T+S+F $k = 50, 100$	0.799	2147	8	3.79
Selective Search Quality	HSV, Lab, rgI, H, I C+T+S+F, T+S+F, F, S $k = 50, 100, 150, 300$	0.878	10,108	80	17.15

Рисунок 2.13

method	MABO	# windows
Arbelaez *et al.* [3]	0.649 ± 0.193	418
Alexe *et al.* [2]	0.694 ± 0.111	1,853
Harzallah *et al.* [16]	-	200 per class
Carreira and Sminchisescu [4]	0.770 ± 0.084	517
Endres and Hoiem [9]	0.791 ± 0.082	790
Felzenszwalb *et al.* [12]	0.829 ± 0.052	100,352 per class
Vedaldi *et al.* [34]	-	10,000 per class
Single Strategy	0.690 ± 0.171	289
Selective search “Fast”	0.804 ± 0.046	2,134
Selective search “Quality”	0.879 ± 0.039	10,097

Рисунок 2.14

Видно також, що найкращий результат досягається, якщо комбінувати

якомога більше стратегій, однак це займає занадто багато часу, тому на практиці використовують щось середнє між використанням лише 1 стратегії та комбінування усіх можливих - наприклад, як показано вище, алгоритм fast selective search використовує 8 стратегій.

2.10.2 Non-maximum suppression

Як зазвичай детектори або алгоритми, які генерують області з можливими об’єктами дають нам на виході занадто багато обмежувальних рамок, тобто на 1 об’єкт фактично йде по декілька обмежувальних рамок тому існує необхідність в алгоритмі, який з цих обмежувальних рамок вибери найбільш адекватні. В ідеалі ми хочемо щоб алгоритм для кожного об’єкта виводив тільки 1 обмежувальну рамку. В якості такого алгоритму зазвичай використовують Non-maximum suppression (NMS).

Вважаємо, що задано обмеження по показнику впевненості (відображає впевненість що у даній обмежувальній рамці є хоч якийсь об‘єкт): якщо показник впевненості більше або дорівнює цьому обмеженню то обмежувальна рамка містить об’єкт, якщо навпаки, то не містить. Отже, нехай нам дано деякий список S обмежувальних рамок, тоді алгоритм NMS складається з таких дій (розглядаються обмежувальні рамки певного фіксованого класу):

1. Видаляємо всі обмежувальні рамки у яких показник впевненості менший заданого обмеження.

2. Вибираємо обмежувальну рамку - M з найбільшим показником впевненості

3. Видаляємо ті обмежувальні рамки з S, у яких IoU з M >= заданого обмеження - K (параметр алгоритму)

4. Записуємо обмежувальну рамку M у список L (на початку алгоритму

це пустий список) та видаляємо її з S

5. Повторюємо кроки 2, 3 та 4 поки з S не видалятися усі обмежувальні рамки.

Дана процедура робиться для кожного класу.

Проблема даного алгоритму полягає в обмеженні K - іноді буває неможливо підібрати таке число K щоб: якщо IoU(A, B) >= K (*), то A точно охоплюють той об'єкт що і B і якщо (*) не виконується, то A точно охоплює інший об‘єкт ніж B. В таких випадках потрібно якимось чином обходитись без чого обмеження і при цьому робити згладжування показників впевненості, тобто якщо обмежувальні рамки занадто сильно перетинаються, то штрафувати їх але при цьому залишаючи їм можливість все ще увійти у список L - тобто бути відповідальним за деякий об‘єкт.

З такою проблемою може впоратися модифікація даного алгоритму - Soft NMS. Нехай f - це функція, яка буде грати роль згладжування показників впевненості. Тоді даний алгоритм по суті мало чим відрізняється від оригінального - лише кроком 3. На кроці 3 згладжуємо наші показники впевненості по такій формулі : $s_i \leftarrow s_i f(iou(\mathcal{M}, b_i))$ (s_i – показник впевненості i-тої обмежувальної рамки b_i), потім видаляємо ті обмежувальні рамки, у яких показник впевненості менше заданого обмеження і переходимо до кроку 4. f часто беруть гаусівською, тобто згладжування набуває форми:

$$s_i = s_i e^{-\frac{\text{iou}(\mathcal{M}, b_i)^2}{\sigma}}$$

(2.39) [13]

2.10.3 Модель R-CNN (Regional convolutional neural network)

Базова модель R-CNN працює таким чином: спочатку використовується алгоритм вибіркового пошуку щоб сформувати області у вигляді обмежувальних рамок, які повинні міститися певні об'єкти. Далі із

зображень виділяється кожна з цих областей і подається на вхід до згорткової нейромережі формуючи таким чином вектор ознак. Для класифікації використовується метод опорних векторів (SVM), який приймає на вхід вектор ознак сформований мережею. Лінійна регресія уточнює координати обмежувальної рамки отриманих за допомогою вибіркового пошуку, точніше лінійна регресія апроксимує змінні t_x, t_y, t_w, t_h для яких виконується $G_x = P_w t_x + P_x, G_y = P_h t_y + P_y, G_w = P_w exp(t_w), G_h = P_h exp(t_h);$ (2.40) [14] де (G_x, G_y) – координати центру реальної обмежувальної рамки, (G_w, G_h) – ширина та висота відповідної обмежувальної рамки, P означає теж саме для обмежувальної рамки запропонованої алгоритмом вибіркового пошуку.

В роботі [14] пропонується така модель (в якості згорткової нейромережі використовується AlexNet – див. рис 2.15 та рис. 2.16 [14]):

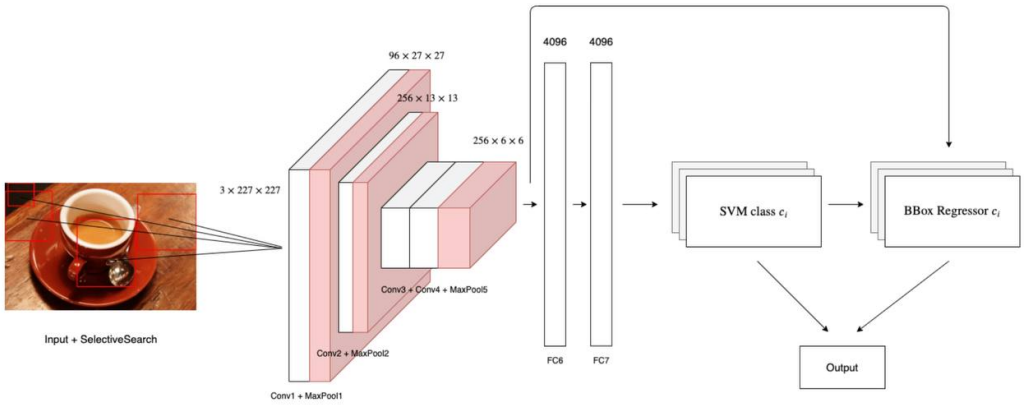


Рисунок 2.15

Більш детальна структура AlexNet (вхід буде мати розмір (227,227,3)):

Рисунок 2.16

Тренування відбувається таким чином: спочатку береться натренована згорткова нерйромережа, яка виконує завдання класифікації, наприклад AlexNet, змінюємо розмірність останнього шару так щоб вона відповідала кількості класів нашої задачі, потім дотреновуємо останні шари, тобто усі інші залишаємо з фіксованими вагами (в якості функції використовуємо перехресну ентропію). Після закінчення тренування цієї нейромережі видаляємо останній шар - таким чином наша нейромережа тепер виводить вектор ознак - цей вектор використовуємо як характеристику для кожного екземпляра в тренувальному наборі. Далі використовуючи ці вектори та номера класів, як мітки до цих векторів, тренуємо SVM. В кінці тренуємо лінійну регресію на вхід до якої подається вихід останньої операції пулінгу згорткової нейромрежі. Для лінійної регресії мінімізуємо таку функцію втрат (точніше кажучи 4 функції втрат - замість * підставляємо x,y,h та w):

$$\sum_{i}^{N}(t_{\star}^{i} - \hat{\mathbf{w}}_{\star}^{\mathrm{T}}\boldsymbol{\phi}_{5}(P^{i}))^{2} + \lambda \left\|\hat{\mathbf{w}}_{\star}\right\|^{2}$$

(2.41) [14]

(2.41) - це звичайна MSE із застосуванням L2 регуляризації.

При тренуванні використовуємо міні-пакети розміром 128, де 32 екземпляри містять об’єкт деякого класу, а 96 - не містять жодного класу (тобто нульовий клас, який фактично відповідає за фон). Ми кажемо що

запропонована обмежувальна рамка A містить деякий об’єкт на зображенні якщо IoU(A, B)>0.5, де B - деяка реальна обмежувальна рамка, якщо такого не виконується, то кажемо, що A містить лише фон. До того ж в якості мітки класу для A будемо брату мітку того B з ким A досягає максимуму по IoU.

Потрібно також зазначити, що як для моделей R-CNN так і для моделей YOLO виконується фільтрація отриманих обмежувальних рамок за допомогою NMS (або Soft NMS), в якості показника впевненості можна брати максимальну ймовірність належності до класу.

Головна проблема R-CNN В тому що ми повинні подавати на вхід кожну область окремо, враховуючи кількість областей, які генерує вибірковий пошук це занадто неефективно. Таким чином дана модель неможливо для застосування в реальному часі (на 1 зображення витрачається 13 секунд на GPU – дані 2014 року) І до того ж алгоритм вибіркового пошуку якого чином не навчається що робить неможливим його покращення робить нас надто сильно незалежними від недоліків цього алгоритму.

2.10.4 Модель Fast R-CNN

Для вирішення недоліків моделі R-CNN пропонується вдосконалена модель - Fast R-CNN. Вдосконалення полягає в тому що тепер ми не подаємо на вхід нашої моделі кожну область запропоновано вибірковим пошуком, ми подаємо на вхід початкове зображення тільки 1 раз. Далі отримуючи таким чином карту ознак з нею вилучається та область, яка вже нас цікавить (була сформована вибірковим пошуком) за допомогою шару RoI (про нього нижче). Таким чином отримали деякий тензор, який стискається в багатовимірний вектор, він проходить через декілька щільних шарів і одночасно подається на вхід до 2 шарів: 1 з них передбачає вірогідність належності до певного класу, інший апроксимує регресійну складову в тих же змінних, в яких це робилося і

для R-CNN. Загалом дана модель має таку структуру (див. рис. 2.17 [15]):

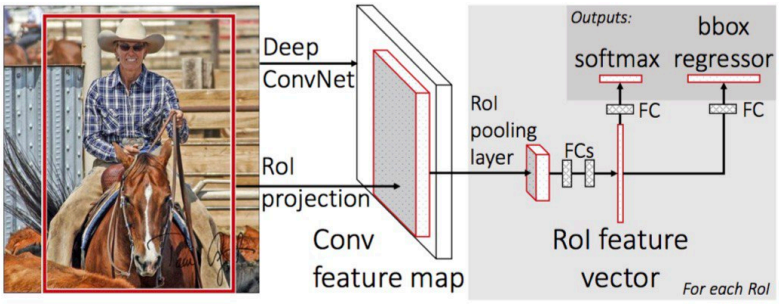


Рисунок 2.17

Шар RoI (Region of interest). До цього шару подається на вхід тензор та обмежувальна рамка, яку буде вилучена з цього тензора (карти ознак). Обмежувальна рамка подається у вигляді (x,y,w,h), де (x,y) - координата лівого верхнього краю обмежувальної рамки, а (w,h) - ширина та висота відповідної рамки. Цей шар по суті виділяє відповідну область на тензорі, розділяє її на матриці утворюючи таким чином решітку із матриць. Ця решітка буде розміром H на W (гіперпараметри для даного шару). Далі з кожної матриці в цій решітці вибирається максимум, при цьому кожна матриця буде розміром приблизно $h/H \times w/W$. Для вхідного тензора така операція робиться для кожного каналу незалежним чином як у випадку з операцією пулінга.

В роботі [15] пропонується використовувати згорткову нейромережу VGG16, далі останній щільний шар замінити на 2 шари: 1 з яких буде виконувати завдання класифікації, інший - завдання регресії (це робиться таким чином як було описано вище). Архітектура VGG16 (див. рис. 2.17 [15]):

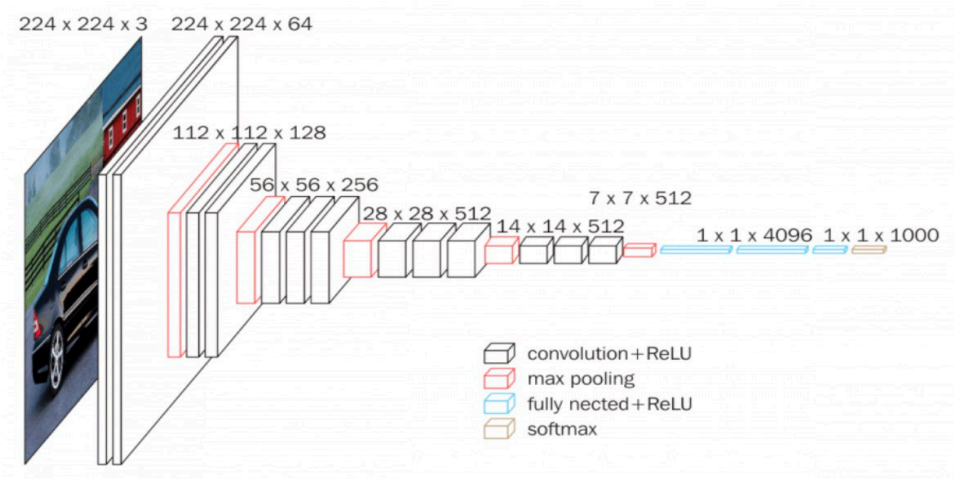


Рисунок 2.17

Для тренуванні Fast R-CNN буде використовуватись функція втрат у вигляді:

$$L(p, u, t^u, v) = L_{\text{cls}}(p, u) + \lambda[u \geq 1]L_{\text{loc}}(t^u, v)$$

(2.42) [15]

p – ймовірності для кожного з класів, які видає нам нейромережа, u – реальний клас об'єкта, t^u- апроксимування нейромережею поправок до обмежувальної рамки (отриманої вибірковим пошуком), v – реальні поправки, [u>=1] = 1 тільки тоді коли u>=1 – тобто коли обмежувальна рамка містить об'єкт, $L_{\text{cls}}(p, u)$ - функція втрат для частини класифікації (можна взяти перехресну ентропію), $L_{\text{loc}}(t^u, v)$ - функція втрат для частини регресії – береться функція Губера такого вигляду:

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}(t_i^u - v_i),$$

(2.43)[15]

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

При тренуванні даної нейромережі ми використовуємо міні пакети кожен з яких сконструйований з 2 зображень - з кожного з них вибирається 64 області і при цьому вибирається таким чином щоб 25 відсотків містили об'єкт,

а інші - ні. При цьому під негативними ми розуміємо ті у кого IoU в [0.1; 0.5) – так званий Hard Negative Mining.

В роботі [15] надана оцінка якості моделі Fast R-CNN порівняно з іншими моделями використовуючи VOC datasets (див. рис. 2.18 [15]):

method	train set	mAP
SPPnet BB [11]†	07 \ diff	63.1
R-CNN BB [10]	07	66.0
FRCN [ours]	07	66.9
FRCN [ours]	07 \ diff	68.1
FRCN [ours]	07+12	**70.0**

method	train set	mAP
BabyLearning	Prop.	63.8
R-CNN BB [10]	12	62.9
SegDeepM	12+seg	67.2
FRCN [ours]	12	66.1
FRCN [ours]	07++12	**68.8**

Рисунок 2.18

В принципі Fast R-CNN вирішує деякі проблеми моделі R-CNN пов’язані з необхідністю подавати на вхід до згорткової нейромережі кожну область запропоновано вибірковим пошуком. Однак все ще є один великий недолік - це сам алгоритм вибіркового пошуку, який доволі повільний і який ми не можемо вдосконалити шляхом навчання. Наступна запропонована модель вирішує і цю проблему.

2.10.5 Модель Faster R-CNN

В даній моделі початкові області будуть генеруватися не вибірковим пошуком, а окремою нейромережею (Regional Proposal Network - RPN), що робить модель Faster R-CNN набагато більш адекватною ніж минулі моделі. В

роботі [16] загальна структура моделі пропонується такою (див. рис. 2.19 [16]):

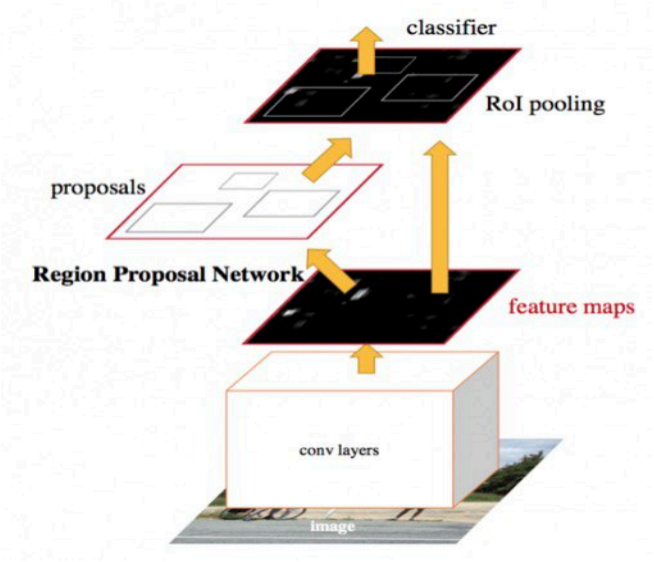


Рисунок 2.19

Структура моделі така сама як і Fast R-CNN, тільки тепер обмежувальні рамки нам пропонує не вибірковий пошук, а інша згорткова нейромережа, яка взаємопов'язана із загальною нейромережею. Спочатку ми пропускаємо вхідне зображення через деяку згорткову нейромережу щоб отримати карту ознак - на цьому етапі все виконується так само як і для Fast R-CNN. Далі RPN бере на вхід карту ознак і виводить обмежувальні рамки, потім застосовується шар RoI і все так само як і для Fast-R-CNN.

Нейромережа RPN. Для початкового апроксимування обмежувальних рамок пропонується ввести так звані якірні рамки - anchor boxes. Вхідне зображення розділяється на прямокутники однакового розміру. Якірні рамка застосовується до кожного з цих прямокутників. Таким чином для кожного з цих прямокутників та для кожної якірної рамки RPN виводитиме:

1. Імовірність того що дана якірна рамка відповідає за певний об'єкт.
2. Поправки до якірної рамки щоб отримати справжню обмежувальну рамку - у тому ж вигляді що і для R-CNN.

Структура RPN пропонується такою (див. рис. 2.20 [16]):

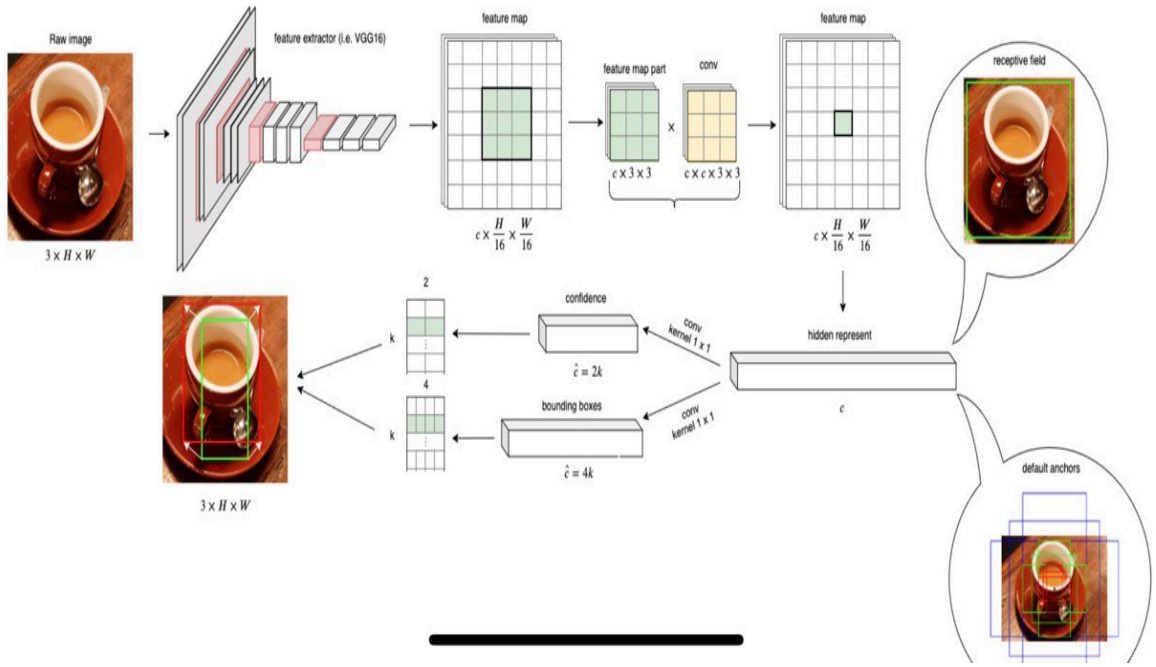


Рисунок 2.20

В кінці описано як діють 2 паралельні шари згортки - ядро згортки розміром 1 на 1 (k-кількість anchor boxes).

Якірна рамка відповідає за певний об’єкт (є позитивною) якщо IoU якірної рамки з обмежувальною рамкою цього об‘єкта > 0.7 або якщо ця якірна рамка має найбільший IoU серед усіх інших якірних рамок з деяким об‘єктом. Якщо ж IoU якірної рамки з обмежувальною рамкою будь-якого об‘єкта < 0.3 , то якірна рамка не відповідає ні за який об‘єкт - є негативною.

При формуванні міні-пакета для тренування RPN ми вибираємо деяке зображення, далі випадковим ним чином вибираємо 256 якірних рамок так щоб позитивних та негативних було порівну, якщо це неможливо доповнюємо міні-пакет негативними. Використовується така функція втрат:

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*).$$

(2.44) [16]

В якості класифікаційної функції втрат можна використати бінарну ентропію (або перехресну), p_i^* = 1 якщо і-та якірна рамка відповідає за певний об’єкт (в іншому випадку = 0), p_i – вивід нейромережі (RPN) ймовірності того

що і-та якірна рамка відповідає за певний об’єкт, регресійна функція втрат така сама як і для Fast R-CNN, N_{reg}таN_{cls} – нормуючі коефіцієнти (N_{cls} =кількість якірних рамок в міні-пакеті=256, N_{reg} =загальна кількість якірних рамок на зображенні= близько 2400, для балансування λ = 10).Усі інші позначення такі самі як і для Fast R-CNN. В якості регресійних міток і-ої якірної рамки беремо обмежувальні рамки тих об’єктів, які є найближчими до і-ої якірної рамки з точки зору IoU.

Процес тренування відбувається ітеративно таким чином:

1. Спочатку формується RPN - Завантажується вже натренована згорткова нейромережа для класифікації з ImageNet, точніше тільки та її частина, яка формує карту ознак, інші шари добудовуються так як це показано на зображенні вище (структура RPN). Запускається процес тренування тільки для цих останніх шарів.

2. Формується Fast R-CNN - Завантажується вже натренована згорткова нейромережа для класифікації, точніше тільки та її частина, яка формує карту ознак (в роботі пропонується взяти наприклад VGG16), інші шари добудовуються так як це показано в минулому пункті. Запускається процес тренування тільки для цих останніх шарів.

3. Останні шари, які є унікальними для RPN, добудовуємо до карти ознак Fast R-CNN і тренуємо ці унікальні шари RPN, усі інші ваги фіксуємо

4. Фіксуємо усі ваги крім останніх щільних шарів загальної нейромережі (по суті це ті, які відносяться до Fast R-CNN) і дотреновуємо їх.

В роботі [16] також сказано, що додаткові ітеративні процедури такого типу не дають значного покращення.

Вибір якірних рамок робиться таким чином щоб вони якнайкраще описували вхідні дані. Тобто це може бути зроблено вручну із урахуванням деякої апріорної інформації або автоматичним чином - наприклад із використанням методів кластеризація (більш детально в пункті щодо моделей YOLO)

Показники якості отриманої нейромережі в порівнянні з іншими

моделями (див. рис. 2.21 [16]):

method	proposals	training data	COCO val		COCO test-dev	
			mAP@.5	mAP@[.5, .95]	mAP@.5	mAP@[.5, .95]
Fast R-CNN [2]	SS, 2000	COCO train	-	-	35.9	19.7
Fast R-CNN [impl. in this paper]	SS, 2000	COCO train	38.6	18.9	39.3	19.3
Faster R-CNN	RPN, 300	COCO train	41.5	21.2	42.1	21.5
Faster R-CNN	RPN, 300	COCO trainval	-	-	**42.7**	**21.9**

Рисунок 2.21

Модель Faster R-CNN є найбільш ефективною з цього сімейства і дає 1 з найкращих результатів і до сьогодення, хоча не з точки зору швидкості опрацювання зображень, тобто для застосувань в реальному часі ця модель все ж таки не набула популярності.

2.11 Сімейство моделей YOLO

2.11.1 Модель YOLO v1

Моделі із сімейства YOLO намагаються знайти ідеальний баланс між швидкістю обробки зображень та якістю моделі. Усі моделі з цього сімейства беруть на вхід зображення, пропускають його 1 раз через нейромережу і одразу видають деяку множину обмежувальних рамок, яка потім додатково обробляється NMS (або в більш сучасних Soft NMS).

Принцип роботи YOLO v1 (робота [17]). Вхідне зображення ділиться на сітку розміром S * S, і для кожної клітини в цій сітці передбачаються B обмежувальних рамок. Для кожної обмежувального рамки передбачаються 5 змінних, перші 2 це координати центру реальної обмежувального рамки, далі йде відповідно ширина та висота даної рамки, 5-а зміна це буде показник впевненості що обмежувальна рамка, яка виведена нейромережою, містить певний об’єкт - визначається таким чином: $\Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}}$. Також до кожної клітини в цій сітці нейромережа виводить ймовірності належності до кожного

з класів (кількість класів = C). Отже нейромережа виводить тензор розмірністю: $S \times S \times (B * 5 + C)$. При тестуванні щоб отримати кінцевий показник впевненості, який буде можливо використовуватись в NMS ми домножиюємо показник впевненості виведений нейромережею на максимальну імовірність належності до класу.

