Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Технологія розпізнавання шрифту Брайля на базі нейронної мережі

(тема)

Виконав:

студент II курсу, групи СПм-22-2

Врублевський В. О.
(прізвище, ініціали)

Спеціальність

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: доц. Бовчалюк С. Я.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А. А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав.
кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Врублевському Володимиру Олекандровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Технологія розпізнавання шрифту Брайля на базі нейронної мережі

затверджена наказом по університету від " 06 " листопада 2023 р. № 1299Ст

2. Термін подання студентом роботи до екзаменаційної комісії 15 січня 2023 р.

3. Вхідні дані до роботи 1) технології оптичного розпізнавання шрифту Брайля
2) технології розробки нейронних мереж
3) алгоритми бінаризації зображень
4) хмарні технології

4. Перелік питань, що потрібно опрацювати у роботі
1) Аналіз необхідності покращення технологій розпізнавання шрифту Брайля
2) Аналіз існуючих методів машинного навчання
3) Аналіз існуючих завдань комп'ютерного зору
4) Аналіз існуючих нейронних мереж
5) Аналіз існуючих методів бінаризації
6) Огляд сучасних інструментів розробки
7) Розробка технології розпізнавання шрифту Брайля на базі нейронної мережі
8) Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 13 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз необхідності покращення технологій	07.11.23-09.11.23	
	розпізнавання шрифту Брайля		
2	Аналіз існуючих методів машинного навчання	10.11.23-11.11.23	
3	Аналіз існуючих завдань комп'ютерного зору	12.11.23-14.11.23	
4	Аналіз існуючих нейронних мереж	15.11.23-16.11.23	
5	Аналіз існуючих методів бінаризації	17.11.23-18.11.23	
6	Огляд сучасних інструментів розробки	19.11.23-30.11.23	
7	Розробка технології для розпізнавання шрифту	01.12.23-01.01.24	
	Брайля		
	Оформлення документації	02.01.24-07.01.24	
9	Подання кваліфікаційної роботи керівникові та	08.01.24-12.01.24	
	та її попередній захист		
10	Подання кваліфікаційної роботи на	13.01.24	
	рецензування		

Дата видачі завдання 6 листопада 2023 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Бовчалюк С. Я.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 112 с., 52 рис., 2 табл., 3 дод., 30 джерел.

НЕЙРОННІ МЕРЕЖІ, ШРИФТ БРАЙЛЯ, ХМАРНІ ТЕХНОЛОГІЇ, БІНАРИЗАЦІЯ ЗОБРАЖЕНЬ, RETINANET

Метою кваліфікаційної роботи є дослідження методів і алгоритмів розпізнавання шрифту Брайля, визначення їхніх переваг і недоліків, та покращення функціонування, шляхом впровадження сучасних технологій на базі нейронної мережі. Практичним результатом проведеного дослідження є розробка програмного рішення для автоматичного перекладу шрифту Брайля, що міститься на зображеннях.

У ході виконання кваліфікаційної роботи було проведено детальний аналіз поточного стану питання розпізнавання шрифту Брайля на зображеннях. Досліджено існуючі методи бінаризації зображень, що можуть бути використані для покращення результатів розпізнавання. Розглянуто розвиток нейронних мереж, починаючи від примітивних, закінчуючи сучасними архітектурами, проаналізовані їх недоліки та переваги. Розглянуто сучасні середовища розробки, технології, фреймворки та мови програмування, що є актуальними для розробки технології детекції об’єктів на зображенні. Реалізовано технологію розпізнавання шрифту Брайля на базі нейронної мережі.

ABSTRACT

Master’s thesis: 112 pages, 52 figures, 2 tables, 3 appendices, 30 sources.

NEURAL NETWORKS, BRAILLE, CLOUD TECHNOLOGIES, BINARIZATION OF IMAGES, RETINANET

The purpose of the qualification work is to research the methods and algorithms of braille recognition, to determine their advantages and disadvantages, and to improve their functioning by implementing modern technologies based on a neural network. The practical result of the conducted research is the development of a software solution for the automatic translation of the Braille font contained in the images.

In the course of the qualification work, a detailed analysis of the current state of Braille recognition on images was carried out. Existing methods of image binarization that can be used to improve recognition results are studied. The development of neural networks, starting from primitive to modern architectures, is considered, their disadvantages and advantages are analyzed. Modern development environments, technologies, frameworks and programming languages, which are relevant for the development of image object detection technology, are considered. Braille recognition technology based on a neural network has been implemented.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ADAM – оцінка адаптивного моменту (англ., Adaptive Moment Estimation)

API – програмний інтерфейс застосунку (англ., Application Programming Interface)

CNN – згорткова нейронна мережа (англ., Convolutional Neural Network)

CRF – умовні випадкові поля (англ., Conditional Random Fields)

DLL – бібліотека динамічних посилань (англ., Dynamic Link Library)

FPN – пірамідальна мережа для признаків (англ., Feature Pyramid Network)

HOG – гістограма орієнтованих градієнтів (англ., Histogram of Oriented Gradients)

ILSVRC – конкурс оптичного розпізнавання ImageNet (англ., ImageNet Large Scale Visual Recognition Challenge)

IOU – перетин над об’єднанням (англ., Intersection Over Union)

JSON – формат об'єктів JavaScript (англ., JavaScript Object Notation)

LBP – локальний бінарний шаблон (англ., Local Binary Pattern)

NMS – пригнічення не максимумів (англ., Non-Maximum Suppression)

OCR – оптичне розпізнавання шрифту Брайля (англ., Optical Braille Recognition)

ReLU – лінійна ректифікація (англ., Rectified Linear Unit)

SVM – машина опорних векторів (англ., Support Vector Machine)

ВСТУП

Шрифт Брайля – це рельєфно-крапковий тактильний шрифт, який використовують незрячі та слабозорі люди для читання і письма. Він був винайдений у 1824 року французьким незрячим юнаком Луїсом Брайлем.

Розпізнавання шрифту Брайля – це завдання, що полягає в перетворенні зображення рельєфних точок у текстовий формат. Розв'язання цієї задачі може бути корисним для різних цілей, таких як розробка програмного забезпечення, що може переводити текст у шрифт Брайля, або створення електронних пристроїв для зчитування шрифту Брайля. Також розв'язання цієї задачі може значно спростити процес навчання людей з обмеженими зоровими здібностями, оскільки незрячі люди спочатку навчаються з використанням шрифту Брайля.

На даний час, одним із перспективних підходів до розв’язання задач розпізнавання шрифтцу Брайля є застосування нейронних мереж. Нейронні мережі – це алгоритми машинного навчання, що можуть навчатися на великих наборах даних і виконувати складні завдання, які важко або неможливо вирішити за допомогою традиційних методів.

Нейронні мережі мають низку переваг перед традиційними методами розпізнавання шрифту Брайля:

- висока точність: нейронні мережі можуть досягати високої точності розпізнавання, навіть якщо зображення шрифту Брайля містить шум або дефекти;
- швидкість: нейронні мережі можуть розпізнавати шрифт Брайля дуже швидко, що робить їх придатними для використання в реальному часі;
- гнучкість: нейронні мережі можуть бути адаптовані для розпізнавання різних типів шрифту Брайля.

Розпізнавання шрифту Брайля на базі нейронної мережі включає в себе наступні етапи:

- попередня обробка зображення: аугментація вихідного зображення з

метою поліпшення якості, виділення необхідних ознак, а також зменшення кількості непотрібної інформації;

- визначення ознак: на цьому етапі із зображення шрифту Брайля витягують ознаки, які будуть використовуватися нейронною мережею для розпізнавання;

- виявлення символів: виявлення символів, які належать до класу шрифту Брайля;

- класифікація: на цьому етапі нейронна мережа класифікує виявлені символи;

- переклад: переклад виявлених символів на бажану мову.

Таким чином завдяки нейронним мережам розпізнавання шрифту Брайля може стати набагато ефективнішим, адже процес розпізнавання символів нейронною мережею надзвичайно швидкий та точний, навіть при наявності шуму або дефектів у зображенні. Гнучкість нейронних мереж дозволяє їм адаптуватися до різних типів цього шрифту, що робить їх незамінним інструментом для полегшення життя людей з обмеженими зоровими можливостями.

1 АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Аналіз необхідності покращення технологій розпізнавання шрифту Браля

Шрифт Брайля – це рельєфно-крапковий тактильний шрифт, який дає змогу людям з обмеженими можливостями зору читати і писати. Нині освітній процес таких людей починається з навчання тактильному сприйняттю цього шрифту, тому тактильне сприйняття такої людини більш чутливе, ніж у звичайної, тому вони читають цей шрифт за допомогою очей.

Прочитання таким методом для непідготовленої людини є достатньо непростим завданням, тому звичайні люди створюють різні способи автоматизації цього процесу.

Існує багато методів розпізнавання шрифту Брайля, що об'єднані під терміном OBR (Optical Braille Recognition). Процес розпізнавання у цих методах розбивається на етапи, обумовлені ключовими відмінностями шрифту Брайля від алфавітів різних мов.

У стандартній варіації шрифту Брайля символи представлені комбінаціями шести точок, розташованих у вузлах уявної сітки (рисунок 1.1). Угруповання точок у літери визначається прив'язкою до цієї сітки. Поза прив'язкою до сітки, що утворюється всіма брайлівськими літерами, точки не мають сенсу. Так, якщо ми бачимо на порожньому аркуші паперу букву А, то легко розуміємо, що це буква А. Однак якщо ми бачимо брайлівську точку окремо від сітки, то достовірно не можна визначити який символ вона представляє і чи є символом взагалі, або ж це просто дефект паперу.

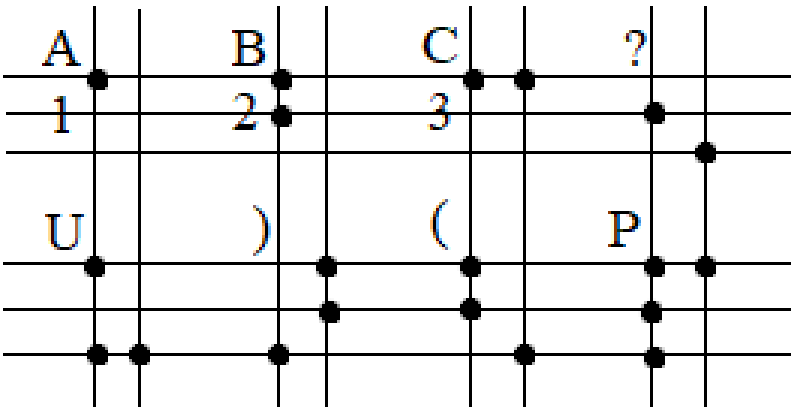


Рисунок 1.1 – Кодування латинського алфавіту шрифтом Брайля

Внаслідок цього, для описаних у літературі алгоритмів OBR практично стандартом виявляється наступний набір кроків [1]:

- знайти точки;
- відновити уявну сітку;
- порівняти знайдені точки з вузлами сітки;
- розпізнати символи;
- конвертувати послідовність символів у звичайний текст.

Різні автори використовують різні методи знаходження точок: динамічний поріг, детектор кіл на основі перетворення Хафа, HOG, LBP, SVM, ознаки Хаара та алгоритм Віоли-Джонса тощо.

Після цього для відновлення сітки використовують лінійну регресію, перетворення Хафа, зміну гістограми розподілу координат точок при покроковому повороті листа.

Однак, при такому підході потрібна єдина на весь аркуш уявна сітка, до якої прив'язані точки. Саме тому сторінка має бути не просто сфотографована, а відсканована. Тільки в цьому випадку по всьому аркушу точки будуть розташовані на паралельних лініях сітки.

Також існує готове комерційне рішення від компанії ЕлекЖест (рисунок 1.2), проте його габарити залишають бажати кращого: ширина 750 мм, довжина 775 мм, висота 510 мм, вага 22 кг.

Рисунок 1.2 – Система OBR компанії ЕлекЖест

Підсумовуючи, можна побачити, що існуючі методи оптичного розпізнавання тексту (OBR) мають низку обмежень. По-перше, вони вимагають сканування зображення, яке може бути ускладненим у разі дефектного паперу або необхідності сканування книжкового розвороту. По-друге, існуючі методи OBR не завжди забезпечують високу точність, зручність та швидкість, а готові рішення можуть бути дорогими та складними у використанні.

1.2 Аналіз існуючих методів машинного навчання

Нейронна мережа – це математична модель, яка використовується для аналізу даних та вирішення завдань, які важко чи неможливо формалізувати. Нейронні мережі мають здатність навчатися на наданих даних і прогнозувати результати на основі отриманого досвіду. Вони складаються з шарів нейронів, які обробляють інформацію та передають її на наступний шар. Типи навчання нейронних мереж можна розділити на три умовні категорії:

- supervised learning;
- unsupervised learning;
- reinforcement learning.

Supervised learning (контрольоване навчання) – це тип машинного навчання, у якому алгоритм навчається на заздалегідь підготовлених датасетах. Датасет при такому типі навчання складається з міток, кожна з яких включає дані, що подаються на вхід нейронної мережі і очікуваний результат. Контрольоване навчання є найбільш поширеним типом машинного навчання. Воно використовується для вирішення широкого спектру завдань, включаючи:

- класифікацію: визначення категорії об'єкта або події. Наприклад, алгоритм класифікації може використовуватися для визначення того, хто знаходиться на зображенні кішка або собака;

- регресію: прогнозування значення змінної на основі інших змінних. Наприклад, алгоритм регресії може використовуватись для прогнозування ціни будинку на основі його характеристик та оточення;

- ранжування: впорядкування елементів відповідно до їх важливості або релевантності, які можуть бути застосовані до різних типів елементів, таких як документи, веб-сторінки, продукти або навіть люди.

Переваги використання supervised learning:

- може використовуватися для вирішення широкого спектру завдань, включаючи класифікацію, регресію та ранжування;

- вимагають менше даних, ніж інші методи навчання;

- результати навчання не важко інтерпретувати.

Недоліки:

- вимагає наявності розмічених даних для навчання;

- може бути менш ефективним для вирішення завдань, які важко формалізувати.

Unsupervised learning – це тип машинного навчання, при якому алгоритм навчається на наборі даних, в якому немає міток, що вказують на правильну відповідь. Алгоритм вчиться виявляти певні закономірності та патерни у даних. Неконтрольоване навчання використовується для наступних типів завдань:

- кластеризація: поділ даних на групи, що базуються на схожості. Наприклад, алгоритм кластеризації може бути використаний для поділу

зображень на групи, засновані на їх колірній гамі;

- виявлення аномалій: виявлення даних, які не відповідають нормальному шаблону. Наприклад, алгоритм виявлення аномалій може використовуватися виявлення шахрайства у фінансових транзакціях;

- зменшення розмірності: зменшення кількості змінних у наборі даних, зберігаючи важливу інформацію. Наприклад, алгоритм зниження розмірності може використовуватися для зменшення розміру набору даних зображень, не втрачаючи при цьому їх ідентичності.

Переваги:

- не потребує наявності розмічених даних для навчання;

- може використовуватись для вирішення завдань, що важко формалізувати;

- може бути більш ефективним для вирішення завдань, що потребують виявлення закономірностей та патернів.

Недоліки:

- результати можуть бути менш інтерпретованими, ніж результати контрольованого навчання;

- може бути менш ефективним для вирішення завдань, що потребують прогнозування.

Reinforcement learning – це тип машинного навчання, в якому агент навчається приймати рішення, отримуючи винагороди за певні дії. Агент взаємодіє із середовищем, виконуючи дії та спостерігаючи за результатами цих дій. Результати дій можуть бути як позитивними так і негативними. Позитивні результати підкріплюються нагородами, а негативні покараннями. Агент використовує нагороди та покарання для навчання моделі поведінки, яка максимізує отримання нагород. Навчання з підкріпленням можна розділити на два основні типи:

- однокрокове навчання: агент отримує нагороду або покарання одразу після виконання дії;

- багатокрокове навчання: агент отримує нагороду або покарання через

декілька дій.

Навчання з підкріпленням є найменш поширеним типом і може використовуватися в наступних сферах:

- ігри: навчання з підкріпленням використовується для навчання агентів грати в ігри, такі як шахи, го та Dota 2;
- робототехніка: навчання з підкріпленням використовується для навчання роботів виконувати завдання у реальному світі, такі як навігація та складання;
- фінанси: навчання з підкріпленням використовується для прийняття рішень у фінансах, таких як торгівля акціями та управління ризиками;
- медицина: навчання з підкріпленням використовується для прийняття рішень у охороні здоров'я, таких як діагностика захворювань та розробка ліків.

Переваги:

- може бути використано для вирішення завдань, які важко чи неможливо розв'язати за допомогою інших методів навчання;
- може бути використано для навчання агентів діяти у складних та постійно мінливих середовищах;
- може бути використано для навчання агентів діяти автономно, без необхідності втручання людини.

Недоліки:

- процес навчання може бути дуже складним та ресурсомістким;
- вимагає значних обчислювальних ресурсів на навчання агентів;
- результати можуть мати вигляд, що важко інтерпретується, що може утруднити розуміння того, як агент приймає рішення.

1.3 Аналіз існуючих завдань комп'ютерного зору

Комп'ютерний зір – це область, що займається вивченням методів отримання високорівневої інформації із цифрових зображень чи відео. З інженерної точки зору, вона спрямована на автоматизацію завдань, що може

виконувати зорова система людини. Завдання комп'ютерного зору включають методи отримання, обробки, аналізу та розуміння цифрових зображень. Розуміння зображень можливо розглядати як виділення символічної інформації з відео даних з допомогою різних методів комп'ютерного зору. Серед основних завдань цієї галузі де використовується машинне навчання можливо назвати наступні:

Класифікація – визначення класу об'єкта, представленого на зображенні. Класичним прикладом є класифікація рукописних символів на зображенні.

Детекція – виявлення об'єктів різних класів на зображенні, у машинному навчанні виявлені об'єкти зазвичай позначаються прямокутниками, які називають bounding boxes чи регіони інтересів (рисунок 1.3).

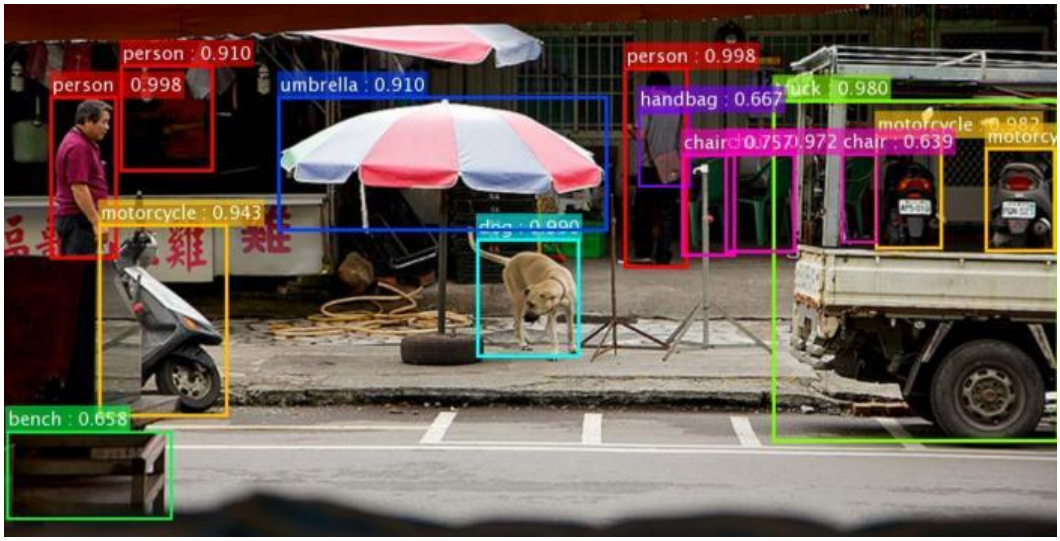


Рисунок 1.3 – Результат детекції за допомогою нейромережі

Розглянемо декілька прикладів задач детекції:

- розпізнавання об'єктів: виявлення об'єктів на зображенні або відео та їх класифікація відповідно до класу. Наприклад, завдання розпізнавання об'єктів може полягати в тому, щоб виявити та класифікувати автомобілі, пішоходів та велосипедистів на дорозі;

- виявлення аномалій: виявлення об'єктів, що не відповідають

очікуваному шаблону. Наприклад, завдання виявлення аномалій може полягати у тому, щоб виявити аномальні ділянки в людини на медичних знімках;

- трекер об'єктів: відстеження об'єктів у серії зображень або відео. Наприклад, трекер об'єктів можна використовувати для відстеження руху людей або автомобілів.

Сегментація – розподіл зображення на різні області (рисунок 1.4). Сегментація може бути використана для наступних цілей:

- виділення об'єктів: сегментація може бути використана для виділення окремих об'єктів у зображенні, наприклад людей на фотографіях або дерев на знімках місцевості;

- кластеризація: сегментація може використовуватися для кластеризації об'єктів на зображенні відповідно до їх подібності. Наприклад, сегментація може використовуватися для кластеризації зображень людей відповідно до їх віку або статі.