Авторами запропонована архітектура схожа на нейромережу GoogleNet[18] (див. рис. 2.22 [17]), згортка 1*1 перед згорткою 3*3 використовуються для економії параметрів - це дозволяє використовувати меншу кількість фільтрів 3*3 і при цьому не втрачати особливо багато точності як це відбувається як випадку з GoogleNet (в даному випадку S=7, B=2, C=20):

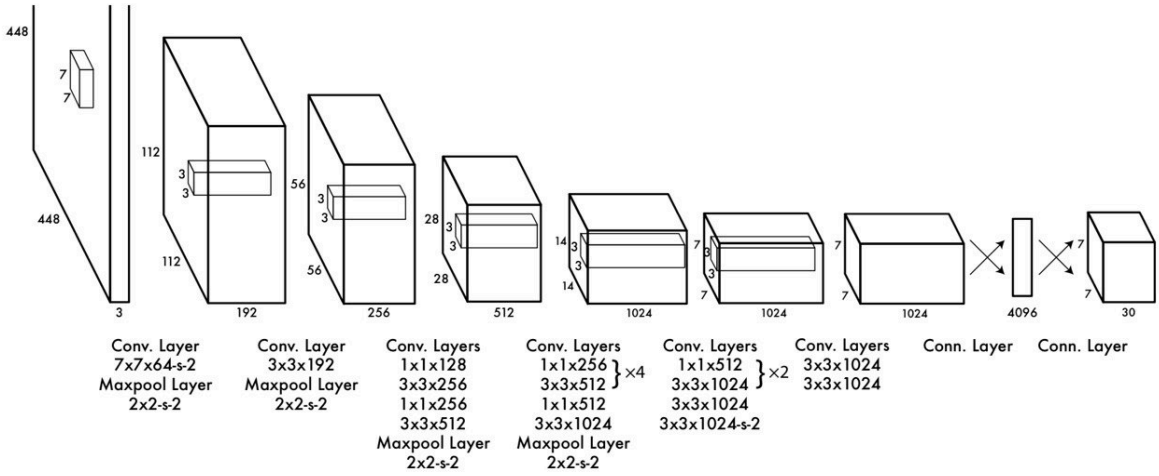


Рисунок 2.22

При тренуваннях ми спочатку використовуємо перші 20 шарів цієї нейромережі, за якими ідуть average pooling та 1 щільний шар для передбачення ймовірності класу. Ми тренуємо цю мережу щоб навчити виконую твої завдання класифікації на датасеті ImageNet competition. Далі ми прибираємо останні 2 шари в цій нейромережі, заміняємо їх на 4 згорткові та 2 щільні шари і тренуємо усю модель близько 135 епох. При цьому використовується така функція втрат [17]:

$$\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$
$$+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$
$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$
$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$
$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \tag{2.45}$$

По факту тут усюди використовуються середньоквадратичне відхилення, для визначення розміру сторін ми ще додатково перед застосуванням середньоквадратичного відхилення до кожної з координат застосовуємо квадратний корінь для того щоб зробити функцію втрат більш інваріантною до масштабу зображень (ширина та висота є нормалізованами - поділили на відповідно ширину та висоту самого зображення). Так як клітин з відсутністю об’єктів набагато більше ніж з об’єктами то встановлюються такі вагові коефіцієнти для балансу: $\lambda_{\text{coord}} = 5$ та $\lambda_{\text{noobj}} = .5.$

В якості функції активації використовується Leaky relu (пункт 2.3). Для запобігання перенавчанню використовується шар Dropout (p=0.5) після першого щільного шару.

Також під час тренування використовувалась техніка збільшення даних. Загалом все техніка включає в себе такі випадкові лінійні операції над зображеннями: обертання (відбувається випадковим чином, як гіперпараметри встановлюються межі кутів повороту), перевертання (горизонтальне та вертикальне), симетричне відображення, зміна розміру, також можливе змішування контексту кількох зображень в одне або занулення пікселів, які утворюють прямокутник (розмір сторін визначається випадковим чином в певних межах) та ще деякі перетворення, про які буде докладніше в розділі YOLO v4. Для даної моделі ми використовуємо лише випадково зміну розміру та насиченість кольору.

Загалом дана мережа має багато мінусів вона не здатна ідентифікувати

об’єкти різних класів якщо вони стоять один біля одного, також є проблема з тим щоб ідентифікувати скупчення малих об’єктів. До того ж через те що нейромережа не уточнює обмежувальні рамки, а фактично передбачає їх з нуля, функція втрат веде себе дещо нестабільно на перших етапах тренування тобто є занадто чутливою до гіперпараметрів. Відсутність технології уточнення обмежувальних рамок також дає низькі результати щодо збіжності навчання, тобто модель потребує занадто багато часу щоб наблизитись до оптимальної точки функції втрат.

2.11.2 Моделі YOLO v2 та YOLO9000

В даному випадку вхідне зображення так само розділяється на сітку, однак тепер кожна клітина в цій сітці також вміщує в себе деяку фіксовану кількість якірних рамок (центр якірної рамки – центр клітини), по 5 на кожну клітину як це запропоновано авторами в роботі [19] (розмір сітки при цьому 13*13).

Кожна якірна рамка передбачає 5 координат t_x, t_y, t_w, t_h, t_0 такі що ($\sigma(z)$ = sigmoida(z)):

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h} \\
Pr(\text{object}) * IOU(b, \text{object}) &= \sigma(t_o)
\end{aligned}
$$

(2.46) [19]

(c_x, c_y) – координати верхнього лівого краю клітини; p_w, p_h – відповідно ширина та висота якірної рамки; b_x, b_y, b_w, b_h – відповідні характеристики реальної обмежувальної рамки.

Також кожна якірна рамка передбачає ймовірності того що об’єкт, за який вона відповідає, належить до певного класу. Якірна рамка відповідає за

певний об’єкт якщо її IoU з обмежувального рамкою даного об’єкта є максимально серед усіх інших якірних рамок. Доцільним є визначати якірні рамка за допомогою методів кластеризація, наприклад методом k-середніх.

В якості архітектури нейромережі даної авторами роботи [19] використовується Darknet-19 (див. рис. 2.23 [19]), який ми спочатку тренуємо для класифікації, потім дещо видозмінюючи в цій нейромережі тренуємо її для виявлення об’єктів. Структура відповідних нейромереж (в Darknet-19 замість останнього згорткового шару було додано декілька нових, після цього також робиться конкатенація передостаннього згорткового шару з більш раннім для того щоб врахувати більш детальні властивості об’єктів):

Darknet-19:

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Рисунок 2.23

YOLO v2 (див. рис. 2.24 [19]):

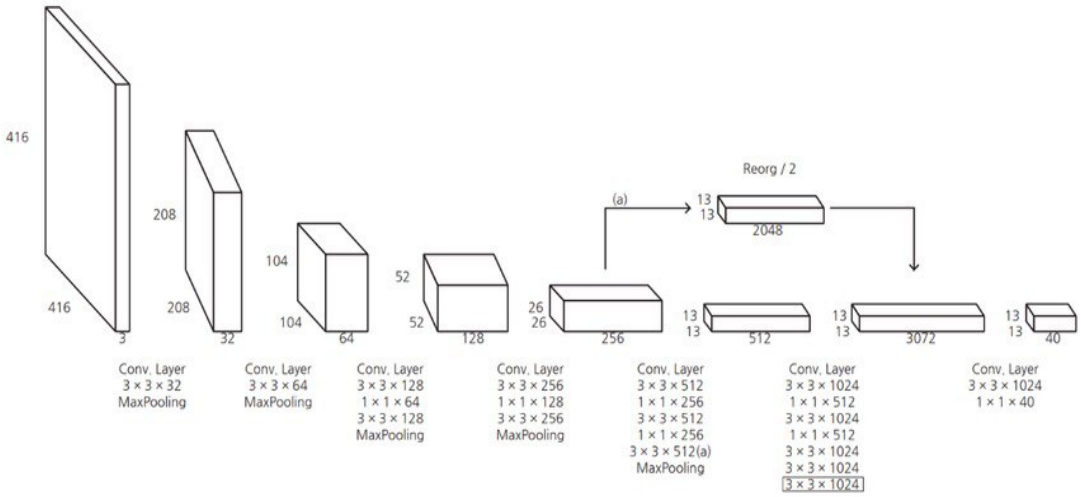


Рисунок 2.24

При класифікації спочатку тренується нейромережа щоб вона якісно класифікувала зображення розміром 224 на 224, далі 10 епох додатково тренуємо на картинках розміром 448. Під час тренування нейромережі для виявлення об’єктів кожні 10 пакетів вибираємо новий розмір зображення випадковим чином з пулу {320, 320+32, 320+64,…,608} та тренуємо на зображеннях цього розміру. Усі інші деталі імплементації такі самі як і для попередньої версії YOLO.

Також варто зазначити що замість блоків Dropout ми вводимо BatchNormalization, що дозволяє прискорити збіжність навчання і відіграє роль регуляризації.

Усі інші особливості такі самі як і для YOLO v1, включаючи функцію втрат, однак ми використовуємо не самі змінні, які видає нейромережа, а змінюємо їх так як описано вище щоб отримати відповідні показники обмежувальної рамки.

Нижче наведені показники якості YOLO v2 з врахуванням швидкості оброблення зображень у вигляді FPS (див. рис. 2.25 та 2.26 [19]):

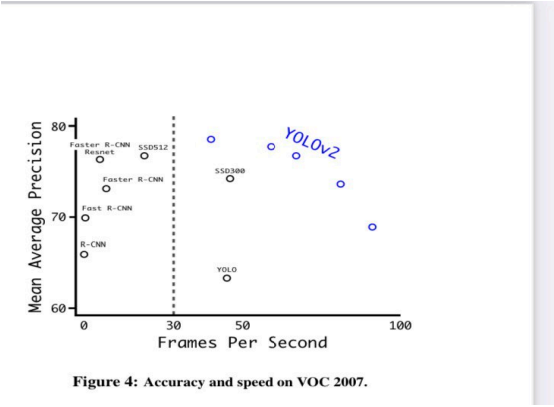


Figure 4: Accuracy and speed on VOC 2007.

Рисунок 2.25

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 288 × 288	2007+2012	69.0	91
YOLOv2 352 × 352	2007+2012	73.7	81
YOLOv2 416 × 416	2007+2012	76.8	67
YOLOv2 480 × 480	2007+2012	77.8	59
YOLOv2 544 × 544	2007+2012	**78.6**	40

Рисунок 2.26

Бачимо дуже гарний баланс між швидкістю та якістю даної моделі.

YOLO9000. Ця модель по суті являє те ж саме що і YOLO v2, однак додатково ще водиться ієрархічна класифікація. При тренування кінцевої нейромережі, яка відповідає за виявлення об’єктів в тренувальному наборі даних будуть міститися зображення як для виявлення об’єктів так і для звичайної класифікації. Тільки тепер клас ми будемо передбачати не звичайним вектором з використанням функції softmax, а використовуючи так зване дерево - кожен ярус цього дерева по суті являє типи об’єктів, за якими можемо розрізняти кожен об’єкт причому кожен тип може мати підтипи. Для вірогідності кожного з цих типів, які відносяться до одного типу ми вже можемо ввести функцію активації softmax. Отже таким чином вводячи цю ймовірність маємо що якщо в тренувальному наборі даних ми бачимо зображення для виявлення об’єктів то виконуємо зворотнє розповсюдження для звичайної функції втрат, якщо бачимо зображення для класифікації, то

виконуємо зворотнє розповсюдження тільки для класифікаційною частини, причому виконуємо це тільки для тих передбачених обмежувальних рамок, які з найбільшою імовірністю містить даний тип об’єкта з тренувальних даних (в якості функції втрат можемо брати перехресну ентропію).

Дана модель комбінує в собі усі плюси звичайної YOLO v2, а також здатна класифікувати більше ніж 9000 типів об’єктів.

2.11.3 Модель YOLO v3

Для цієї версії в роботі [20] була введена нова згорткова нейромережа для класифікації як основа, з якої потім будується детектор, - Darknet-53. Це доволі велика нейромережа, тому щоб не було проблем із деградацію точності вводяться skip connections. Структура Darknet-53 (див. рис. 2.27 [20]):

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Рисунок 2.27

В даній версії YOLO ми передбачаємо ті ж самі змінні що і в минулій, однак робимо це в 3 масштабах, тобто карта ознак, яку дає нам Darknet-53, обробляється декількома згортковими шарами і виводиться сітка передбачень з якірними рамками як це робилось раніше. Далі ми беремо таку ж карту ознак,

збільшуємо її ширину та висоту вдвічі, робимо конкатенацію з минулим шаром Darknet-53 по каналам (конкатенація робиться з тим шаром щоб ця операція було в принципі можлива). Після цього ми так само застосовуємо декілька згорткових шарів не змінюючи висоту та ширину отриманої карти знак і таким чином отримаємо сітку передбачення але вже в 2-му масштабі, вона буде в 2 рази більше ніж минула, по глибині (кількості каналів) такою ж самою. Аналогічними діями отримаємо передбачення і в 3-му масштабі.

Структура даної нейромережі описана нижче (див. рис. 2.28 [20]):

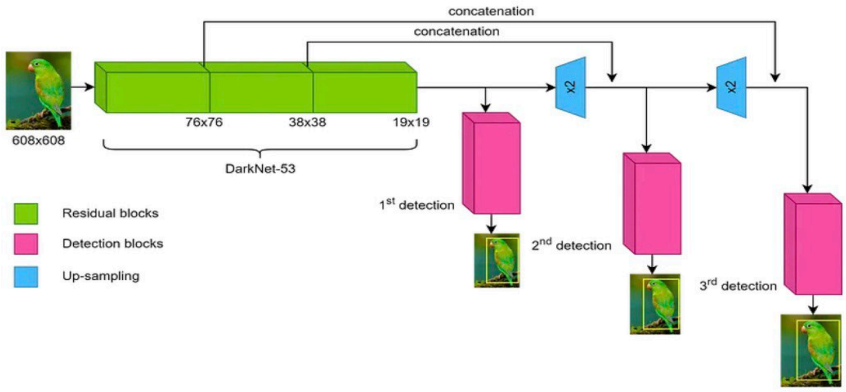


Рисунок 2.28

Також для тренування застосовують більший набір аргументації даних (збільшення масиву даних) ніж в попередній версії- ці прийоми були коротко описані в минулому пункті.

Усі інші особливості цієї моделі такі самі як і у минулої, функція втрат – сума середньоквадратичних відхилень, однак в даному випадку середньоквадратичне відхилення застосовується вже до самих змінних, які виводить нейромережа, без зайвих перетворень.

В подальшому кожна наступна модель YOLO буде намагатися імплементувати усі найбільш ефективні сучасні техніки в нейромережах для досягнення балансу між швидкістю та якістю.

2.11.4 Моделі YOLO v4 та YOLO EDGE

Загалом детектори можна поділити на однорівневі та дворівневі по такій схемі (див. рис. 2.29 [21]) наведеній в роботі [21]:

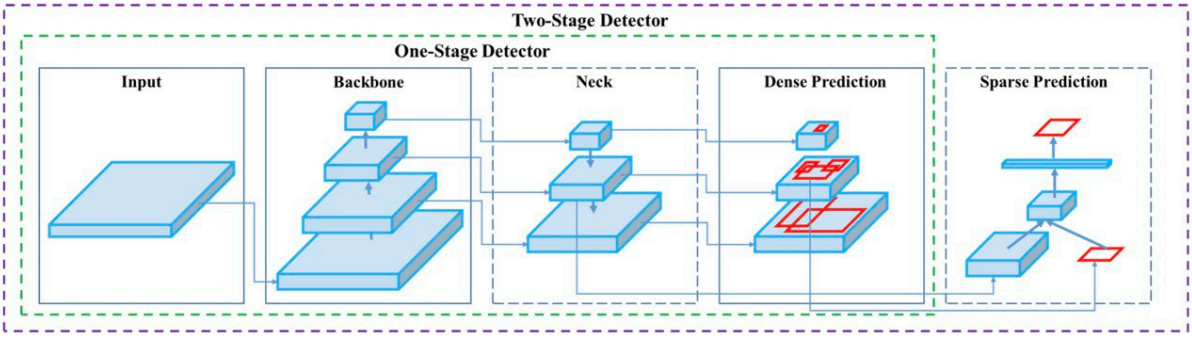


Рисунок 2.29

Моделі YOLO належать до однорівневих, R-CNN - до дворівневих.

В YOLO v4 авторами [21] запропоновано таку структуру нейромережі (див. рис. 2.30 [21]):

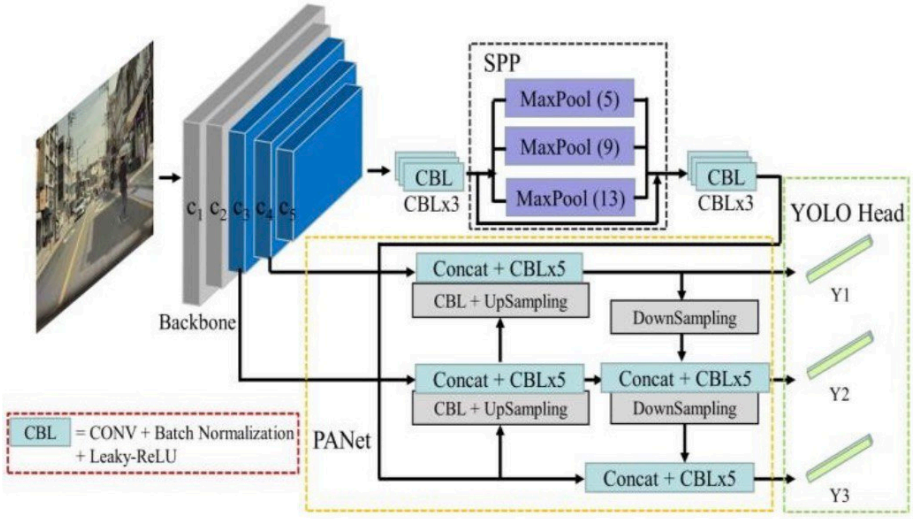


Рисунок 2.30

В якості основної частини (backbone) використовували CSPDarknet-53 - це дещо модифікована основа минулої версії YOLO із використанням функції активації Mish та skip connections типу CSP (перевага цієї функції

активації та даного типу skip connections перераховані у пунктах “Skip connections” та “Функції активації” відповідно). Архітектура CSPDarknet-53 (див. рис. 2.31 [21]):

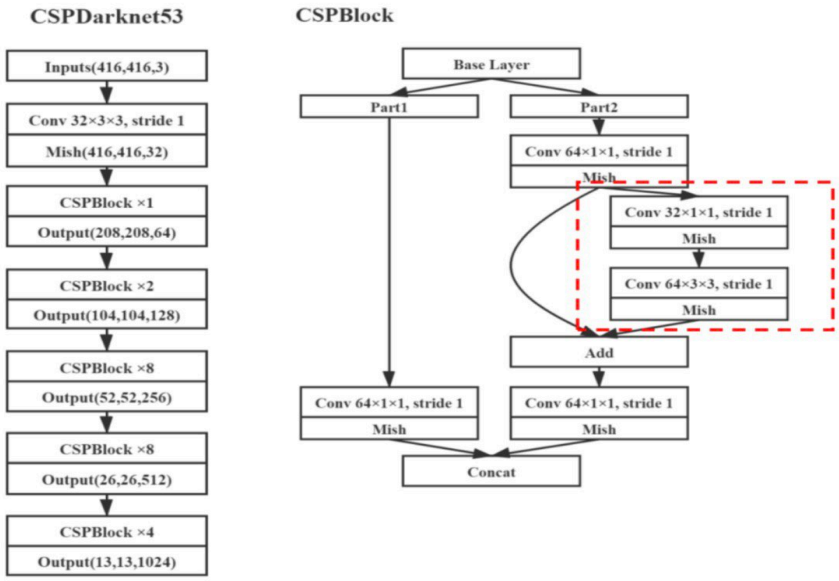


Рисунок 2.31

В якості шиї використовуються SPP блок та PANet. Блок допомагає отримати з вхідної карти ознак довільного розміру вектор фіксованого розміру. В даному випадку SPP блок працює трохи інакше - для фіксованого зображення це просто 3 паралельні операції пулінгу із подальшою конкатенацією, однак вікно пулінгу буде збільшуватись пропорційно тому як збільшується вихідна карта ознак. Таким чином для великих зображень ми не будемо виділяти надто непомітні ознаки, а для зображень маленького розміру не пропустимо дрібні ознаки, які мають значення. Мережа PANet робить підвищення дискретизації щоб ми змогли ефективно навчити загальну нейромережу виводити передбачення в 3 масштабах як це робилося для YOLO v3.

При тренуванні використовується такі функції втрат (їх доцільність було описано в пункті «функцій втрат»): бінарна ентропія для класифікації, mse для показника впевненості, CIoU для обмежувальної рамки.

Також під час тренування застосовуються різноманітні техніки

збільшення даних, які були описані в минулих пунктах, а також один специфічний для цієї нейромережі - мозаїк. По суті це змішування контексту випадково взятих 4 зображень. Також застосовується техніка змагання із самим собою, тобто ми зашумлюємо зображення таким чином щоб максимізувати функцію втрат, тобто по суті змінюємо незначним шумами зображення таким чином щоб нейромережа розпізнавала її гірше, однак потім виконуємо зворотнє розповсюдження на цьому екземплярі - навчаючи нейромережу таким чином краще робити передбачення на складних нечітких зображеннях.

Варто зазначити що в якості пакетної нормалізації використовуються перехресна міні пакетна нормалізації (більш детально описано в розділі «Шари нормалізації»). При фільтрації отриманих обмежувальних рамок ми використовуємо таку функцію в алгоритмі Soft NMS [21]:

$$s_i = \begin{cases} s_i, & IoU - \mathcal{R}_{DIoU}(\mathcal{M}, B_i) < \varepsilon, \\ 0, & IoU - \mathcal{R}_{DIoU}(\mathcal{M}, B_i) \geq \varepsilon, \end{cases} \quad \mathcal{R}_{DIoU} = \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2}, \tag{2.46}$$

Таким чином ще враховуються відстань між центрами обмежувальних рамок: якщо вона достатньо велика то ми ніяк не впливаємо на показник впевненості, тобто ніяк його не зменшимо і відповідно не відфільтровуємр дану обмежувального рамку.

Показники якості даної нейромережі (див. рис. 2.32 [21]):

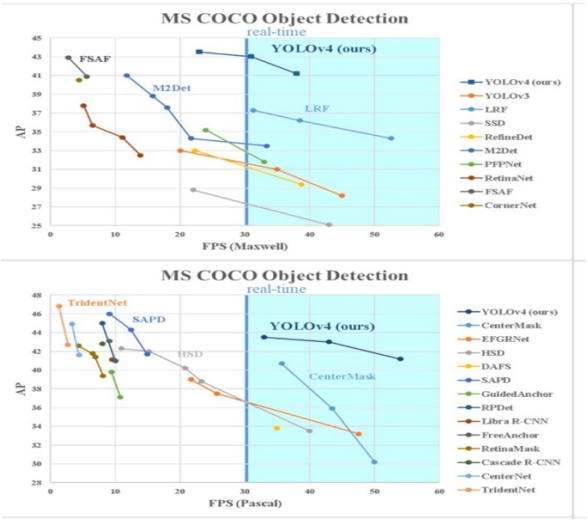


Рисунок 2.32

Бачимо значне покращення точності в залежності від попередньої версії за зовсім непомітний спад FPS.

YOLO Edge. Дана версія YOLO - це спроба зменшити розмір YOLO v4, при цьому зберегти її ефективність, тобто зробити її доступним для використання на периферійних девайсах без сторонніх серверів (тільки тренування здійснюється з використанням додаткових серверів). В архітектурі YOLO Edge в роботі [22] запропоновано використати основну частину з YOLO v4 дещо її скоротивши, в шиї замість PANet використовується простіша структура FPN, яка не робить зайвих операцій по зменшенню дискретизації, що може бути обтяжливим для периферійних пристроїв. Також замість функції активації Mish використовується Leaky relu. Загалом її структура має такий вигляд (див. рис. 2.33 [22]):

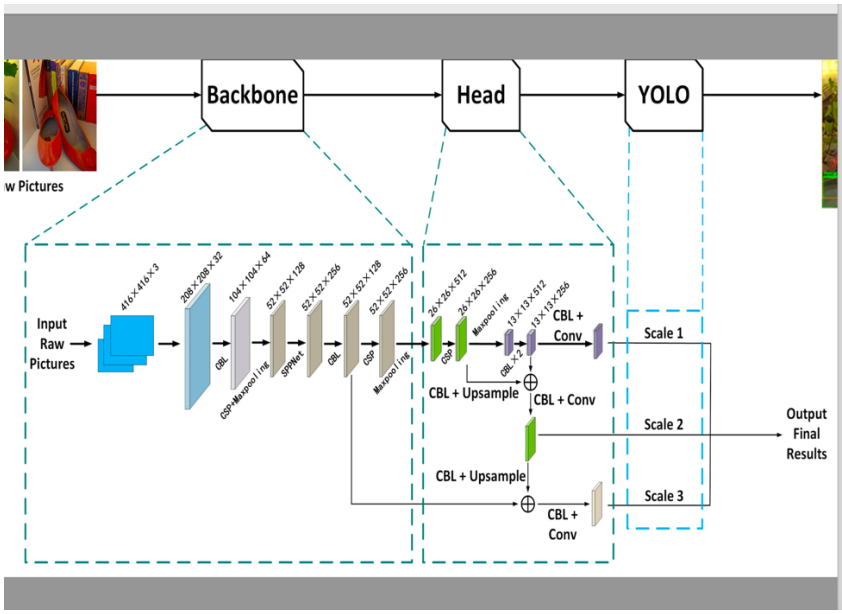


TABLE I
IMPROVED BACKBONE OF EDGE YOLO FOR EACH LAYER

Layer	Type	Size/Stride	Filters	Output	BFLOPS
0	Conv	3 × 3/2	32	208 × 208 × 32	0.075
1	Conv	3 × 3/2	64	104 × 104 × 64	0.399
2	Conv	3 × 3/2	64	104 × 104 × 64	0.797
3	Route		2	104 × 104 × 32	
4	Conv	3 × 3/1	32	104 × 104 × 32	0.199
5	Conv	3 × 3/1	32	104 × 104 × 32	0.199
6	Route		54	104 × 104 × 64	
7	Conv	1 × 1/1	64	104 × 104 × 64	0.089
8	Route		27	104 × 104 × 128	
9	Max	2 × 2/2		52 × 52 × 128	0.001
10	Conv	1 × 1/1	128	52 × 52 × 128	0.089
11	Max	5 × 5/1		52 × 52 × 128	0.009
12	Route		10	52 × 52 × 128	
13	Max	9 × 9/1		52 × 52 × 128	0.028
14	Route		10	52 × 52 × 128	
15	Max	13 × 13/1		52 × 52 × 128	0.058
16	Route		15 13 11 10	52 × 52 × 512	

TABLE II
NETWORK ARCHITECTURE OF IMPROVED FPN FOR EACH LAYER

Layer	Type	Size/Stride	Filters	Output	BFLOPS
17	Conv	1 × 1/1	256	52 × 52 × 256	0.709
18	Conv	3 × 3/1	128	52 × 52 × 128	1.595
19	Route		18	52 × 52 × 64	
20	Conv	3 × 3/1	64	52 × 52 × 64	0.199
21	Conv	3 × 3/1	64	52 × 52 × 64	0.199
22	Route		21 20	52 × 52 × 128	
23	Conv	1 × 1/1	128	52 × 52 × 128	0.089
24	Route		18 23	52 × 52 × 256	
25	Max	2 × 2/2		26 × 26 × 512	0.001
26	Conv	1 × 1/1	128	26 × 26 × 128	0.044
27	Conv	3 × 3/1	256	26 × 26 × 256	0.399
28	Route		27	26 × 26 × 128	
29	Conv	3 × 3/1	128	26 × 26 × 128	0.199
30	Conv	3 × 3/1	128	26 × 26 × 128	0.199
31	Route		30 29	26 × 26 × 256	
32	Conv	1 × 1/1	256	26 × 26 × 21	0.089
33	Route		32 27	26 × 26 × 512	
34	Max	2 × 2/2		13 × 13 × 512	0.000

Рисунок 2.33

CSP skip connections, які використовувалися для YOLO Edge (див. рис. 2.34 [22]):

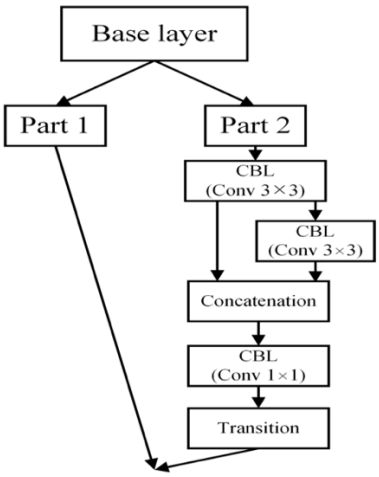


Рисунок 2.34

Всі інші особливості залишаються без змін (в порівнянні з YOLO v4).