Рисунок 1.4 – Результат сегментації за допомогою нейромережі

1.4 Аналіз існуючих нейронних мереж

У зв'язку з бурхливим розвитком штучного інтелекту, все частіше можна побачити застосування нейронних мереж у цій галузі. Нейронні мережі здатні виконувати такі завдання комп'ютерного зору, як класифікація, детекція та сегментація. Найбільш популярними нейронними мережами в даній області

вважаються згорткові нейронні мережі, тому що вони вирішують проблеми які пов’язані з використанням повнозв'язних нейронних мереж для роботи з зображеннями:

- при розгортанні матриці пікселів зображення в одновимірний вектор для подання на вхід мережі, виходить занадто велика кількість нейронів на вхідному шарі, що робить її дуже важкою і схильною до швидкого перенавчання;
- неможливість зберегти просторові відносини зображення під час переведення в одновимірний вектор.

Згорткові нейронні мережі зазвичай складаються з шарів згортки, пулінгу та повнозв'язних лінійних шарів. Згорткові шари являють собою прохід по зображенню фільтром, який називають ядро згортки, підсумовуючи результати поелементного твору для кожного фрагмента зображення і ядра згортки. Потім результат такого проходу подається на функцію активації, яка є деякою нелінійною функцією. Найбільш поширеним варіантом такої функції є ReLU [2]:

$$f(x) = \max(0, x), \tag{1.1}$$

На виході отримуємо карту активації і чим вище в ній значення по модулю, тим ймовірніше знаходження необхідної ознаки на даній ділянці зображення. Далі зображено процес згортки зображення (рисунок 1.5–1.7) 9x9 ядром 5x5 зі значенням stride = 1, результатом є карта активації розміром 5x5.

Шар пулінгу є нелінійним стиском карти активації, при якому група пікселів стискається до одного. Найбільш поширеними типами пулінгу є:

- максимальний: серед групи пікселів обирається піксель із максимальною яскравістю (рисунок 1.8);
- мінімальний: серед групи пікселів обирається піксель із мінімальною яскравістю;
- усереднений пулінг: результатом буде середнє значення групи пікселів.

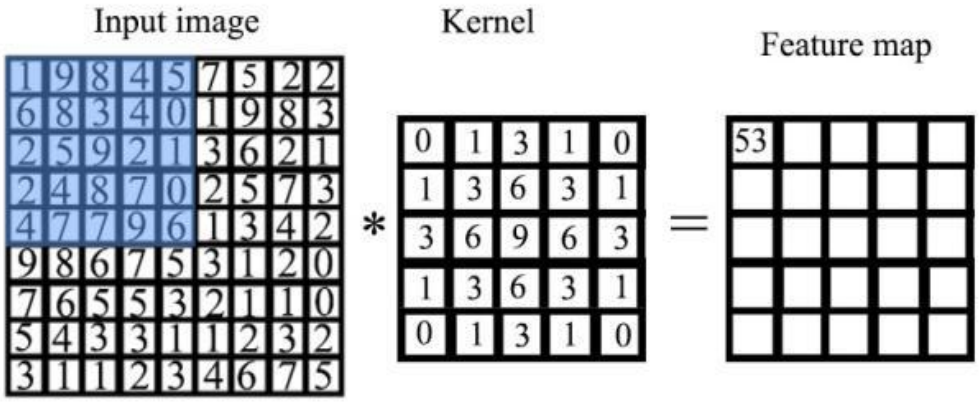


Рисунок 1.5 – Процес згортання, крок 1

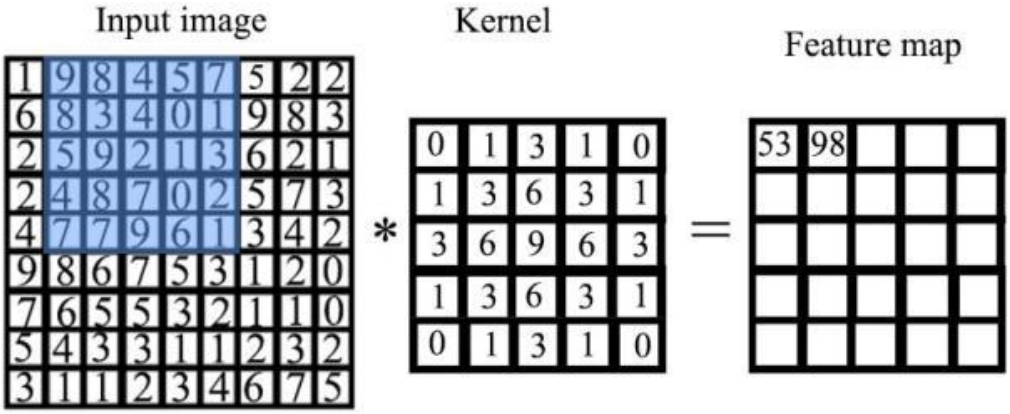


Рисунок 1.6 – Процес згортання, крок 2

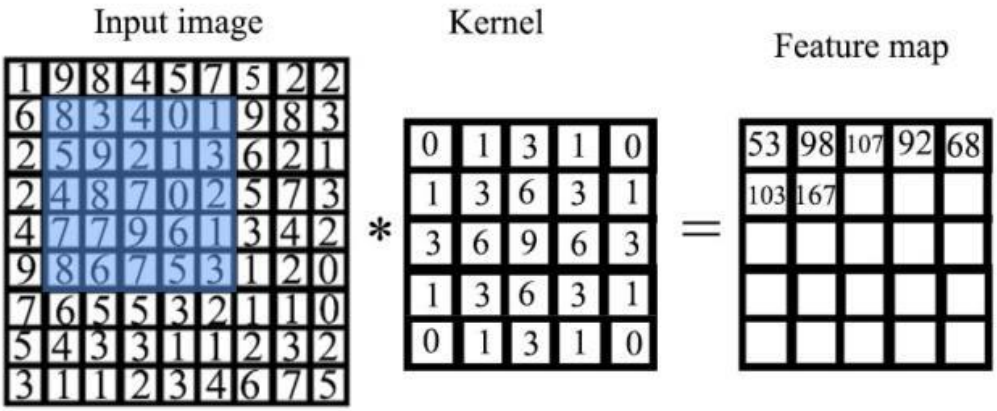


Рисунок 1.7 – Процес згортання, крок 7

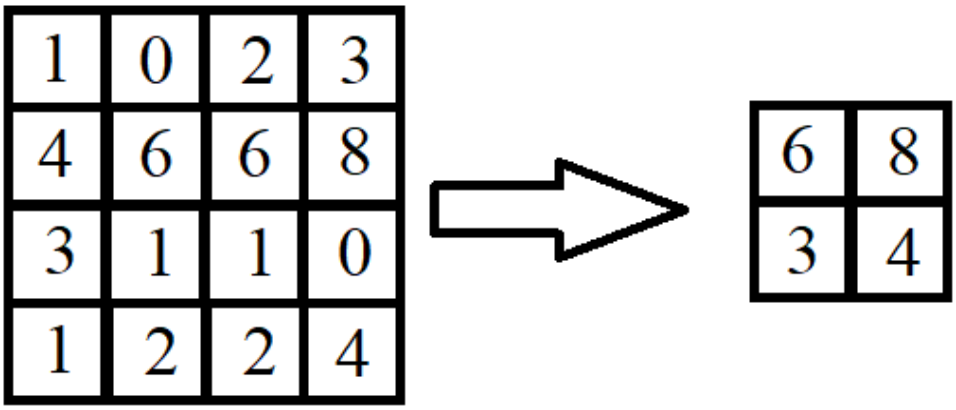


Рисунок 1.8 – Максимальний пулінг

З кожним новим шаром згортки і пулінга, карти активацій виділяють більш високорівневу інформацію, так якщо на перших шарах мережа реагує на примітивні образи, наприклад точки або лінії, то на останніх можуть бути виділені цілі об'єкти або образи. Завдяки такому підходу система перебудовується від конкретної сітки пікселів з високою роздільною здатністю до більш абстрактних карт активації. При цьому на кожному наступному шарі розмір карток активації зменшується, але збільшується кількість каналів (рисунок 1.9).

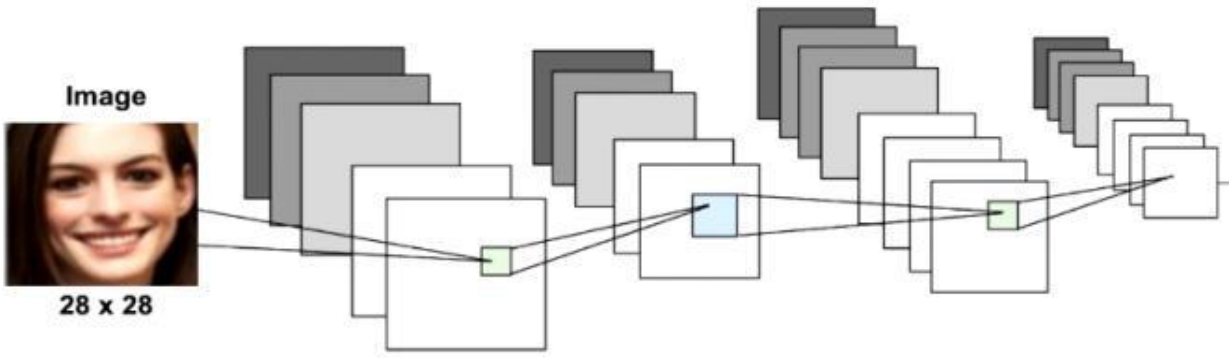


Рисунок 1.9 – Процес обробки зображення згортковими шарами

Ці дані об'єднуються і передаються на повнозв'язкову нейронну мережу, яка також може складатися з кількох шарів. При такому підході на вхід подається вектор, який має значно меншу розмірність порівняно з вихідним зображенням. Перевагами цього підходу є:

- один з найкращих алгоритмів класифікації зображень;
- значне скорочення кількості параметрів, що налаштовуються, оскільки в процесі навчання бере участь тільки ядро згортки, та декілька повноз’язкових шарів;
- можливість використання паралельних обчислень;
- стійкість до розташування об'єкта на зображенні.

Недоліком цього методу є велика кількість параметрів, які можливо підібрати тільки емпіричним шляхом. Однак цей недолік вирішується використанням готових архітектур згорткових мереж.

З розвитком згорткових нейронних мереж з'являлися дедалі складніші архітектури. Найефективніші виявлялися кожні кілька років на конкурсі ILSVRC. Так однією з перших згорткових нейронних мереж, яка перемогла у цьому конкурсі, стала AlexNet [3]. Вона складалася з восьми шарів: п'яти шарів згортки та трьох повнозв'язкових шарів (рисунок 1.10). Її перевагами, порівняно з раніше представленими методами, були:

- використання функції активації ReLU замість Tanh;
- аугментація даних;
- можливість використовувати декілька графічних процессорів для навчання.

Наступним переможцем стала архітектура VGGNet[4], розроблена Visual Geometry Group в Університеті Единбурга. Вона складається з 19 шарів згортки та 3 повнозв'язкових шарів (рисунок 1.11). На відміну від AlexNet, вхідний згортковий шар оброблявся фільтром 3x3, а не 11x11, що дозволило картам активації виділяти менш узагальнену інформацію, що дозволило мережі розпізнавати більш дрібні деталі.

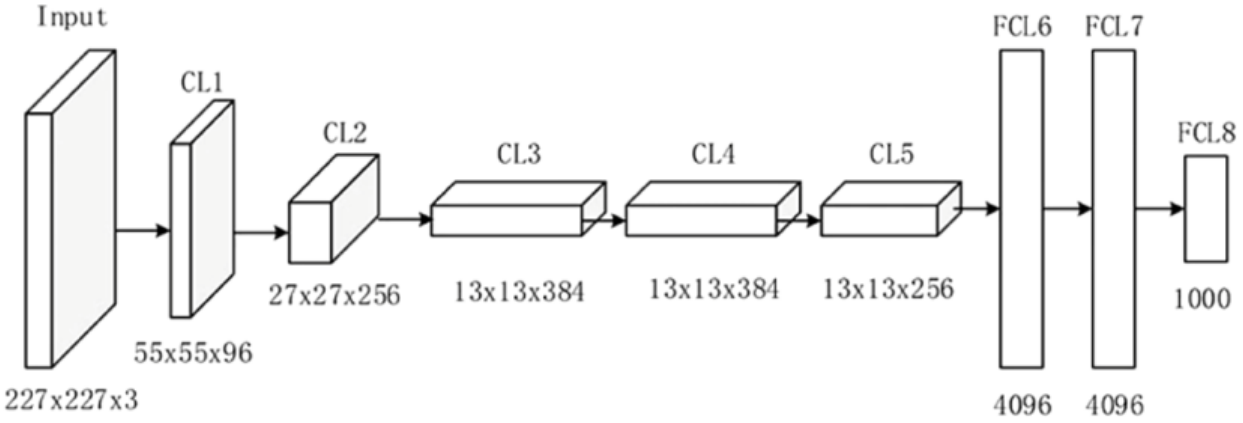


Рисунок 1.10 – Архітектура AlexNet

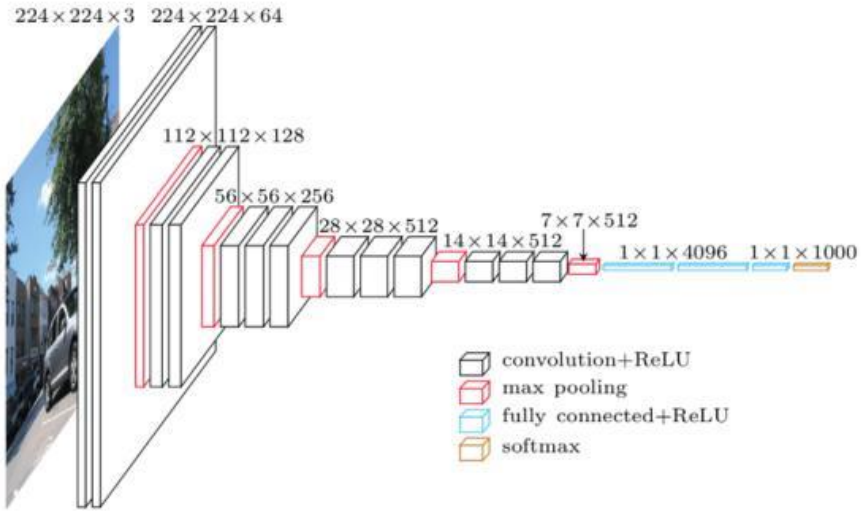


Рисунок 1.11 – Архітектура VGG

Пізніше Google представила свою нейромережу GoogleNet [5], ключовою відмінністю якої стали так звані Inception блоки (рисунок 1.12). Логіка таких блоків полягала в тому, щоб проходити зображення кількома фільтрами згортки, а потім конкатенувати отримані карти активації і відправляти на наступний шар. Такий підхід дозволяє мережі використовувати уявлення різних рівнів абстракції. Приклад такого блоку, що складається з трьох фільтрів

різного розміру:

- фільтр 1x1 виявляє контури або грані об'єктів;
- фільтр 3x3 виявляє дрібні деталі;
- фільтр 5x5 знаходить великі деталі та особливості об'єктів.

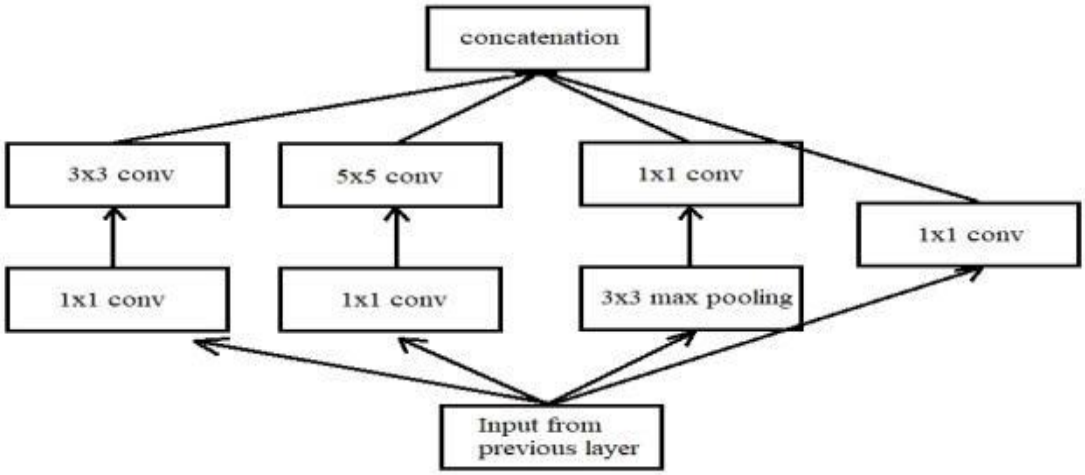


Рисунок 1.12 – Приклад структури inception блоку

Крім того, у GoogleNet використовувалася методика Dropout [6]. Це прийом регуляризації, який допомогає боротися з проблемою перенавчання, випадково відключаючи певні нейрони під час навчання. Це забезпечує більш рівномірний розподіл навчання між усіма нейронами, запобігаючи перекладанню відповідальності за певний клас на окремі нейрони.

Проте з розвитком згорткових нейронних мереж виникла проблема згасання градієнтів оскільки цей тип мереж навчається з допомогою функції зворотного поширення помилки (рисунок 1.13).

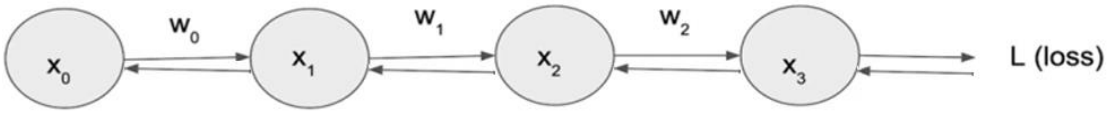


Рисунок 1.13 – Оновлення ваг за допомогою зворотнього поширення помилки

Для того щоб оновити коефіцієнти ваг для кожного нейрону, необхідно підрахувати градієнт на основі отриманого значення помилки:

$$w_2 = w_2 - \alpha \frac{\delta L}{\delta x_3} \frac{\delta x_3}{\delta w_2}, \tag{1.2}$$

$$w_1 = w_1 - \alpha \frac{\delta L}{\delta x_3} \frac{\delta x_3}{\delta x_2} \frac{\delta x_2}{\delta w_1}, \tag{1.3}$$

$$w_0 = w_0 - \alpha \frac{\delta L}{\delta x_3} \frac{\delta x_3}{\delta x_2} \frac{\delta x_2}{\delta x_1} \frac{\delta x_1}{\delta w_0}, \tag{1.4}$$

де x – значення нейрону;

w – коефіцієнт ваг нейрону;

α – learning rate.

Найчастіше градієнти та learning rate, які безпосередньо впливають на значення ваг, є числами меншими ніж одиниця. В результаті, коефіцієнти шарів, розташованих далеко від виходу мережі, будуть оновлюватися дуже повільно або взагалі не будуть оновлюватися. Це може призвести до того, що ці шари не будуть навчатися, і мережа не зможе досягти бажаної точності.

Цю проблему було вирішено в архітектурі ResNet [7] шляхом запровадження технології skip connection (рисунок 1.14). Суть skip connection полягає у створенні обхідних шляхів, які пропускають один або кілька шарів, тим самим дозволяючи градієнтам ефективніше поширюватися через мережу, забезпечуючи більш глибоке навчання.

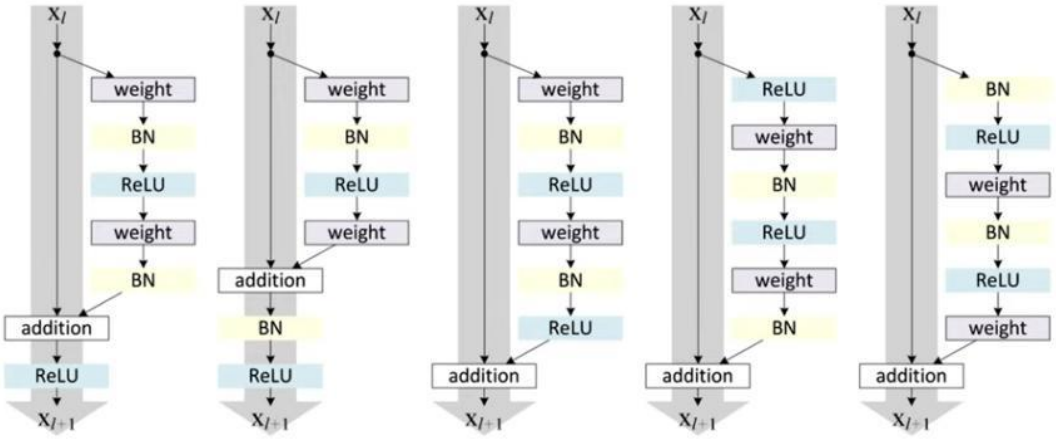


Рисунок 1.14 – Варіанти блоків з використанням skip connection

Архітектура має кілька варіантів, які різняться за кількістю шарів, наприклад, ResNet18 містить 18 шарів згортання і 1 повнозв'язний шар. У той час як ResNet34 має 34 згорткових шари та 1 повнозв'язковий шар (рисунок 1.15). Такі різновиди архітектури з різною кількістю шарів включають ResNet50, ResNet101, ResNet152 та інші. На рисунку 5.6 зображена архітектура ResNet34.

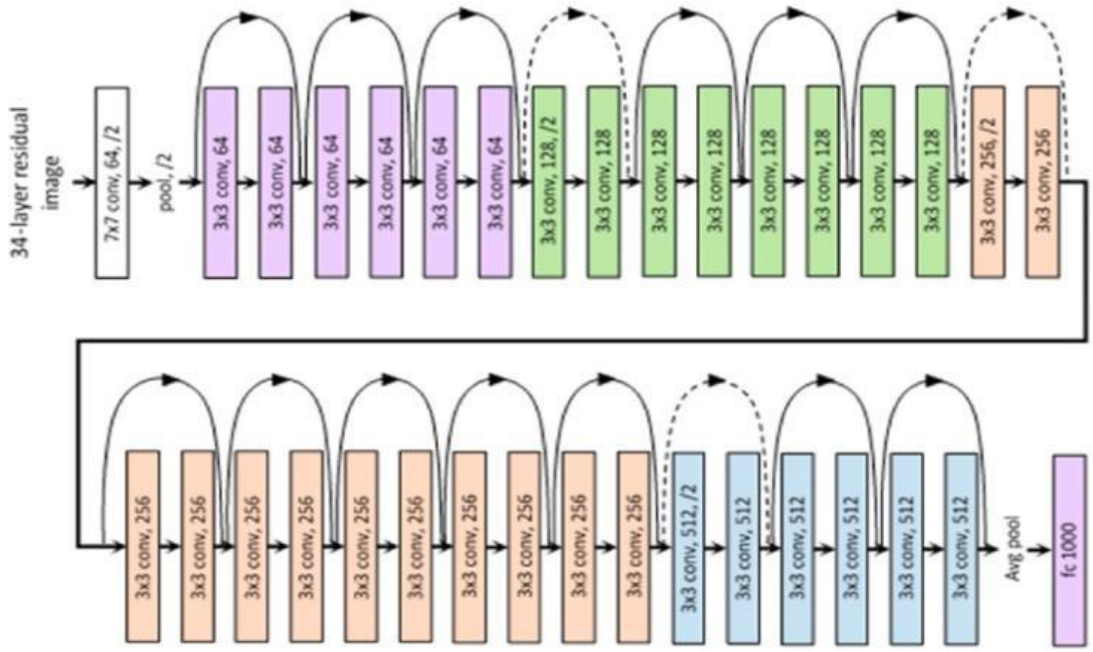


Рисунок 1.15 – Архітектура ResNet34

Ще один цікавий підхід, який використовує технологію skip connection, був представлений у 2017 в моделі DenseNet [8] (рисунок 1.16). Ключовою особливістю даної моделі стало використання dense блоків, суть яких полягала в тому, щоб подавати карти активації з попередніх шарів згортки блоку на наступні. Таким чином шари блоку мають найбільш повне уявлення про зображення завдяки картам активації з різними рівнями абстракції. Між кожним блоком вставляються перехідні шари, що складаються з шарів згортки та пулінгу, які приводять карти активацій до необхідного розміру. Такий підхід так само добре справляється з проблемою градієнтного згасання завдяки меншій кількості шарів і параметрів, а також зв'язку всіх шарів у dense блоках.

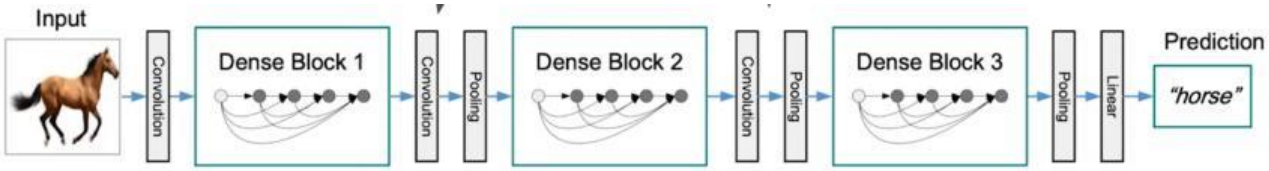


Рисунок 1.16 – Архітектура DenseNet

Нижче представлена діаграма top-5 error переможців конкурсу ILSVRC з 2012 до 2015 року (рисунок 1.17).

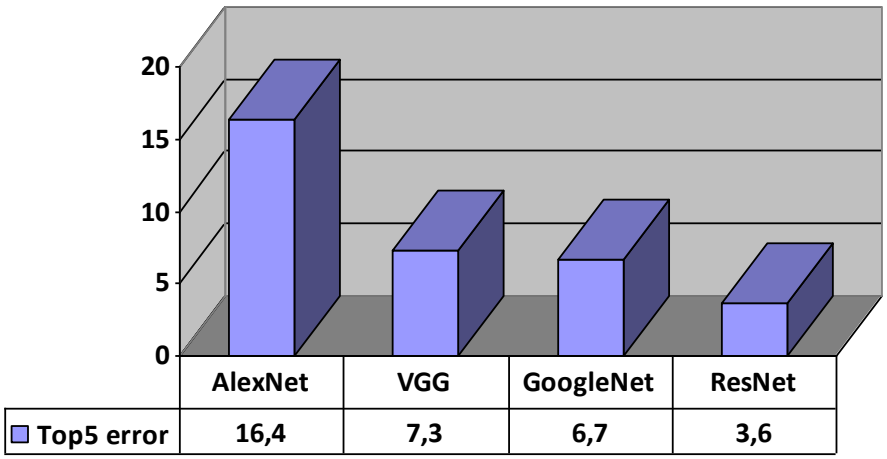


Рисунок 1.17 – Діаграма top-5 error переможців конкурсу ILSVRC

Нейромережі, які розв'язують задачу детекції, можна розділити на два класи: двостадійні та ондостадійні.

Двостадійні архітектури складаються з трьох модулів – генератора регіонів інтересу, регрессора та класифікатора. Двостадійними вони називаються, тому що спочатку генеруються регіони інтересу, а потім відбувається їх регресія та класифікація. Однією з перших моделей, яка розв'язувала задачу детекції була RCNN [9] (рисунок 1.18). Алгоритм її роботи можливо поділити на такі кроки:

- згенерувати якусь кількість регіонів інтересу, де ймовірно знаходження об'єкта;
- привести їх до єдиного розміру;
- для кожного регіону провести регресію і класифікацію;
- обрати передбачення з найбільшим ступенем упевненості.

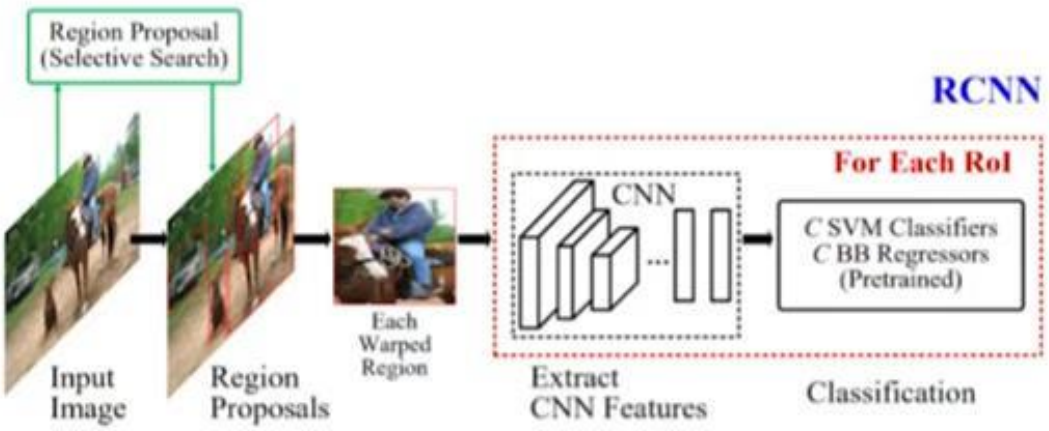


Рисунок 1.18 – Архітектура RCNN

Однак обчислювальна складність такого методу була дуже висока, оскільки кожен передбачений регіон оброблявся згортковою нейронною мережею. Тому в наступній версії архітектури Fast RCNN [10] (рисунок 1.19) поміняли ці етапи місцями, тобто спочатку витягували карту ознак, а потім генерували регіони інтересу.

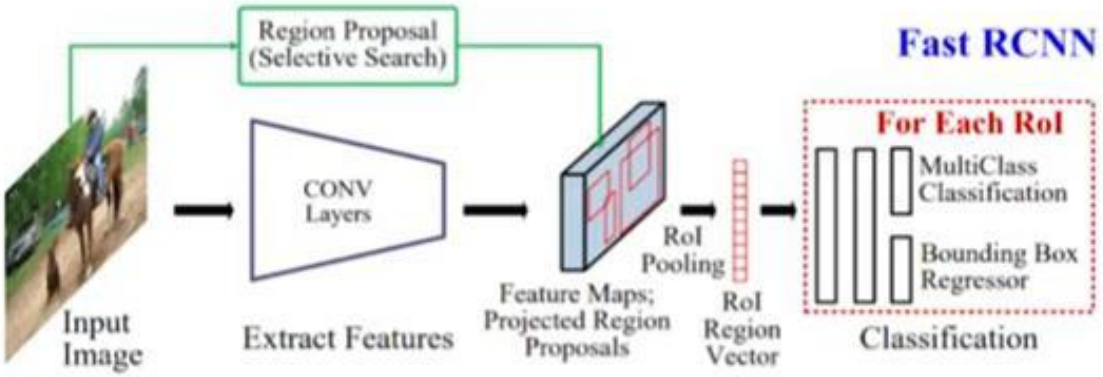


Рисунок 1.19 – Архітектура Fast RCNN

Наступною модифікацією стала Faster RCNN[11] (рисунок 1.20), де детерменований алгоритм пошуку прямокутників замінили на невеликий модуль, що складається з декількох шарів згортки, що зробило архітектуру більш гнучкою для навчання на різних даних. Також відмовилися від ідеї генерації регіонів інтересів на ходу, замість цього є деяка заготовлена кількість цих регіонів, які називають anchor boxes. Таким чином, замість генерації, модуль регресії підганяє вже існуючі регіони, з яких вибирається один з найбільшим перетином.

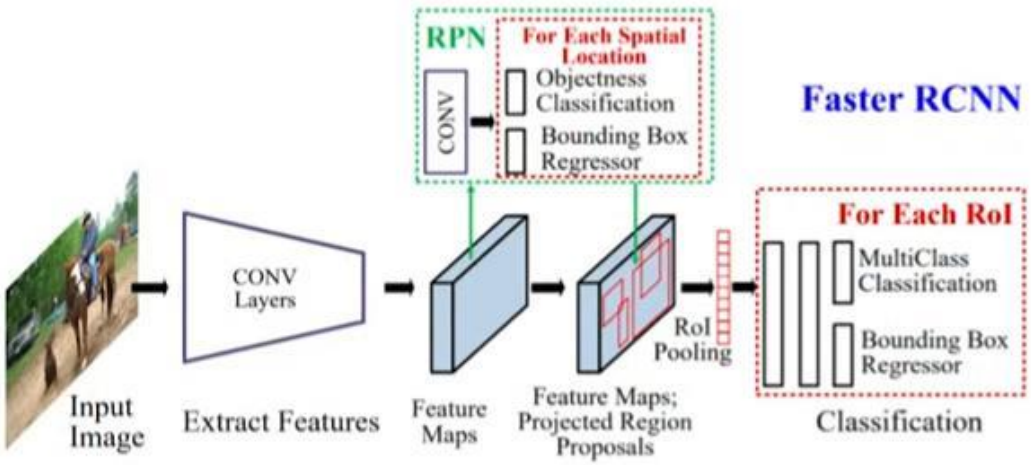


Рисунок 1.20 – Архітектура Faster RCNN

Перевагою двостадійних архітектур є висока точність, тому їх використання релевантно в областях де швидкість цінується менше точності, наприклад медицина. Однак недоліками є складність навчання, а також висока обчислювальна складність.

На зміну двостадійним архітектурам прийшли одностадійні, які вирішують завдання детекції за один крок. Однією із таких мереж є YOLO (You Look Only Once) [12] (рисунок 1.21). Загальний алгоритм роботи мережі полягає у наступному:

- отримання карт ознак за допомогою згорткових шарів;
- отримання сітки;
- передбачення регіонів інтересу;
- вибір регіонів із найбільшим перетином.

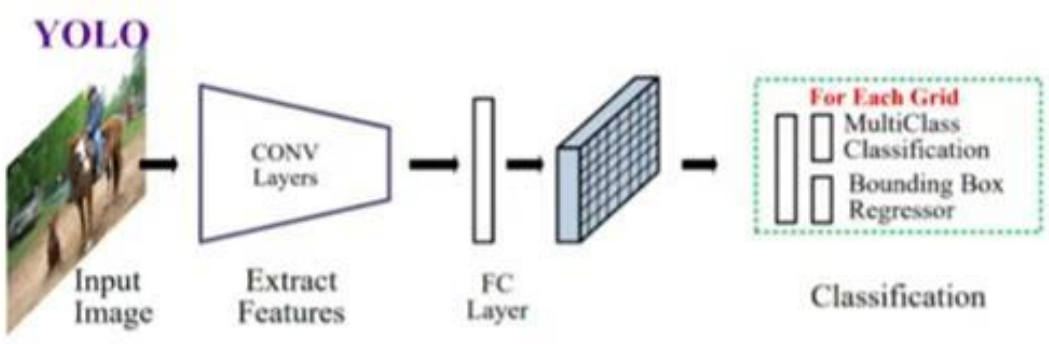


Рисунок 1.21 – Архітектура YOLO

Далі з’явилася SSD (Single Shot Multibox Detector) [13] (рисунок 1.22). Основна ідея архітектури полягає в тому, щоб передбачати розташування об'єктів різних розмірів використовуючи карти ознак різних згорткових шарів.

Потім з'явилася RetinaNet [14] (рисунок 1.23), ключовою особливістю якої стало використання функції втрат Focal Loss, яка дозволяє зосередити увагу мережі на прикладах, які важко класифікувати (1.2):

$$FL = -(1 - P_t)^{\gamma} \log(P_t), \tag{1.5}$$

де P_t – імовірність, прогнозована мережею для правильного класу;

γ – параметр, що контролює вплив фокусного доданка.

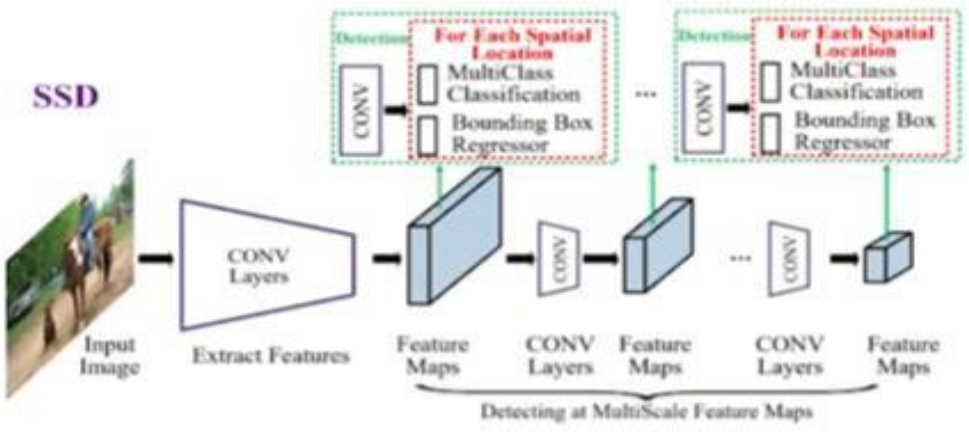


Рисунок 1.22 – Архітектура SSD

Коли об'єкт класифікується правильно, log(Pt) сходиться до нуля, і вплив спадної частини стає сильнішим зі збільшенням γ. Це означає, що зі збільшенням γ, функція втрат більш сконцентрована на неправильно класифікованих прикладах.

Також в цій архітектурі було використано технологію FPN (Feature pyramid network), яка схожа на принцип вилучення ознак у SSD, але зі зв'язками між згортковими шарами як у dense блоці, оскільки конткатенує карти ознак різних масштабів.

Одним з перших, для вирішення задачі сегментації зображень, з'явився метод використання повністю згорткової нейронної мережі (рисунок 1.24), при цьому ядра згортки мали таку ж розмірність як на оригінальному зображенні. На виході для кожного пікселя зображення з карт активації вибирався той, у класі якого мережа найбільш впевнена.

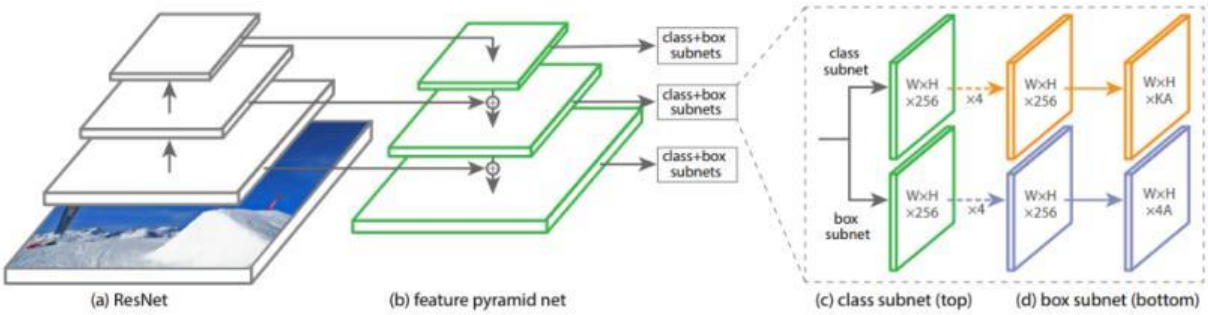


Рисунок 1.23 – Архітектура RetinaNet

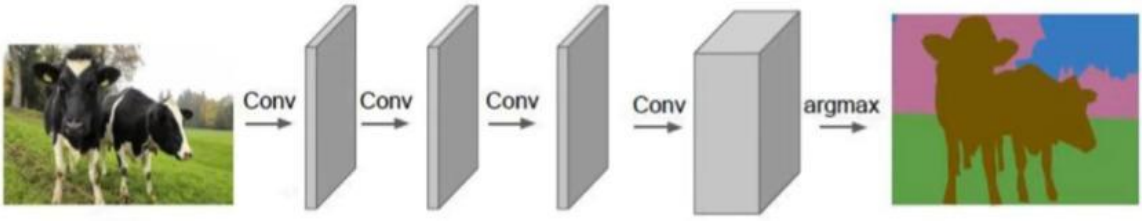


Рисунок 1.24 – Fully convolutional network

Однак у такого підходу проблема така сама, як і з використанням повної мережі для обробки зображень – занадто багато параметрів.

Потім з'явилася ідея використання звичайної згорткової мережі, так само без використання повнозв'язкових шарів. Після отримання останньої карти активації застосовується upsampling для розширення зображення до оригінального формату. Цей метод вже більш прийнятний, проте має кілька недоліків:

- upsampling не може якісно відновити інформацію;
- пулінг відчутно спотворює просторову інформацію;
- спотворення розмірів об'єктів.

Перша вдала спроба вирішення проблеми upsampling'a була застосована у DeConvNet [15]. Суть полягала у використанні DeConvolution шарів для збільшення розмірів останньої карти активації. Принцип роботи цих шарів полягав у почерговому використанні upsampling'a та згортки (рисунок 1.24).

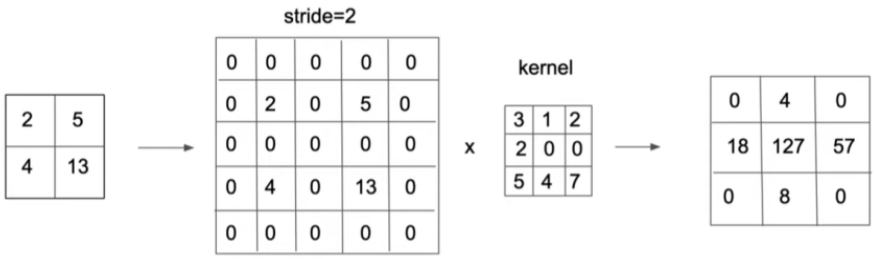
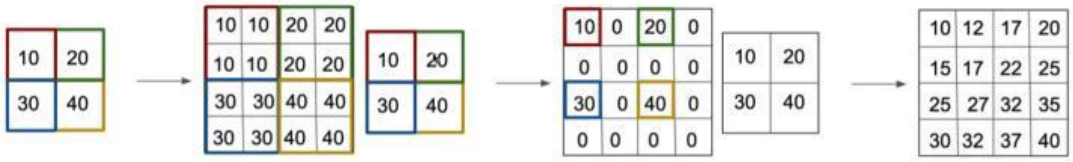


Рисунок 1.24 – Deconvolution layer

При цьому тестуються різні методи upsampling'a (рисунок 1.25) для досягнення найкращого результату.