Показники якості (див. рис. 2.35 [22]):

Model	Sizes(MB)	mAP@0.5	FPS^{ave}_{Xavier}	FPS^{ave}_{Nano}	FLOP/S	mAP@0.5 (TensorRT)	FPS^{ave}_{Xavier} (TensorRT)	FPS^{ave}_{Nano} (TensorRT)
YOLOV3	236.52	0.553	4.9	2.0	65.879G	0.373	31.2	4.93
YOLOV4	245.78	0.657	2.1	1.0	128.459G	0.459	26.1	4.62
MobileNetV3 SSD	115.09	0.362	10.3	9.2	17.863G	0.248	32.9	20.9
YOLOV3-Tiny	33.79	0.331	50.3	15.2	5.571G	0.077	67.5	35.8
Edge YOLO	25.27	0.473	26.6	11.4	10.250G	0.325	49.5	26.6

Рисунок 2.35

Для свого розміру нейромережа дає непогані результати (для YOLO v4 ці показники відрізняються від тих, які були вище, так як там використовувалася $mAP_{0.5\text{-}0.95}$).

2.11.5 Модель YOLO v5

Загальна структура даної версії YOLO (див. рис. 2.36 [23]):

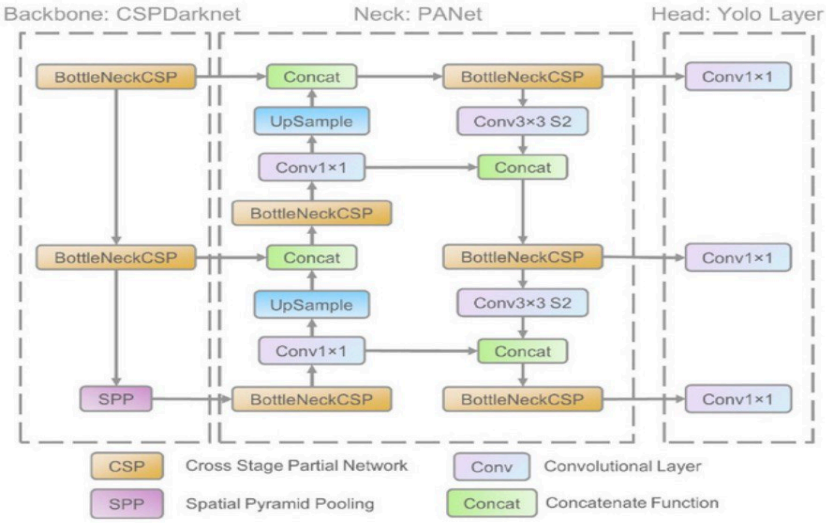


Рисунок 2.36

Насправді дана версія має багато модифікацій, які розподіляються по розміру нейромережі: нано-версія, маленька-версія, середня-версія, велика-версія, екстра-велика-версія.

Кожна з них дещо вдосконалює загальну структуру заради або підвищення ефективності або швидкості обробки зображень. За основу береться техніка CSP skip connections, як і в минулій версії, однак із використанням згортки 1*1 для зниження кількості параметрів таким чином (див. рис. 2.37 [23]):

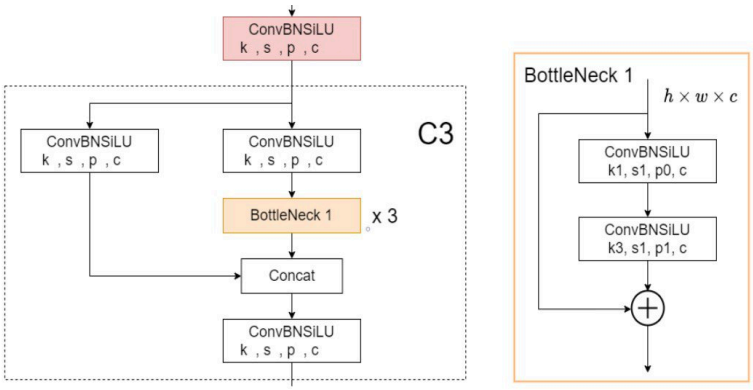


Рисунок 2.37

В шиї замість SPP використовується SPPF для підвищення швидкості (див. рис. 2.38 [23]):

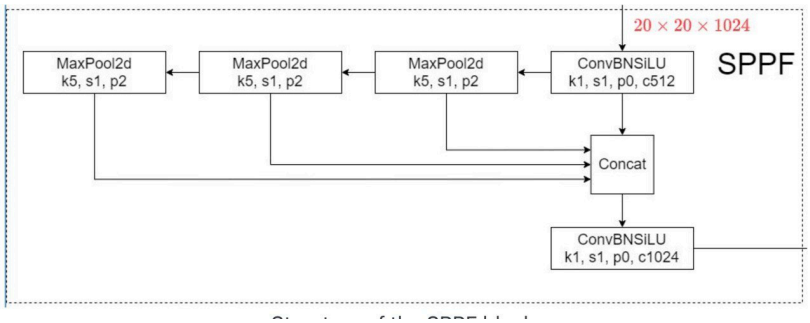


Рисунок 2.38

У внутрішніх шарах зазвичай використовується функція активації – Silu.

Кожна якірна ранку передбачає, окрім імовірностей класів, 5 змінних які співвідносяться з типовими показниками обмежувальної рамки таким чином [23]:

$$
\begin{aligned}
b_x &= (2 \cdot \sigma(t_x) - 0.5) + c_x \\
b_y &= (2 \cdot \sigma(t_y) - 0.5) + c_y \\
b_w &= p_w \cdot (2 \cdot \sigma(t_w))^2 \\
b_h &= p_h \cdot (2 \cdot \sigma(t_h))^2
\end{aligned}
$$

(2.47) [23]

Формули відрізняються трохи від тих, які були в минулих версіях, тому що для минулих версій ми фактично не могли отримати центр обмежувального рамки, який лежав точно на краю клітини, через те що сігмоїда лише асимптотично наближається до 0 або 1 але не дорівнює жодному з цих значень. З використанням даних формул ця проблема відсутня.

Приклад архітектури YOLO v5 великого розміру можна подивитися за посиланням: https://docs.ultralytics.com/yolov5/tutorials/architecture_description/ .

Також для пошуку гіперпараметрів застосовуються генетичні алгоритми. Це алгоритми, які дозволяють вирішити складні задачі оптимізації, ті які важко або неможливо вирішити звичайним градієнтом спуском. Суть в

тому що спочатку в області визначення цільової функції ініціалізуєтьчя множина початкових точок. Далі підраховується значення цільової функції в цих точках, ті точки в яких функції ближче до оптимального значення одразу переходять на наступний рівень (до наступної популяції) , ті в яких цільова функція трохи менше певного заданого рівня відбираються як батьки для наступне популяції тобто видозмінюючи координати цих точок отримаємо точки для наступної популяції. Ці видозміни можуть відбуватися випадковим зашумленням додаючи Гаусівський вектор до 1 точки, в такому разі це називається мутація. Або це може бути перехрещення - коли для кожної координати випадковим чином з 2 батьків вибирається кординати одного з батьків і таким чином формується точка для наступних популяцій. Батьки вибираються з певними ймовірностями, які зазвичай обернено пропорційні тому наскільки цільова функція близька до оптимального значення. Таким чином генерується сотні популяцій (точок), серед кінцевої генерації вибираюся точка, в якій цільова функція буде досягати найближчого до оптимального значення.

Всі інші деталі ідентичні попередній моделі YOLO.

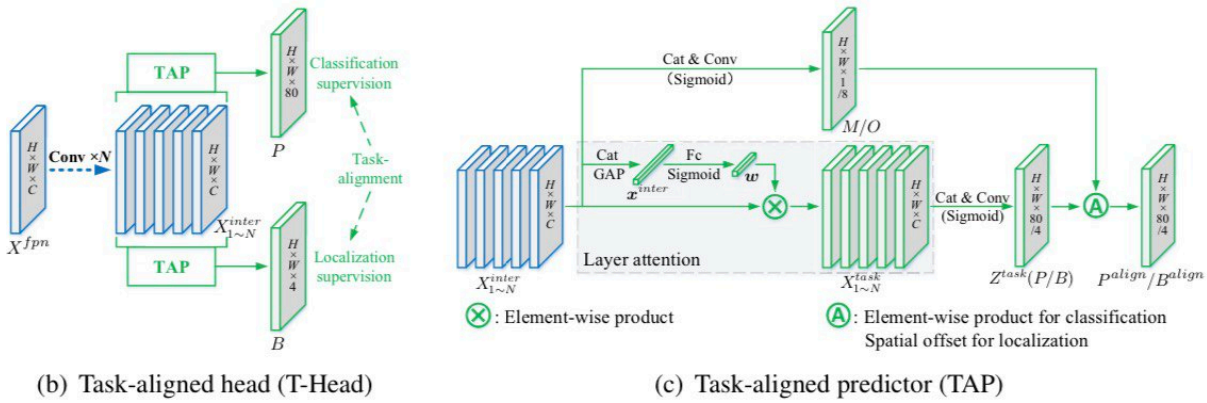
2.11.6 Модель YOLO v6

Так само як і для минулої версії YOLO, для цієї не існує єдиної нейромережі - існують різні модифікації. Отже так само опишемо базову техніку створення YOLO v6.

Ця версія застосовує новий метод оптимізації на відміну від звичайного градієнтного спуску - перепараметризація оптимізатора. Метод полягає в тому що у нас є деяка реальна модель, яку ми сконструювали, також ми уявляємо собі дещо більш складну модель, яка краще апроксимує шукані змінні і замість того щоб її будувати ми видозмінюємо градієнтний крок таким чином щоб

реальна модель до якої ми прийдемо була рівносильна складній уявній моделі але при звичайному градієнтному кроці. Цей процес чудово описано в роботі [24] .

Також застосовується техніка розділення голови нашої нейромережі, тобто в кінці ми робимо деяке розділення нашої карти ознак по каналам, можливо додатково ще застосовуючи операцію згортки - це робиться для того щоб регресійну та класифікаційну частину передбачати окремо в різних тензорах (див. рис. 2.38 [25]). Це є розумний хід, оскільки регресійна та класифікаційна частини є доволі різними по своїй природі функціями. До того ж враховується взаємозалежність шляхом вилучення ознак зі спільних карт ознак. Виглядає це таким чином (більш детально в роботі [25]):



(b) Task-aligned head (T-Head) (c) Task-aligned predictor (TAP)

Рисунок 2.39

З урахуванням розвитку в останні роки технологій нейромереж, які не використовують якірні рамки, доцільним є їх застосувати і в цій роботі як мінімум для економії кількості параметрів та часу тренування і таким чином досягненні кращих результатів збіжності з урахуванням вищеописаного нововведеного алгоритму оптимізації. До того ж робота нейромережі у випадку з якірними рамками занадто сильно залежить від якості методів кластеризація, які визначають ці якірні рамки. Варто зазначити, що в даному випадку класифікаційна частина нейромережі буде передбачати не ймовірності класів, а їх показники впевненості для кожного класу (тобто фактично це ймовірність певного класу помножена на IoU_{truth}^{pred}).

В якості основної частини даної версії YOLO використовуються різноманітні поєднання Rep блоків (див. рис. 2.40 [24]):

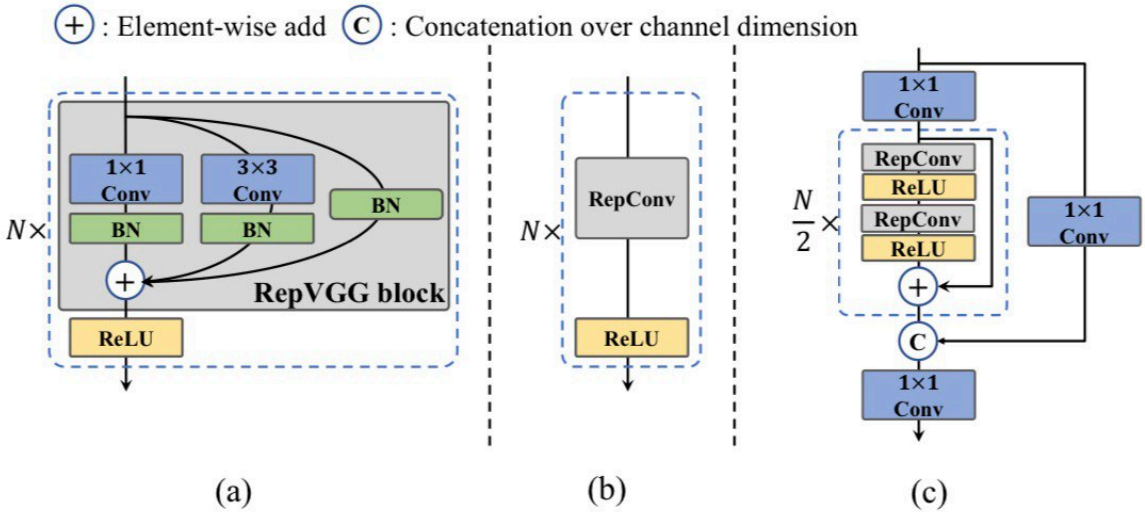


Рисунок 2.40

a, b - для малих нейромереж; c - використання стратегії CSPNet для середніх та великих нейромереж.

Загальна структура цієї нейромережі виглядає таким чином (див. рис. 2.41 [24]):

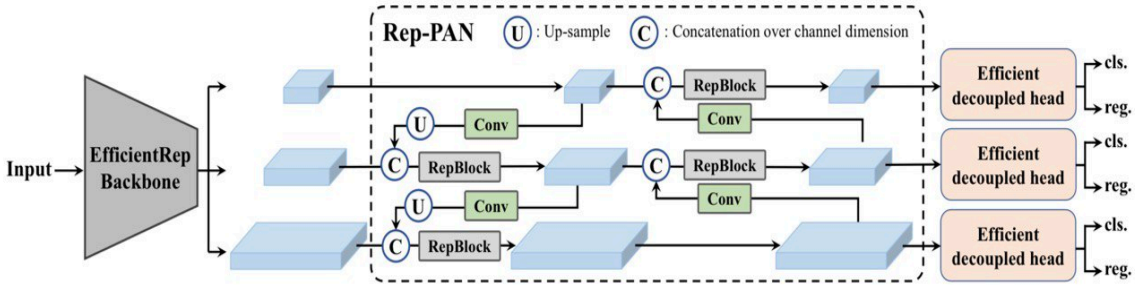


Рисунок 2.41

Так як ми вже не використовуємо якірні рамки, то тепер вхідне зображення розділяється на сітку клітин (центр кожної з цих клітин називається якірною точкою) і як для класифікаційного так і для регресійного виводу кожна клітина передбачає певні змінні того об’єкта за який відповідає. В регресійний частині передбачається відстані від центра відповідної клітини до 4 сторін обмежувальні рамки.

Для регресійної частини зазвичай для цієї версії YOLO використовується така IoU-функція втрат: $IoU - \frac{|C \setminus (A \cup B)|}{|C|}$, де C- найменша обмежувальна рамка, яка включає A і B.

Для класифікаційної частини застосовується така функція втрат (VariFocal loss) [24]:

$$\mathrm{VFL}(\mathrm{p},\mathrm{q}) = \begin{cases} -\mathrm{q}(\mathrm{q}\log(\mathrm{p}) + (1-\mathrm{q})\log(1-\mathrm{p})) & \mathrm{q} > 0 \\ -\alpha \mathrm{p}^{\gamma}\log(1-\mathrm{p}) & \mathrm{q} = 0, \end{cases} \quad (2.48)$$

q – IoU між передбаченням та реальністю, якщо p - показник впевненості класу, який вказаний у тренувальному екземплярі; q=0 в іншому випадку; p – показник впевненості класу, виведений нейромережею; γ- гіперпараметр, який контролює баланс між класами.

Застосовуються також деякий варіанти техніки самотренування. Вона полягає в тому що спочатку ми тренуємо модель з використанням двох попередніх функцій втрат, зберігаємо цю модель, а потім заново тренуємо модель, однак тепер вже із використанням трьох функцій втрат, третя з яких буде мати вигляд (вимірює різницю між розподілами) [24]:

$$L_{KD} = KL(p_t^{cls}||p_s^{cls}) + KL(p_t^{reg}||p_s^{reg}), \; L_{total} = L_{det} + \alpha L_{KD}, \quad (2.49)$$

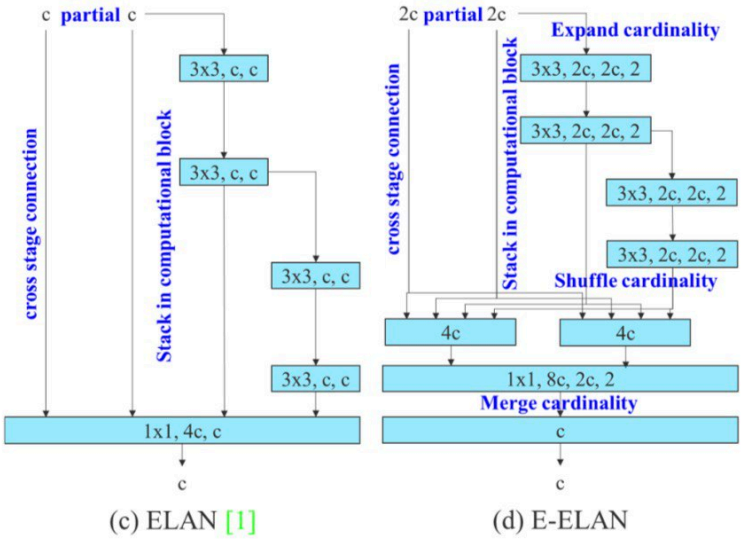
KL – розходження Кульбака-Лейблера (більш детально як її вводити для неперервних величин використовуючи функцію Дірака – робота [26]), L_{det} – лінійна комбінація двох функцій втрат описаних вище.

Нетренувальний параметр α поступово змешується при тренуванні так як згодом студент (нова не натренована модель) буде як мінімум на рівні вчителя (натренована модель), а з самого початку легше навчатися шляхом мінімізації різниці в розподілі з вчителем.

Мітки при тренуванні розподіляються таким чином: В околі центральної точки певного реального об’єкта вибирається декілька якірних точок з найвищими показниками впевненості класу даного об‘єкта. Ці якірні точки будуть позитивними по відношенню до даного екземпляру класу, усі інші - негативними.

2.11.7 Модель YOLO v7

В якості архітектури дана версія використовує удосконалену структуру ELAN - E-ELAN. Ця структура використовує операцію групової згортки для розширення каналу та збільшення потужності обчислювальних блоків і при цьому контролювати довжину найкоротшого градієнтного шляху. Структура ELAN в свою чергу намагається просто контролювати довжину найкоротшого градієнтного шляху в кожному шарі. Структура ELAN та E-LAN (див. рис. 2.42 [27]):



(c) ELAN [1] (d) E-ELAN

Рисунок 2.42

Як і в попередній версії YOLO v7 використовує RepConv блок, однак авторами [27] було досліджено, що третя гілка в цьому блоці, яка ніяк не діє на тензор, дає радше негативний ефект, тому її було викинуто і новоутворений блок тепер має назву RepConvN.

Також в цій версії було створено новий метод масштабування моделі із врахуванням того що при збільшенні глибини в моделях на основі конкатенації також збільшується і ширина.

При тренуванні використовується так звана додатково голова яка

слугує асистентом для тренування. Суть механізму у тому що справжня голова, яка відповідає за кінцевий вивід, під час тренування буде визначати які клітини є позитивними по відношенню до деякого екземпляра, а які - ні, при цьому для додаткової голови кількість позитивних клітин дозволена більшою ніж для основної голови. В результаті зворотне розповсюдження буде виконуватись 2 рази - для основної та додаткової голови. Схематичне зображення даного методу (див. рис. 2.43 [27]):

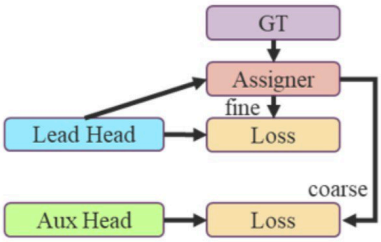


Рисунок 2.43

Так як остання версія YOLO v8 вийшла у січні цього року і ще немає офіційних робіт щодо неї, то відомо досить мало - це є модель, яка не використовує якірні рамки, в своїй архітектурі запозичила деякі елементи з п’ятої версії, зокрема, bottleneck та SPPF.

2.12 DeepSort

2.12.1 Звичайний Sort

Sort - це алгоритм для покадрового відслідковування об’єктів. В ідеальному варіанті: кожному об’єкту на кожному кадрі надається ідентифікатор таким чином щоб один і той самий об’єкт на різних кадрах мав один і той самий ідентифікатор. Виявленням об’єктів на кожному кадрі займається детектор.

Для реалізації процедури надання правильних ідентифікаторів

використовуються відстань Махаланобіса в квадраті та фільтр Калмана.

Метрика Махаланобіса (в квадраті) для двох точок (x та y) з багатовимірного простору працює таким чином [30]:

$$d(x,y) = (x-y)^T S^{-1}(x-y) \quad (2.50)$$

S - коваріаційна матриця деякого розподілу. Нехай, наприклад, x точка з цього розподілу тоді дана величина вказує наскільки дисперсій у віддалено від x, причому по кожному з напрямків дисперсій (під напрямком дисперсії мається на увазі напрямок одного з власних векторів матриці S, які утворюють базис, - по факту розкид даних відбувається саме в цих напрямках). Тобто якщо ця відстань є дуже малою, то це означає що 2 точки близькі як точки зору Евклідової відстані так і з точки зору розподілу, отже, скоріш за все ми таким чином 2 рази просто виміряли одну й ту саму величину однак другий раз на вимірювання повпливали деякі незначні похибки.

Фільтр Калмана уточнює деяку величину, за якою ми спостерігаємо з рахуванням власне самих спостережень та попередніх уточнень. Причому значення самої величини нам невідомі. Це відбувається по такій формулі [29]:

$$\hat{\mathbf{x}}(k) = \mathbf{A}(k)\hat{\mathbf{x}}(k-1) + \mathbf{K}(k)[\mathbf{z}(k) - \mathbf{H}(k)\mathbf{A}(k)\hat{\mathbf{x}}(k-1)]. \quad (2.51)$$

$\hat{\mathbf{x}}(k)$ - оцінка спостережуваного вектора x в момент k за спостереженнями z(k) = H(k)x(k) + noise, x(k) = A(k) + noise. Таким чином відбувається згладжування величини - її фільтрації від зайвого шуму та оцінка з урахуванням усієї наявної інформації. Матричний коефіцієнт підбирається таким чином щоб мінімізувати математичне очікування суму квадратів похибки реальної величини від виходу фільтра Калмана по всім координатам.

Для отримання рівнянь стану використовується модель постійної швидкості. Припускається що координат обмежувального рамки між часовими проміжками спостереження змінюється дуже повільно і тоді в розкладі в ряд Тейлора в точці t_{k-1} для точки t_k можна обмежитись другою похідною. Також припускається що 2-а похідна є нічим іншим як шумом. Використовуючи дані припущення отримуємо [28]:

$$x_{b,k} = x_{b,k-1} + \triangle T \dot{x}_{b,k-1} + \frac{\triangle T^2}{2} w_{x,k-1} \tag{2.52}$$

Використовуємо цю формулу для трьох інших координат, а також розкладаємо вже до першої похідної швидкість для кожної з чотирьох координат. Таким чином в якості спостережуваної величини у нас буде 8 координат, 4 з яких є швидкості зміни інших чотирьох координат; w_x – білий шум, тобто некорельований з попередніми вимірами та має Гаусівський розподіл. В спостереженнях z(k) будуть міститися лише 4 координати; z(k) = Hx(k) + noise, H = [I 0], I має розмір (4,4), H – (4,8). Більш детально про модель постійної швидкості та про фільтр Калмана в роботах [28] та [29] відповідно.

Алгоритм Sort працює таким чином: У нас є вихід фільтра Калмана для спостережуваних величини для минулого кадру. Далі використовуємо отримані вище рівняння стану щоб передбачити яким має бути спостереження на поточному кадрі. Після цього для кожного передбачення спостереження з минулого кадру підбираємо спостереження з поточного таким чином щоб між ними відстань Махаланобіса була найменшою застосовуючи Угорський алгоритм. При цьому ця відстань не повинна перевищувати деякого обмеження - зазвичай воно виводиться з використанням 95% довірчого інтервалу для випадкової величини з розподілу хі-квадрат.

2.12.2 Алгоритм DeepSort

В роботі [30] авторами алгоритму DeepSort пропонується таке нововведення до алгоритму Sort: В якості метрики будемо використовувати не тільки відстань Махаланобіса, а й метрику, яка буде вимірювати зовнішню схожість об’єктів. Для цього використовують натренована нейромережа для задачі класифікації, відкидається останній шар, який передбачає ймовірності, в якості зовнішньої характеристики кожного об’єкта буде виступати вихід

останнього щільного шару. Цей вихід також додатково нормується за нормою L2. Ця нейромережа також має назву дескриптор. Авторами [30] пропонується такий дескриптор (див. рис. 2.44 [30]):

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
Batch and ℓ_2 normalization		128

Рисунок 2.44

Відстань для 2 об’єктів (i та j) за цим дескриптором розраховується таким чином [30]:

$$\min\{1 - \boldsymbol{r}_j{}^{\mathrm{T}} \boldsymbol{r}_k^{(i)} \mid \boldsymbol{r}_k^{(i)} \in \mathcal{R}_i\} \tag{2.53}$$

R_i – зберігає останні 100 показників дескриптора для і-го об’єкта. Для цієї відстані так само водиться певне обмеження.

В якості остаточної відстані (cosine distance) приймається опукла комбінація відстані Махаланобіса та відстані за дескриптором [30]:

$$c_{i,j} = \lambda\, d^{(1)}(i,j) + (1-\lambda) d^{(2)}(i,j) \tag{2.54}$$

Також вводиться час життя для кожного об’єкта – якщо на A_{max} кадрах підряд не спостерігається об’єкт з певним ідентифікатором, то вважається що цей об’єкт зник з кадру і його ідентифікатор більше не використовується для цього відео файлу. λ та A_{max} – гіперпараметри алгоритму DeepSort.

Висновки до розділу 2

В даному розділі було викладеного необхідний матеріал для загального ознайомлення зі згортковими нейромережами та деякими особливостями їх тренування, а також побудови, яка сприяє кращий збіжності під час тренування (наприклад skip connections). Далі було проаналізовано основні види детекторів - однорівневі та дворівневі. В якості однорівневих були розглянуті моделі із сімейства YOLO, які шукають компроміс між швидкістю та якістю роботи. Саме тому вони є дуже ефективними при виявленні та відслідковуванні об’єктів в онлайн режимі. Як приклад дворівневівих детекторів були взяті моделі із сімейства R-CNN. Вони відрізняються гарною точністю, однак є занадто повільними для застосування в реальному часі. Для кожної з цих моделей є свої нюанси пов’язані зі зіставлення міток при тренуванні, вибором функції активації, архітектури, а також вибором функцій втрат. В кінці цього розділу було розглянуто один з найпопулярніших алгоритмів для відслідковування об‘єктів - DeepSort. Основна його особливість в тому що він використовує додаткову згорткову нейромережу для отримання вектора зовнішності для кожного об’єкта.

3 ПРАКТИЧНІ РЕЗУЛЬТАТИ

3.1 Вибір мови програмування, фреймворків та додаткових програм

У якості мови програмування буде використовуватись Python, оскільки він містить такі пакети як numpy, matplotlib та pandas, які дають мінімально необхідний інструментарій для роботи з нейронними мережами. Так як усі версії YOLO, які будуть розглядатися в практичній частині, були написані та натреновані за допомогою фреймворку Pytorch, то візьмемо його в якості нашого фреймворку. Він є надзвичайно поширеним при роботі з нейромережами так як автоматизує багато алгоритмів пов’язаних з навчанням нейромереж та імплементацію найбільш відомих шарів. В цьому плані він схожий з Tensorflow, однак деякі дослідники зазначають, що в окремих випадках він є дещо швидшим в роботі.

Відео доволі не зручно зберігати просто у послідовності фотографій зображень так як це займає більше місця, до того ж часто буває потрібно як мінімум візуально оцінити специфіку роботи певного алгоритму. Однак саме в такій формі у нас буде датасет для тестування алгоритму DeepSort, тому також буде використовуватись програма Anchorpoint для склеєння всіх цих фотографій в один відеоряд. Швидкість відтворення відео при цьому буде 25 fps.

3.2 Вибір наборів даних

Для тренування та тестування нам необхідні будуть різні набори даних з відповідними мітками. Зокрема, для тестування детекторів буде використовуватись COCO датасет, він був введений у 2014 році. Інші відомі набори даних для тренування детекторів це VOC, однак порівняно з COCO він

містить в рази менше корисних даних, це можна побачити на відповідній діаграмі (кількість екземплярів на категорію) та графіку (див. рис. 3.1 та 3.2 відповідно) :

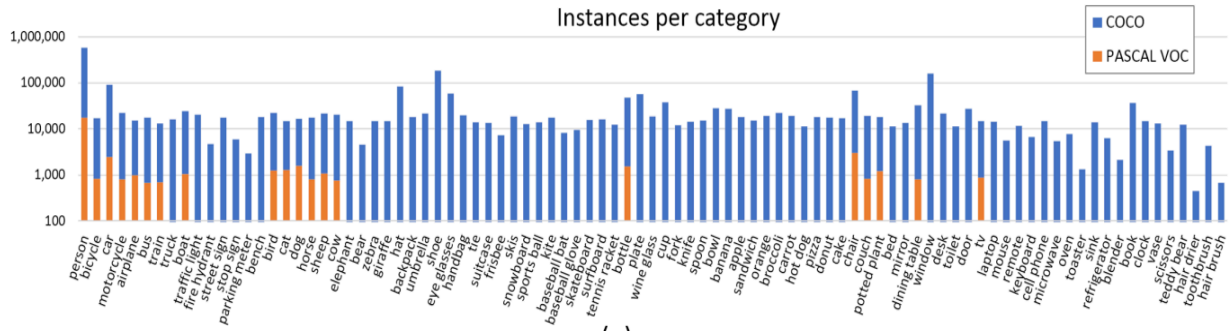


Рисунок 3.1

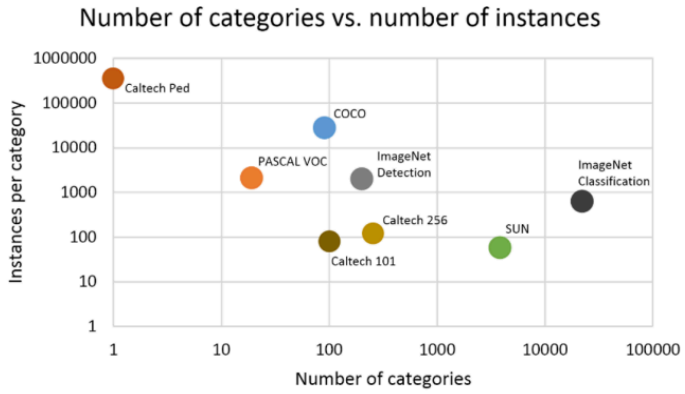


Рисунок 3.2

COCO датасет вмістить порядку 320 000 зображень для задач виявлення об’єктів, які можна використовувати для навчання та тестування детекторів причому навчальний набір займає 50 відсотків від цієї вибірки, валідаційний – 25 та тестовий - 25. На відміну від нього VOC містить лише 11 000 зображень для тренування детекторів.

В ідеалі дескриптор в алгоритмі DeepSort повинен видавати для кожного об’єкта унікальний вектор ознак, тому коли ми формуємо датасет для його тренування то по суті ми робимо багато знімків різних людей і кожній людині ставимо у відповідність унікальний ідентифікатор і навчаємо дескриптор на задачі класифікації таким чином щоб він міг чітко класифікувати до якого ідентифікатору належить об’єкт, який подається на

вхід дескриптору. Саме для цього і були сформовані такі дата сети як Mars та Market1501. В оригінальній версії алгоритму DeepSort дескриптор був натренований на Mars датасеті, який містить 1 100 000 зображень близько 1700 людей з різних ракурсів. Market1501 в принципі виконує ту саму задачу, однак зображення вже бралася в інших місцях і в меншій кількості, таким чином його більш реально використовувати для тестування не маючи достатньо потужних технічний засобів. Відео та мітки для тестування кінцевої моделі (реалізація алгоритму DeepSort в комбінації з певним детектором) будуть братися з MOT16 Challenge. Для тестування якості наших моделей буде використовуватись декілька відео з датасету MOT17.

3.3 Розроблений програмний продукт

Даний ПП було розроблено з використанням кодів на гітхабі для реалізації алгоритму DeepSort [32] та моделей yolo [33, 34, 35, 36, 23]. Власноруч було скомбіновано усі ці коди для формування єдиної програми, в якій можна простим чином вибирати з яким саме детектором буде взаємодіяти DeepSort для обробки обраного відео. Також було створено алгоритми для тестування моделей yolo та кінцевої моделі виявлення та відслідковування об’єктів. Для тестування також додатково використовувався ресурс [42]. До того ж прописано код для тренування різних дескрипторів щоб перевірити чи дає оригінальний дескриптор найбільш якісні результати.

Кінцевий програмний продукт може використовуватись для вирішення задач виявлення та відслідковування об’єктів в різних сферах людської діяльності як це було описано в розділі 1. При цьому варто зазначити що реально ефективної роботи алгоритму бажано мати дані специфічні для вашої місцевості або виду діяльності щоб на них дотренерувати детектор в fkujhsnvb DeepSort.

3.4 Вибір детекторів

Будемо ставити собі в завдання побудувати таку модель, яка здатна виявляти та відслідковувати об’єкти в реальному часі, а тому вибір детекторів проводитиметься серед моделей YOLO. До того ж моделі YOLO версії 4 не буде розглядатися так як за показниками якості вона є не кращою за 5-у версію але є значно більш громіздкою і відносно інших версій YOLO вона давно не оновлювалась (навіть 3-я версія останні роки повсякчас іноді вдосконалюється). За аналогічними судженнями не буде розглядатися модель Faster R-CNN (після довгого тренування вона може дати гарні результати, однак у нас відсутнє технічне обладнання для такого довгого та складного з обчислювальної точки зору тренування, до того ж в будь-якому разі це є занадто повільна модель і в контексті швидкості вона не може конкурувати з моделями yolo). Таким чином враховуючи що з третьої версії починається використовуватись потужна основа (backbone) з використанням skip connections - будуть розглядатися такі версії YOLO: 3, 5, 6, 7 та 8. У кожної з моделей починаючи з п’ятої версії є по декілька модифікацій на одну версію - таким чином загальна кількість нейронних мереж, які будуть проходити валідацію, сягає 19. Даний процес буде відбуватися на валідаційному COCO датасеті, кількість зображень в якому = 5000, однак в цій роботі буде використовуватися лише 1700 зображень з цього набору даних щоб не перенавантажувати нашу систему з обмеженими ресурсами. Тестування проводиться на сервісі Colab з використанням відеокарти Nvidia Tesla T4 на мові програмування Python за допомогою фреймворку PyTorch. Враховуючи розміри та кількість даних, на яких повинні бути натреновані моделі, щоб досягти гарних показників якості моделі YOLO брались вже натренованими.

При навчанні здебільшого використовувався алгоритм Adam, гіперпараметри для тренування здебільшого встановлювались з використанням генетичних алгоритмів (початкова ініціалізація відбувалася з

використанням значень гіперпараметрів за замовчуванням), до таких гіперпараметрів належить, наприклад, швидкість навчання, моменти 1-го та 2-го порядків (Adam), а також weight decay.

Для yolo v3 маємо такі результати тестування (див. рис. 3.3):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov3	0.648	0.463	42.918	37.175	119.0	640.0

Рисунок 3.3

Після тестування yolo v5 отримали такі результати (див. рис. 3.4):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov3	0.648	0.463	42.918	37.175	119.0	640.0
yolov5s	0.563	0.376	192.308	119.048	14.1	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov5l	0.664	0.485	46.296	39.683	89.3	640.0
yolov5x	0.681	0.503	24.331	22.831	166.0	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov5l6	0.709	0.533	11.086	10.661	147.0	1280.0

Рисунок 3.4

Видаляємо модель yolov3 так як модель yolov5l досягає кращого результату при меншому розмірі та при більшій швидкості (39 проти 37 fps). Також приберемо з розгляду модель yolov5s, як модель з надто низькими показниками якості map50 та map50-95, та модель yolov5x, розмір якої не відповідає показниками якості, - yolov5m6 досягає кращого результату при набагато меншому розмірі і незначному падінню в швидкості (швидкість падає через те що цю модель необхідно застосувати до зображень розміром 1280*1280 пікселів - за необхідності перед застосуванням yolov5m6 зображення форматують до відповідного розміру).

Після тестування yolov6 та видалення усіх зайвих моделей отримали такі результати (див. рис. 3.5):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov5l	0.664	0.485	46.296	39.683	89.3	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov5l6	0.709	0.533	11.086	10.661	147.0	1280.0
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov6m	0.651	0.486	70.423	63.694	72.6	640.0
yolov6l	0.682	0.511	43.668	41.152	114.0	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov6l_mbla	0.677	0.506	53.476	49.751	88.7	640.0
yolov6x_mbla	0.692	0.520	33.670	32.154	151.0	640.0

Рисунок 3.5

Для більшої наочності відсортуємо цю таблицю за зростанням метрики map50 (так як вона по суті грає основну роль при виборі детектора) та отримаємо (див. рис. 3.6):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov6m	0.651	0.486	70.423	63.694	72.6	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov5l	0.664	0.485	46.296	39.683	89.3	640.0
yolov6l_mbla	0.677	0.506	53.476	49.751	88.7	640.0
yolov6l	0.682	0.511	43.668	41.152	114.0	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov6x_mbla	0.692	0.520	33.670	32.154	151.0	640.0
yolov5l6	0.709	0.533	11.086	10.661	147.0	1280.0

Рисунок 3.6

Викинемо модель yolov5l так як yolov6l_mbla досягає кращих результатів за значно більшої швидкості (49.8 проти 39.7 fps) і має при цьому менший розмір. За аналогічних причин приберемо з розгляду модель yolov6m порівнюючи її з yolov6m_mbla. Після відкидання усіх недоцільних моделей з вищеописаних причин та тестування yolo v7 маємо таку відсортовану таблицю (див. рис. 3.7):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov6l_mbla	0.677	0.506	53.476	49.751	88.7	640.0
yolov6l	0.682	0.511	43.668	41.152	114.0	640.0
yolov7	0.684	0.496	98.039	86.957	72.1	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov6x_mbla	0.692	0.520	33.670	32.154	151.0	640.0
yolov7x	0.698	0.512	54.054	50.505	136.0	640.0
yolov5l6	0.709	0.533	11.086	10.661	147.0	1280.0
yolov7w6	0.715	0.528	34.247	32.154	134.7	1280.0

Рисунок 3.7

Викинемо модель yolov5l6 так як yolov7w6 випереджає її з точки зору усіх критеріїв - краща якість за map, більша швидкість, менша кількість параметрів (тобто розмір з точки зору розташування в пам'яті комп'ютера). За аналогічних суджень прибераємо модель yolov6x_mbla порівнюючи її з yolov7x. Модель yolov7 досягає приблизно таких самих результатів як і yolov6l але при цьому займає в 2 рази менше місця в пам'яті і є десь в стільки ж разів швидшою, тому викинемо yolov6l. Так само порівнюючи з yolov7 приберемо і yolov6l_mbla.

Після відкидання усіх недоцільних моделей з вищеописаних причин та тестування yolo v8 маємо (див. рис. 3.8):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov8m	0.659	0.497	64.103	57.803	49.7	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov7	0.684	0.496	98.039	86.957	72.1	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov8l	0.688	0.528	37.736	34.722	83.7	640.0
yolov7x	0.698	0.512	54.054	50.505	136.0	640.0
yolov8x	0.702	0.538	22.321	21.368	131.0	640.0
yolov7w6	0.715	0.528	34.247	32.154	134.7	1280.0

Рисунок 3.8

Приберемо модель yolov8x так як вона є занадто повільною у порівнянні з аналогічною по розміру і вищій по якості yolov7w6. За аналогічних причин приберемо модель yolov8m. Остаточний список моделей матиме такий вигляд (див. рис. 3.9):

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov7	0.684	0.496	98.039	86.957	72.1	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov8l	0.688	0.528	37.736	34.722	83.7	640.0
yolov7x	0.698	0.512	54.054	50.505	136.0	640.0
yolov7w6	0.715	0.528	34.247	32.154	134.7	1280.0

Рисунок 3.9

Даний процес тестування також можна зобразити за допомого графіків таким чином (див. рис. 3.10 та 3.11):

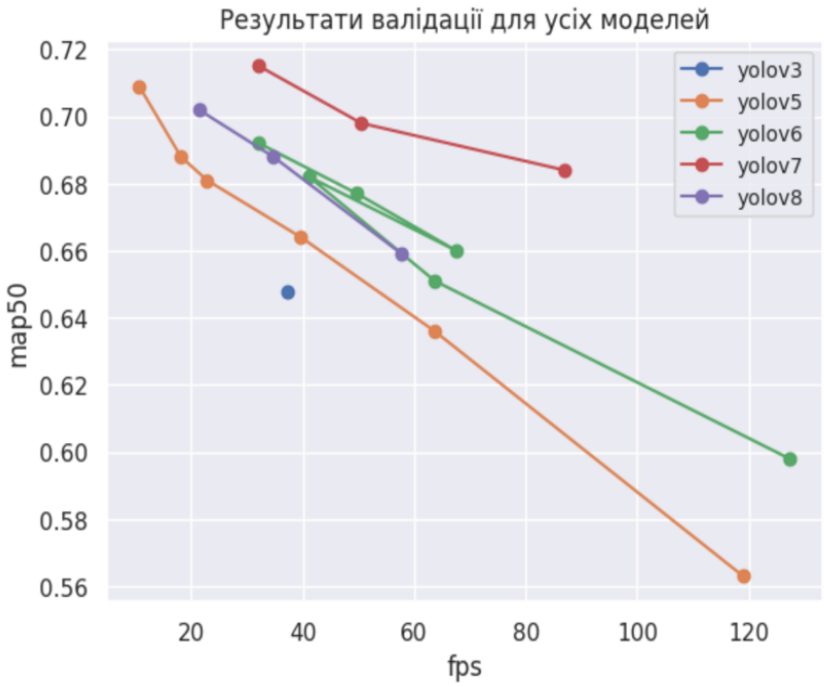


Рисунок 3.10

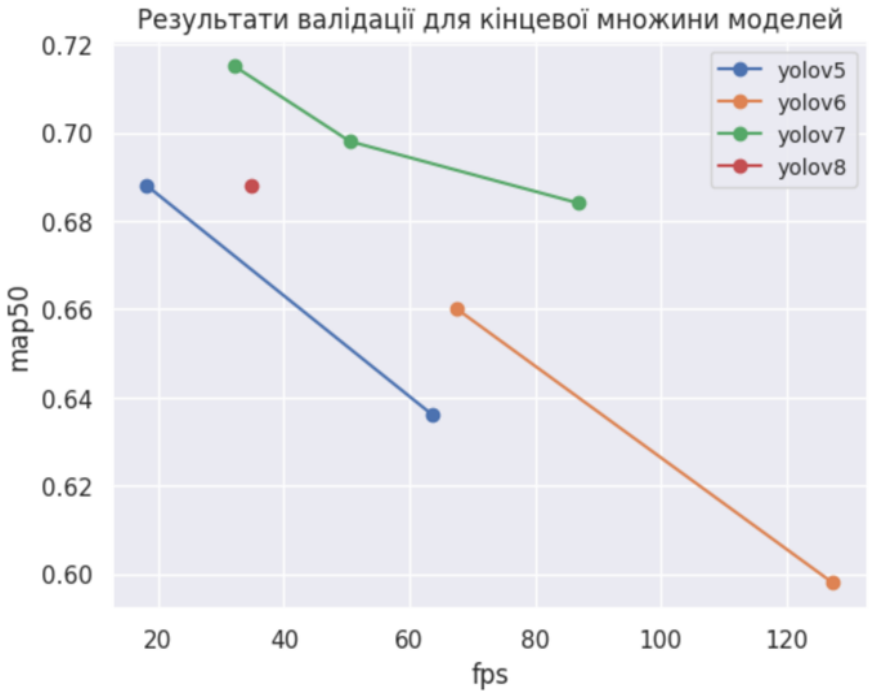


Рисунок 3.11

3.5 Тестування алгоритму DeepSort

3.5.1 Метрики MOTA

Для тестування, як вже частково згадувалося, буде використовуватись три відео з MOT17 Dataset – MOT17-02-DPM, MOT17-04-FRCNN, MOT-13-FRCNN. До того ж виявлення та відслідковування буде проводитись тільки для людей з двох причин: по-перше щоб не перенавантажувати нашу систему, так як все ж таки в нашому розпорядженні дуже обмежені ресурси; по-друге, COCO dataset містить найбільше екземплярів саме людей, тому можливо в цьому випадку моделі YOLO будуть працювати найбільш ефективним чином. В якості детекторів будуть використовуватись ті детектори, які ми відібрали у минулому пункті. За метрики якості будемо брати метрики MOTA.

Також було проведено декілька серій експериментів, в ході яких встановлено, що одними з найбільш оптимальних гіперпараметрів для DeepSort є максимальна cosine відстань = 0.3, вклад метрики Махаланобіса є

нульовим, однак вона бере участь у відкиданні непридатних спостережень (тобто використовується її граничне значення) – за безпосередню відстань між виходом детектора та проекцією фільтра Калмана відповідає вектор ознак, який формується дескрипторм. Використаємо вже натренований дескриптор з оригінальної моделі, тобто той що використовувався авторами самого алгоритму DeepSort.

Residual і позначає такий блок (див. рис. 3.12):

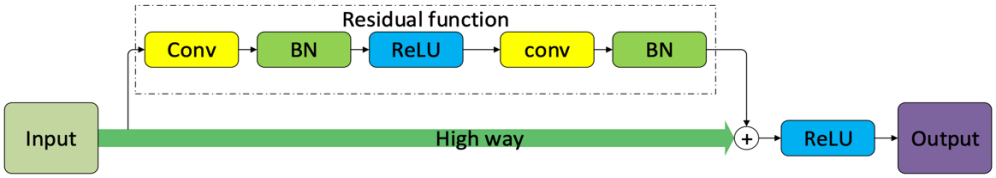


Рисунок 3.12

Результати тестування алгоритму DeepSort з yolov5m (див. рис. 3.13):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.166460	0.731199	3330	15251	213	6	45	3543	18581	38	62	21.30000
1	MOT17-04-FRCNN	0.192085	0.747781	9996	37561	826	5	55	10822	47557	53	83	19.40000
2	MOT17-13-FRCNN	0.197475	0.704500	2648	8994	320	8	71	2968	11642	63	110	23.50000
3	COMBINED	0.186770	0.737150	15974	61806	1359	19	171	17333	77780	154	255	21.15625

Рисунок 3.13

Результати тестування алгоритму DeepSort з yolov5m6 (див. рис. 3.14):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.216350	0.727750	4310	14271	261	7	40	4571	18581	50	62	15.30000
1	MOT17-04-FRCNN	0.261286	0.751465	13665	33892	1198	6	48	14863	47557	60	83	14.00000
2	MOT17-13-FRCNN	0.267222	0.698807	3471	8171	320	10	62	3791	11642	75	110	16.70000
3	COMBINED	0.251440	0.738176	21446	56334	1779	23	150	23225	77780	185	255	15.16875

Рисунок 3.14

Результати тестування алгоритму DeepSort з yolov6s (див. рис. 3.15):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.204079	0.721698	4305	14276	485	7	40	4790	18581	53	62	21.1000
1	MOT17-04-FRCNN	0.348887	0.743327	18407	29150	1731	8	37	20138	47557	88	83	16.7000
2	MOT17-13-FRCNN	0.240337	0.697898	3342	8300	510	11	60	3852	11642	74	110	24.1000
3	COMBINED	0.298046	0.733926	26054	51726	2726	26	137	28780	77780	215	255	20.1125

Рисунок 3.15

Результати тестування алгоритму DeepSort з yolov6m_mbla (див. рис. 3.16):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.220548	0.721378	5407	13174	1255	6	37	6662	18581	74	62	16.00000
1	MOT17-04-FRCNN	0.381773	0.740098	21319	26238	3067	11	32	24386	47557	93	83	12.70000
2	MOT17-13-FRCNN	0.311630	0.692165	4372	7270	658	12	50	5030	11642	109	110	19.20000
3	COMBINED	0.332759	0.730104	31098	46682	4980	29	119	36078	77780	276	255	15.55625

Рисунок 3.16

Результати тестування алгоритму DeepSort з yolov7 (див. рис. 3.17):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.220763	0.724370	4668	13913	531	6	38	5199	18581	55	62	16.60000
1	MOT17-04-FRCNN	0.283218	0.751338	14759	32798	1236	5	43	15995	47557	62	83	15.00000
2	MOT17-13-FRCNN	0.296770	0.703086	3827	7815	337	14	57	4164	11642	82	110	18.30000
3	COMBINED	0.270327	0.737984	23254	54526	2104	25	138	25358	77780	199	255	16.43125

Рисунок 3.17

Результати тестування алгоритму DeepSort з yolov7_w6 (див. рис. 3.18):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.195576	0.727244	3908	14673	251	7	42	4159	18581	45	62	16.9
1	MOT17-04-FRCNN	0.201569	0.751234	10549	37008	930	3	56	11479	47557	50	83	16.3
2	MOT17-13-FRCNN	0.242570	0.698421	3165	8477	314	10	66	3479	11642	67	110	18.7
3	COMBINED	0.206274	0.736429	17622	60158	1495	20	164	19117	77780	162	255	17.2

Рисунок 3.18

Результати тестування алгоритму DeepSort з yolov7x (див. рис. 3.19):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.251547	0.724397	5008	13573	289	6	38	5297	18581	57	62	13.0000
1	MOT17-04-FRCNN	0.272473	0.753904	14320	33237	1320	5	44	15640	47557	62	83	11.7000
2	MOT17-13-FRCNN	0.319017	0.702659	4095	7547	344	14	56	4439	11642	87	110	13.5000
3	COMBINED	0.274441	0.738636	23423	54357	1953	25	138	25376	77780	206	255	12.5875

Рисунок 3.19

Результати тестування алгоритму DeepSort з yolov8l (див. рис. 3.20):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
0	MOT17-02-DPM	0.194715	0.736298	3808	14773	166	6	45	3974	18581	42	62	15.40000
1	MOT17-04-FRCNN	0.202746	0.756160	10565	36992	895	5	55	11460	47557	52	83	14.60000
2	MOT17-13-FRCNN	0.247638	0.710353	3213	8429	298	9	64	3511	11642	74	110	16.10000
3	COMBINED	0.207547	0.743490	17586	60194	1359	20	164	18945	77780	168	255	15.26875

Рисунок 3.20

Загальні результати тестування (див. рис. 3.21):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
yolov5m	COMBINED	0.186770	0.737150	15974	61806	1359	19	171	17333	77780	154	255	21.15625
yolov7-w6	COMBINED	0.206274	0.736429	17622	60158	1495	20	164	19117	77780	162	255	17.20000
yolov8l	COMBINED	0.207547	0.743490	17586	60194	1359	20	164	18945	77780	168	255	15.26875
yolov5m6	COMBINED	0.251440	0.738176	21446	56334	1779	23	150	23225	77780	185	255	15.16875
yolov7	COMBINED	0.270327	0.737984	23254	54526	2104	25	138	25358	77780	199	255	16.43125
yolov7x	COMBINED	0.274441	0.738636	23423	54357	1953	25	138	25376	77780	206	255	12.58750
yolov6s	COMBINED	0.298046	0.733926	26054	51726	2726	26	137	28780	77780	215	255	20.11250
yolov6m_mbla	COMBINED	0.332759	0.730104	31098	46682	4980	29	119	36078	77780	276	255	15.55625

Рисунок 3.21

Отже, DeepSort є доволі ефективним з точки зору метрики MOTP – середнього показника IoU між передбаченою обмежувальної рамкою та справжньою. Однак метрика MOTA є досить низькою – не піднімається вище 35 відсотків. Причина такої роботи алгоритму ховається в його головному недоліку – відсутності залежності дескриптора та детектора (більш детально у висновках).