а) б) в)

Рисунок 1.25 – Методи upsampling'a: а) nearest neighbors; б) bed of nails; в) bilinear

Також існує метод з розташуванням елементів карт активації на місця, де вони знаходилися до операції пулінга в згортковій мережі, який називається unpooling [16] (рисунок 1.26):

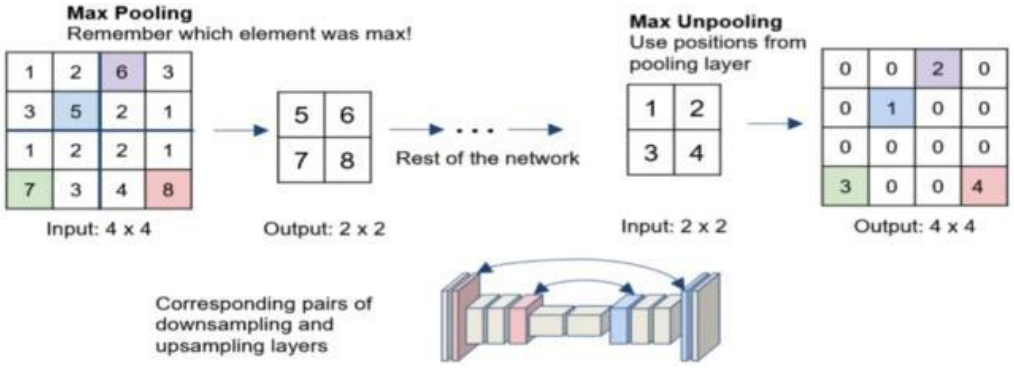


Рисунок 1.26 – Unpooling

Для вирішення проблеми спотворення просторової інформації було запропоновано метод Multiscale Context Aggregator [17]. У цьому методі звичайні шари згортки замінили на dilated шари (рисунок 1.27). Відрізняються дані шари наявністю відступів, які дозволяють збільшити область захоплення інформації згорткового ядра, при цьому зберігаючи обчислювальну складність. Dilation коефіцієнт визначає, скільки пікселів буде пропущено між елементами ядра згортки.

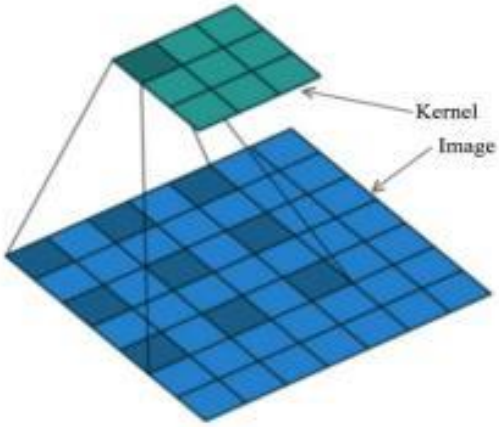


Рисунок 1.27 – Dilated convolution

Такі шари об'єднали в блок схожий структурою з inception блоком і назвали Multiscale Context Aggregator (рисунок 1.28).

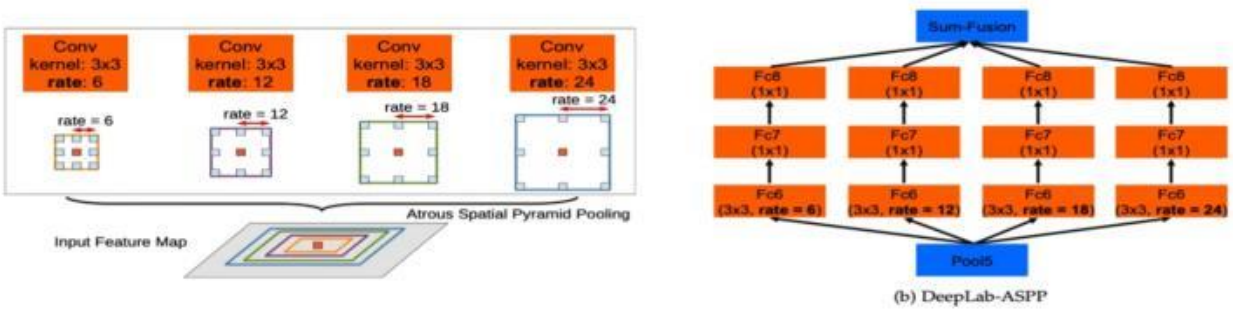


Рисунок 1.28 – Multiscale context agregator

Альтернативним вирішенням проблеми спотворення просторової інформації є Pyrimid Pooling [18] (рисунок 1.29). Основна ідея даного методу теж схожа з inception блоком. Завдяки шарам пулінгу різного розміру, об'єднаним в один блок, мережа може отримати інформацію про зображення на різних масштабах.

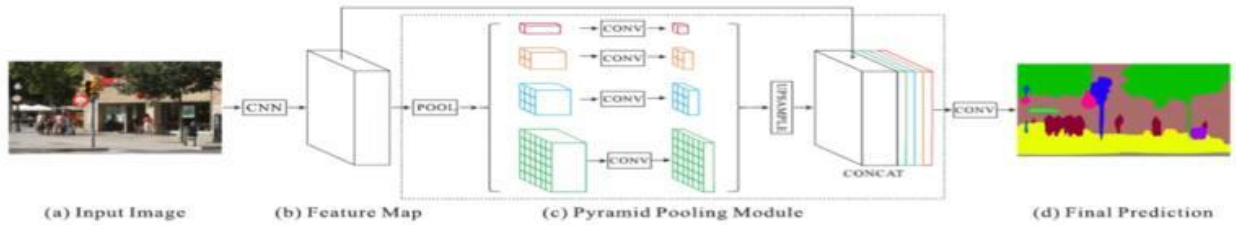


Рисунок 1.29 – Pyramid pooling

Однак, навіть застосування цих методів не змогло дати бажаного результату, тому люди вирішили використовувати постобробку зображень. Одним із найпопулярніших методів для цієї задачі є CRF [19]. Conditional Random Fields – ймовірна графічна модель, яка моделює умовний розподіл ймовірностей для вихідних даних, враховуючи вхідні ознаки та їх взаємозв'язки.

CRF складає граф, де вузлами є пікселі зображення. Вона моделює зв'язки між сусідніми пікселями на основі потенціалів, що визначають ймовірність різних значень пікселів з урахуванням сусідів. Потенціали можуть бути визначені на основі вхідних ознак (наприклад, колір, текстура, яскравість) та функцій, які оцінюють схожість пікселів усередині сегмента та відмінності між сегментами.

Завданням CRF є знаходження комбінації значень пікселів, яка максимізує ймовірність сегментації при заданих потенціалах.

Архітектури нейромереж, які вирішують задачу сегментації:

DeepLab – це сімейство глибоких архітектур для семантичної сегментації зображень, розроблене дослідницькою групою Google Research. Основні

технології які були застосовані в DeepLabv3 [20] (рисунок 1.30):

- блок Aspp, який є різновидом Multiscale Context Aggregator;
- dilation згортка;
- deconvolution;
- CRF.

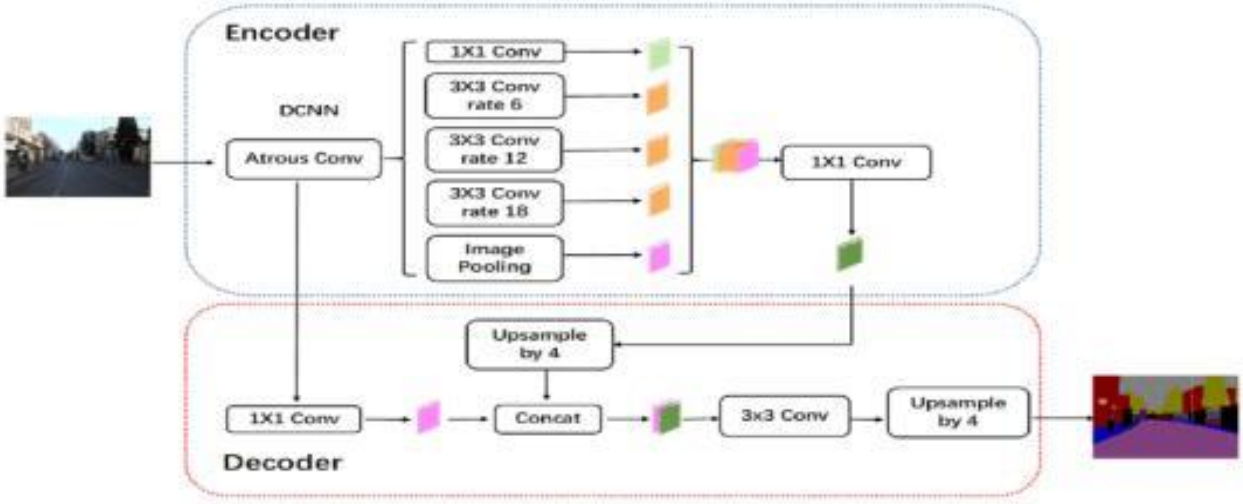


Рисунок 1.30 – Архітектура DeepLabv3

UNet [21] – це нейронна мережа, розроблена для семантичної сегментації зображень, пов'язаних із обробкою біомедичних зображень (рисунок 1.31). UNet здобула широке визнання і залишається популярною в галузі комп'ютерного зору та обробки медичних зображень. Ключовими особливостями даної архітектури стали використання технології skip connection і регулювання ваг для чутливості до дрібних деталей.

Одним із досить нових підходів у розв'язанні задачі сегментації є вирішення задачі детекції перед сегментацією, таким чином значно зменшуючи область для якої необхідно провести сегментацію, проте підходить він лише для завдання інстанс сегментації. Такий підхід було реалізовано в архітектурі Mask RCNN [22] (рисунок 1.32). Працює аналогічно Faster RCNN тільки з додаванням модуля сегментації для регіонів інтересу.

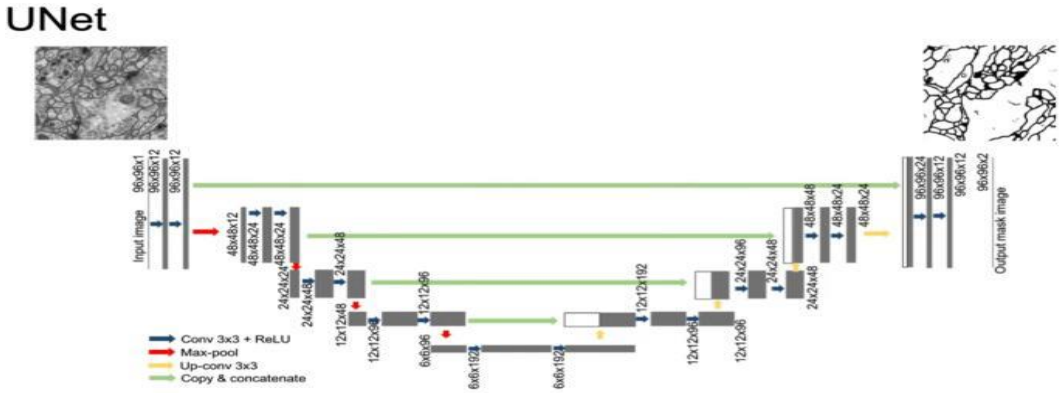


Рисунок 1.31 – Архітектура Unet

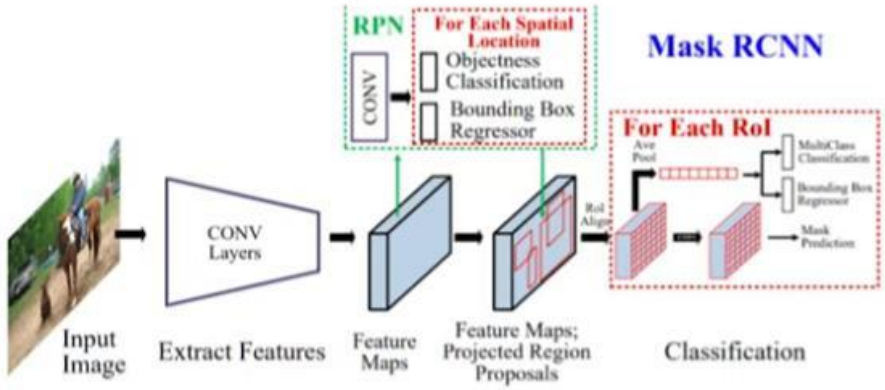


Рисунок 1.32 – Архітектура Mask RCNN11

1.5 Аналіз існуючих методів бінаризації

Бінаризація – це процес перетворення зображення з градацій сірого на двоколірне (бінарне) зображення, пікселі якого можуть мати лише два значення: чорний або білий. У контексті обробки зображень, бінаризація має на увазі використання порогового значення, для диференціювання класів пікселів чорного (об'єкт) і білого (фон), пікселі, чиє значення яскравості перевищує поріг бінаризації, стають білими, нижче – чорними.

Бінаризація є одним зі стандартних методів попередньої обробки зображень і широко використовується для виділення ознак, об'єктів на зображенні та відокремлення їх від фону, а також радикального зменшення

кількості інформації. Цей метод корисний у сфері машинного зору і може використовуватися в розпізнаванні тексту, виділенні контурів, обробці зображень у медичних системах, а також у багатьох інших галузях.

Процес бінаризації складається з двох етапів: конвертація зображення у градації сірого та бінаризація сірого зображення. Методи конвертації зображення у градації сірого:

Середнє значення. Для кожного пікселя обчислюється середнє значення його каналів червоного, зеленого і синього (RGB) і потім це середнє значення присвоюється каналу сірого.

$$L = \frac{R + G + B}{3}, \tag{1.6}$$

де L – це вихідне значення пікселя;

R – це значення червоного каналу пікселя;

G – це значення зеленого каналу пікселя;

B – це значення синього каналу пікселя.

Канал яскравості (Luminosity). У цьому методі використовується тільки канал яскравості, який являє собою зважене середнє значення каналів RGB, що враховує сприйняття людиною яскравості різних кольорів.

$$L = w_r * R + w_g * G + w_b * B, \tag{1.7}$$

де w – це ваговий коефіцієнт для кожного каналу зображення.

Зазвичай використовують наступні значення вагів:

- rec BT.601: w_r = 0.299, w_g = 0.587, w_b = 0.114;
- rec BT.709: w_r = 0.2126, w_g = 0.7152, w_b = 0.0722;
- rec BT.2020: w_r = 0.2627, w_g = 0.6780, w_b = 0.0593.

Десатурація. Для кожного пікселя знаходиться середнє між мінімальним і максимальним значенням яскравості його каналів.

Максимальна декомпозиція. Для знаходження яскравості пікселя цим методом використовується максимальне значення його каналів.

Мінімальна декомпозиція: для знаходження яскравості пікселя цим методом використовується мінімальне значення його каналів.

Одноканальне перетворення. У якості значення пікселя при перетворенні кольорового зображення в градації сірого цим методом використовується значення одного з його каналів (в основному зеленого).

Методи знаходження порога бінарізації можливо поділити на глобальні та локальні.

Метод Оцу [23] є одним із найбільш відомих методів глобальної бінаризації. Цей метод обчислює оптимальний поріг t, який мінімізує середню помилку сегментації. Під час аналізу зображень, яскравості пікселів можна розглядати як випадкові величини і використовувати їх гістограму для оцінки ймовірнісного розподілу. Гістограма будується за значеннями:

$$p_i = \frac{n_i}{N}, \tag{1.8}$$

де N – це кількість пікселів на зображенні;

n_i – це кількість пікселів із рівнем яскравості i.

З використанням гістограми відбувається поділ усіх пікселів на дві категорії: об'єктні та фонові. Для кожної з цих категорій визначаються відповідні відносні частоти $\omega 0 \omega 1$:

$$\omega_0(k) = \sum_{i=1}^{k} p_i, \tag{1.9}$$

де $i \in (0, t)$.

$$\omega_1(k) = \sum_{i=k+1}^{L} p_i = 1 - \omega_0(k)$$, (1.10)

де $i \in (t+1, L-1)$.

Обчислити середні значення яскравості для кожного класу можливо за допомогою наступних формул:

$$\mu_0(k) = \sum_{i=1}^{k} \frac{ip_i}{\omega_0}$$, (1.11)

де $i \in (0, t)$.

$$\mu_1(k) = \sum_{i=k+1}^{L} \frac{ip_i}{\omega_1}$$, (1.12)

де $i \in (t+1, L-1)$.

Внутрішньокласові дисперсії обчислюються за наступними формулами:

$$\sigma_0^2 = \sum_{i=1}^{k} \frac{(i - \mu_0)^2 * p_i}{\omega_0}$$, (1.13)

де $i \in (0, t)$.

$$\sigma_1^2 = \sum_{i=k+1}^{L} \frac{(i-\mu_1)^2 * p_i}{\omega_1}, \quad (1.14)$$

Наступним кроком необхідно обчислити зважену суму внутрішньокласових дисперсій:

$$\sigma^2(t) = \omega_0 * \sigma_0^2 + \omega_1 * \sigma_1^2, \quad (1.15)$$

Після обчислення міри поділу класів для всіх можливих порогових значень, алгоритм обирає порогове значення t_{opt} ,що максимізує цю міру поділу.

Локальні методи пошуку порога бінаризації є більш стійкими до зміни інтенсивності освітлення на зображенні, оскільки в цих методах зображення ділиться на сектори, де для кожного сектора знаходиться локальний поріг бінаризації. Найбільш популярні локальні методи:

Метод Бернсена [24] передбачає виділення локальної області для кожного пікселя на зображенні, такої як, наприклад, квадратна або кругла область навколо нього. Потім на основі цієї області обчислюються мінімальне і максимальне значення пікселів, і поріг встановлюється між цими значеннями.

Метод Ніблека [25] також використовує локальне оточення пікселя для визначення порога бінаризації, однак для обчислення порога використовується середнє значення і стандартне відхилення яскравостей. Для обчислення середнього значення m і математичного відхилення σ використовується формула:

$$m(x,y) = \frac{1}{WH} \sum_{i=1}^{W} \sum_{j=1}^{H} I(x+i, y+j), \quad (1.16)$$

$$\sigma(x,y) = \sqrt{\frac{1}{WH}\sum_{i=1}^{W}\sum_{j=1}^{H}(I(x+i,y+j) - m(x,y))^2}, \quad (1.17)$$

де *I(x,y)* – яскравість пікселя в позиції (*x*, *y*);

W і *H* – ширина і висота локальної області.

Для обчислення порогу бінаризації використовується наступна формула:

$$T(x,y) = m(x,y) + k * \sigma(x,y), \quad (1.18)$$

де *k* – коефіцієнт чутливості бінаризації.

Метод Сауоли [26] аналогічний методу Ніблека, ключова відмінність полягає в обчисленні порога бінаризації:

$$T(x,y) = m(x,y)(1 + k\left(\frac{\sigma(x,y)}{R} - 1\right)), \quad (1.19)$$

де *R* – коефіцієнт максимального стандартного відхилення в локальній області;

k – коефіцієнт чутливості бінаризації.

Метод Крістіана [27] є модифікацією методу Ніблєка. Цей метод також враховує локальне оточення кожного пікселя для визначення порога бінаризації. А поріг обчислюється за формулою:

$$T = (1-k) * m + k * M + k * \frac{s}{R} * (m - M), \quad (1.20)$$

де M – це максимальна яскравість усього зображення;

k – коефіцієнт чутливості бінаризації.

Метод Бредлі-Рота [28] використовує інтегральні зображення, які дозволяють ефективно визначити суму піксельних значень і середню яскравість в обраному фрагменті зображення, використовуючи лише одну ітерацію. Значення елемента інтегрального зображення розраховується за формулою:

$$I(x,y) = f(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1), \quad (1.21)$$

де I – результат попередніх ітерацій для даної позиції пікселя;

$f(x,y)$ – яскравість пікселя вихідного зображення.

Для того щоб знайти сумарну яскравість S у деякій прямокутній ділянці, де (x_1, y_1) є верхнім лівим кутом, а правий нижній (x_2, y_2), необхідно скористатися формулою:

$$\sum_{x=x_1}^{x_2}\sum_{y=y_1}^{y_2} f(x,y) = I(x_2,y_2) - I(x_2,y_1-1) - I(x_1-1,y_2) + I(x_1-1,y_1-1), \quad (1.22)$$

де $f(x,y)$ – значення елементів інтегральної матриці.

Далі для обчислення порогу бінаризації необхідно знайти середнє від сумми значень пікселей у локальній області:

$$T(x,y) = \frac{\sum i(x,y)}{n}, \quad (1.23)$$

де i – значення яскравості пікселя у точці (x,y);

n – кількість пікселей у локальній області.