3.5.2 Метрики ідентифікації

Загалом існують декілька класів метрик для оцінки якості моделей виявлення та відслідковування. Метрики MOTA не враховують того факту, що трекер міг лише на пару кадрів втратити об’єкт, а потім повернутися до його відслідковування, оскільки як відхилення так і саме повернення до відслідковування правильного об’єкта він рахує як помилку, і таким чином по суті трекер ніяк не винагороджується з точки зору метрики за повернення до свого найбільш частого стану, що і означає відслідковування правильного об’єкта. Головна метрика в даному класі метрик - IDF1 (identification F1-score). Вона для кожної передбаченої траєкторії підраховує кількість кадрів, на яких передбачений ідентифікатор був на правильному місці (тобто належав об'єкту за яким слідкував найбільшу частину своєї траєкторії) по відношенню до загальної кількості зроблених прогнозів та наявних справжніх об'єктів. Загалом чим ближче дана метрика до 1 тим меншу кількість кадрів трекер відволікався на сторонні об'єкти. Також вводяться метрики IDP та IDR (indetification precision та recall). В даному випадку IDTP - це ті передбачення, які відповідають за правильний об'єкт згідно з принципом описаним вище, IDFP - це усі інші передбачення, а IDFN - це усі інші правдиві об'єкти, які не були захоплені передбаченнями із IDTP.

Результати тестування алгоритму DeepSort з yolov5m (див. рис. 3.22):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.230790	2553	16028	990	3543	18581	38	62
1	MOT17-04-FRCNN	0.293633	8571	38986	2251	10822	47557	53	83
2	MOT17-13-FRCNN	0.310609	2269	9373	699	2968	11642	63	110
3	COMBINED	0.281623	13393	64387	3940	17333	77780	154	255

Рисунок 3.22

Результати тестування алгоритму DeepSort з yolov5m6 (див. рис. 3.23):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.282999	3276	15305	1295	4571	18581	50	62
1	MOT17-04-FRCNN	0.359116	11208	36349	3655	14863	47557	60	83
2	MOT17-13-FRCNN	0.374522	2890	8752	901	3791	11642	75	110
3	COMBINED	0.344023	17374	60406	5851	23225	77780	185	255

Рисунок 3.23

Результати тестування алгоритму DeepSort з yolov6s (див. рис. 3.24):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.292243	3415	15166	1375	4790	18581	53	62
1	MOT17-04-FRCNN	0.423015	14318	33239	5820	20138	47557	88	83
2	MOT17-13-FRCNN	0.362850	2811	8831	1041	3852	11642	74	110
3	COMBINED	0.385586	20544	57236	8236	28780	77780	215	255

Рисунок 3.24

Результати тестування алгоритму DeepSort з yolov6m_mbla (див. рис. 3.25):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.327378	4132	14449	2530	6662	18581	74	62
1	MOT17-04-FRCNN	0.484050	17412	30145	6974	24386	47557	93	83
2	MOT17-13-FRCNN	0.442059	3685	7957	1345	5030	11642	109	110
3	COMBINED	0.443166	25229	52551	10849	36078	77780	276	255

Рисунок 3.25

Результати тестування алгоритму DeepSort з yolov7 (див. рис. 3.26):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.325315	3868	14713	1331	5199	18581	55	62
1	MOT17-04-FRCNN	0.388029	12330	35227	3665	15995	47557	62	83
2	MOT17-13-FRCNN	0.407061	3217	8425	947	4164	11642	82	110
3	COMBINED	0.376486	19415	58365	5943	25358	77780	199	255

Рисунок 3.26

Результати тестування алгоритму DeepSort з yolov7_w6 (див. рис. 3.27):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.289182	3288	15293	871	4159	18581	45	62
1	MOT17-04-FRCNN	0.293651	8668	38889	2811	11479	47557	50	83
2	MOT17-13-FRCNN	0.355268	2686	8956	793	3479	11642	67	110
3	COMBINED	0.302218	14642	63138	4475	19117	77780	162	255

Рисунок 3.27

Результати тестування алгоритму DeepSort з yolov7x (див. рис. 3.28):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.350281	4182	14399	1115	5297	18581	57	62
1	MOT17-04-FRCNN	0.388468	12275	35282	3365	15640	47557	62	83
2	MOT17-13-FRCNN	0.438032	3522	8120	917	4439	11642	87	110
3	COMBINED	0.387355	19979	57801	5397	25376	77780	206	255

Рисунок 3.28

Результати тестування алгоритму DeepSort з yolov8l (див. рис. 3.29):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs
0	MOT17-02-DPM	0.288450	3253	15328	721	3974	18581	42	62
1	MOT17-04-FRCNN	0.322958	9530	38027	1930	11460	47557	52	83
2	MOT17-13-FRCNN	0.362041	2743	8899	768	3511	11642	74	110
3	COMBINED	0.321034	15526	62254	3419	18945	77780	168	255

Рисунок 3.29

Загальні результати тестування (див. рис. 3.30):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs	fps
yolov5m	COMBINED	0.281623	13393	64387	3940	17333	77780	154	255	21.15625
yolov7-w6	COMBINED	0.302218	14642	63138	4475	19117	77780	162	255	17.20000
yolov8l	COMBINED	0.321034	15526	62254	3419	18945	77780	168	255	15.26875
yolov5m6	COMBINED	0.344023	17374	60406	5851	23225	77780	185	255	15.16875
yolov7	COMBINED	0.376486	19415	58365	5943	25358	77780	199	255	16.43125
yolov6s	COMBINED	0.385586	20544	57236	8236	28780	77780	215	255	20.11250
yolov7x	COMBINED	0.387355	19979	57801	5397	25376	77780	206	255	12.58750
yolov6m_mbla	COMBINED	0.443166	25229	52551	10849	36078	77780	276	255	15.55625

Рисунок 3.30

Отже, ситуація дещо покращилася для метрик ідентифікації, при цьому найкращими залишаються ті ж самі моделі – yolov6. Yolov7x можна не враховувати так як вона значно поступається у швидкості для yolov6s.

3.6 Тестування алгоритму DeepSort з дескриптором – resnet50

3.6.1 Тренування нового дескриптора

Спробуємо поекспериментувати з іншим дескрипором - resnet50. Це є перша нейромережа, яка почала ефективно використовувати skip connections, її структура має такий вигляд (див. рис. 3.31):

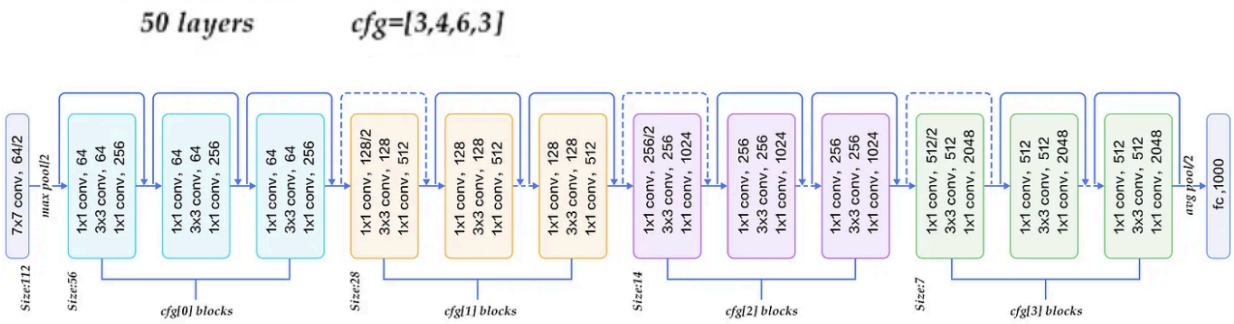


Рисунок 3.31

Вона більша за розмірами ніж оригінальний дескриптор, однак вона в кожному згортковому шарі вона використовує 2 згортки 1 на 1, що в свою чергу економить кількість параметрів без особливої втрати репрезентативності даної нейромережі.

Тренування здійснювалося на датасеті Market1501, з якого було взято близько 13 000 тренувальних зображень та 3 300 – валідаційних. Перед тренуванням був змінений останній шар, який відповідає за класифікацію так як в нашому випадку у валідаційному наборі даних буде міститися 750 класів. Після тренування цей останній шар було замінено на звичайну L2-нормалізацію як і у випадку з оригінальним дескриптором. Також варто зазначити що для тренування використовувався звичайний метод стохастичного градієнту зі швидкістю навчання = 0.1, яка кожні 20 епох зменшується в 10 раз. Усього тренування тривало 40 епох. Саме тренування та його результати гарно відображаються на цьому графіку (на наступній

сторінці – див. рис. 3.32):

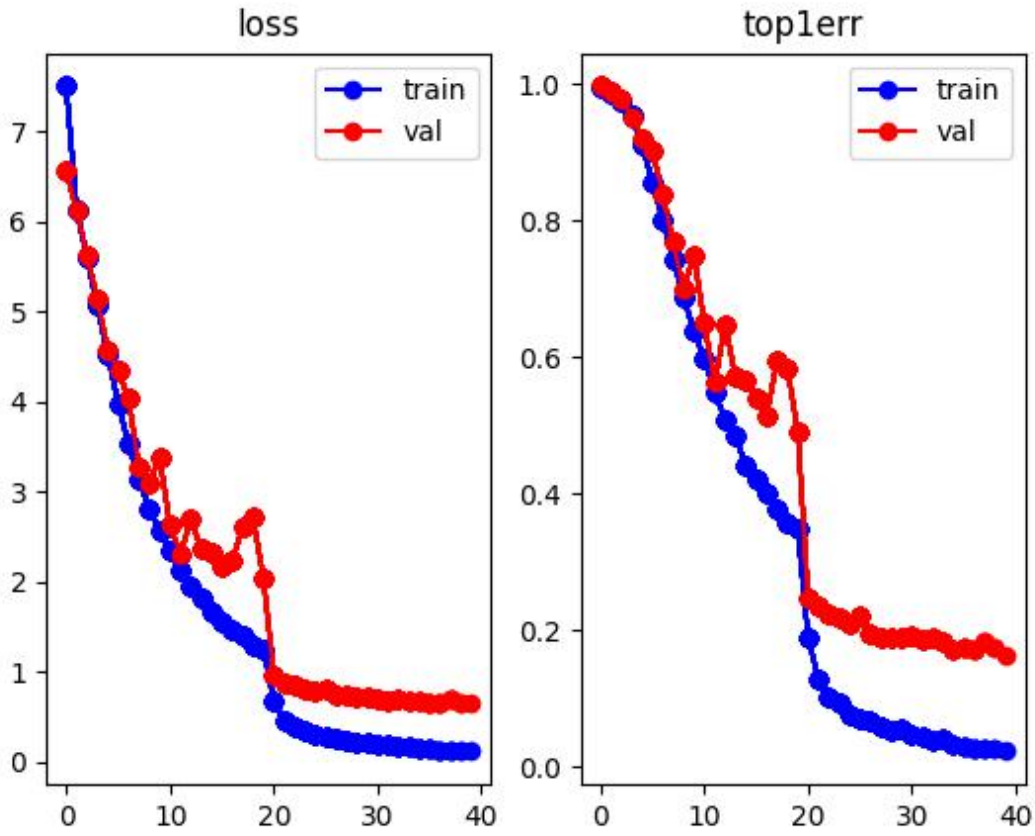


Рисунок 3.32

Top1err – це просто (1-accuracy) на кінець епохи. Розмір пакету = 64.

3.6.2 Метрики якості з новим дескриптором

Проведемо тестування на тих же метриках що і з минулим дескриптором та отримаємо:

Метрики MOTA (див. рис. 3.33):

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
yolov7-w6	COMBINED	0.207187	0.735007	17822	59958	1596	20	161	19418	77780	102	255	12.38750
yolov8l	COMBINED	0.210607	0.742068	17939	59841	1433	21	161	19372	77780	100	255	11.43750
yolov5m6	COMBINED	0.253394	0.736692	21739	56041	1875	24	145	23614	77780	117	255	11.33125
yolov5m	COMBINED	0.264297	0.734829	22763	55017	2042	23	145	24805	77780	119	255	13.68125
yolov7	COMBINED	0.272821	0.736639	23629	54151	2231	27	138	25860	77780	116	255	11.50000
yolov7x	COMBINED	0.275534	0.736848	23757	54023	2135	27	134	25892	77780	119	255	9.60000
yolov6s	COMBINED	0.298676	0.732317	26408	51372	2936	26	138	29344	77780	128	255	13.02500
yolov6m_mbla	COMBINED	0.333209	0.730205	31597	46183	5364	34	113	36961	77780	144	255	10.60000

Рисунок 3.33

Метрики ідентифікації (див. рис. 3.34):

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs	fps
yolov7-w6	COMBINED	0.274327	13332	64448	6086	19418	77780	102	255	12.38750
yolov8l	COMBINED	0.281888	13693	64087	5679	19372	77780	100	255	11.43750
yolov5m6	COMBINED	0.318066	16125	61655	7489	23614	77780	117	255	11.33125
yolov7x	COMBINED	0.330581	17136	60644	8756	25892	77780	119	255	9.60000
yolov5m	COMBINED	0.331277	16992	60788	7813	24805	77780	119	255	13.68125
yolov7	COMBINED	0.334794	17349	60431	8511	25860	77780	116	255	11.50000
yolov6s	COMBINED	0.343079	18376	59404	10968	29344	77780	128	255	13.02500
yolov6m_mbla	COMBINED	0.393722	22588	55192	14373	36961	77780	144	255	10.60000

Рисунок 3.34

Бачимо що за усіма метриками кардинально ситуація не змінилася, лише для моделі yolov5m якість відчутно підвищилася з 18 до 26 відсотків з метрикою MOTA та з 28 до 33 відсотків за метрикою IDF1. Однак все одно загальна якість по всім моделям не змінилася і ті моделі, які були найкращими, такими і залишилися, хоча як можна побачити порівнюючи остаточні таблиці результатів для двох дескрипторів – у випадку з resnet50 алгоритм працює занадто повільно. Отже, можна зробити висновок, що одним з найбільш оптимальних виборів дескриптора є саме той, який був запропонований авторами алгоритму DeepSort.

Висновки до розділу 3

Найбільш ефективним чином з алгоритмом DeepSort взаємодіють моделі yolo версії 6. Алгоритм DeepSort відмітає деякі передбачення зроблені детектором і таким чином в загальному випадку він занижує показник повноти. Відмінність шостої версії полягає у тому що у нього дуже мала різниця між повнотою та точністю і обидва зберігаються на доволі високому рівні, тому навіть при взаємодії з DeepSort він дає найкращий показник по усім метрикам. Дамо, наприклад, середні показники точності та повноти для шостої версії yolov6s (див. рис. 3.35):

```
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.598
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.475
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.164
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.415
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.601
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.345
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.569
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.619
```

Рисунок 3.35

Така негативна поведінка алгоритму DeepSort пов’язана з тим що дескриптор під час тренування по суті ніяким чином не встановлює зв’язок з детектором. Схожа проблема була в моделях R-CNN при виявленні об’єктів і Faster R-CNN по суті вирішували її спільним тренуванням декількох нейромереж і таким чином по суті поєднанням в 1 нейромережу. Аналогічні підходи в сучасних технологіях застосовуються і в сфері відслідковування об’єктів де детектор та дескриптор суті зв’язуються в одну нейромережу та тренуються деяким специфічним чином - це є основна ідея алгоритму FairMOT. Вивчення даного алгоритму було б непоганою ідеєю для подальшого дослідження. Він буде можливо дещо повільніше ніж DeepSort, однак все одно буде мати здатність працювати в реальному часі і при цьому підвищувати метрики якості значним чином.

Якщо подивитися на детальний результат роботи DeepSort, наприклад, шляхом аналізу відео фрагментів, які він видає то можна побачити що він

набагато краще працює для близьких об’єктів ніж для тих скупчень, які знаходяться далеко. До того ж його роботу можна значно покращити якщо у нас є дані в тій місцевості або схожих місцевостях схожих об’єктів, які ми будемо намагатися відслідковувати. В цьому випадку можна використати ці дані для тренування детекторів, а в якості початкової точки його тренування використовувати вже натреновані версії yolo, можна почати з yolo версії шостої, так як при незалежному виборі дескриптора вона дає найкращий результат і далі продовжити перебір, наприклад, серед тих моделей yolo, які були запропоновані в цій роботі. Також варто зазначити що що ставишся його останній раз оновлювались в 2023р. тому не дивно що вона дає найкращі результати. Ще одна важлива деталь - це швидкість. Тестування відбувалося з використанням відеокарти NVIDIA Tesla T4 і при цьому були по суті досягнуті результати трохи більше 20 fps, їх можна значно покращити якщо використовувати більш потужні відеокарти, такі як наприклад: NVIDIA V100 або навіть краще A100.

4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ

4.1 Постановка задачі проектування

Розробити ПП, який виявляє та відстежує об’єкти з допустимою точністю на відео-фрагментах знятих в різних місцях, з різних ракурсів, а також з рухомою та нерухомою відеокамерою. При цьому обробка кожного кадру даним ПП повинна здійснюватися приблизно в реальному часі, тобто зі швидкістю >=20 fps. Для проведення техніко-економічного аналізу ПП застосовується метод ФВА.

4.2 Обґрунтування функцій програмного продукту

Функції програмного продукту

F1 - мова програмування;

F2 - програмний пакет (або фреймворк) для реалізації алгоритму DeepSort;

F3 - програмний пакет для нанесення міток на відповідні кадри;

F4 - детектор, який буде виконувати функцію виявлення об’єктів з подальшим використанням алгоритму DeepSort;

F5 - нейромережа для виведення вектора ознак для кожного з об‘єктів обмежувальні рамки яких передбачає детектор.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) Python.

б) C++.

с) R.

Функція F_2:

а) Tensorflow.

б) Pytorch.

с) Власна реалізація.

Функція F_3:

а) cv2.

Функція F_4 (згідно з результатами практичного розділу маємо такі детектори):

а) yolov6s.

б) yolov6m mbla.

Функція F_5:

а) Оригінальна (тобто та яка використовувалася автрами в алгоритмі DeepSort).

б) ResNet50 (глибина = 50 шарів).

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

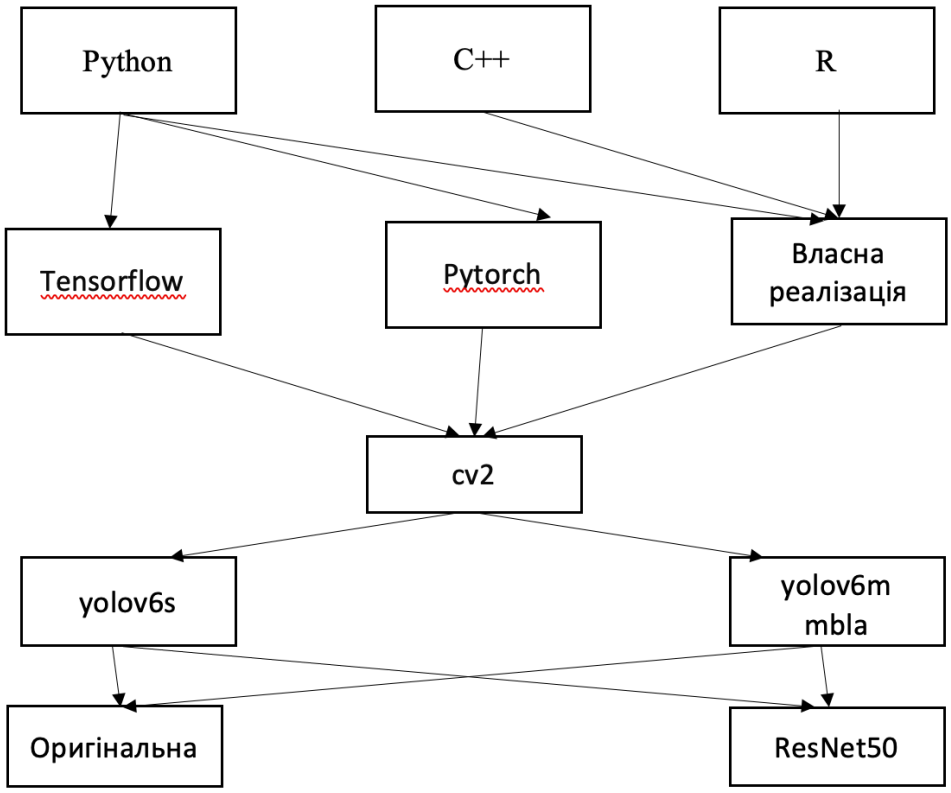


Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіантів основних функцій. Позитивно-негативна матриця показана в таблиці 4.1.

Таблиця 4.1 - Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F1	а	Простий у користуванні	Повільний
	б	Швидкий	Складний у користуванні
	с	Швидкий та зручний для статистичних обчислень	Складно комбінувати вже з існуючими реалізаціями програм на інших мовах програмування
F2	а	Містить багато моделей для глибокого навчання	Не існує технології рощподіленого навчання
	б	Існує технологія розподіоеного навчання+велика кількість технологій глибокого навчання	Не існує технологій web-серверів для зручного розгортання моделей у виробництві
	с	Чіткий контроль за усіма можливими недоліками	Надто багато часц йтиме на реалізацію сже готових рішень, до того ж буде складність із використанням навчених моделей
F3	а	Зручний для обробки відео кадрів	Для побудови графіків краще використовувати інші бібліотеки
F4	а	Швидка модель для обробки зображень	Не є найбільш точною серед усіх моделей yolo
	б	Найбільш точна серед усіх моделей yolo	Не є найбільш швидкою серед усіх моделей yolo
F5	а	Не мітсить надто великої кількості параметрів, однак здатна давати адекватну точність	Не оновлювалась з 2017 року
	б	Потужна та глибока нейромережа	Надто глибока і не дає відповідного приросту в точності в порівнянні з оригінадьною нейромережею

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F_1:

Python сумісний з Pytorch та Tensorflow, тому усі інші мови програмування відкинемо.

Функція F_2:

Моделі yolo, особливо останні версії, реалізовані на Pytorch. До того ж з урахуванням наявності технології розподіленого навчання – виберемо фреймворк Pytorch.

Функція F_5:

З практичного розділу відомо що ResNet50 не дає значних переваг, тому залишимо тільки оригінальну нейромережу.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

F_1а – F_2б – F_3а– F_4а– F_5а

F_1а – F_2б – F_3а– F_4б– F_5а

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів програмного продукту

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X1 - розмір детектора;
- X2 - швидкість алгоритму;

- X3 - якість метрик MOTA та IDF1;

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту, як показано у таблиці 4.2.

Таблиця 4.2 - Основні параметри програмного продукту

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Розмір детектора	X1	Мб	115	70	45
Швидкість алгоритму	X2	Fps (кадр/сек)	10	20	30
Якість метрик MOTA та IDF1	X3	відсотки	25	36	65

За даними таблиці 4.3 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.4.

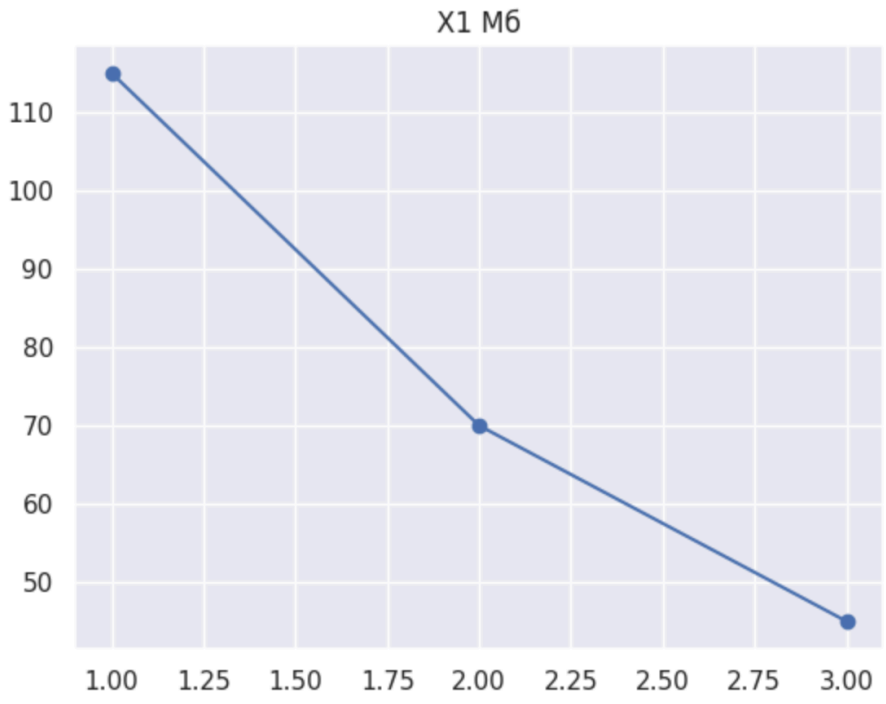


Рисунок 4.2

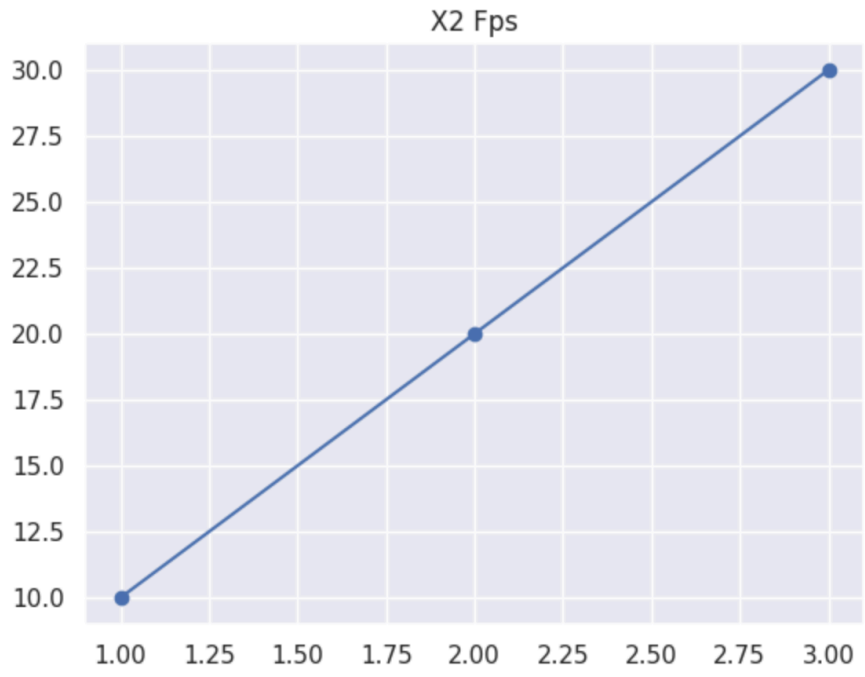


Рисунок 4.3

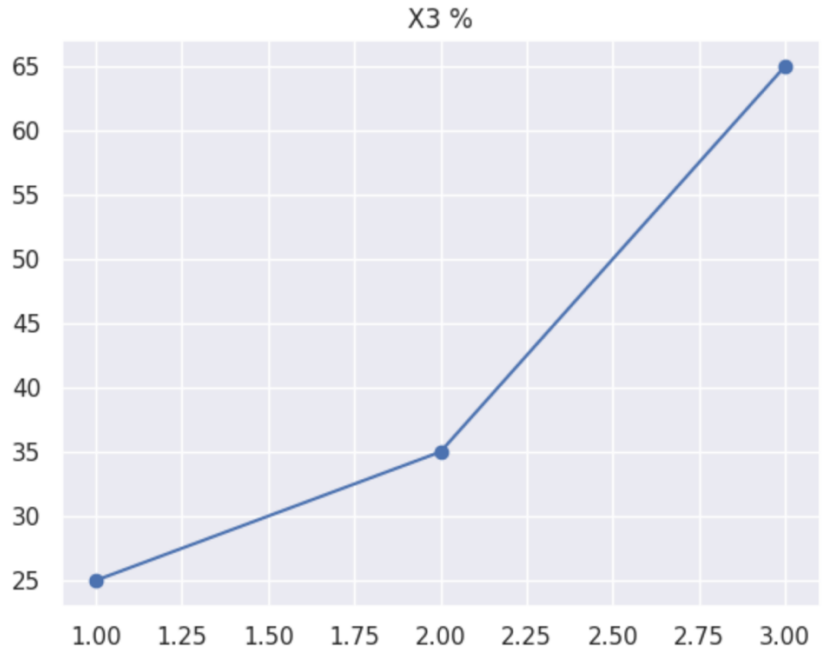


Рисунок 4.4

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Таблиця 4.3 - Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів Ri	Відхи-лення Δi	*Δi^2*
			1	2	3	4	5	6	7			
X1	Розмір детектора	Мб	1	2	1	2	4	1	1	12	-5,333333333	28,44444444
X2	Швидкість алгоритму	Fps (кадр/сек)	2	1	5	1	1	2	4	16	-1,333333333	1,777777778

Таблиця 4.3 - продовження

X3	Якість метрик MOTA та IDF1	%	4	5	4	3	3	3	2	24	6,666666667	44,44444444
	Разом		7	8	10	6	8	6	7	52	0	74,66666667

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 52, \tag{4.1}$$

де N – число експертів,

n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} \tag{4.2}$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \tag{4.3}$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 74,(6). \tag{4.4}$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 197}{7^2(3^3 - 3)} = 0{,}76 > W_k = 0{,}67. \quad (4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Таблиця 4.4 - Попарне порівняння параметрів.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	>	<	>	>	<	<	<	0,5
X1 і X3	<	<	<	<	<	<	<	<	0,5
X2 і X3	<	<	>	<	<	<	>	<	0,5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}. \quad (4.6)$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \quad (4.7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \tag{4.8}$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \tag{4.9}$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \tag{4.10}$$

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 - Розрахунок вагомості параметрів

Параметри	Параметри			Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	bi	Ki	bi	Ki	bi	Ki
X1	1	0,5	0,5	2	0,2222222222	5,5	0,22	15,25	0,2210144928
X2	1,5	1	0,5	3	0,3333333333	8	0,32	22	0,3188405797
X3	1,5	1,5	1	4	0,4444444444	11,5	0,46	31,75	0,4601449275
Всього:				9	1	25	1	69	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X1* (розмір детектора), *X2* (швидкість кінцевої моделі) та *X3* (якість метрик MOTA та IDF1) були вибрані згідно з характеристиками, які ми отримали на етапі тестування ПП, отже.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j}\,, \qquad (4.11)$$

де n – кількість параметрів;

$K_{вi}$– коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Таблиця 4.6 - Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	X2	17,85	4,1	0,32	1,312
F2	А	X2	17,85	4,1	0,32	1,312
F3	А	X2	17,85	4,1	0,32	1,312
F4	А	X1	38,9	10	0,22	2,2
	А	X2	20,1	5,1	0,32	1,632
	А	X3	34,5	4,5	0,46	2,07
	Б	X1	50,8	8,8	0,22	1,936
	Б	X2	15,6	3,2	0,32	1,024
	Б	X3	38,5	5,4	0,46	2,484
F5	А	X2	17,85	4,1	0,32	1,312

За даними з таблиці 4.6 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}], \quad (4.12)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 3*1{,}312 + 2{,}2 + 1{,}632 + 2{,}07 + 1{,}312 = 11{,}15;$$

$$K_{K2} = 3*1{,}312 + 1{,}936 + 1{,}024 + 2{,}484 + 1{,}312 = 10{,}692.$$

Як видно з розрахунків, кращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Перший варіант включає в себе завдання:

1. Розробка моделі виявлення об’єктів (детектор) з використанням більш простої та швидшої моделі;

Другий варіант включає в себе завдання:

2. Розробка моделі виявлення об’єктів (детектор) з використанням більш складної та якісної моделі;

Усі варіанти включають в себе завдання:

3. Розробка моделі відслідковування об’єктів за допомогою алгоритму DeepSort з детектором з минулих пунктів;

Завдання 1 за ступенем новизни відноситься до групи Б, завдання 2 – до групи Б, завдання 3 – до групи Б. За складністю алгоритми, які

використовуються в завданні 1 належать до групи 3; в завданні 2 – до групи 1; а в завданні 3 – до групи 1.

Для реалізації усіх завдань використовується інформація у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як:

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \quad (4.13)$$

де T_P – трудомісткість розробки ПП;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

K_M – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни Б та групи складності алгоритму 3, трудомісткість дорівнює: T_P =19 людино-днів. Поправочний коефіцієнт, який враховує вид інформації у вигляді даних для першого завдання: $K_П$ = 0,75. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх завдань рівний 1: $K_{СК}$ = 1. Використання стандартних модулів для усіх завдань врахуємо за допомогою коефіцієнта $K_{СТ}$ = 0,9 . Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 19 \cdot 0{,}75 \cdot 0{,}9 = 12{,}825 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання ($Т_Р$ =64 людино-днів, $К_П$ =1,01 , $К_{СК}$ = 1, $К_{СТ}$ =0,9):

$$Т_2 = 64 \cdot 1,01 \cdot 0,9 = 58,176 \text{ людино-днів.}$$

Для третього завдання ($Т_Р$ =64 людино-днів, $К_П$ =1,01 , $К_{СК}$ = 1, $К_{СТ}$ =0,9):

$$Т_3 = 64 \cdot 1,01 \cdot 0,9 = 58,176 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$Т_I = (12,825 + 58,176) \cdot 8 = 568,008 \text{ людино-годин.}$$

$$Т_{II} = (58,176 + 58,176) \cdot 8 = 930,816 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь один програміст з окладом 20000 грн., два аналітики в області комп'ютерного зору з окладом 23000. Визначимо середню зарплату за годину за формулою:

$$С_Ч = \frac{М}{T_m \cdot t} \text{ грн.,} \qquad (4.14)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$С_ч = \frac{23000 + 23000 + 20000}{3 \cdot 21 \cdot 8} = 131 \text{ грн.} \quad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_ч \cdot Т_i \cdot К_Д, \quad (4.16)$$

де $С_ч$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 131 \cdot 568{,}008 \cdot 1{,}2 = 89290{,}86$ грн.

II. $С_{ЗП} = 131 \cdot 930{,}816 \cdot 1{,}2 = 146324{,}28$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 89290{,}86 \cdot 0.22 = 19643{,}99$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 146324{,}28 \cdot 0.22 = 32191{,}34$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 20000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 20000 \cdot 0{,}2 = 48000$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1 + К_З) = 48000 \cdot (1 + 0.2) = 57600$ грн.

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 57600 \cdot 0,22 = 12672 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 30000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1,4 \cdot 0,25 \cdot 30000 = 10500 \text{ грн.},$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1,4 \cdot 30000 \cdot 0,08 = 3360 \text{ грн.},$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 16) \cdot 8 \cdot 0.35 =$$
$$= 627,2 \text{ години},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 627,2 \cdot 0,2 \cdot 0,3 \cdot 4,87 = 183,27$ грн.,

де $N_С$ – середньо-споживча потужність приладу;
$К_З$– коефіцієнтом зайнятості приладу;
$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$С_Н = Ц_{ПР} \cdot 0.67 = 30000 \cdot 0,67 = 20100$ грн.

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н, \quad (4.17)$$

$С_{ЕКС} = 57600 + 12672 + 10500 + 3360 + 183,27 + 20100 = 104415,27$ грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 104415,27 / 627,2 = 166,48$ грн/год.

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$С_М = С_{М-Г} \cdot Т, \quad (4.18)$$

I. $С_М = 166,48 \cdot 568,008 = 94561,97$ грн.

II. $С_М = 166,48 \cdot 930,816 = 154962,25$ грн.

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0,67, \quad (4.19)$$

I. $С_Н = 89290,86 \cdot 0,67 = 59824,88$ грн.

II. $С_Н = 146324,28 \cdot 0,67 = 98037,27$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н, \quad (4.20)$$

I. $С_{ПП} = 89290,86 + 19643,99 + 94561,97 + 59824,88 = 263321,7$ грн.

II. $С_{ПП} = 146324,28 + 32191,34 + 154962,25 + 98037,27 = 431515,14$ грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР_j} = К_{Кj} / С_{Фj}, \quad (4.21)$$

$К_{ТЕР1} = 11,15 / 263321,7 = 4,2344 \cdot 10^{-5}$,

$К_{ТЕР2} = 10,692 / 431515,14 = 2,4778 \cdot 10^{-5}$.

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 4,2344 \cdot 10^{-5}$.

Висновки до розділу 4

Після виконання функціонально-вартісного аналізу програмного продукту отримали що найбільший коефіцієнт техніко-економічного рівня має перший варіант реалізації, тобто той який використовує в якості детектора модель yolov6s. Це означає що найбільш прибутковим є варіант використання меншого та простішого детектора в даній системі відслідковування об’єктів, отже, приріст ефективності є незначним при використанні більш складних моделей із сімейства yolo.

ВИСНОВКИ

В ході цієї роботи було реалізовано програмний продукт для виявлення та відслідковування об’єктів. Для цього був застосований алгоритм DeepSort з детекторами із сімейства yolo. Даний алгоритм було протестовано як з оригінальним дескриптором так із дещо ускладненим вигляді resnet50, що по суті підтвердило ефективність використання саме оригінального дескриптора. Серед детекторів найбільш ефективними виявилися yolo версія 6. Це можна пояснити тим що це є перша версія yolo, яка використовує інший алгоритм навчання, який працює так наче ускладнюючи структуру нейромережі - перепараметризація. Також дану версію було оновлено в цьому році, що зробило її ще більш ефективною.

Розроблений програмний продукт можна використати наприклад для системи автономного керування бажано із дотренуванням на даних, які будуть зібрані конкретно для цієї задачі. Згідно з показниками метрик MOTA та метрик ідентифікації цей програмний продукт не є надто ефективним, однак показники метрик сильно погіршуються через те що на відео фрагментах присутньо багато маленьких об’єктів, тобто об’єктів, які знаходяться на далекій відстані і при цьому помилка при виявленні або відслідковуванні далеких об’єктів рахується так само як і для великих об’єктів. Хоча для багатьох задач більш важливим може бути саме об’єкти, які знаходяться в досить близькому околі до нас. В такому разі DeepSort буде давати кращі результати, що можна побачити на відповідних оброблених відео фрагментах. Загалом дотренування даного ПП на даних специфічних для конкретної задачі і при цьому можливо попередньо оброблених (відкинувши наприклад мітки для занадто далеко об’єктів) може доволі непогано підвищити ефективність цієї моделі виявлення та відслідковування об’єктів.

Також був проведений функціонально-вартісний аналіз даного програмного продукту в ході якого було визначено, що найбільш ефективною

моделлю детектора є саме yolo v6s (інші можуть давати лише незначний приріст в якості за надто високі обчислювальні витрати) – її коефіцієнт техніко-економічного рівня = $4{,}31943 \cdot 10^{-5}$.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Deep Learning (Ian J. Goodfellow, Yoshua Bengio and Aaron Courville), MIT Press, 2016.

2. Diganta Misra. Mish: A Self Regularized Non-Monotonic Activation Function, 2020.

3. Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, & Dongwei Ren. Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression, 2019.

4. Sergey Ioffe, & Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

5. Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. Deep Residual Learning for Image Recognition, 2015.

6. Zhuliang Yao, Yue Cao, Shuxin Zheng, Gao Huang, & Stephen Lin. Cross-Iteration Batch Normalization, 2021.

7. Haikun Wei, Jun Zhang, Florent Cousseau, Tomoko Ozeki, Shun-ichi Amari; Dynamics of Learning Near Singularities in Layered Networks. *Neural Comput* 2008; 20 (3): 813–843. doi: https://doi.org/10.1162/neco.2007.12-06-414.

8. A. Emin Orhan, & Xaq Pitkow. Skip Connections Eliminate Singularities, 2018.

9. Falong Shen, & Gang Zeng. Weighted Residuals for Very Deep Networks, 2016.

10. Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, & Jun-Wei Hsieh. CSPNet: A New Backbone that can Enhance Learning Capability of CNN, 2019.

11. Uijlings, J., Sande, K., Gevers, T., & Smeulders, A. Selective Search for Object Recognition. *International Journal of Computer Vision,* 2013, *104, 154-171.*

12. P. F. Felzenszwalb and D. P. Huttenlocher. Efficient GraphBased

Image Segmentation. IJCV, 59:167–181, 2004.

13. Navaneeth Bodla, Bharat Singh, Rama Chellappa, & Larry S. Davis. Soft-NMS – Improving Object Detection With One Line of Code, 2017.

14. Ross Girshick, Jeff Donahue, Trevor Darrell, & Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.

15. Ross Girshick. Fast R-CNN, 2015.

16. Shaoqing Ren, Kaiming He, Ross Girshick, & Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, 2016.

17. Joseph Redmon, Santosh Divvala, Ross Girshick, & Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection, 2016.

18. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. CoRR, abs/1409.4842, 2014

19. Joseph Redmon, & Ali Farhadi. YOLO9000: Better, Faster, Stronger, 2016.

20. Joseph Redmon, & Ali Farhadi. YOLOv3: An Incremental Improvement, 2018.

21. Alexey Bochkovskiy, Chien-Yao Wang, & Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection, 2020.

22. S. Liang *et al*., "Edge YOLO: Real-Time Intelligent Object Detection System Based on Edge-Cloud Cooperation in Autonomous Vehicles," in *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 25345-25360, Dec. 2022, doi: 10.1109/TITS.2022.3158253.

23. https://github.com/ultralytics/yolov5

24. Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, Yiduo Li, Bo Zhang, Yufei Liang, Linyuan Zhou, Xiaoming Xu, Xiangxiang Chu, Xiaoming Wei, & Xiaolin Wei. YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications, 2022.

25. Chengjian Feng, Yujie Zhong, Yu Gao, Matthew R Scott, and Weilin

Huang. Tood: Task-aligned one-stage object detection. In ICCV, 2021.

26. Xiang Li, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection. Advances in Neural Information Processing Systems, 33:21002–21012, 2020.

27. Chien-Yao Wang, Alexey Bochkovskiy, & Hong-Yuan Mark Liao. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, 2022.

28. Nathanael L. Baisa. Derivation of a Constant Velocity Motion Model for Visual Tracking, 2020.

29. Hamilton J.D. Time Series Analysis. – New Jersey, Princeton University Press, 1994. – 890 p.

30. Nicolai Wojke, Alex Bewley, & Dietrich Paulus. Simple Online and Realtime Tracking with a Deep Association Metric, 2017.

31. Luiten, J., Ošep, A., Dendorfer, P. *et al.* HOTA: A Higher Order Metric for Evaluating Multi-object Tracking. *Int J Comput Vis* **129**, 548–578, 2021. https://doi.org/10.1007/s11263-020-01375-2

32. https://github.com/xuarehere/yolo_series_deepsort_pytorch

33. https://github.com/ultralytics/yolov3

34. https://github.com/meituan/YOLOv6

35. https://github.com/WongKinYiu/yolov7

36. https://github.com/ultralytics/ultralytics

37. S. C. et al, Fundamentals of object tracking. Cambridge University Press, 2011.

38. T.-Y. Lin et al., Microsoft COCO: Common Objects in Context. 2015.

39. P. Dendorfer et al., MOTChallenge: A Benchmark for Single-Camera Multiple Target Tracking. 2020.

40. L. Zheng et al., MARS: A Video Benchmark for Large-Scale Person Re-identification. Springer, 2016.

41. L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang, and Q. Tian, “Scalable

Person Re-identification: A Benchmark," 2015.

42. https://github.com/JonathonLuiten/TrackEval

ДОДАТОК А ЛІСТІНГ ПРОГРАМИ

Файл для оцінки якостей моделей yolo (yolovalidation.py):

```
!nvidia-smi
import numpy as np
import pandas as pd
df = []
index = []
cols = ['mAP50', 'mAP50-95', 'fps (without NMS)', 'fps (with
NMS)','model size (MB)', 'image size (resolution)']
!pip install fiftyone
!pip install pyyaml
import fiftyone as fo
import fiftyone.zoo as foz
dataset = foz.load_zoo_dataset(
    "coco-2017",
    split="validation",
    dataset_name="detector-recipe"
)
!fiftyone convert \
            --input-dir /root/fiftyone/coco-2017/validation
\
            --input-type
fiftyone.types.COCODetectionDataset \
            --output-dir /content/base-datasets/coco \
            --output-type fiftyone.types.YOLOv5Dataset
"""Validation of YOLO v3"""
import os
if not os.path.exists('/content/datasets'):
  os.mkdir('/content/datasets')
if not os.path.exists('/content/datasets/coco'):
  os.mkdir('/content/datasets/coco')
if not os.path.exists('/content/datasets/coco/images'):
  os.mkdir('/content/datasets/coco/images')
```

```
      if not os.path.exists('/content/datasets/coco/labels'):
        os.mkdir('/content/datasets/coco/labels')
      if not os.path.exists('/content/datasets/coco/images/val'):
        os.mkdir('/content/datasets/coco/images/val')
      if not os.path.exists('/content/datasets/coco/labels/val'):
        os.mkdir('/content/datasets/coco/labels/val')
      # Commented out IPython magic to ensure Python compatibility.
      !git clone https://github.com/ultralytics/yolov3
      # %cd yolov3
      !pip install -r requirements.txt
      import yaml
      with open('data/coco128.yaml', 'r') as f:
        file = yaml.safe_load(f)
      names_true = file['names']
      names_true = list(names_true.values())
      with  open('/content/base-datasets/coco/dataset.yaml',  'r')
as f:
        file = yaml.safe_load(f)
      names = file['names']
      #names
      import os
      l = os.listdir('/content/base-datasets/coco/labels/val')
      for i in range(len(l)):
        f  =  open('/content/base-datasets/coco/labels/val/'+l[i],
'r')
        lines = f.readlines()
        f.close()
        lines_true = lines.copy()
        k = 0
        for t in lines:
          el = str(names_true.index(names[int(t.split(' ')[0])]))
          lis = lines_true[k].split(' ')
          lis[0] = el
          lines_true[k] = ' '.join(lis)
          k += 1
        f  =  open('/content/base-datasets/coco/labels/val/'+l[i],
```

```
'w')
        f.writelines(lines_true)
        f.close()
      import shutil
      destination = '/content/datasets/coco/labels/val'
      # moving txt files
      for i in range(1700):
        source = '/content/base-datasets/coco/labels/val/'+l[i]
        if not os.path.exists(destination+'/'+l[i]):
          shutil.move(source, destination)
      destination = '/content/datasets/coco/images/val'
      # moving jpg files
      for i in range(1700):
        source                  =                  '/content/base-
datasets/coco/images/val/'+l[i][:-4]+'.jpg'
        if not os.path.exists(destination+'/'+l[i][:-4]+'.jpg'):
          shutil.move(source, destination)
      with open('data/coco128.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['path'] = '../datasets/coco'
      f_old['train'] = 'images/val'
      f_old['val'] = 'images/val'
      #f_old = yaml.safe_load(file)
      with open('data/coco1700.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python train.py --data coco1700.yaml --epochs 300 --weights
yolov3.pt  --batch-size 128
      !python val.py --weights yolov3.pt --data coco1700.yaml --
img 640
      df.append([0.648,     0.463,     np.round(1000/23.3,     3),
np.round(1000/26.9, 3), 119, 640])
      np.round(1000/23, 3)
      """Validation of yolo v5"""
      # Commented out IPython magic to ensure Python compatibility.
      # %cd ..
      # Commented out IPython magic to ensure Python compatibility.
```

```
      !git clone https://github.com/ultralytics/yolov5  # clone
      # %cd yolov5
      !pip install -r requirements.txt  # install
      with open('data/coco128.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['path'] = '../datasets/coco'
      f_old['train'] = 'images/val'
      f_old['val'] = 'images/val'
      #f_old = yaml.safe_load(file)
      with open('data/coco1700.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python val.py --weights yolov5s.pt --data coco1700.yaml --
img 640
      df.append([0.563,     0.376,     np.round(1000/5.2,     3),
np.round(1000/8.4, 3), 14.1, 640])
      !python val.py --weights yolov5m.pt --data coco1700.yaml --
img 640
      df.append([0.636,     0.45,     np.round(1000/12.5,     3),
np.round(1000/15.7, 3), 40.8, 640])
      !python val.py --weights yolov5l.pt --data coco1700.yaml --
img 640
      df.append([0.664,     0.485,     np.round(1000/21.6,     3),
np.round(1000/25.2, 3), 89.3, 640])
      !python val.py --weights yolov5x.pt --data coco1700.yaml --
img 640
      df.append([0.681,     0.503,     np.round(1000/41.1,     3),
np.round(1000/43.8, 3), 166, 640])
      !python val.py --weights yolov5m6.pt --data coco1700.yaml -
-img 1280
      df.append([0.688,     0.511,     np.round(1000/51.7,     3),
np.round(1000/55.1, 3), 69, 1280])
      !python val.py --weights yolov5l6.pt --data coco1700.yaml -
-img 1280
      df.append([0.709,     0.533,     np.round(1000/90.2,     3),
np.round(1000/93.8, 3), 147, 1280])
      index.extend(['yolov3',  'yolov5s',  'yolov5m',  'yolov5l',
```

```
'yolov5x', 'yolov5m6', 'yolov5l6'])
      """Validation of yolo v6"""
      # Commented out IPython magic to ensure Python compatibility.
      # %cd ..
      # Commented out IPython magic to ensure Python compatibility.
      !git clone https://github.com/meituan/YOLOv6
      # %cd YOLOv6
      !pip install -r requirements.txt
      with open('data/coco.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      #f_old['path'] = '../datasets/coco'
      f_old['train'] = '../datasets/coco/images/val'
      f_old['val'] = '../datasets/coco/images/val'
      f_old['test'] = '../datasets/coco/images/val'
      f_old['is_coco'] = False
      del f_old['anno_path']
      #f_old = yaml.safe_load(file)
      with open('data/coco1700.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python tools/eval.py --data data/coco1700.yaml  --weights
yolov6s.pt --task val

      df.append([0.598,     0.437,     np.round(1000/6.33,     3),
np.round(1000/7.85, 3), 38.9, 640])
      !python tools/eval.py --data data/coco1700.yaml  --weights
yolov6m.pt --task val
      df.append([0.651,     0.486,     np.round(1000/14.2,     3),
np.round(1000/15.7, 3), 72.6, 640])
      !python tools/eval.py --data data/coco1700.yaml  --weights
yolov6l.pt --task val
      df.append([0.682,     0.511,     np.round(1000/22.9,     3),
np.round(1000/24.3, 3), 114, 640])
      !wget
https://github.com/meituan/YOLOv6/releases/download/0.4.0/yolov6
m_mbla.pt
      !python tools/eval.py --data data/coco1700.yaml  --weights
```

```
yolov6m_mbla.pt --task val --img-size 640
      df.append([0.66,     0.488,     np.round(1000/13.22,     3),
np.round(1000/14.8, 3), 50.8, 640])
      !wget
https://github.com/meituan/YOLOv6/releases/download/0.4.0/yolov6
l_mbla.pt
      !python tools/eval.py --data data/coco1700.yaml  --weights
yolov6l_mbla.pt --task val --img-size 640
      df.append([0.677,     0.506,     np.round(1000/18.7,     3),
np.round(1000/20.1, 3), 88.7, 640])
      !wget
https://github.com/meituan/YOLOv6/releases/download/0.4.0/yolov6
x_mbla.pt
      !python tools/eval.py --data data/coco1700.yaml  --weights
yolov6x_mbla.pt --task val --img-size 640
      df.append([0.692,     0.520,     np.round(1000/29.7,     3),
np.round(1000/31.1, 3), 151, 640])
      index.extend(['yolov6s',       'yolov6m',       'yolov6l',
'yolov6m_mbla', 'yolov6l_mbla', 'yolov6x_mbla'])
      """Validation of yolo v7"""
      # Commented out IPython magic to ensure Python compatibility.
      # %cd ..
      # Commented out IPython magic to ensure Python compatibility.
      !git clone https://github.com/WongKinYiu/yolov7
      # %cd yolov7
      !pip install -r requirements.txt
      with open('data/coco.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      #f_old['path'] = '../datasets/coco'
      f_old['train'] = '../datasets/coco/images/val'
      f_old['val'] = '../datasets/coco/images/val'
      f_old['test'] = '../datasets/coco/images/val'
      #f_old = yaml.safe_load(file)
      with open('data/coco1700.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python test.py --data data/coco1700.yaml --img 640 --device
```

```
0 --weights yolov7.pt --name yolov7_640_val
      df.append([0.684,     0.496,     np.round(1000/10.2,     3),
np.round(1000/11.5, 3), 72.1, 640])
      !python test.py --data data/coco1700.yaml --img-size 640 --
device 0 --weights yolov7x.pt
      df.append([0.698,     0.512,     np.round(1000/18.5,     3),
np.round(1000/19.8, 3), 136, 640])
      !wget
https://github.com/WongKinYiu/yolov7/releases/download/v0.1/yolo
v7-w6.pt
      !python test.py --data data/coco1700.yaml --img-size 1280 -
-device 0 --weights yolov7-w6.pt
      df.append([0.715,     0.528,     np.round(1000/29.2,     3),
np.round(1000/31.1, 3), 134.7, 1280])
      index.extend(['yolov7', 'yolov7x', 'yolov7w6'])
      """Validation of yolo v8"""
      # Commented out IPython magic to ensure Python compatibility.
      # %cd ..
      # Commented out IPython magic to ensure Python compatibility.
      !git clone https://github.com/ultralytics/ultralytics
      # %cd ultralytics
      !pip install -r requirements.txt
      !pip install ultralytics
      with open('ultralytics/datasets/coco128.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['path'] = '/content/datasets/coco'
      f_old['train'] = '/content/datasets/coco/images/val'
      f_old['val'] = '/content/datasets/coco/images/val'
      with  open('ultralytics/datasets/coco1700.yaml',  'w')  as
f_new:
        yaml.dump(f_old, f_new)
      df.append([0.659,     0.497,     np.round(1000/15.6,     3),
np.round(1000/17.3, 3), 49.7, 640])
      from ultralytics import YOLO
      # Load a model
      modelm = YOLO("yolov8m.pt")  # load an official model
```

```
      modell = YOLO("yolov8l.pt")
      modelx = YOLO("yolov8x.pt")
      # Validate the model
      metricsm                                                    =
modelm.val(data='ultralytics/datasets/coco1700.yaml')
      df.append([0.688,     0.528,     np.round(1000/26.5,     3),
np.round(1000/28.8, 3), 83.7, 640])
      metricsl                                                    =
modell.val(data='ultralytics/datasets/coco1700.yaml')
      df.append([0.702,     0.538,     np.round(1000/44.8,     3),
np.round(1000/46.8, 3), 131, 640])
      metricsx                                                    =
modelx.val(data='ultralytics/datasets/coco1700.yaml')
      index.extend(['yolov8m', 'yolov8l', 'yolov8x'])
      """Save results"""
      lines = [' '.join([str(j) for j in i])+'\n' for i in df]
      table = [' '.join(cols)+'\n']+lines+[' '.join(index)]
      with open('/content/val_models.txt', 'w') as f:
        f.writelines(table)
```