Для того, щоб визначити найбільш ефективний алгоритм бінаризації, буде обрано різні комбінації з пар: режим конвертації зображення в градації сірого та методу бінаризації. Далі для зручності назви зображень матимуть такий вигляд: (метод переведення в сірий)_(метод бінаризації) (рисунок 1.33–1.44).

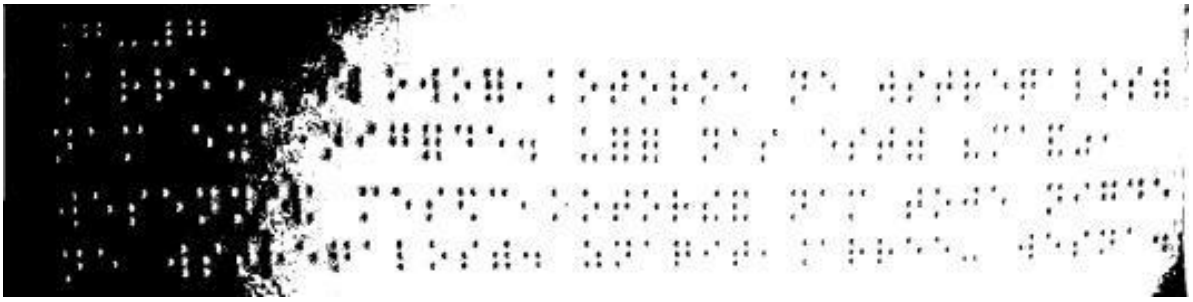


Рисунок 1.33 – Max_Otsu

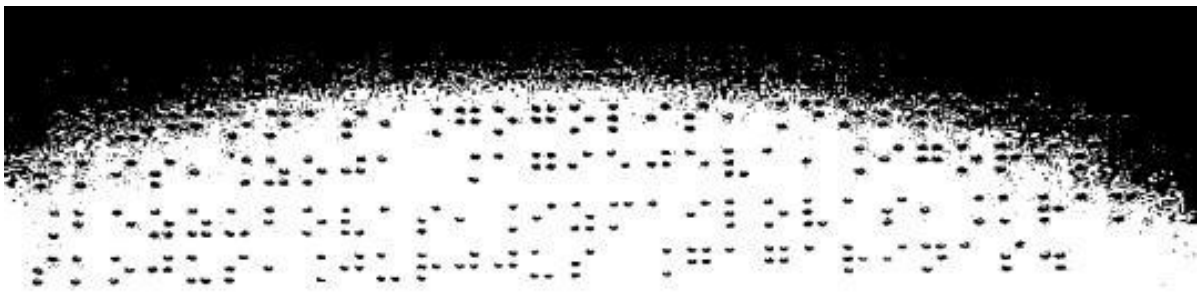


Рисунок 1.34 – Luma601_Otsu

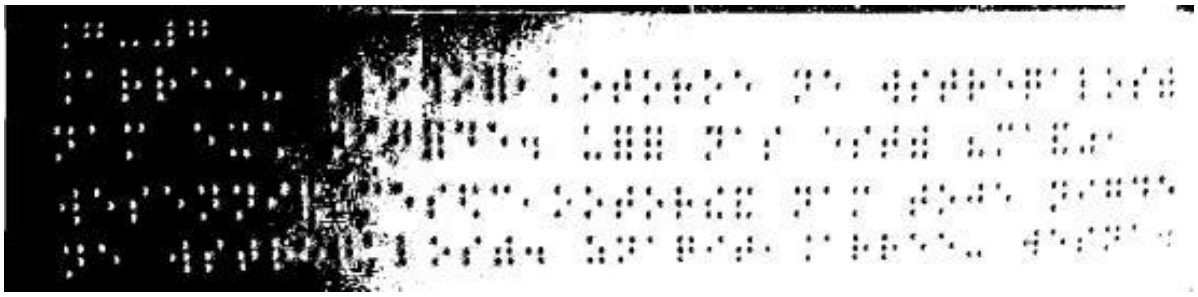


Рисунок 1.35 – Blue_Bernsen

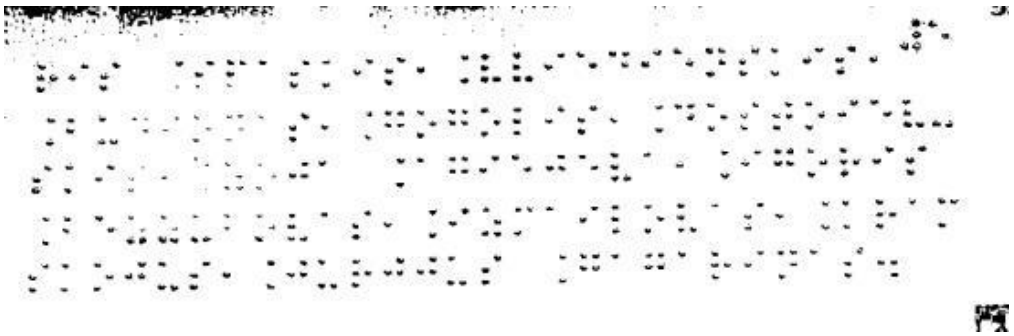


Рисунок 1.36 – Max_Bernsen

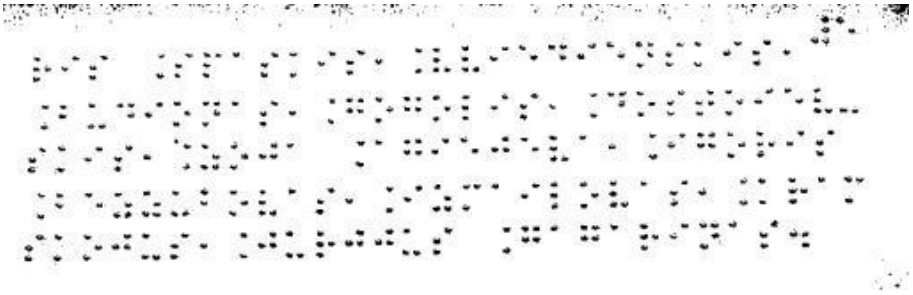


Рисунок 1.37 – Min_Niblack

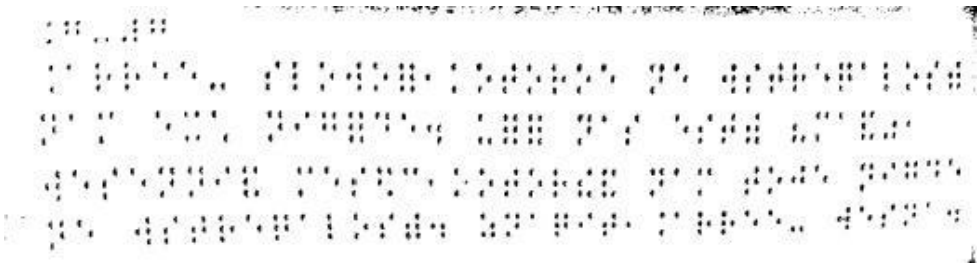


Рисунок 1.38 – Luma2020_Niblack

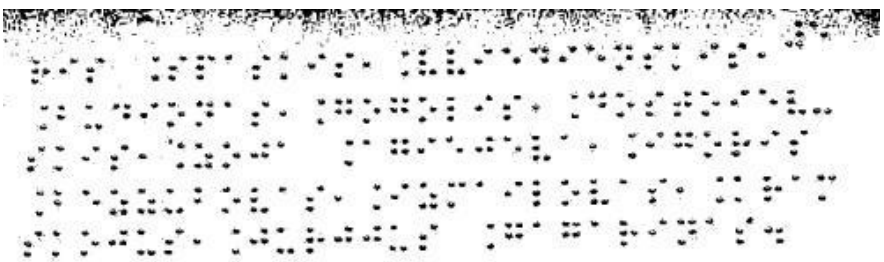


Рисунок 1.39 – Luma601_Sauola

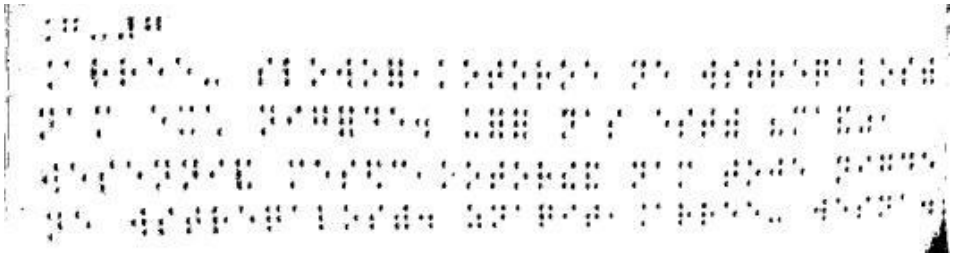


Рисунок 1.40 – Max_Sauola

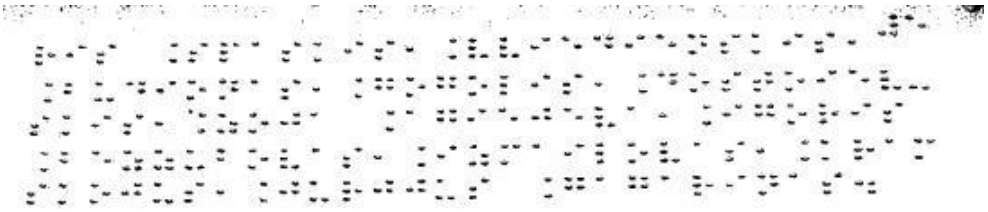


Рисунок 1.41 – Red_Cristian

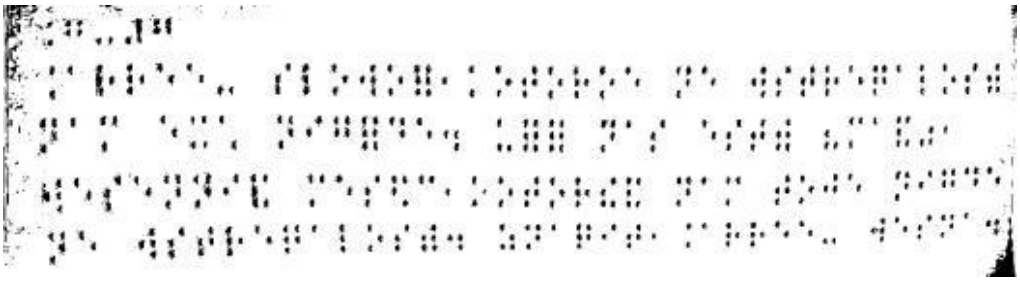


Рисунок 1.42 – Max_Cristian

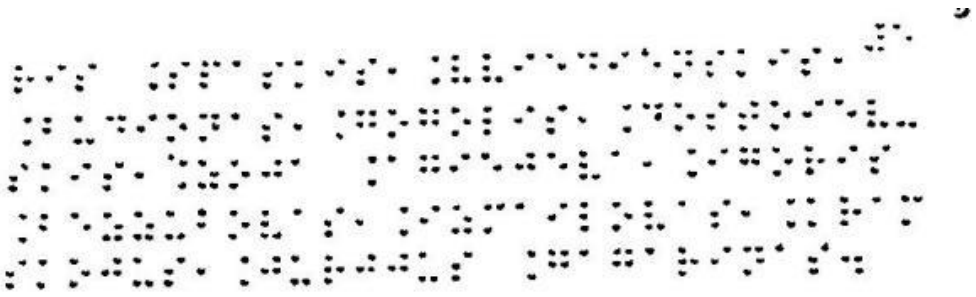


Рисунок 1.43 – Luma601_Bradley

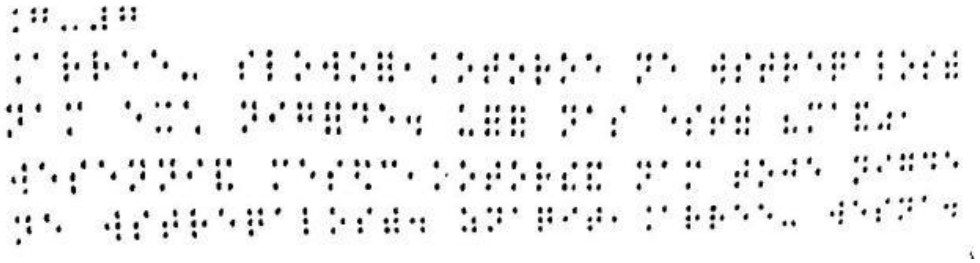


Рисунок 1.44 – Green_Bradley

З отриманих зображень можна зробити висновок, що алгоритм Бредлі є найбільш оптимальним, оскільки кінцевий результат містить найменшу кількість шуму, збережена ключова інформація з вихідного зображення, а обчислювальна складність поступається лише методам глобальної бінаризації. Також використання різних методів перетворення в градації сірого не мало відчутного впливу, тому буде доречним використання стандартів luma.

1.6 Постановка завдання дослідження

Шрифт Брайля є життєво важливим інструментом для людей з вадами зору, тому що він є основним способом читання та письма. Тому дуже важливою є проблема транскрипції цього шрифту із зображень у читабельний текст, який можуть зрозуміти як люди з вадами зору, так і зрячі. Як було вказано раніше, існуючі методи перекладу шрифту Брайля, що міститься на зображенні, не є достатньо точними, потребують попереднього опрацювання, а також не зовсім зручні. У наслідок цього, існує загальна потреба у використанні нових технологій, що зможуть більш ефективно виконувати поставлене завдання. Однією з таких технологій є застосування нейронних мереж, що показали багатообіцяючі результати у задачах подібного класу.

Також для підвищення продуктивності та ефективності моделі нейронної мережі, важливо розглянути методи аугментації зображення, що служить двом ключовим цілям у цьому контексті: виділення ключових ознак, притаманних

зображенням що містять шрифт Брайля, і зменьшення обсягу обробляємих даних без значної втрати ключових ознак.

Враховуючи вищевказане, основною метою кваліфікаційної роботи роботи є дослідження методів і алгоритмів розпізнавання шрифта Брайля, визначення їхніх переваг і недоліків, та покращення функціонування, шляхом впровадження сучасних технологій на базі нейронної мережі. У якості практичного результату дослідження має стати розробка програмного рішення для автоматичного перекладу шрифту Брайля, що міститься на зображеннях.

Для досягнення поставленої мети дослідження необхідно розв’язати наступні часткові задачі:

- дослідження існуючих методів аугментації зображень;
- розробка датасету, що буде складатися з зображень, що містять шрифт Брайля та відповідних транскрипцій, для полегшення навчання та оцінки моделі нейронної мережі;
- дослідження та впровадження сучасних архітектур нейронних мереж, що використовуються у сфері компютерного зору, зосереджуючись на їх адаптації до розпізнавання символів Брайля;
- навчання та налаштування моделі нейронної мережі на створеному датасеті, для досягненя високої точності розпізнавання символів;
- оцінка продуктивності моделі за допомогою різних показників, таких як точність та запам’ятовування;
- дослідження потенційних вдосконалень та оптимізацій для покращення надійності та ефективності моделі у реальних сценаріях.

2 ОГЛЯД СУЧАСНИХ ІНСТРУМЕНТІВ РОЗРОБКИ

2.1 Огляд сучасних середовищ розробки і технологій для створення та навчання нейронних мереж

Нейронні мережі – це потужний інструмент, який знаходить застосування в різних областях, від обробки природної мови до машинного зору. Створення та навчання нейронних мереж потребує використання спеціалізованих інструментів та технологій. В останні роки з'явилося безліч нових середовищ розробки та технологій, які значно спрощують та прискорюють цей процес.

Мови програмування стають незамінним інструментом розробки будь-якого програмного проекту. Вибір правильної мови програмування може суттєво полегшити та прискорити весь процес від ідеї до реалізації. В даний момент існує значна кількість мов програмування, розглянемо найбільш популярні:

- C;
- C++;
- C#;
- Java;
- Rust;
- Go;
- Python.

C – статично типізована мова програмування, яка компілюється в машинний код, що тісно відповідає типовим машинним інструкціям. Через це він знайшов широке застосування в проектах, де потрібне низькорівневе програмування, таких як операційні системи та різноманітне прикладне програмне забезпечення для безлічі пристроїв – від суперкомп'ютерів до вбудованих систем. Мова C вплинула на розвиток індустрії програмного забезпечення і стала основою для створення інших мов програмування, таких

як C++, C# і Java.

C++ – мова програмування, яка компілюється в машинний код і використовується для різних цілей. Він має велику стандартну бібліотеку, що включає поширені контейнери та алгоритми, інструменти введення-виведення, роботу з регулярними виразами, підтримку багатопоточності та інші можливості. Ця мова поєднує особливості як високорівневих, так і низькорівневих мов програмування, надаючи розробникам потужні інструменти для роботи на різних рівнях абстракції.

Java – це об'єктно-орієнтована мова програмування зі строгою типізацією, призначена для різних завдань. Однією з його переваг є те, що байт-код, який вона генерує, незалежний від конкретної операційної системи та обладнання. Це означає, що Java-програми можуть виконуватися на різних пристроях, якщо вони мають відповідну віртуальну машину Java. Це забезпечує переносимість програм і зручність використання на різних платформах без необхідності переписувати код під кожен пристрій.

Rust – компільована мова програмування, створена Mozilla Research. Вона була розроблена з упором на безпеку, швидкість та паралелізм. Мова використовується для широкого спектру програм, включаючи вбудоване програмне забезпечення, системне програмування, веб-розробку, криптографію, ігрову індустрію та багато іншого.

Go – компільована мова програмування, яка була розроблена в Google. Вона була створена з упором на простоту, ефективність та масштабованість для вирішення проблем сучасної розробки програмного забезпечення. Go розроблена з упором на чистий та лаконічний синтаксис, що робить її простою для вивчення та використання. Вона має мінімалістичну і зрозумілу структуру, що допомагає розробникам писати чистий і підтримуваний код.

Python – це високорівнева інтерпретована мова програмування, яка відома своєю простотою у вивченні та використанні. Python має величезну екосистему бібліотек і модулів, які покривають практично будь-яку область: від веб-розробки та наукових обчислень до машинного навчання та обробки даних.

Це дозволяє розробникам швидко створювати програми, використовуючи готові рішення. Незважаючи на свої переваги, Python може бути не найшвидшою мовою через свій метод компіляції коду. Така особливість може створювати проблеми для деяких високопродуктивних програм.

При вивченні мов програмування відкриваються їх особливості та можливості, проте для повного розкриття їхнього потенціалу потрібні спеціалізовані інструменти, такі як фреймворки, спрямовані на прискорення процесу розробки. Перехід від вивчення мов програмування до розгляду різноманіття фреймворків супроводжується розумінням, як ці інструменти доповнюють функціональність мов, полегшуючи і прискорюючи розробку додатків. Розглянемо найбільш популярні фреймворки для створення та навчання нейронних мереж.

TensorFlow – відкрита opens-source бібліотека для машинного навчання та штучного інтелекту [29]. Вона може бути використана для різних завдань, проте головним напрямком є створення та тренування нейронних мереж. TensorFlow була розроблена дослідницькою командою Google Brain для внутрішнього використання, проте пізніше набула суспільного поширення. Ця бібліотека підтримується на наступних мовах програмування: Python, JavaScript, C++ і Java.

PyTorch – фреймворк машинного навчання, заснований на бібліотеці Torch [30]. Він підтримує різні типи нейронних мереж та надає безліч інструментів та функцій для їх створення та навчання. Спочатку розроблявся компанією Meta AI, а зараз є частиною Linux Foundation. Серед мов, що підтримуються, Python і C++, проте найбільшу популярність має версія для Python.

Keras – високорівнева бібліотека, яка виступає у ролі інтерфейсу для роботи з TensorFlow та PyTorch. Націлена на оперативну роботу з мережами глибинного навчання, при цьому спроектована так, щоб бути компактною, модульною та розширюваною. Бібліотека містить численні реалізації широко застосовуваних будівельних блоків нейронних мереж, таких як шари, цільові та

передавальні функції, оптимізатори та безліч інструментів для спрощення роботи із зображеннями та текстом.

MMDetection – це відкрита бібліотека для виявлення об'єктів на зображеннях, яка базується на PyTorch. Ця бібліотека надає інструменти навчання та тестування різних моделей комп'ютерного зору, таких як виявлення об'єктів, сегментація та інші завдання, пов'язані з обробкою зображень.

Розробка подібних програм нерозривно пов'язана з розміткою зображень. Якість і швидкість всього процесу залежать від інструментів, використовуваних для розмітки даних, оскільки вони формують основу для навчання та роботи нейронних мереж. Ці інструменти дозволяють виявляти та виділяти ключові елементи у зображеннях, створюючи коректні та різноманітні набори даних, що у свою чергу впливає на точність та ефективність моделей машинного навчання.

Microsoft Paint – простий растровий редактор, що постачається разом із усіма версіями Windows. Він є корисним, коли потрібно швидко підправити певний параметр, не запускаючи більш професійні та великовагові програми.

LabelImg – інструмент розмітки графічних зображень, написаний на Python, що використовує Qt для графічного інтерфейсу. Підтримує збереження розмітки у форматах XML, Pascal Voc Format та YOLO.

Label Studio – це open-source платформа для розмітки даних, яка підтримує різні типи анотацій, включаючи класифікацію, сегментацію, виявлення об'єктів, ключові точки та інші. Цей інструмент підтримує спільну роботу над набором даних, що дозволяє користувачам працювати разом над одним набором даних.

2.2 Вибір та обґрунтування обраних технологій

Правильне планування деталей проекту заздалегідь є надзвичайно важливим аспектом, на якому не варто економити час. Випадковий вибір без ретельного обговорення може призвести до втрати значно більшої кількості

часу для команди розробників, ніж той, який би вони витратили на уважне попереднє планування.

Одним з ключових факторів є обрання технології розробки. Навіть при наявності широкого спектру сучасних технологій, важливо враховувати, що різноманітні особливості того чи іншого язику програмування чи фреймворку можуть мати вагому роль для майбутнього успіху проекту.

У цій галузі мови програмування стають незамінним інструментом для створення та навчання нейронних мереж. Python, з його багатою екосистемою бібліотек для машинного навчання, таких як TensorFlow, PyTorch, Keras та MMDetection, став основним вибором для багатьох фахівців. Його простота та ефективність дозволяють легко приступити до створення нейронних мереж та експериментувати з різними моделями. Однак у зв'язку з його типом переведення інструкцій в машинний код, він не є оптимальним з точки зору швидкості. Якщо в проекті необхідно реалізувати якийсь ресурсомісткий процес, який не надає велика бібліотека готових рішень, то найбільш оптимальним вибором буде його реалізація на більш низькорівневому аналогу.

Python надає офіційні інструменти для інтеграції коду C++ у свої програми. Цей механізм, відомий як Python/C API, забезпечує простий доступ до функцій і структур даних, визначених на рівні C++. Завдяки цій можливості, розробники можуть створювати модулі на C++ і без проблем інтегрувати їх у свої Python-додатки, поєднуючи гнучкість та зручність Python з продуктивністю та можливостями низькорівневого програмування, що надаються C++. Такий підхід відкриває можливості використання оптимізованого коду на C++ всередині Python-додатків для підвищення їх швидкості та ефективності роботи.

При виборі бібліотек та фреймворків для розробки нейронних мереж слід звертати увагу на такі аспекти:

- ціль проекту;
- обчислювальні ресурси;
- підтримка;

- розмір спільноти;
- рівень знань та досвід розробників.

Залежно від завдання, яке переслідує проект, можуть вибиратися різні технології розробки нейронних мереж. Це може бути завдання класифікації, прогнозування, обробки зображень, навчання без учителя тощо. Так Pytorch, TensorFlow та Keras підходять для більшості завдань, а MMDetection спеціалізується на задачах детекції та сегментації зображень.

Всі перераховані вище фреймворки добре оптимізовані для підтримки обчислень на графічних процесорах компанії Nvidia, проте якщо планується навчання на процесорах інших компаній, то краще розібрати інші варіанти. Наприклад, бібліотека TensorFlow є найкращим вибором для роботи з TPU, оскільки ці процесори були спеціально створені компанією Google, так само як і сама бібліотека TensorFlow, що робить їх оптимальною комбінацією для використання.

Як і в попередньому випадку, всі перераховані фремворки мають досить високий кредит довіри оскільки розробляються великими технічними компаніями, тому питань із підтримкою протягом тривалого часу немає.

TensorFlow – це один з найпопулярніших фреймворків для машинного навчання та глибокого навчання. Має величезну спільноту, яка підтримується Google, що забезпечує широку базу знань та обмін досвідом. Він має велику кількість користувачів, активні форуми та гарну документацію.

Незважаючи на те, що PyTorch відносно новий фреймворк в порівнянні з TensorFlow, його спільнота також значна. Він популярний серед дослідників та фахівців у галузі штучного інтелекту, завдяки гнучкості та зручності використання. Має активну спільноту, форуми і велику документацію.

Keras є високорівневим інтерфейсом для роботи з нейронними мережами, і він часто використовується разом з TensorFlow. Його спільнота включає як користувачів TensorFlow, так і інших фреймворків, і хоча воно може бути менше, ніж у TensorFlow і PyTorch, воно все ж таки активно і підтримує безліч різних обговорень і ресурсів.

MMDetection – спеціалізована бібліотека для виявлення об'єктів на зображеннях. Порівняно із загальними фреймворками машинного навчання, її спільнота є відносно невеликою. Вона має свою базу користувачів, включаючи фахівців з комп'ютерного зору та задач обробки зображень.

Гнучкість фреймворку також є важливим аспектом. Наприклад, TensorFlow використовує статичну структуру обчислювального графа, де всі компоненти моделі компілюються до початку навчання. Такий підхід не дозволяє отримувати результати проміжних обчислень, що ускладнює налаштування та налагодження різних модулів. У PyTorch використовується динамічна структура графа обчислень, що забезпечує більш гнучке і ефективне налаштування компонентів моделі.

Оскільки Keras є інтерфейсом, він націлений на простоту використання, що робить його менш гнучким у порівнянні з TensorFlow та PyTorch, але доступнішим для початківців.

MMDetection є спеціалізованим фреймворком для роботи в галузі комп'ютерного зору, має гнучкість щодо алгоритмів детекції, але може бути менш гнучким в інших аспектах машинного навчання через спеціалізацію.

За всіма перерахованими параметрами, PyTorch здається найбільш підходящим інструментом. Його гнучкість, простота використання, можливості налаштування та навчання моделей, а також активна та широка спільнота роблять його найбільш привабливим вибором. Крім того, наявний досвід роботи з PyTorch дає додаткові переваги у розумінні інструментарію та швидкої адаптації до завдань поточного дослідження.

Так само важливим аспектом у розробці додатків такого типу є бібліотеки для роботи із зображеннями. Нижче представлені найбільш популярні представники на Python.

Pillow – одна з найпопулярніших бібліотек для роботи із зображеннями. Вона надає широкий спектр функцій для обробки та аналізу зображень.

OpenCV – бібліотека з відкритим кодом, яка надає ширший спектр інструментів, проте є важчою ніж Pillow. Також бібліотека має деякі готові

рішення на базі нейронних мереж, що може стати в нагоді для простих завдань.

Matplotlib – бібліотека для візуалізації даних, яка також може використовуватися для роботи із зображеннями.

Вибір Pillow як бібліотеки для роботи із зображеннями є більш очевидним. Так як у OpenCV немає відповідних готових рішень, які змогли б полегшити роботу з розробки даного проекту, її використання не є доцільним, а Matplotlib чудово підходить для складання різних типів графіків, але не надто зручний для роботи із зображеннями. Також OpenCV є більш складною в освоєнні, що робить її ще менш привабливою для вибору.

При виборі інструментів для розмітки зображень Microsoft Paint і Label Studio привертають найбільше уваги. Label Studio пропонує великий набір інструментів і дозволяє зберігати розмітку у більшості популярних форматів, що відрізняє його від LabelImg. Крім того, у Label Studio підтримуються гарячі клавіші, що значно підвищує ефективність процесу розмітки. Що стосується Microsoft Paint, він виявляється зручним, коли потрібні невеликі редагування і немає необхідності запускати більш ресурсомісткі програми.

2.3 Огляд хмарних обчислювальних сервісів

З постійним удосконаленням нейронних мереж вони ставали глибшими і складнішими, що сильно збільшило вимоги до ресурсів для їх розробки. Так у сучасних архітектурах кількість параметрів може досягати кількох мільярдів і тільки для їхнього інференсу можуть знадобитися цілі обчислювальні кластери. На допомогу приходять хмарні послуги, які надають доступ до сучасного обчислювального обладнання. Розглянемо деякі найпопулярніші сервіси.

Amazon Web Services (AWS) – це найпопулярніша в світі хмарна платформа, що пропонує понад 200 повнофункціональних сервісів для обробки даних у різних куточках світу. AWS надає значно більше сервісів та функціоналу порівняно з будь-яким іншим постачальником хмарних послуг: від інфраструктурних технологій, таких як рішення для обчислення, сховища і бази

даних, до інновацій, таких як машинне навчання, штучний інтелект, аналітика великих обсягів даних та Інтернет речей.

Microsoft Azure – платформа обчислення у хмарі, яка пропонує широкий спектр послуг. Сервіси Azure можуть включати прості веб-сервіси для розміщення бізнесу у хмарі або повністю віртуалізовані комп'ютери для виконання програмних рішень. Ця платформа пропонує різноманітні хмарні послуги, такі як віддалені сховища, бази даних та централізоване керування обліковими записами. Крім цього, Azure впроваджує нові можливості, такі як штучний інтелект (AI) та Інтернет речей (IoT). Є вигідним рішенням для організацій, які вже користуються продуктами Microsoft, такими як Windows, Office та інші.

Google Cloud Platform (GCP) – набір сервісів хмарних обчислень, який надає широкий спектр модульних хмарних сервісів, включаючи обчислення, розробку додатків, машинне навчання, аналіз даних та їх зберігання. Розробники, хмарні адміністратори та інші IT-спеціалісти можуть отримати доступ до GCP через загальнодоступні або виділені мережі. GCP відомий своїми технологічними новаціями та просунутими інструментами для штучного інтелекту, машинного навчання та аналізу даних.

IBM Cloud – це хмарна платформа від компанії IBM, що надає широкий спектр хмарних послуг для бізнесу та розробки програм. IBM відомий своєю увагою до безпеки та можливостями роботи з великими обсягами даних. Вона пропонує гібридні рішення для організацій, що використовують як хмарні, так і локальні обчислення.

DigitalOcean – хмарний провайдер, що спеціалізується на наданні простих, доступних та високопродуктивних хмарних послуг для розробників та компаній. DigitalOcean спрямований на розробників, що шукають легкість використання та управління хмарними ресурсами. Вони спеціалізуються на віртуальних серверах та наданні простих та швидких рішень.

Також досить важливим аспектом є ціна послуг. Інформація про вартість оренди обладнання за годину представлена нижче (таблиця 2.1).

Таблиця 2.1 – Вартість оренди обладнання у різних хмарах за годину

Сервіс	Nvidia V100	Nvidia P100
Amazon Web Services	$3.06	-
Microsoft Azure	$3.42	$1.05
Google Cloud Platform	$2.28	$1.26
IBM Cloud	$2.6	$1.6
DigitalOcean	$2.30	$1.10

Хоча у всіх хмарних сервісів приблизно однакові характеристики, сервіс Google здається найбільш підходящим для цього завдання. Google Cloud Platform забезпечує інтеграцію з хмарним сховищем Google Drive. Це означає, що ви можете зберігати свої проекти, датасети та файли прямо у своєму Google Drive, що зручно для організації та доступу до даних.

Google Colab надає доступ до середовища Jupyter Notebook, де можна виконувати код Python прямо в браузері. Цей сервіс забезпечує доступ до обчислювальних потужностей у хмарі, включаючи можливість використання графічних процесорів та тензорних процесорів для задач машинного навчання та обробки даних.

Інтерфейс Colab простий у використанні та володіє різними функціями, такими як додавання текстових осередків, вставка графіки та зображень, а також зручність у роботі з бібліотеками Python для наукових обчислень та машинного навчання, що робить його привабливим для дослідників, розробників та студентів.

3 РОЗРОБКА ТЕХНОЛОГІЇ РОЗПІЗНАВАННЯ ШРИФТУ БРАЙЛЯ

На основі попередньої інформації, можна припустити, що алгоритм роботи технології складатиметься з наступних етапів:

- бінаризація зображення;
- детекція символів шрифту Брайля за допомогою нейронної мережі;
- фільтрація результатів;
- переклад отриманих символів шрифту Брайля;
- відображення результатів.

3.1 Підготовка датасету

Так як навчання нейронних мереж, які вирішують задачу детекції об'єктів на зображеннях, зазвичай проводиться за допомогою supervised learning. Необхідно зібрати відповідний датасет, який складатиметься із зображень та розмітки до них.

На жаль, у відкритих джерелах вдалося знайти лише штучно згенерований датасет на платформі Kaggle, який містить чорно-білі символи Брайля (рисунок 3.1) та ніяк не підходить для поточного завдання. Тому доведеться розмітити датасет самостійно.

а) б) в)

Рисунок 3.1 – Приклади зображень з датасету Kaggle: а) буква з; б) буква о; в) буква а

Для розмітки зображення необхідно відзначити координати кожного символу Брайля та призначити відповідні їм класи. Існують різні способи завдання координат прямокутних областей, такі як запис координат центру області, ширини та висоти або координат верхнього лівого та правого нижнього кута. Незважаючи на те, що особливої різниці між ними немає, більш звичним та зручним є другий варіант. При корекційній розмітці можна відразу виправити дві точки, не витрачаючи часу обчислення координат прямокутної області. Процес розмітки зображення за допомогою Label Studio (рисунок 3.2).

Рисунок 3.2 – Процес розмітки зображення за допомогою Label Studio

Для перекладу значень символів Брайля до класів необхідно перевести їх у чисельний вигляд. Для цього кожна з шести точок буде закодована в двійковій системі, де ліва верхня точка представлятиме нижній біт, а ліва нижня представлятиме верхній біт (рисунок 3.3).

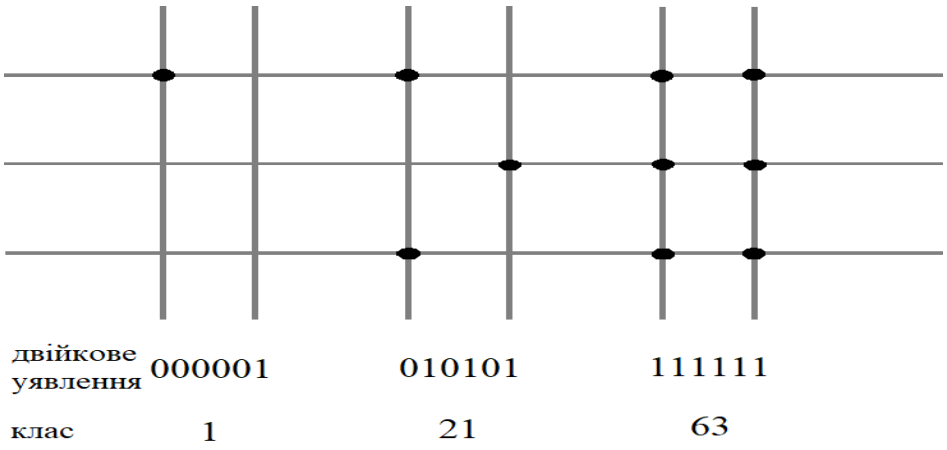


Рисунок 3.3 – Кодування символів шрифту Брайля

Отримані результати пропонується зберігати у форматі JSON, оскільки він має схожу структуру зі списками Python. Нижче наведено шаблон файлу розмітки (лістинг 3.1).

Лістинг 3.1 – Шаблон файлу розмітки

```
{
    "image_path": "path to image",
    "labels": [
        {
            "box": [X0,Y0,X1,Y1],
            "class": 1
        },
        {
            "box": [X2,Y2,X3,Y3],
            "class": 2
        }
    ]
}
```

Далі необхідно реалізувати алгоритм попередньої обробки зображень, який був обраний раніше. Так як готової реалізації алгоритму бінаризації Бредлі немає на Python, то необхідно визначити, чи є це доцільним засобами даної мови, чи варто використовувати більш низькорівневий аналог для збільшення продуктивності.

Оскільки часова складність алгоритму залежить від розміру зображень, для вимірів було обрано кілька різних зображень. Результати зведено у таблицю 3.1.

Таблиця 3.1 – Порівняння часу виконання алгоритму

Мова програмування	Розмір зображення	Час виконання, сек
Python	538x217	0.3649
C++	538x217	0.0161
Python	854x640	1.7979
C++	854x640	0.0889
Python	1158x218	0.785
C++	1158x218	0.033

Як і очікувалося, реалізація алгоритму на Python виявилася недостатньо ефективною, тому для реалізації такого ресурсомісткого алгоритму буде доцільніше використовувати мову C++.

Для того, щоб використовувати вставки C++ на Python, необхідно реалізувати алгоритм з використанням Pyton/C API. Взаємодія з API відбувається за допомогою використання класів об'єктів з Python (лістинг 3.2).

Лістинг 3.2 – Оголошення функції та ініціалізація змінних

```
extern "C" void bradley_binarization(PyObject *pixels, PyObject
*width, PyObject *height, PyObject *bradley_param){
    long w = PyLong_AsLong(width);
    long h = PyLong_AsLong(height);
    long bradley_p = PyLong_AsLong(bradley_param);

    bradley_p = std::clamp(bradley_p, 1L, 32L);

    const int S = w / bradley_p;
    int s2 = S / 2;
    const float t = 0.15;
    unsigned long* integral_image = (unsigned
long*)malloc(sizeof(unsigned long) * w * h);
    long sum = 0;
```

```
    int count = 0;
    int index;
    int x1, y1, x2, y2;
```

Команда extern "C" C++ вказує компілятору, що функція, змінна або тип даних, оголошені за допомогою цієї команди, мають посилання на C. Це означає, що вони будуть мати угоду про виклики, типи даних та інші особливості, сумісні з мовою C. У цій ситуації це необхідно для того, щоб компілятор C++ зберіг оригінальну назву функції, щоб її можна було легко знайти при портуванні DLL на Python. Для роботи з простими типами даних, які були експортовані з Python типу int, float, char необхідно перевести їх до типів даних C++ за допомогою спеціальних функцій, наприклад, PyLong_AsLong перекладає цілий тип даних Python в C++ long. Потім необхідно обчислити інтегральну матрицю зображення (лістинг 3.3).

Лістинг 3.3 – Обчислення інтегральної матриці зображення

```
for (int x = 0; x < w; x++){
        sum = 0;
        for (int y = 0; y < h; y++){
            index = y * w + x;
            sum += PyLong_AsLong(PyList_GetItem(pixels, index));
            if (x == 0)
                integral_image[index] = sum;
            else
                integral_image[index] = integral_image[index - 1]
+ sum;
        }
    }
```

Взаємодія з більш складними структурами даних імпортованих з Python, таких як списки, словники та кортежі, відбувається за допомогою спеціальних функцій, так для отримання значення яскравості зі списку pixels за індексом використовується функція PyList_GetItem. Для підрахунку інтегрального зображення необхідно здійснити прохід по всьому зображенню та обчислити нові значення яскравості.

Якщо поточний стовпець дорівнює 0, значення яскравості пікселя

дорівнює сумі значень всіх попередніх пікселів у поточному стовпці.

В іншому випадку, значення яскравості пікселя дорівнює сумі значень усіх попередніх пікселів у поточному стовпці та значення яскравості попереднього пікселя у поточному рядку.

Маючи інтегральну матрицю зображення, можна визначити оптимальний поріг бінаризації кожної області (лістинг 3.4).

Лістинг 3.4 – визначення оптимального порога бінаризації у локальній області

```
    for (int x = 0; x < w; ++x) {
        for (int y = 0; y < h; ++y) {
            index = y * w + x;
            x1 = x - s2;
            x2 = x + s2;
            y1 = y - s2;
            y2 = y + s2;
            if (x1 < 0)
                x1 = 0;
            if (x2 >= w)
                x2 = w - 1;
            if (y1 < 0)
                y1 = 0;
            if (y2 >= h)
                y2 = h - 1;
            count = (x2 - x1) * (y2 - y1);
            sum = integral_image[y2 * w + x2] - integral_image[y1
* w + x2] - integral_image[y2 * w + x1] + integral_image[y1 * w +
x1];
            long pixel = PyLong_AsLong(PyList_GetItem(pixels,
index));
            if ((pixel * count) < (long)(sum * (1.0 - t)))
                PyList_SetItem(pixels, index, PyLong_FromLong(0));
            else
                PyList_SetItem(pixels, index,
PyLong_FromLong(255));
        }
    }
```

Поточний піксель вважається класом об'єкта, якщо добуток яскравості поточного пікселя (pixel) і кількості пікселів у локальній області (count) менше ніж добуток суми значень пікселів у локальній області (sum) та коефіцієнта бінаризації (1 – t), в інших випадках піксель вважається класом фону.

Отриманий після компіляції dll файл необхідно портувати на python за допомогою бібліотеки ctype (лістинг 3.5).

Лістинг 3.5 – Портування DLL на python

```
import ctypes as ct
DLL = ct.CDLL(path_to_dll)

def bradley_binarization(pixels, width, height, bradley_param):
    bradley_bin = ct.PYFUNCTYPE(None, ct.py_object, ct.py_object,
ct.py_object, ct.py_object)(('bradley_binarization', DLL))
    bradley_bin(pixels, width, height, bradley_param)
```

ct.CDLL завантажує DLL за вказаним шляхом у пам'ять, ct.PYFUNCTYPE шукає необхідну функцію в dll файлі і при успішному знаходженні повертає покажчик на неї. Тепер отриману функцію можна використовувати у Python.