Файл для відбору найбільш адекватних моделей yolo по різним критеріям (yolovalresults.py):

```
      # Mounting GDrive.
      import sys
      from google.colab import drive

      MOUNT_POINT = "/content/drive"
      drive.mount(MOUNT_POINT)

      # !! Please provide a path to a homework directory. !! #
      # !! E.g. my instance of source code is located at  !! #
      # !! "RL_Homework/HW1_Part2/" on GDrive             !! #
      # /Understanding-Multiple-Object-Tracking-using-DeepSORT
      PATH_TO_CODE_FOLDER     =     'DeepSort'          #     like
"RL_Homework/HW1_Part2"
```

```
      assert PATH_TO_CODE_FOLDER, "Please, provide a path to the
code on your GDrive"

      # Commented out IPython magic to ensure Python compatibility.
      # Full mounted path to the code then is:
      FULL_PATH = f"{MOUNT_POINT}/My Drive/{PATH_TO_CODE_FOLDER}"

      # Adding a path to the HW code to a list of sources for
importing.
      sys.path.append(FULL_PATH)

      # Changing a working directory.
      # %cd $FULL_PATH

      """## Finsh the validation of the yolo models"""

      import numpy as np
      import pandas as pd

      with open('val_models.txt', 'r') as f:
        lines = f.readlines()
      cols = lines[0][:-1].split(" ")
      index = lines[-1].split(" ")
      arr = np.array([np.float128(np.array(line[:-1].split(" ")))
for line in lines[1:-1]])
      df = pd.DataFrame(arr, columns=cols, index=index)
      df_all = df.copy()
      yol3 = index[0]
      yol5 = index[1:7]
      yol6 = index[7:13]
      yol7 = index[13:16]
      yol8 = index[16:19]
      #pd.DataFrame(arr)

      df
```

"""Для yolo v3 маємо такі результати тестування: ... . Після тестування yolo v5 отримали такі результати:"""

df.loc['yolov3':'yolov5l6']

"""Видаляємо модель yolov3 так як модель yolov5l досягає кращого результату при меншому розмірі та при більшій швидкості (39 проти 37 fps). Також приберемо з розгляду модель yolov5s, як модель з надто низькими показниками якості map50 та map50-95, та модель yolov5x, розмір якої не відповідає показниками якості, - yolov5m6 досягає кращого результату при набагато меньшому розмірі і незначному падінню в швидкості (швидкість падає через те що цю модель необхідно застосувати до зображень розміром 1280*1280 пікселів - за необхідності перед застосуванням yolov5m6 зображення форматують до відповідного розміру).

"""

df = df.drop(['yolov3', 'yolov5s', 'yolov5x'])

"""Після тестування yolov6 та видалення усіх зайвих моделей отримали такі резульати:"""

df.loc['yolov5m':'yolov6x_mbla']

df.loc['yolov5m':'yolov6x_mbla'].sort_values(by=['mAP50'])

"""Викинемо модель yolov5l так як yolov6l_mbla досягає кращих резульатів за значно більшої швидкості (49.8 проти 39.7 fps) і має при цьому менший розмір. За аналогічних причин приберемо з розгляду модель yolov6m порівнюючи її з yolov6m_mbla."""

df = df.drop(['yolov5l', 'yolov6m'])

"""Після відкидання усіх недоцільних моделей з вищеописаних

```
причин та тестування yolo v7 маємо:"""

    df.loc['yolov5m':'yolov7w6'].sort_values(by=['mAP50'])

    """Викинемо модель yolov5l6 так як yolov7w6 випереджає її з точки зору усіх критеріїв - краща якість за map, більша швидкість, менша кількість параметрів (тобто розмір з точки зору розташування в пам'яті комп'ютера). За аналогічних суджень прибераємо модель yolov6x_mbla порівнюючи її з yolov7x. Модель yolov7 досягає приблизно таких самих результатів як і yolov6l але при цьому займайє в 2 рази менше місця в пам'яті і є десь в стільки ж разів швидшою, тому викинемо yolov6l. Так само порівнюючи з yolov7 приберемо і yolov6l_mbla. """

    df = df.drop(['yolov5l6', 'yolov6x_mbla', 'yolov6l', 'yolov6l_mbla'])

    """Після відкидання усіх недоцільних моделей з вищеописаних причин та тестування yolo v8 маємо:"""

    df.sort_values(by=['mAP50'])

    """Приберемо модель yolov8x так як вона є занадто повільною у порівнянні з аналогічною по розміру і вищій по якості yolov7w6. За аналогічних причин приберемо модель yolov8m. """

    df = df.drop(['yolov8x', 'yolov8m'])
    df

    df.sort_values(by=['mAP50'])

    """##Plot results"""

    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_theme()
```

```
      fig,ax = plt.subplots()
      tit1 = 'Результати валідації для усіх моделей'
      tit2 = 'Результати валідації для кінцевої множини моделей'
      ax.set_title(tit1)
      ax.plot(df_all.loc[yol3, 'fps(with_NMS)'], df_all.loc[yol3,
'mAP50'], label='yolov3', marker='o')
      ax.plot(df_all.loc[yol5, 'fps(with_NMS)'], df_all.loc[yol5,
'mAP50'], label='yolov5', marker='o')
      ax.plot(df_all.loc[yol6, 'fps(with_NMS)'], df_all.loc[yol6,
'mAP50'], label='yolov6', marker='o')
      ax.plot(df_all.loc[yol7, 'fps(with_NMS)'], df_all.loc[yol7,
'mAP50'], label='yolov7', marker='o')
      ax.plot(df_all.loc[yol8, 'fps(with_NMS)'], df_all.loc[yol8,
'mAP50'], label='yolov8', marker='o')
      ax.set_xlabel("fps")
      ax.set_ylabel("map50")
      ax.legend(fontsize='10')

      fig,ax = plt.subplots()

      ax.set_title(tit2)
      ax.plot(df_all.loc[['yolov5m','yolov5m6'],
'fps(with_NMS)'],  df_all.loc[['yolov5m','yolov5m6'],  'mAP50'],
label='yolov5', marker='o')
      ax.plot(df_all.loc[['yolov6s',            'yolov6m_mbla'],
'fps(with_NMS)'],    df_all.loc[['yolov6s',    'yolov6m_mbla'],
'mAP50'], label='yolov6', marker='o')
      ax.plot(df_all.loc[['yolov7',   'yolov7x',   'yolov7w6'],
'fps(with_NMS)'], df_all.loc[['yolov7', 'yolov7x', 'yolov7w6'],
'mAP50'], label='yolov7', marker='o')
      ax.plot(df_all.loc['yolov8l',            'fps(with_NMS)'],
df_all.loc['yolov8l', 'mAP50'], label='yolov8', marker='o')
      ax.set_xlabel("fps")
      ax.set_ylabel("map50")
```

```
ax.legend(fontsize='10')
```

Файл для тестування різних моделей yolo з deepsort (deepsort_main.py):

```
import sys
from google.colab import drive
MOUNT_POINT = "/content/drive"
drive.mount(MOUNT_POINT)
PATH_TO_CODE_FOLDER = 'DeepSort/yolovx_deepsort_pytorch'  # like "RL_Homework/HW1_Part2"
assert PATH_TO_CODE_FOLDER, "Please, provide a path to the code on your GDrive"
# Commented out IPython magic to ensure Python compatibility.
# Full mounted path to the code then is:
FULL_PATH = f"{MOUNT_POINT}/My Drive/{PATH_TO_CODE_FOLDER}"

# Adding a path to the HW code to a list of sources for importing.
sys.path.append(FULL_PATH)
# Changing a working directory.
# %cd $FULL_PATH
"""## Cloning Repositories"""
!nvidia-smi
!pip install pyyaml
"""## Installing Requirements"""
!pip install -r requirements.txt
"""# Integrating DeepSORT
# Inference (on MOT17 videos)
##YOLO v5
Running DeepSORT with yolo v5m.
"""
import yaml
with open('configs/yolov5.yaml', 'r') as f:
  f_old = yaml.safe_load(f)
f_old['YOLOV5']['WEIGHT'] = "./models/yolov5/yolov5m.pt"
f_old['YOLOV5']['IMG_SIZE_HEIGHT'] = 64*10
```

```
      f_old['YOLOV5']['IMG_SIZE_WIDTH'] = 64*10
      with open('configs/yolov5.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov5m/data --config_detection ./configs/yolov5.yaml --
detect_model yolov5
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov5m/data --config_detection ./configs/yolov5.yaml --
detect_model yolov5
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov5m/data --config_detection ./configs/yolov5.yaml --
detect_model yolov5
      """Running DeepSORT with yolo v5m6."""
      import yaml
      with open('configs/yolov5.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['YOLOV5']['WEIGHT'] = "./models/yolov5/yolov5m6.pt"
      f_old['YOLOV5']['IMG_SIZE_HEIGHT'] = 64*14
      f_old['YOLOV5']['IMG_SIZE_WIDTH'] = 64*14
      with open('configs/yolov5.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov5m6/data --config_detection ./configs/yolov5.yaml -
-detect_model yolov5
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov5m6/data --config_detection ./configs/yolov5.yaml -
-detect_model yolov5
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov5m6/data --config_detection ./configs/yolov5.yaml -
-detect_model yolov5
      """##YOLO v6
      Running DeepSORT with yolo v6s.
      """
      import yaml
      with open('configs/yolov6.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
```

```
      f_old['YOLOV6']['WEIGHT'] = "./models/yolov6/yolov6s.pt"
      with open('configs/yolov6.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov6s/data --config_detection ./configs/yolov6.yaml --
detect_model yolov6
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov6s/data --config_detection ./configs/yolov6.yaml --
detect_model yolov6
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov6s/data --config_detection ./configs/yolov6.yaml --
detect_model yolov6
      """Running DeepSORT with yolo v6m_mbla.
      """
      import yaml
      with open('configs/yolov6.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['YOLOV6']['WEIGHT']                                    =
"./models/yolov6/yolov6m_mbla.pt"
      with open('configs/yolov6.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov6m_mbla/data                    --config_detection
./configs/yolov6.yaml --detect_model yolov6
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov6m_mbla/data                    --config_detection
./configs/yolov6.yaml --detect_model yolov6
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov6m_mbla/data                    --config_detection
./configs/yolov6.yaml --detect_model yolov6
      """##YOLO v7
      Running DeepSORT with yolo v7.
      """
      import yaml
      with open('configs/yolov7.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
```

```
      f_old['YOLOV7']['WEIGHT'] = "./models/yolov7/yolov7.pt"
      f_old['YOLOV7']['IMG_SIZE_HEIGHT'] = 640
      f_old['YOLOV7']['IMG_SIZE_WIDTH'] = 640
      with open('configs/yolov7.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov7/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov7/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov7/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      """Running DeepSORT with yolo v7w6.
      """
      import yaml
      with open('configs/yolov7.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['YOLOV7']['WEIGHT'] = "./models/yolov7/yolov7-w6.pt"
      f_old['YOLOV7']['IMG_SIZE_HEIGHT'] = 64*20
      f_old['YOLOV7']['IMG_SIZE_WIDTH'] = 64*20
      with open('configs/yolov7.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov7-w6/data --config_detection ./configs/yolov7.yaml
--detect_model yolov7
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov7-w6/data --config_detection ./configs/yolov7.yaml
--detect_model yolov7
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov7-w6/data --config_detection ./configs/yolov7.yaml
--detect_model yolov7
      """Running DeepSORT with yolo v7x.
      """
      import yaml
```

```
      with open('configs/yolov7.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['YOLOV7']['WEIGHT'] = "./models/yolov7/yolov7x.pt"
      f_old['YOLOV7']['IMG_SIZE_HEIGHT'] = 64*10
      f_old['YOLOV7']['IMG_SIZE_WIDTH'] = 64*10
      with open('configs/yolov7.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov7x/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov7x/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov7x/data --config_detection ./configs/yolov7.yaml --
detect_model yolov7
      """##YOLO v8
      Running DeepSORT with yolo v8l.
      """
      import yaml
      with open('configs/yolov8.yaml', 'r') as f:
        f_old = yaml.safe_load(f)
      f_old['YOLOV8']['WEIGHT'] = "./models/yolov8/yolov8l.pt"
      with open('configs/yolov8.yaml', 'w') as f_new:
        yaml.dump(f_old, f_new)
      !python3   deepsort.py   ./MOT17-02-DPM.mp4   --save_path
./output/yolov8l/data --config_detection ./configs/yolov8.yaml --
detect_model yolov8
      !python3   deepsort.py   ./MOT17-04-FRCNN.mp4   --save_path
./output/yolov8l/data --config_detection ./configs/yolov8.yaml --
detect_model yolov8
      !python3   deepsort.py   ./MOT17-13-FRCNN.mp4   --save_path
./output/yolov8l/data --config_detection ./configs/yolov8.yaml --
detect_model yolov8
```

Файл для тренування дескриптора resnet50 (traindeepsort.py):

```
<h3>Prepare data</h3>
"""
!nvidia-smi
!pip install gdown
!gdown https://drive.google.com/uc\?id\=0B8-rUzbwVRk0c054eEozWG9COHM
import zipfile
zip_ref = zipfile.ZipFile("Market-1501-v15.09.15.zip", 'r')
zip_ref.extractall("Market")
zip_ref.close()
!git clone https://github.com/xuarehere/yolovx_deepsort_pytorch.git
# Commented out IPython magic to ensure Python compatibility.
# %cd yolovx_deepsort_pytorch
!pip install -r requirements.txt
# Commented out IPython magic to ensure Python compatibility.
# %cd deep_sort/deep
# Commented out IPython magic to ensure Python compatibility.
# %%writefile model.py
# import torch
# import torch.nn as nn
# import torch.nn.functional as F
# from torch.nn import init
# from torchvision import models
# from torch.autograd import Variable
# def weights_init_kaiming(m):
#     classname = m.__class__.__name__
#     # print(classname)
#     if classname.find('Conv') != -1:
#                 init.kaiming_normal_(m.weight.data,  a=0, mode='fan_in') # For old pytorch, you may use kaiming_normal.
#     elif classname.find('Linear') != -1:
#                 init.kaiming_normal_(m.weight.data,  a=0, mode='fan_out')
```

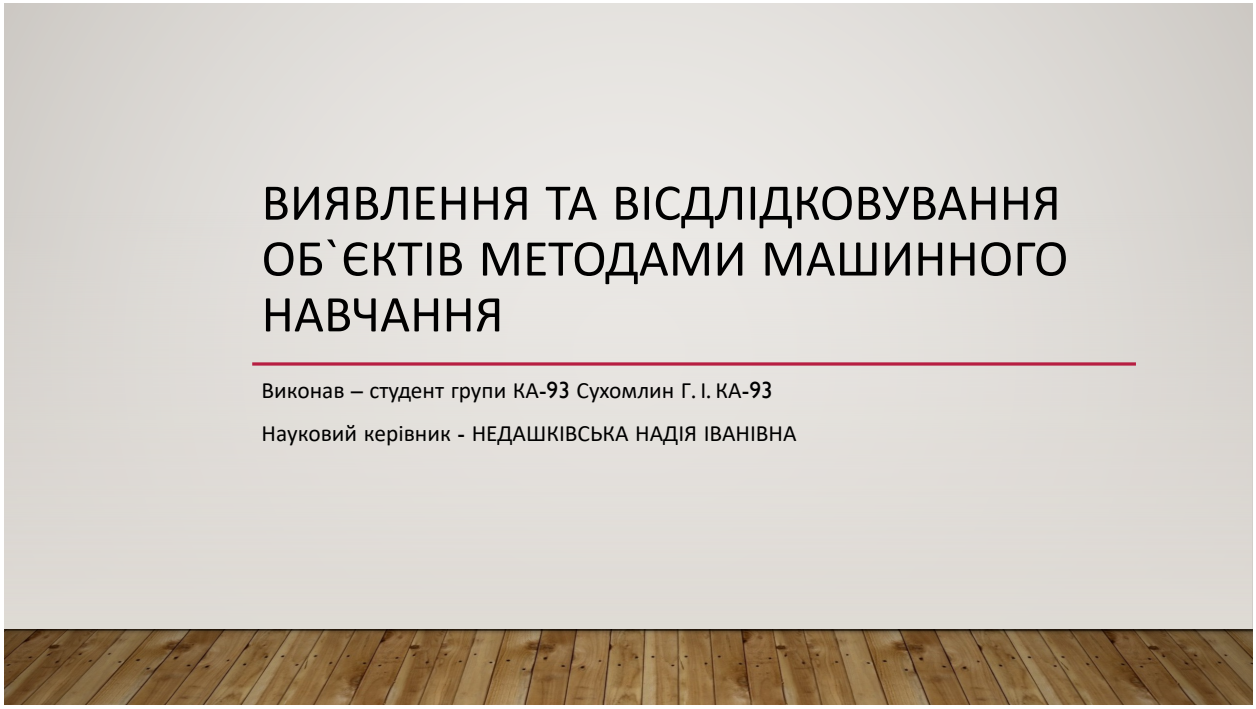
```
        #     elif classname.find('BatchNorm1d') != -1:
        #         init.normal_(m.weight.data, 1.0, 0.02)
        #     if hasattr(m, 'bias') and m.bias is not None:
        #         init.constant_(m.bias.data, 0.0)
        #
        # def weights_init_classifier(m):
        #     classname = m.__class__.__name__
        #     if classname.find('Linear') != -1:
        #         init.normal_(m.weight.data, std=0.001)
        #         init.constant_(m.bias.data, 0.0)
        #
        #
        # def activate_drop(m):
        #     classname = m.__class__.__name__
        #     if classname.find('Drop') != -1:
        #         m.p = 0.1
        #         m.inplace = True
        #
        # # Defines the new fc layer and classification layer
        # # |--Linear--|--bn--|--relu--|--Linear--|
        # class ClassBlock(nn.Module):
        #       def __init__(self, input_dim, class_num, droprate,
relu=False, bnorm=True, linear=512, return_f = False):
        #         super(ClassBlock, self).__init__()
        #         self.return_f = return_f
        #         add_block = []
        #         if linear>0:
        #             add_block += [nn.Linear(input_dim, linear)]
        #         else:
        #             linear = input_dim
        #         if bnorm:
        #             add_block += [nn.BatchNorm1d(linear)]
        #         if relu:
        #             add_block += [nn.LeakyReLU(0.1)]
        #         if droprate>0:
        #             add_block += [nn.Dropout(p=droprate)]
```

```
        #         add_block = nn.Sequential(*add_block)
        #         add_block.apply(weights_init_kaiming)
        #
        #         classifier = []
        #         classifier += [nn.Linear(linear, class_num)]
        #         classifier = nn.Sequential(*classifier)
        #         classifier.apply(weights_init_classifier)
        #
        #         self.add_block = add_block
        #         self.classifier = classifier
        #     def forward(self, x):
        #         x = self.add_block(x)
        #         if self.return_f:
        #             f = x
        #             x = self.classifier(x)
        #             return [x,f]
        #         else:
        #             x = self.classifier(x)
        #             return x
        #
        # # Define the ResNet50-based Model
        # class Net(nn.Module):
        #
        #       def __init__(self, num_classes=751, droprate=0.5,
stride=2, circle=False, ibn=False, linear_num=512, reid=False):
        #         super(Net, self).__init__()
        #         model_ft = models.resnet50(pretrained=True)
        #         if ibn==True:
        #               model_ft = torch.hub.load('XingangPan/IBN-
Net', 'resnet50_ibn_a', pretrained=True)
        #         # avg pooling to global pooling
        #         if stride == 1:
        #           model_ft.layer4[0].downsample[0].stride = (1,1)
        #             model_ft.layer4[0].conv2.stride = (1,1)
        #         model_ft.avgpool = nn.AdaptiveAvgPool2d((1,1))
        #         self.model = model_ft
```

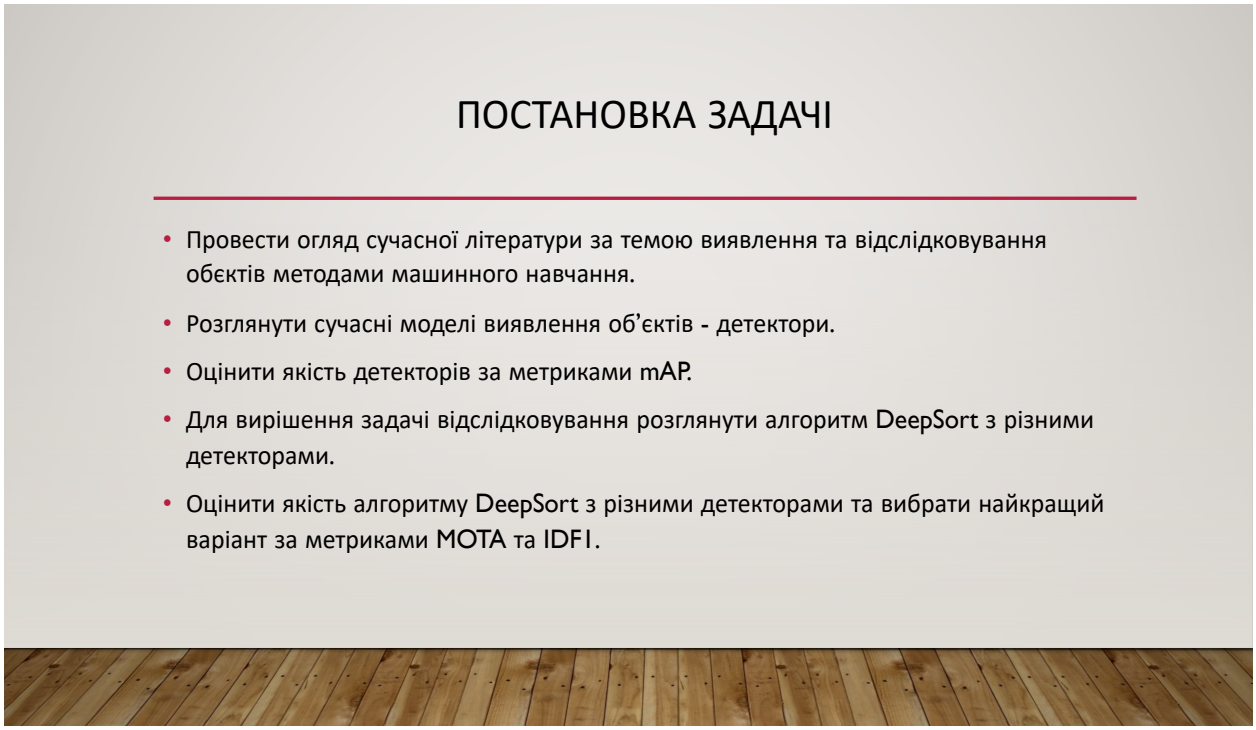
```
      #         self.reid = reid
      #         self.circle = circle
      #           self.classifier = ClassBlock(2048, num_classes,
droprate, linear=linear_num, return_f = circle)
      #
      #     def forward(self, x):
      #         x = self.model.conv1(x)
      #         x = self.model.bn1(x)
      #         x = self.model.relu(x)
      #         x = self.model.maxpool(x)
      #         x = self.model.layer1(x)
      #         x = self.model.layer2(x)
      #         x = self.model.layer3(x)
      #         x = self.model.layer4(x)
      #         x = self.model.avgpool(x)
      #         x = x.view(x.size(0),-1)
      #         # B x 128
      #         if self.reid:
      #             x = x.div(x.norm(p=2,dim=1,keepdim=True))
      #             return x
      #         x = self.classifier(x)
      #         return x

      !python          prepare_market1501.py          --data_dir
/content/Market/Market-1501-v15.09.15

      """<h3>Train descriptor</h3>"""

      !python  train.py  --data-dir  /content/Market/Market-1501-
v15.09.15/pytorch/ --interval 10  --gpu-id 0
```

ДОДАТОК Б ДЕМОНСТРАЦІЙНИЙ МАТЕРІАЛ

Слайд 1:



Слайд 2:



Слайд 3:

ПОСТАНОВКА ЗАДАЧІ (ПРАКТИЧНА ЧАСТИНА)

- Необхідно реалізувати програмний продукт, який буде виконувати виявлення та відстежування об'єктів на відеофрагментах.
- В якості детекторів (моделей виявлення об'єктів) розглядаються здебільшого детектори із сімейтства yolo.
- Для реалізації відстежування використовується алгоритм DeepSort.
- Для тренування дескриптора використовується Market1501 dataset&
- Для тренування та тестування детекторів береться COCO dataset.
- Для тестування кінцевої моделі використовується датасет MOT17.

Слайд 4:

МОДЕЛІ ВИЯВЛЕННЯ ОБ`ЄКТІВ (ДЕТЕКТОРИ)

- Існують дворівневі та одноріневі детектори: сімейство моделей R-CNN – дворівневі, YOLO – одноріневі.
- Дворівневі є занадто громіздкими, що робить надзвичайно обтяжливим з обчислювальної точки зору отримати результати в реальному часі, тому будемо тестувати моделі YOLO.
- При цьому для практичного тестування будемо використовувати останні версії YOLO: v3,v5, v6, v7, v8.