3.2 Архітектура і навчання нейронної мережі

Для даної технології було обрано архітектуру RetinaNet у якості нейронної мережі. Цей вибір зумовлений позитивними результатами, які ця архітектура продемонструвала у подібних завданнях. RetinaNet також є однією з найбільш популярних архітектур і відмінно піддається навчанню на різноманітних даних завдяки своїм унікальним особливостям.

Спочатку розглянемо архітектуру RetinaNet трохи докладніше. Складається вона з кількох модулів: backbone, feature pyramid network, classification head, regression head, anchor generator і focal loss.

Backbone – це нейронна мережа, яка отримує ознаки із зображення і надає їх FPN для подальшої обробки. Класичним прикладом для даної архітектури є використання згорткової нейронної мережі ResNet50. ResNet 50 складається з 5 шарів, кожен із яких складається з так званих bottleneck блоків (рисунок 3.4).

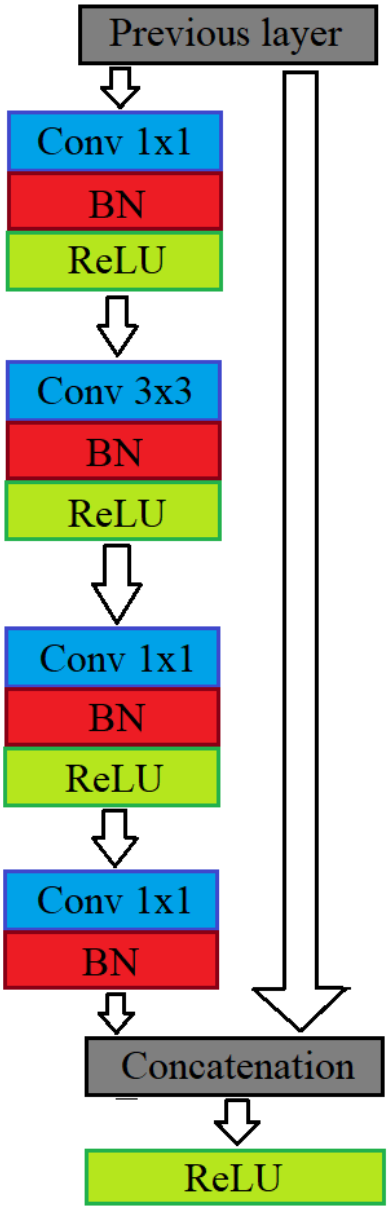


Рисунок 3.4 – Структура bottleneck блоку

Feature Pyramid Network – це основа RetinaNet, що забезпечує механізми роботи із зображеннями різного масштабу (лістинг 3.6). Витягуючи ознаки з різних шарів згорткової нейронної мережі, даний механізм робить два типи дій: downsampling, використовуючи звичайні згорткові шари та upsampling. Потім отримані таким чином карти активації подаються на вхід classification head та regression head.

Лістинг 3.6 – Ініціалізація ResNet50 with FPN

```
def resnet_fpn_extractor(
    backbone, trainable_layers = 5, returned_layers = None,
extra_blocks = None, norm_layer = None) -> BackboneWithFPN:
    layers_to_train = ["layer4", "layer3", "layer2", "layer1",
"conv1"][:trainable_layers]
    if trainable_layers == 5:
        layers_to_train.append("bn1")
    for name, parameter in backbone.named_parameters():
        if all([not name.startswith(layer) for layer in
layers_to_train]):
            parameter.requires_grad_(False)
    if extra_blocks is None:
        extra_blocks = LastLevelMaxPool()
    if returned_layers is None:
        returned_layers = [2, 3, 4]
    return_layers = {f"layer{k}": str(v) for v, k in
enumerate(returned_layers)}
    in_channels_stage2 = backbone.inplanes // 8
    in_channels_list = [in_channels_stage2 * 2 ** (i - 1) for i in
returned_layers]
    out_channels = 256
    return BackboneWithFPN(
        backbone, return_layers, in_channels_list, out_channels,
extra_blocks=extra_blocks, norm_layer=norm_layer
    )
```

Функція resnet_fpn_extractor ініціалізує об'єднаний модуль ResNet50 та FPN. На вхід подається об'єкт згорткової нейронної мережі, кількість шарів backbone, що будуть навчатися, номери шарів після яких FNP буде витягувати карти ознак, додатковий блок і блок нормалізації. За замовчуванням усі шари згорткової нейронної мережі будуть навчатися, FNP витягуватиме карти ознак з усіх шарів, крім першого, а додатковим блоком наприкінці мережі буде MaxPool з розмірами ядра 2x2. In_channels_list – це розміри вхідних каналів кожного шару FPN, а out_channels – це розмір вихідного каналу згорткової нейронної мережі.

Classification Head – це частина мережі, яка на основі отриманих ознак з FPN передбачає, чи належить область зображення якомусь класу об'єкта (рисунок 3.5).

Замість звичного BatchNorm використовуються GroupNorm. GroupNorm –

це метод нормалізації, основна ідея якого в об'єднанні каналів усередині карт активації в групи, на відміну від об'єднання батчів у BatchNorm. Для кожної групи каналів обчислюється середнє та стандартне відхилення. Після цього відбувається нормалізація активацій у кожній групі з використанням цих статистичних параметрів. На відміну від BatchNorm, GroupNorm працює незалежно від розміру батча під час навчання, що робить його більш стійким у разі маленьких батчів або коли розмір батча змінюється. Кількість вихідних каналів дорівнює добутку кількості класів на кількість якірних боксів у кожному вихідному каналі, а виходом модулю є вірогідність приналежності регіону до класу.

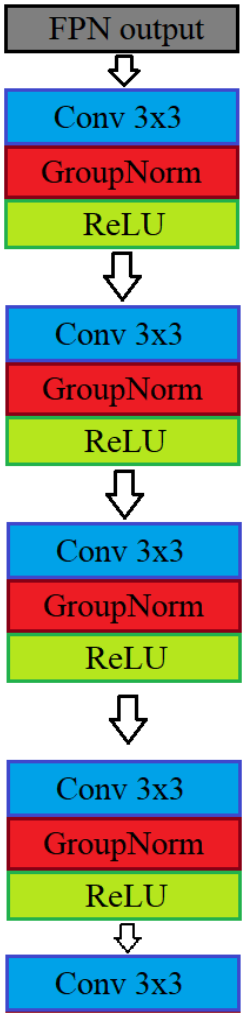


Рисунок 3.5 – Структура модулю класифікації

Regression Head – це компонент мережі, який визначає рамки (bounding boxes) для виявлених об'єктів, вказуючи їх положення на зображенні. Він також використовує ознаки з FPN для виконання цього завдання і має таку саму структуру, як і classification head (рисунок 3.5). Ключовою різницею є кількість виходів, яка залежить від кількості якірних прямокутників і обчислюэться як добуток кількості якірних прямокутників у вихідному каналі на 4. А виходом є зміщення (x, y) для кожного якірного прямокутника та зміна висоти і ширини. Для зручності ці два модулі будуть обь'єднані у один клас RetinaNetHead (лістинг 3.7).

Лістинг 3.7 – Клас RetinaNetHead

```
class RetinaNetHead(torch.nn.Module):
    def __init__(self, in_channels, num_anchors, num_classes,
norm_layer = None):
        super().__init__()
        self.classification_head = RetinaNetClassificationHead(
            in_channels, num_anchors, num_classes,
norm_layer=norm_layer
        )
        self.regression_head =
RetinaNetRegressionHead(in_channels, num_anchors,
norm_layer=norm_layer)

    def compute_loss(self, targets, head_outputs, anchors,
matched_idxs):
        return {
            "classification":
self.classification_head.compute_loss(targets, head_outputs,
matched_idxs),
            "bbox_regression":
self.regression_head.compute_loss(targets, head_outputs, anchors,
matched_idxs),
        }

    def forward(self, x):
        return {"cls_logits": self.classification_head(x),
"bbox_regression": self.regression_head(x)}
```

Клас RetinaNetHead об'єднує в собі модулі RegressionHead та ClassificationHead. До методу forward подаються карти ознак, отримані з FNP, а

результатом є словник, що містить координати виявлених об'єктів та ймовірності їх приналежності до класів.

Anchor generator – це частина мережі, що відповідає за початкову генерацію рамок, що обмежують регіони интересу. На відміну від інших компонентів мережі, не змінюється з навчанням, а задається користувачем на основі поставленого завдання (лістинг 3.8).

Лістинг 3.8 – Створення класу AnchorGenerator

```
def anchorgen():
    anchor_sizes = tuple((x, int(x * 2 ** (1.0 / 3)), int(x * 2 **
(2.0 / 3))) for x in [32, 64, 128, 256, 512])
    aspect_ratios = ((0.5, 1.0, 2.0),) * len(anchor_sizes)
    anchor_generator = AnchorGenerator(anchor_sizes,
aspect_ratios)
    return anchor_generator
```

Функція anchorgen визначає кількість та розмір якірних прямокутників для кожного вихідного каналу.

Усі отримані модулі після об'єднання становлять модель RetinaNet (лістинг 3.9).

Лістинг 3.9 – Створення RetinaNet

```
def create_retinanet(*,num_classes = 64,trainable_backbone_layers
= 5,**kwargs) -> RetinaNet:
    backbone = ResNet(Bottleneck, [3, 4, 6, 3])
    backbone = resnet_fpn_extractor(backbone,
trainable_backbone_layers, returned_layers=[2, 3, 4])
    anchor_generator = anchorgen()
    head = RetinaNetHead(backbone.out_channels,
anchor_generator.num_anchors_per_location()[0], num_classes,
norm_layer=partial(torch.nn.GroupNorm, 32))
    model = RetinaNet(backbone, num_classes,
anchor_generator=anchor_generator, head=head,
detections_per_img=800, topk_candidates=1600, **kwargs)
    return model
```

Функція create_retinanet формує модель RetinaNet, поєднуючи отримані модулі. Параметр detections_per_img визначає кількість кандидатів, які зберігаються після процедури Non-Maximum Suppression (NMS), тоді як topk_candidates вказує на кількість кандидатів, які залишаються до застосування NMS.

NMS – це метод, що використовується в завданнях виявлення об'єктів, щоб відсіяти зайві передбачувані кандидати, залишивши лише найдостовірніші або ті, які мають найбільший перетин з цільовими пророкуваннями. Враховуючи, що на кожному зображенні потрібно виявити символи шрифту Брайля, максимальна кількість яких у датасеті становить 577, рекомендується встановити параметри detections_per_img і topk_candidates з надлишком.

Далі потрібно підготувати дані для навчання моделі, які складаються із зображень та їх розмітки. Ці дані повинні бути перетворені на тензори, а значення пікселів зображень повинні бути змінені так, щоб вони знаходилися в діапазоні від -1 до 1. Для цього створимо клас Braille Dataset, який успадковується від класу torch.utils.data.Dataset (лістинг 3.10).

Лістинг 3.10 – Клас BrailleDataset

```
class BrailleDataset(Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        img_path = self.data[idx]['image']
        image = {"image": Image.open(img_path), "path": img_path}
        targets = self.data[idx]['targets']

        if self.transform:
            image['image'] = self.transform(image['image'])
            targets['boxes'] = torch.tensor(targets['boxes'])
            targets['labels'] = torch.tensor(targets['labels'],
dtype=torch.int)
        return image, targets
```

Клас BrailleDataset має три методи: конструктор, геттер та метод отримання довжини. У конструктор подаються дані та перетворення, які необхідно до них застосувати. Так, при зверненні до елемента датасету буде викликаний геттер, який поверне перетворені зображення та розмітку. Маючи такий клас, можливо створити ітератор з об'єктів датасета (лістинг 3.11).

Лістинг 3.11 – Створення ітераторів

```
def load_data(path):
    transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])
    targets = []
    for label_file in os.listdir(path):
        with open(os.path.join(labels_path, label_file)) as file:
            labels_dict = json.load(file)
            boxes = []
            labels = []
            for label in labels_dict["labels"]:
                boxes.append(label["box"])
                labels.append(label["class"])
            targets.append({"image": labels_dict["image"],
"targets": {"boxes": boxes, "labels": labels}})

    trainset_len = round(len(targets) * 0.9)

    trainset = BrailleDataset(targets[:trainset_len], transform)
    testset = BrailleDataset(targets[trainset_len:], transform)

    trainloader = DataLoader(trainset, batch_size=1, shuffle=True,
num_workers=1, collate_fn=collate_fn)
    testloader = DataLoader(testset, batch_size=1, shuffle=False,
num_workers=1, collate_fn=collate_fn)

    return trainloader, testloader
```

Функція load_data проходить по всіх файлах міток у зазначеній папці, завантажуючи інформацію про рамки та мітки символів Брайля для кожного зображення. Отримані дані обертаються в клас BrailleDataset, з яких потім створюється завантажувач для навчальної вибірки trainloader та тестуючої вибірки testloader.

Маючи модель і завантажувач даних відбувається навчання (лістинг 3.12).

Лістинг 3.12 – Навчання неромережі

```
def train():
    torch.cuda.empty_cache()
    trainloader, testloader = load_data()
    model = create_retinanet(num_classes=64,
trainable_backbone_layers=5, detections_per_img=800,
topk_candidates=1600).cuda()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    model.train()
    for epoch in range(epochs):
        i = 0
        train_loader_len = len(trainloader)
        for i, data in enumerate(trainloader, 0):
            images, targets = data
            optimizer.zero_grad()

            # targets = [x.cuda() for x in targets]
            images = [x.cuda() for x in images]
            for target in targets:
                target['boxes'] = target['boxes'].cuda()
                target['labels'] = target['labels'].cuda()
            loss = model(images, targets)
            loss = sum(loss for loss in loss.values())
            loss.backward()
            optimizer.step()
            if i % 10 == 0:
                with open("results.txt", "+w") as file:
                    file.write(f"Epoch {epoch + 1}/100, Batch
{i}/{train_loader_len}, Loss: {loss.item()}")
    torch.save(model, f"model_{epoch}.pth")
```

Функція train організує процес навчання нейронної мережі. Перед початком навчання очищується кеш графічного процесора. Потім створюються завантажувачі даних та об'єкт моделі, які згодом поміщаються на пам’ять графічного процесора.

У ролі алгоритму оптимізації був обран Adaptive Moment Estimation – оптимізаційний алгоритм, що використовується в глибокому навчанні для оновлення вагів моделі на основі градієнтів, обчислених під час зворотного поширення помилки. Він поєднує у собі ідеї двох інших оптимізаційних методів Momentum та RMSProp. Як і метод Momentum Adam використовує момент для прискорення навчання. Він зберігає експоненційно спадне середнє попередніх градієнтів і використовує його для вирівнювання шумів у

градієнтах. Adam також зберігає експоненційно спадне середнє квадратів градієнтів, щоб нормалізувати оновлення параметрів, як RMSProp. Перевагами цього методу є:

- ефективно працює з великими наборами даних та моделями з великою кількістю параметрів;
- адаптується до різних швидкостей навчання для різних параметрів;
- зазвичай потребує меншої кількості налаштувань гіперпараметрів у порівнянні з іншими методами.

У циклі відбувається ітерація по завантажувачу навчальних даних і кожної ітерації витягуються зображення і мітки до них. Ці дані подаються на вхід нейронної мережі, відповіддю якої є classification та regression loss. Отримані значення втрат сумуються і на їхній основі обчислюються градієнти методом зворотнього поширення помилки. На основі обчислених градієнтів алгоритм оптимізації оновлює ваги моделі.

Отримані результати помилок зберігаються для подальшої побудови графіка навчання та контролю за навчанням. Після закінчення навчання отримані ваги моделі зберігаються для подальшого використання.

Однак враховуючи специфіку даних у датасеті, класичний спосіб навчання здається дуже тривалим та недоцільним завданням. На відміну від класифікації або детекції, де на зображенні є декілька об'єктів, на кожному зображенні необхідно розмітити кожен символ шрифту Брайля, число яких може перевищувати декілька сотен.

Тому оптимальнішим буде підхід active learning, суть якого полягає у навчанні моделі на невеликій кількості розмічених даних та подальшому застосуванні на нерозмічених. Потім відбираються дані, у яких модель має найменшу ступінь впевненості чи має багато помилок. Такі дані розмічаються та повторюється навчання на розширеному наборі даних.

Зрештою, за допомогою такого підходу вдалося розмітити 168 зображень. Отриманий датасет був розділений на 150 зображень тренувальної вибірки та 18 валідаційних зображень.

Для того, щоб визначити, чи нейронна мережа навчилася розпізнавати символи шрифту Брайля на зображеннях, а не просто запам’ятала дані, необхідно побудувати графік навчання, який відображає кількість помилок, які мережа робила під час навчання. Якщо значення помилок не зменьшуються з часом чи навпаки збільшуються, то це сигналізує про перенавчання мережі. Графік навчання нейронної мережі на тренувальній вибірці (рисунок 3.5). На графіку навчання видно, що модель поступово зменшувала кількість помилок. Це означає, що модель ефективно навчалася та не піддавалася перенавчанню.

Рисунок 3.5 – Графік навчання моделі

3.3 Оцінка якості розробленої технології

Ключовим показником якості технологій на базі нейронних мереж є її точність, для справедливої оцінки якої необхідно обчислити два показники, це точність детекції та класифікації на валідаційній вибірці. Для оцінки точності детекції буде використано параметр Intersection Over Union(IOU), для розрахунку якого необхідно знайти площу перетину отриманого прямокутника та відповідного прямокутника, який був розмічений при створенні датасета, отриману величину необхідно розділити на їх загальну площу (лістинг 3.13).

Лістинг 3.13 – Обчислення значення IOU

```
def calculate_iou(targets, predictions):
    xA = max(targets[0], predictions[0])
    yA = max(targets[1], predictions[1])
    xB = min(targets[2], predictions[2])
    yB = min(targets[3], predictions[3])
    inter_area = max(0, xB - xA + 1) * max(0, yB - yA + 1)
    boxA_area = (targets[2] - targets[0] + 1) * (targets[3] -
targets[1] + 1)
    boxB_area = (predictions[2] - predictions[0] + 1) *
(predictions[3] - predictions[1] + 1)
    return inter_area / float(boxA_area + boxB_area - inter_area)
```

Передбачення можливо умовно поділити на три типи: True positive, False positive та False negative (рисунок 3.6). True positive – коли модель чітко передбачила клас і коефіцієнт IOU більше 0.5. False positive – коли модель помилилася із класом, чи коефіцієнт IOU менше 0.5. False negative – коли модель не виявила потрібний об'єкт. Тому за наявності прогнозів типу false positive чи false negative, необхідно реалізувати алгоритм пошуку передбачень, який відповідають цільовим.

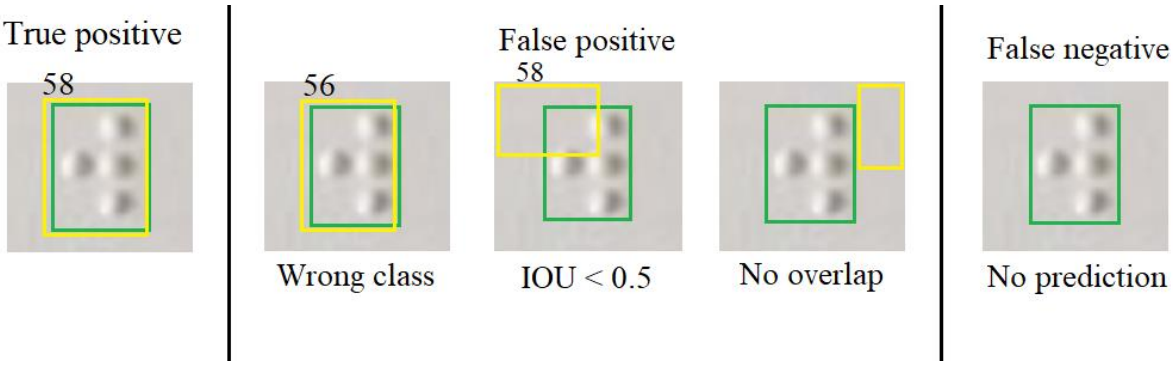


Рисунок 3.6 – Типи помилок передбачень

Для аналізу результатів детекції необхідно їх упорядкувати по рядках, а також відкинути передбачення одного й того ж об'єкта, ґрунтуючись на впевненості моделі. Угруповання передбачень рядками зліва направо (лістинг 3.14).

Лістинг 3.14 – Угруповання передбачень

```
min_height = min(map(lambda x: x[3] - x[1], boxes)) * 0.8
predictions = [{"box": box, "label": label, "score": score} for
box, label, score in zip(boxes, labels, scores)]
lines = {}
for pred in predictions:
    y = pred["box"][1]
    key = list(filter(lambda x: abs(x - y) < min_height,
lines.keys()))
    key = None if len(key) == 0 else key[0]
    if key is None:
        lines[y] = [pred]
    else:
        lines[key].append(pred)
for key in lines:
    lines[key].sort(key=lambda x: x["box"][0])
```

Сортування передбачень здійснюється за наступним алгоритмом:

- обчислення мінімальної висоти min_height;
- для кожного передбачення виконується пошук ключів у словнику lines, у яких значення висоти близькі на основі min_height;
- якщо відповідний ключ не знайдено, то створюється новий запис у словнику lines;
- якщо знайдено відповідний ключ, то поточне передбачення додається до відповідної групи у словнику;
- сортування всіх ліній за значенням x координати лівого верхнього кута.

Далі необхідно відфільтрувати множинні передбачення для одного об'єкта (лістинг 3.15).

Лістинг 3.15 – Фільтрація зайвих передбачень

```
nms_predictions = {key: [] for key in lines}
for i in range(0, len(line)):
    left_x1 = line[i]["box"][0]
    left_y1 = line[i]["box"][1]
    left_x2 = line[i]["box"][2]
    left_y2 = line[i]["box"][3]
    left_width = left_x2 - left_x1
    left_height = left_y2 - left_y1
```

```
    score_left = line[i]["score"]
    duplicate_i = None
    for k in range(len(nms_predictions[key])):
        right_x1 = nms_predictions[key][k]["box"][0]
        right_y1 = nms_predictions[key][k]["box"][1]
        right_x2 = nms_predictions[key][k]["box"][2]
        right_y2 = nms_predictions[key][k]["box"][3]
        right_width = right_x2 - right_x1
        right_height = right_y2 - right_y1
        if ((right_x1 - left_x1) ** 2 + (right_y1 - left_y1) ** 2)
** 1/2 < min(left_height, left_width, right_height, right_width) /
2:
            duplicate_i = k
            break
    if duplicate_i is not None:
        score_right = nms_predictions[key][duplicate_i]["score"]
        if score_left > score_right:
            nms_predictions[key][duplicate_i] = line[i]
    else:
        nms_predictions[key].append(line[i])
```

Алгоритм Non-Maximum Supression для поточного завдання складається з наступних етапів:

- ітерація по рядках;
- для кожного передбачення у поточному рядку перевіряються перетини з іншими передбаченнями того ж рядка;
- якщо виникає перетин з будь-яким із передбачень, вибирається передбачення на основі впевненості моделі;
- якщо немає перетинів, то поточне передбачення додається до кінцевого списку.

Маючи передбачення, які впорядковані за рядками, ми можемо оцінити ефективність технології. Для кожного передбачення необхідно обчислити IOU і порівняти класи з цільовими (лістинг 3.16).

Лістинг 3.16 – Обчислення ефективності технології

```
def calculate_precision(predictions, targets):
    correct_detections = 0
    correct_classifications = 0
    for key_p, key_t in zip(predictions.keys(), targets.keys()):
        for num_p, box_p in enumerate(predictions[key_p]):
```

```
            min_distance = [10000, 0]
            for num_t, box_t in enumerate(targets[key_t]):
                distance = abs(box_p['box'][0] - box_t['box'][0])
                if distance < min_distance[0]:
                    min_distance = [distance, num_t]
            iou_value = calculate_iou(box_p['box'],
targets[key_t][min_distance[1]]['box'])
            if iou_value > 0.7:
                correct_detections += 1
                if box_p['label'] ==
targets[key_t][min_distance[1]]['label']:
                    correct_classifications += 1

    return correct_detections, correct_classifications
```

Алгоритм підрахунку точності технології на базі нейронної мережі можна розділити на такі етапи:

- для кожного передбачення, зробленого нейронною мережею, шукається найближча мітка в цільових даних, у межах поточної лінії;
- для обраної пари передбачення і цільового прямокутника обчислюється значення IOU;
- якщо значення IOU більше 0.7, збільшується лічильник правильних детекцій;
- якщо клас передбачення збігається з класом цільового прямокутника та значення IOU більше 0.7, збільшується лічильник правильних класифікацій та детекцій.

При аналізі валідаційної вибірки (лістинг 3.17), яка складається з 18 зображень та містить 4630 символів Брайлю, нейронна мережа здійснила 4158 правильних передбачень. Отже, остаточна точність детекції на валідаційній вибірці склала приблизно 90%. З цих 4158 правильних передбачень, 4095 були успішними в класифікації, що становить приблизно 98% точності.

Лістинг 3.17 – Тест нейронної мережі на валідаційній вибірці

```
def validate():
    model = load_model().eval().cuda()
    total_detections = 0
    total_classifications = 0
```

```
    total_correct_detections = 0
    total_correct_classifications = 0
    trainloader, testloader = load_data()
    with torch.no_grad():
        for images_dict, targets in testloader:
            images = [x['image'].cuda() for x in images_dict]
            for target in targets:
                target['boxes'] = target['boxes'].cuda()
                target['labels'] = target['labels'].cuda()
            outputs = model(images)[0]
            boxes = outputs['boxes'].tolist()
            labels = outputs['labels'].tolist()
            scores = outputs['scores'].tolist()
            boxes, labels, scores, lines = filter_res(boxes,
labels, scores)
targets_lines = targets_to_lines(targets[0]['boxes'].tolist(),
targets[0]['labels'].tolist(), [1.0 for x in
range(len(targets[0]["boxes"]))])
            total_detections += len(targets[0]["boxes"])

            correct_predictions, correct_classifications =
calculate_precision(lines, targets_lines)
            total_classifications += correct_predictions
            total_correct_detections += correct_predictions
            total_correct_classifications +=
correct_classifications
```

Функція valdiate організовує тестування нейронної мережі на валідаційній вибірці. load_model().eval().cuda() здійснює завантаження навченої нейронної мережі, переводить її в режим оцінки і поміщає на графічний процесор. Потім створюються лічильники кількості виявлених об'єктів та правильних передбачень, а також ітератори з датасету. Отримані передбачення нейронної мережі фільтруються та сортуються. Для кожного зображення підраховуються кількість символів та кількість вірних передбачень.

Щоб візуалізувати результати передбачення, потрібно виконати кілька кроків. Спочатку потрібно накласти рамки на вихідне зображення у місцях, де було виявлено об'єкти. Після цього необхідно перекласти отримані символи шрифту Брайля і так як символи універсальні – не змінюються для різних мов, то перед початком перекладу необхідно знати якою мовою був написан вихідний текст. Для перекладу необхідно скористатись спеціальною таблицею символів для конкретної мови (рисунок 3.7).

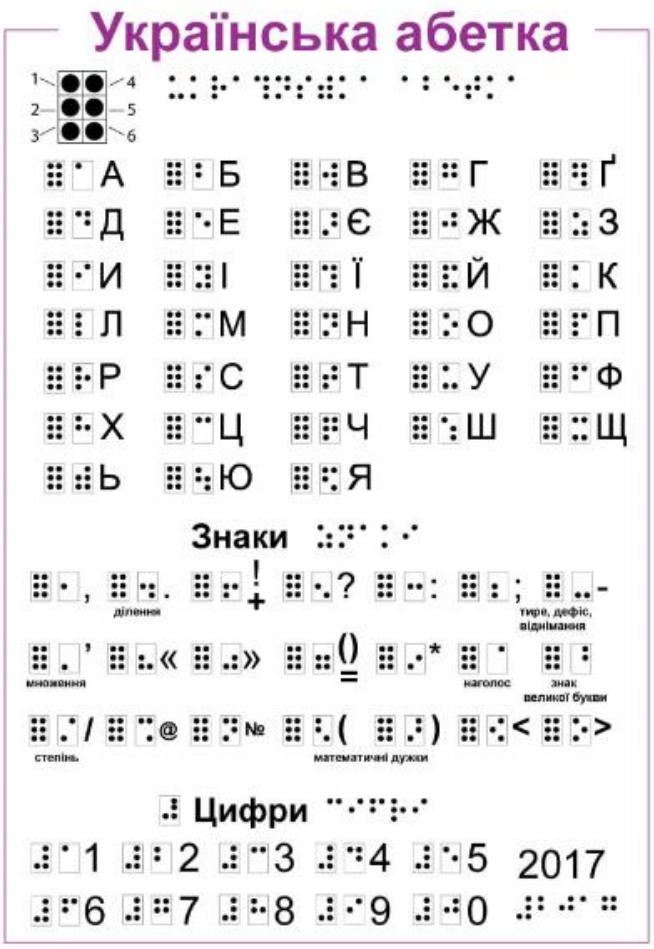


Рисунок 3.7 – Українська абетка шрифту Брайля

Після цього можливо відобразити символи, виявлені усередині цих рамок, щоб візуально показати, що було виявлено на зображенні (лістинг 3.18).

Лістинг 3.18 – Візуалізація результатів

```
def draw(img, boxes, labels, scores):

    head, tail = os.path.split(img)
    img = Image.open(os.path.join(sources_path, tail))
    draw = ImageDraw.Draw(img)
    color = (0, 255, 0)
    width = 2
    text_color = (50, 50, 255)
    braille_font = ImageFont.truetype(path_to_braile_font, 15)
    font = ImageFont.truetype("arial.ttf", 15)

    for box, score, label in zip(boxes, scores, labels):
```

```
        if score < 0.5:
            continue
        x1, y1, x2, y2 = box
        width = x2 - x1
        height = y2 - y1
        draw.rectangle((x1, y1, x2, y2), outline=color, width=2)

        label_text = russian_braille_dict.get(braille[label],
braille[label])
        my_text_size_x, my_text_size_y = draw.textsize(label_text,
font=font)
        braille_text_size_x, braille_text_size_y =
draw.textsize(braille[label], font=braile_font)
        centred_x = x1 + width / 2 - my_text_size_x
        centred_y = y1 + height / 2 - my_text_size_y / 2
        draw.text((centred_x, centred_y), label_text,
fill=text_color, font=font)
        draw.text((x1 + width / 2, y1 + height / 2 -
braille_text_size_y / 2), braille[label], fill=text_color,
font=braile_font)

    img.show()
```

За допомогою бібліотеки Pillow завантажується вихідне зображення, а також створюється об'єкт draw для відтворення результатів. Проходячи по кожній рамці (box), оцінці впевненості (score) та мітці класу (label), перевіряється впевненість передбачення. Якщо впевненість передбачення перевищує 50%, то малюється прямокутник навколо об'єкта, центрована текстова мітка об'єкта, яка складається із символу шрифту Брайля та перекладеного символу. Після всіх операцій над зображенням, воно відображається за допомогою img.show() (рисунок 3.8).

3.4 Можливості покращення технології

Незважаючи на те, що розроблена технологія показала непогані результати, стислий термін дослідження не дозволив повністю розкрити її потенціал. Розробка технологій на базі нейронних мереж потребує великої кількості емпіричних тестів, а також розмітки даних для навчання, які потребують багато часу. Таким чином, у розробників технології є можливість її поліпшення, при подальшій розробці.

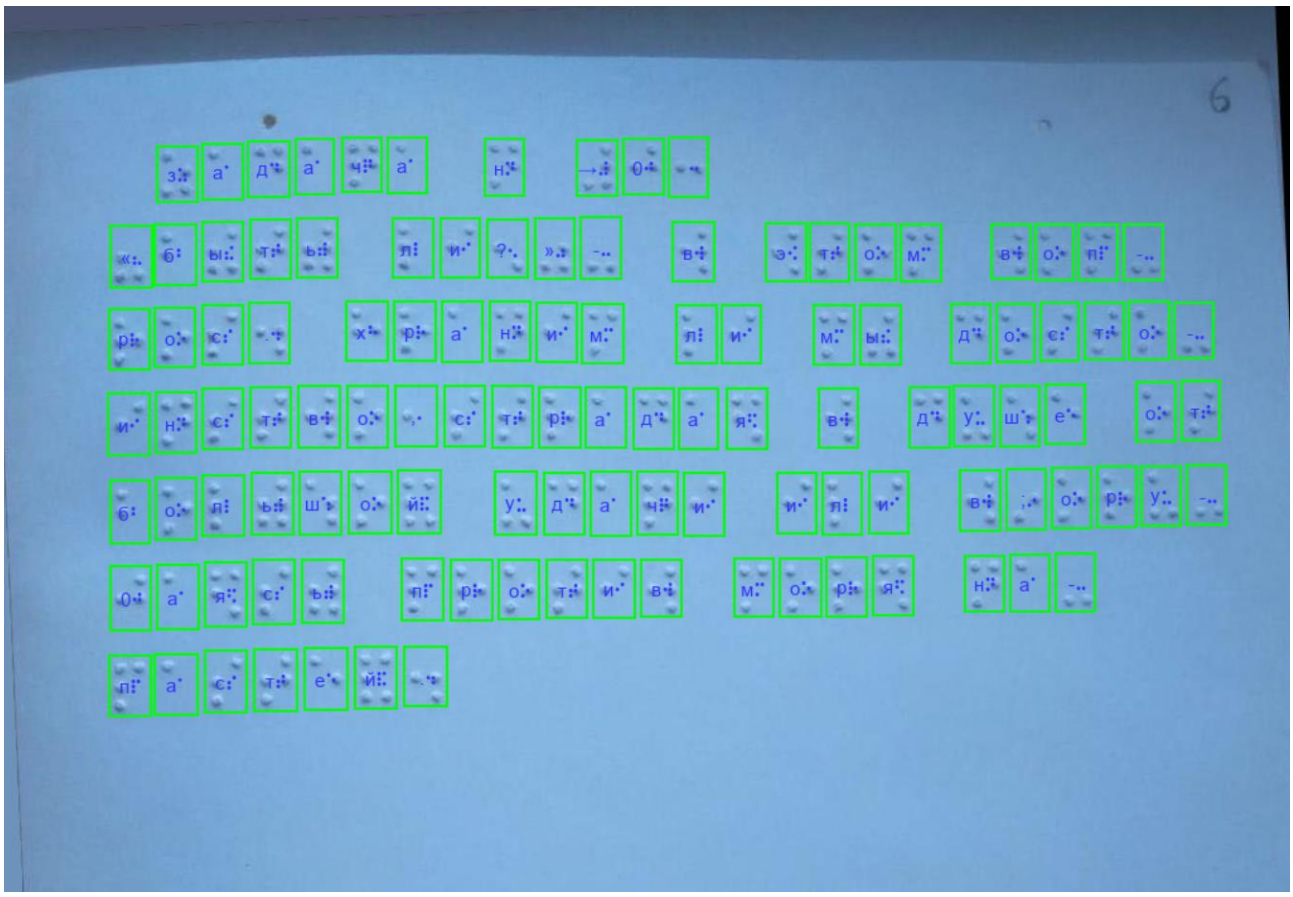


Рисунок 3.8 – Візуалізація результатів передбачень

Так одним з можливих покращень є пошук та розмітка більшої кількості даних для навчання, що дозволить нейронній мережі краще узагальнювати дані та робити більш точні передбачення.

Також можна спробувати покращити процес навчання нейронної мережі, наприклад, використовуючи ефективніші методи оптимізації помилки або використовуючи потужніші обчислювальні ресурси.

Ще одним варіантом є використання більш легковажних згорткових нейронних мереж у якості backbone для моделі. Це дозволить використовувати технологію на менш потужних обчислювальних машинах або мобільних пристроях.

Також можливо розробити спеціальний алгоритм розпізнавання мови, якою був закодован шрифт Брайлю з метою покращення взаємодії з користувачем та розширення можливостей перекладу на різні мови.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було розглянуто актуальність завдання перекладу шрифту Брайля на зображенні, розглянуті існуючі методи вирішення завдання та готові комерційні рішення. Було проаналізовано існуючі підходи машинного навчання і завдання комп'ютерного зору для яких ефективним є застосування нейронних мереж. Досліджено технології, що використовуються в сучасних архітектурах нейронних мереж, та найбільш сучасні архітектури, що найактивніше застосовуються у цій галузі.

У роботі наведено огляд сучасних середовищ розробки та засобів для розробки технологій на базі нейронної мережі, наведено обґрунтування вибору технологій для реалізації програмного продукту.

Проведено аналіз існуючих методів бінаризації. Знайдено найбільш оптимальний алгоритм для бінаризації зображень, що містять шрифт Брайля. Виконано оптимізацію бистродії бінаризації зображень, завдяки реалізації алгоритму на більш низькорівневій мові програмування та подальшому портуванні за допомогою Python/C API.

Розроблено технологію на базі нейронної мережі RetinaNet, що виконує детекцію та переклад символів Брайля на зображеннях. Завдяки використанню найбільш сучасних хмарних провайдерів здійснено швидкісне навчання нейронної мережі та її тестування.

При аналізі ефективності, за допомогою обчислення коефіцієнту IOU, розроблена технологія показала 90% точності детекції та 98% точності класифікації символів шрифту Брайля на зображеннях.

За допомогою підходу active learning створений датасет, що складається із 168 розмічених зображень шрифту Брайля.

За результатами досліджень кваліфікаційної роботи опубліковано тези доповіді на одинадцятій міжнародній науково-технічній конференції «Проблеми інформатизації» 16-17 листопада 2023 року.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Isayed, S. A review of optical Braille recognition / S. Isayed, R. Tahboub //2015 2nd World Symposium on Web Applications and Networking (WSWAN). – IEEE. 2015. –С. 1–6.

2. Hahnloser, R., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., & Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature, 405, 947–951.

3. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. В Advances in Neural Information Processing Systems 25 (с. 1097–1105).

4. Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. В International Conference on Learning Representations (ICLR).

5. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. В Proceedings of the IEEE conference on computer vision and pattern recognition (с. 1–9).

6. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15(56), 1929–1958.

7. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. В 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (с. 770–778).

8. Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. В Proceedings of the IEEE conference on computer vision and pattern recognition (с. 4700–4708).

9. Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. Proceedings of

the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), c. 580–587.

10. Girshick, R. (2015). Fast RCNN. Proceedings of the IEEE International Conference on Computer Vision (ICCV), c. 1440–1448.

11. Ren, S., Sun, J., Wang, X. (2015). Faster RCNN: Towards Realtime Object Detection with Region Proposal Networks. Advances in Neural Information Processing Systems (NIPS), c. 91–99.

12. Redmon, J., Divvala, S., Girshick, R., Farhadi, A. (2015). You Only Look Once: Unified, Realtime Object Detection. arXiv preprint arXiv:1506.02640.

13. Liu, W., Anguelov, D., Szegedy, C., Reed, S., Fu, C. Y., Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. Proceedings of the European Conference on Computer Vision (ECCV), c. 21–37.

14. Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., Belongie, S. (2017). Focal Loss for Dense Object Detection. Proceedings of the IEEE International Conference on Computer Vision (ICCV), c. 2999–3007.

15. Sheikh, Y., Darrell, T., Efros, A. A. (2014). Deconvolutional Networks for Image Restoration. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

16. Yang, Y., Zhang, X., Yang, J., Wang, X. (2010). Unpooling Operations for Convolutional Neural Networks. Proceedings of the IEEE International Conference on Computer Vision (ICCV), c. 1330–1337.

17. Zhao, H., Yu, J., Shi, J., Wang, X., Jia, J. (2017). Multiscale Context Aggregation for Semantic Segmentation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), c. 5591–5599.

18. He, K., Zhang, X., Ren, S., Sun, J. (2014). Spatial pyramid pooling in deep convolutional networks for visual recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), c. 740–748.

19. Lafferty, J., McCallum, A., Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Proceedings of the 18th International Conference on Machine Learning (ICML), c.

282–289.

20. Chen, L. C., et al. (2017). DeepLab v3: Encoder-Decoder with Atrous Convolution for Semantic Image Segmentation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), c. 2912–2921.

21. Ronneberger, O., Fischer, P., Brox, T. (2015). UNet: Convolutional networks for biomedical image segmentation. In Medical image computing and computer-assisted intervention (MICCAI), c. 234–241.

22. He, K., et al. (2017). Mask RCNN: Object detection via region-based convolutional neural networks. Proceedings of the IEEE International Conference on Computer Vision (ICCV), c. 2980–2988.

23. Otsu, N. (1979). A threshold selection method from grey level histograms. IEEE Trans. Syst. Man Cybernet. SMC–9, c. 62–66.

24. Bernsen, J. (1986). Dynamic thresholding of gray-level images. In Int. Conf. Pattern Recognition, vol. 2, c. 1251–1255.

25. Niblack, W. (1986). An Introduction to Digital Image Processing, c. 115–116. Englewood Cliffs, N.J.: Prentice Hall.

26. Sauvola, J., Pietikainen, M. (2000). Adaptive Document Image Binarization. The Journal of the Pattern Recognition Society, PR 33, c. 225–236.

27. Wolf, C., Jolion, J. M., Chassaing, F. (2002). Text localization, enhancement and binarization in multimedia documents. In Proceedings of the 16th International Conference on Pattern Recognition (Vol. 2, c. 1037–1040). IEEE.

28. Bradley, D., Roth, G. (2007). Adaptive Thresholding using the Integral Image. Journal of Graphics Tools, 12(2), 13–21.

29. Abadi, M., Barham, P., & Chen, J. (2016). TensorFlow: A System for Large-Scale Machine Learning. Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

30. Ketkar, Nikhil (2017). "Introduction to PyTorch". Deep Learning with Python. Apress, Berkeley, CA. pp. 195–208.