Слайд 5:

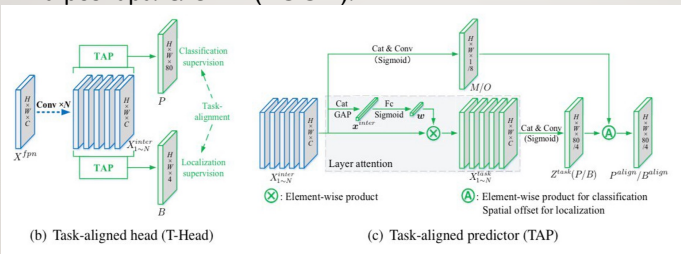
МОДЕЛІ YOLO

- Основною особливістю yolo є те що зображення подається на вхід тільки один раз і одразу отримуються усі передбачувальні обмежувальні рамки.
- Загальною особливістю ранніх моделей є використання якірних рамок, співвідношення їх сторін визначається за допомогою методів кластерного аналізу, такий підхід тривав до п'ятої версії включно.
- Починаючи з шостої версії якірні рамки не використовуються – замість цього вибирають деякі ключові точки, по відношенню до яких визначають відхилення. При цьому використовується розпаралелення голови (точніше техніка TOOD) – окремо передбачаються регресійні змінні та класифікаційні змінні, при цьому з урахуванням взаємозв'язку між цими видами змінних.

Слайд 6:

МОДЕЛІ YOLO

- Техніка розпаралелення (TOOD):

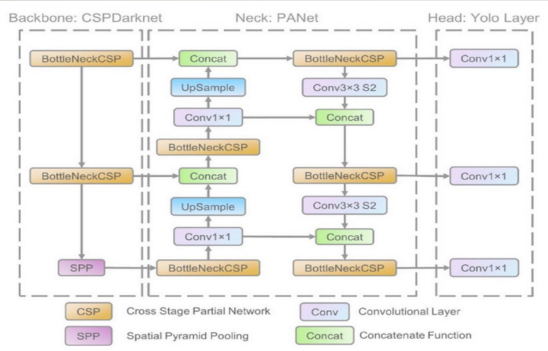


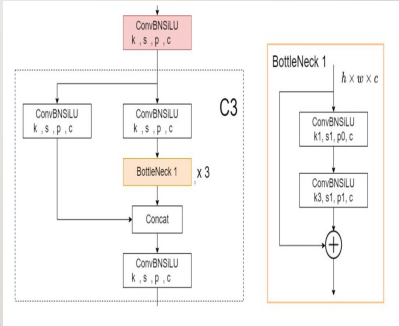
- Ця техніка паралельно обчислює класифікаційну та регресійні складові (схема (c) для кожної складової), однак при цьому враховується взаємозв'язок між ними (спільний отраний за схемою (b))

Слайд 7:

ПРИКЛАДИ АРХІТЕКТУР

- Загальна архітектура yolov5:

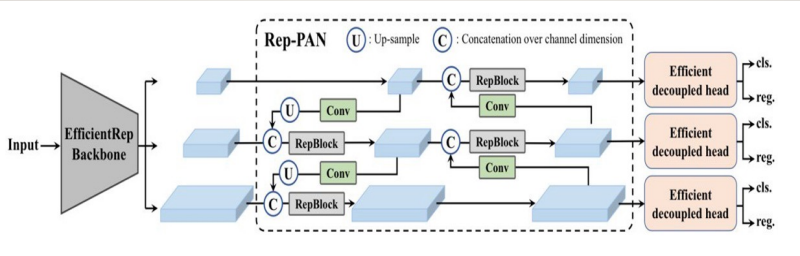




Слайд 8:

ПРИКЛАДИ АРХІТЕКТУР

- Загальна архітектура yolov6:



- Під Efficient decoupled head мається на увазі техніка розпералелення голови – TOOD.

Слайд 9:

МЕТРИКИ ЯКОСТІ ДЛЯ ДЕТЕКТОРІВ

- Головна метрика якості – mAP.
- mAP – це є усереднений показник AP по всім класам.
- AP – площа під кривою PR (Precisio-Recall).
- Базовим поняттям в даній тематиці також є метрика IoU.
- IoU = (площа перетину)/(площа об'єднання).
- mAP50 – mAP при IoU_threshold = 0.5 (якщо IoU>IoU_threshold, то передбачена обмежувальна рамка справді містить деякий об'єкт).
- mAP50-95 – середній mAP при зміні IoU_threshold від 0.5 до 0.95 .

Слайд 10:

ВІДБІР ДЕТЕКТОРІВ.ПОЧАТКОВИЙ ЕТАП.

- Результати тестування детекторів на COCO validation dataset розміром 1700 зображень:

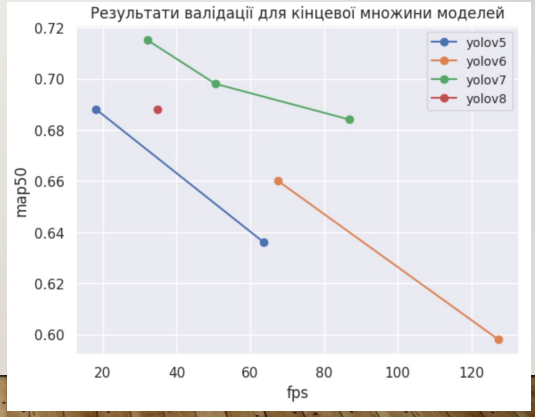
	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov3	0.648	0.463	42.918	37.175	119.0	640.0
yolov5s	0.563	0.376	192.308	119.048	14.1	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov5l	0.664	0.485	46.296	39.683	89.3	640.0
yolov5x	0.681	0.503	24.331	22.831	166.0	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov5l6	0.709	0.533	11.086	10.661	147.0	1280.0
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov6m	0.651	0.486	70.423	63.694	72.6	640.0
yolov6l	0.682	0.511	43.668	41.152	114.0	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov6l_mbla	0.677	0.506	53.476	49.751	88.7	640.0
yolov6x_mbla	0.692	0.520	33.670	32.154	151.0	640.0
yolov7	0.684	0.496	98.039	86.957	72.1	640.0
yolov7x	0.698	0.512	54.054	50.505	136.0	640.0
yolov7w6	0.715	0.528	34.247	32.154	134.7	1280.0
yolov8m	0.659	0.497	64.103	57.803	49.7	640.0
yolov8l	0.688	0.528	37.736	34.722	83.7	640.0
yolov8x	0.702	0.538	22.321	21.368	131.0	640.0

Слайд 11:

ВІДБІР ДЕТЕКТОРІВ.КІНЦЕВИЙ ЕТАП.

- Після відбору тих детекторів, які явно домінують над деякими іншими детекторами, маємо такі результати:

	mAP50	mAP50-95	fps(without_NMS)	fps(with_NMS)	model_size(MB)	image_size(resolution)
yolov6s	0.598	0.437	157.978	127.389	38.9	640.0
yolov5m	0.636	0.450	80.000	63.694	40.8	640.0
yolov6m_mbla	0.660	0.488	75.643	67.568	50.8	640.0
yolov7	0.684	0.496	98.039	86.957	72.1	640.0
yolov5m6	0.688	0.511	19.342	18.149	69.0	1280.0
yolov8l	0.688	0.528	37.736	34.722	83.7	640.0
yolov7x	0.698	0.512	54.054	50.505	136.0	640.0
yolov7w6	0.715	0.528	34.247	32.154	134.7	1280.0



Слайд 12:

АЛГОРИТМ DEEPSORT

- DeepSort – це алгоритм для відслідковування об’єктів з використанням вже навченого детектора.
- Він надає кожному об’єкту унікальний ідентифікатор.
- Співвіднесення ідентифікаторів між двома сусідніми кадрами відбувається із застосуванням відстані Махаланоубіса та вектра ознак об’єкта, який видається іншою згортковою нейромережею.
- Для врахування спострежень (виходу детектора) та проекції координат об’єкта на наступний кадр використовується фільтр Калмана.
- Для тренування детекторів в цій роботі використовувався Market1501 dataset.

Слайд 13:

МЕТРИКИ MOTA

- Основні метрики з класу метрик MOTA для задач відслідковування це MOTA, MOTP, ID(-S).
- MOTA - правильність відслідковування за багатьма об’єктами (максимум = 1).

$$MOTA = 1 - \frac{\sum_t FN_t + FP_t + IDS_t}{\sum_t GT_t}$$

- MOTP (точність відслідковування за багатьма об’єктами) – враховує лише якість регресійної складової моделі.

$$\text{MOTP} = \frac{\sum_{i,t} d_t^i}{\sum_t c_t}.$$

- ID - кількість неправильних перенесень id з t-1 - го кадру в наступний.

Слайд 14:

МЕТРИКИ ІДЕНТИФІКАЦІЇ

- Інший широковикористовуваний клас метрик – метрики ідентифікації.
- Головна метрика в даному класі метрик - IDF1 (identification F1-score).

$$\text{IDF1} = \frac{|\text{IDTP}|}{|\text{IDTP}| + 0.5\,|\text{IDFN}| + 0.5\,|\text{IDFP}|}$$

- IDF1 для кожної передбаченої траєкторії підраховує кількість кадрів, на яких передбачений ідентифікатор був на правильному місці (тобто належав об'єкту за яким слідкував найбільшу частину своєї траєкторії) по відношенню до загальної кількості зроблених прогнозів та наявних справжніх об'єктів.
- Також вводяться метрики IDP та IDR (indetification precision та recall).

Слайд 15:

РЕЗУЛЬТАТИ АЛГОРИТМУ DEEPSORT. МЕТРИКИ MOTA.

- За реультатами тестування алгоритму маємо, що найкращими за метриками MOTA виявилися моделі yolo v6:

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
yolov5m	COMBINED	0.186770	0.737150	15974	61806	1359	19	171	17333	77780	154	255	21.15625
yolov7-w6	COMBINED	0.206274	0.736429	17622	60158	1495	20	164	19117	77780	162	255	17.20000
yolov8l	COMBINED	0.207547	0.743490	17586	60194	1359	20	164	18945	77780	168	255	15.26875
yolov5m6	COMBINED	0.251440	0.738176	21446	56334	1779	23	150	23225	77780	185	255	15.16875
yolov7	COMBINED	0.270327	0.737984	23254	54526	2104	25	138	25358	77780	199	255	16.43125
yolov7x	COMBINED	0.274441	0.738636	23423	54357	1953	25	138	25376	77780	206	255	12.58750
yolov6s	COMBINED	0.298046	0.733926	26054	51726	2726	26	137	28780	77780	215	255	20.11250
yolov6m_mbla	COMBINED	0.332759	0.730104	31098	46682	4980	29	119	36078	77780	276	255	15.55625

Слайд 16:

РЕЗУЛЬТАТИ АЛГОРИТМУ DEEPSORT. МЕТРИКИ IDF1.

- Показники за метриками IDF1 демонструють майже аналогічний результат:

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs	fps
yolov5m	COMBINED	0.281623	13393	64387	3940	17333	77780	154	255	21.15625
yolov7-w6	COMBINED	0.302218	14642	63138	4475	19117	77780	162	255	17.20000
yolov8l	COMBINED	0.321034	15526	62254	3419	18945	77780	168	255	15.26875
yolov5m6	COMBINED	0.344023	17374	60406	5851	23225	77780	185	255	15.16875
yolov7	COMBINED	0.376486	19415	58365	5943	25358	77780	199	255	16.43125
yolov6s	COMBINED	0.385586	20544	57236	8236	28780	77780	215	255	20.11250
yolov7x	COMBINED	0.387355	19979	57801	5397	25376	77780	206	255	12.58750
yolov6m_mbla	COMBINED	0.443166	25229	52551	10849	36078	77780	276	255	15.55625

Слайд 17:

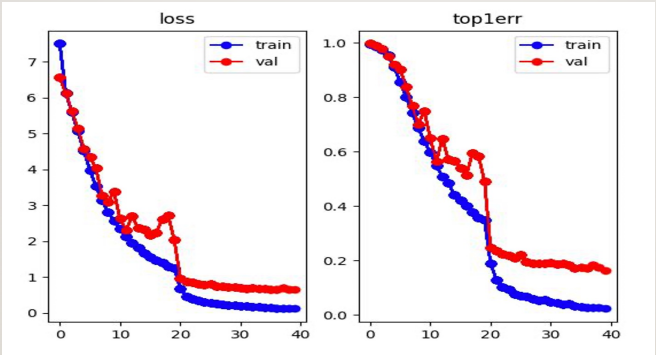
ЗМІНА ДЕСКРИПТОРА В DEEPSORT

- Змінимо дескриптор на більш складний (глибокий), однак з використанням технік економії кількості параметрів – resnet50. Архітектура цього дескриптора зображена нижче:

Слайд 18:

ЗМІНА ДЕСКРИПТОРА В DEEPSORT

- Процес тренування та валідації resnet50 зображено нижче:



Слайд 19:

АЛГОРИТМУ DEEPSORT З ДЕСКРИПТОРОМ RESNET50. МЕТРИКИ MOTA.

- Результати з новим дектриптором для метрик MOTA:

	seq	MOTA	MOTP	CLR_TP	CLR_FN	CLR_FP	MT	ML	Dets	GT_Dets	IDs	GT_IDs	fps
yolov7-w6	COMBINED	0.207187	0.735007	17822	59958	1596	20	161	19418	77780	102	255	12.38750
yolov8l	COMBINED	0.210607	0.742068	17939	59841	1433	21	161	19372	77780	100	255	11.43750
yolov5m6	COMBINED	0.253394	0.736692	21739	56041	1875	24	145	23614	77780	117	255	11.33125
yolov5m	COMBINED	0.264297	0.734829	22763	55017	2042	23	145	24805	77780	119	255	13.68125
yolov7	COMBINED	0.272821	0.736639	23629	54151	2231	27	138	25860	77780	116	255	11.50000
yolov7x	COMBINED	0.275534	0.736848	23757	54023	2135	27	134	25892	77780	119	255	9.60000
yolov6s	COMBINED	0.298676	0.732317	26408	51372	2936	26	138	29344	77780	128	255	13.02500
yolov6m_mbla	COMBINED	0.333209	0.730205	31597	46183	5364	34	113	36961	77780	144	255	10.60000

Слайд 20:

АЛГОРИТМУ DEEPSORT З ДЕСКРИПТОРОМ RESNET50. МЕТРИКИ IDF1.

- Результати з новим дектриптором для метрик IDF1:

	seq	IDF1	IDTP	IDFN	IDFP	Dets	GT_Dets	IDs	GT_IDs	fps
yolov7-w6	COMBINED	0.274327	13332	64448	6086	19418	77780	102	255	12.38750
yolov8l	COMBINED	0.281888	13693	64087	5679	19372	77780	100	255	11.43750
yolov5m6	COMBINED	0.318066	16125	61655	7489	23614	77780	117	255	11.33125
yolov7x	COMBINED	0.330581	17136	60644	8756	25892	77780	119	255	9.60000
yolov5m	COMBINED	0.331277	16992	60788	7813	24805	77780	119	255	13.68125
yolov7	COMBINED	0.334794	17349	60431	8511	25860	77780	116	255	11.50000
yolov6s	COMBINED	0.343079	18376	59404	10968	29344	77780	128	255	13.02500
yolov6m_mbla	COMBINED	0.393722	22588	55192	14373	36961	77780	144	255	10.60000

Слайд 21:

ПОКАЗНИКИ ЕФЕКТИВНОСТІ ІНШИХ ТРЕКЕРІВ

Table 1: The *MOT15* leaderboard. Performance of several trackers according to different metrics.

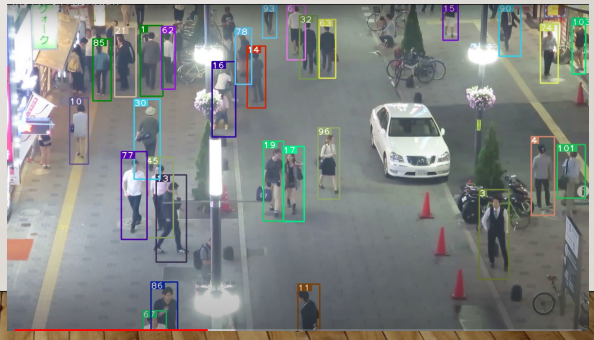
Method	MOTA	IDF1	MOTP	FAR	MT	ML	FP	FN	IDSW	FM	IDSWR	FMR
MPNTrack [Brasó and Leal-Taixé, 2020]	51.54	58.61	76.05	1.32	225	187	7620	21780	375	872	5.81	13.51
Tracktor++v2 [Bergmann et al., 2019]	46.60	47.57	76.36	0.80	131	201	4624	26896	1290	1702	22.94	30.27
TrctrD15 [Xu et al., 2020]	44.09	45.99	75.26	1.05	124	192	6095	26917	1347	1868	23.97	33.24
Tracktor++ [Bergmann et al., 2019]	44.06	46.73	75.03	1.12	130	189	6477	26577	1318	1790	23.23	31.55
KCF [Chu et al., 2019]	38.90	44.54	70.56	1.27	120	227	7321	29501	720	1440	13.85	27.76
AP_HWDPL_p [Long et al., 2017]	38.49	47.10	72.56	0.69	63	270	4005	33203	586	1263	12.75	27.48
STRN [Xu et al., 2019]	38.06	46.62	72.06	0.94	83	241	5451	31571	1033	2665	21.25	54.82
AMIR15 [Sadeghian et al., 2017]	37.57	46.01	71.66	1.37	114	193	7933	29397	1026	2024	19.67	38.81
JointMC [Keuper et al., 2018]	35.64	45.12	71.90	1.83	167	163	10580	28508	457	969	8.53	18.08
RAR15pub [Fang et al., 2018]	35.11	45.40	70.94	1.17	94	305	6771	32717	381	1523	8.15	32.58
HybridDAT [Yang and Jia, 2017]	34.97	47.72	72.57	1.46	82	304	8455	31140	358	1267	7.26	25.69
INARLA [Wu et al., 2019]	34.69	42.06	70.72	1.71	90	216	9855	29158	1112	2848	21.16	54.20
STAM [Chu et al., 2017]	34.33	48.26	70.55	0.89	82	313	5154	34848	348	1463	8.04	33.80
QuadMOT [Son et al., 2017]	33.82	40.43	73.42	1.37	93	266	7898	32061	703	1430	14.70	29.91
NOMT [Choi, 2015]	33.67	44.55	71.94	1.34	88	317	7762	32547	442	823	9.40	17.50
DCCRP [Zhou et al., 2018a]	33.62	39.08	70.91	1.02	75	271	5917	34002	866	1566	19.39	35.07
TDAM [Yang and Jia, 2016]	33.03	46.05	72.78	1.74	96	282	10064	30617	464	1506	9.25	30.02
CDA_DDALpb [Bae and Yoon, 2018]	32.80	38.79	70.70	0.86	70	304	4983	35690	614	1583	14.65	37.77
MHT_DAM [Kim et al., 2015]	32.36	45.31	71.83	1.57	115	316	9064	32060	435	826	9.10	17.27
LFNF [Sheng et al., 2017]	31.64	33.10	72.03	1.03	69	301	5943	35095	961	1106	22.41	25.79
GMPHD_OGM [Song et al., 2019]	30.72	38.82	71.64	1.13	83	275	6502	35030	1034	1351	24.05	31.43
PHD_GSDL [Fu et al., 2018]	30.51	38.82	71.20	1.13	55	297	6534	35284	879	2208	20.65	51.87
MDP [Xiang et al., 2015]	30.31	44.68	71.32	1.68	94	277	9717	32422	680	1500	14.40	31.76
MCF_PHD [Wojke and Paulus, 2016]	29.89	38.18	71.70	1.54	86	317	8892	33529	656	989	14.44	21.77
CNNTCM [Wang et al., 2016]	29.64	36.82	71.78	1.35	81	317	7786	34733	712	943	16.38	21.69
RSCNN [Mahgoub et al., 2017]	29.50	36.97	73.07	2.05	93	262	11866	30474	976	1176	19.36	23.33
TBSS15 [Zhou et al., 2018b]	29.21	37.23	71.28	1.05	49	316	6068	36779	649	1508	16.17	37.57
SCEA [Yoon et al., 2016]	29.08	37.15	71.11	1.05	64	341	6060	36912	604	1182	15.13	29.61
SiameseCNN [Leal-Taixe et al., 2016]	29.04	34.27	71.20	0.89	61	349	5160	37798	639	1316	16.61	34.20
HAM_INTP15 [Yoon et al., 2018a]	28.62	41.45	71.13	1.30	72	317	7485	35910	460	1038	11.07	24.98
GMMA_intp [Song et al., 2018]	27.32	36.59	70.92	1.36	47	311	7848	35817	987	1848	23.67	44.31
oICF [Kieritz et al., 2016]	27.08	40.49	69.96	1.31	46	351	7594	36757	454	1660	11.30	41.32
TO [Manen et al., 2016]	25.66	32.74	72.17	0.83	31	414	4779	40511	383	600	11.24	17.61
LP_SSVM [Wang and Fowlkes, 2016]	25.22	34.05	71.68	1.45	42	382	8369	36932	646	849	16.19	21.28
HAM_SADF [Yoon et al., 2018a]	25.19	37.80	71.38	1.27	41	420	7330	38275	357	745	9.47	19.76
ELP [McLaughlin et al., 2015]	24.99	26.21	71.17	1.27	54	316	7345	37344	1396	1804	35.60	46.00
AdTobKF [Loumponias et al., 2018]	24.82	34.50	70.78	1.07	29	375	6201	39321	666	1300	18.50	36.11
LINF1 [Fagot-Bouquet et al., 2016]	24.53	34.82	71.33	1.01	40	466	5864	40207	298	744	8.62	21.53
TENSOR [Shi et al., 2018]	24.32	24.13	71.58	1.15	40	336	6644	38582	1271	1304	34.16	35.05
TFMOT [Boragule and Jeon, 2017]	23.81	32.30	71.35	0.78	35	447	4533	41873	404	792	12.69	24.87
JPDA_m [Rezatofighi et al., 2015]	23.79	33.77	68.17	1.10	36	419	6373	40084	365	869	10.50	25.00
MotiCon [Leal-Taixé et al., 2014]	23.07	29.38	70.87	1.80	34	375	10404	35844	1018	1061	24.44	25.47
DEEPDA_MOT [Yoon et al., 2019a]	22.53	25.92	70.92	1.27	46	447	7346	39092	1159	1538	31.86	42.28
SegTrack [Milan et al., 2015]	22.51	31.48	71.05	1.36	42	461	7890	39020	697	737	19.10	20.20
EAMTpub [Sanchez-Matilla et al., 2016]	22.30	32.84	70.79	1.37	39	380	7924	38982	833	1485	22.79	40.63
SAS_MOT15 [Maksai and Fua, 2019]	22.16	37.15	71.10	0.97	22	444	5591	41531	700	1240	21.60	38.27
OMT_DFH [Ju et al., 2017]	21.16	37.34	69.94	2.29	51	335	13218	34657	563	1255	12.92	28.79
MTSTracker [Nguyen Thi Lan Anh et al., 2017]	20.64	31.87	70.32	2.62	65	266	15161	32212	1387	2357	29.16	49.55
TC_SIAMESE [Yoon et al., 2018b]	20.22	32.59	71.09	1.06	19	487	6127	42596	294	825	9.59	26.90
DCO_X [Milan et al., 2016]	19.59	31.45	71.39	1.84	37	396	10652	38232	521	819	13.79	21.68
CEM [Milan et al., 2014]	19.30	N/A	70.74	2.45	61	335	14180	34591	813	1023	18.60	23.41
RNN_LSTM [Milan et al., 2017]	18.99	17.12	70.97	2.00	40	329	11578	36706	1490	2081	37.01	51.69
RMOT [Yoon et al., 2015]	18.63	32.56	69.57	2.16	38	384	12473	36835	684	1282	17.08	32.01
TSDA_OAL [Ju et al., 2017]	18.61	36.07	69.68	2.83	68	305	16350	32853	806	1544	17.32	33.18
GMPHD_15 [Song and Jeon, 2016]	18.47	28.38	70.90	1.36	28	399	7864	41766	459	1266	14.33	39.54
SMOT [Dicle et al., 2013]	18.23	0.00	71.23	1.52	20	395	8780	40310	1148	2132	33.38	61.99
ALExTRAC [Bewley et al., 2016b]	16.95	17.30	71.18	1.60	28	378	9233	39933	1859	1872	53.11	53.48
TBD [Geiger et al., 2014]	15.92	0.00	70.86	2.58	46	345	14943	34777	1939	1963	44.68	45.23
GSCR [Fagot-Bouquet et al., 2015]	15.78	27.90	69.38	1.31	13	440	7597	43633	514	1010	17.73	34.85
TC_ODAL [Bae and Yoon, 2014]	15.13	0.00	70.53	2.24	23	402	12970	38538	637	1716	17.09	46.04
DP_NMS [Pirsiavash et al., 2011]	14.52	19.69	70.76	2.28	43	294	13171	34814	4537	3090	104.69	71.30

Слайд 22:

ІЛЮСТАЦІЯ РОБОТИ DEEPSORT+YOLOV6M_MBLA

Фрагмент відео MOT17-04
9-а секунда

Фрагмент відео MOT17-04
23-я секунда



Слайд 23:

ВИСНОВКИ

- Моделі yolo, які найкраще взаємодіють з алгоритмом DeepSort, виявились yolov6s та yolov6m_mbla.
- У результаті тестування було виявлено, що оригінальний дескриптор є доволі непоганим балансом між розміром та якістю тому що подальше його ускладнення (наприклад, у вигляді resnet50) не дає значних покращень.
- Даний програмний продукт може бути застосований у моніторинг в різних сферах людської діяльності - від відслідковування автомобілів (або номерних знаків та їх подальше розпізнавання) на дорогах до відстеження гравців в спортивних іграх.
- Якщо застосовувати даний продукт для деякої конкретної вузькоспеціалізованої задачі то варто зібрати датасет, який стосується саме цієї задачі та дотренувати нейромережу yolo на ньому. Через вузькоспеціалізованність задачі важливо ретельно підбирати датасет щоб не було перенавчання.

Слайд 24:

ВИСНОВКИ

- За показниками якості для детекторів можна стверджувати, що модель yolov7-w6 чудово підходить для виявлення об'єктів (map50 = 0.715).
- За показниками ефективності інших трекерів – комбінація DeepSort + yolov6 дає краще за середній результат.
- Отже, даний програмний продукт можна використовувати для відслідковування найближчих об'єктів і при цьому досягати відносно непоганої точності.
- Якщо перед вами стоїть задача виявлення то достатньо використовувати графічний процесор Nvidia Tesla T4 щоб досягти real-time результатів, для задачі відслідковування можна взяти Nvidia V100 або Nvidia A100.

Слайд 25: