ДВНЗ «Донецький національний технічний університет»

Факультет комп'ютерно-інформаційних технологій та автоматизації

Кафедра прикладної математики та інформатики

«До захисту допущено»

Завідувач кафедри ПМІ

_____ **О. А. Дмитрієва**
(підпис) (ініціали, прізвище)

«____»_____2021 р.

Випускна кваліфікаційна робота магістра

на тему «Удосконалення технології розпізнавання облич за допомогою згорткових нейронних мереж»

Виконав: студент 2 курсу, групи КНм-20
(шифр групи)

спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

Мякшина О.Д. _____
(прізвище та ініціали) (підпис)

Керівник проф. каф. ПМІ, д.т.н., проф. Башков Є.О. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Нормоконтроль:

_____ *І. А. Назарова*
(підпис)

_____
(дата)

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

Студент _____
(підпис)

Покровськ – 2021 р.

ДВНЗ «Донецький національний технічний університет»

Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики
Освітній ступінь магістр
Спеціальність (напрям підготовки) 122 Комп'ютерні науки

(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМІ
_____ /О. А. Дмитрієва/
«____»_____2021 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Мякшину Олександру Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Удосконалення технології розпізнавання облич за допомогою згорткових нейронних мереж
керівник роботи проф. каф. ПМІ, д.т.н., Башков Євген Олександрович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом від « 22 » вересня 20 21 року № 570
2. Строк подання студентом роботи 7 грудня 2021 року
3. Вихідні дані до роботи результати переддипломної практики
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) аналітичний огляд принципів розпізнавання обличь за допомогою згорткових нейронних мереж; 2) аналіз сучасних моделей згорткових нейронних мереж для розпізнавання обличь, вибір модифікації; 3) розробка та оцінка модифікованої системи розпізнавання обличь

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) 1) загальна постановка задачі; 2) архітектура системи розпізнавання облич; 3) виділення признаків; 4) модифікація мережі; 5) використані датасети; 6) розробка та навчання моделей; 7) тестування моделей; 8) висновки; 9) список використаних джерел; 10) список публікацій

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 4 жовтня 2021 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналітичний огляд предметної області, дослідження теоретичної бази, вивчення джерел інформації	04.10.21-17.10.21	
2	Аналіз сучасних моделей згорткових нейронних мереж, вибір модифікацій	18.10.21-31.10.21	
3	Розробка та навчання модифікованої моделі нейронної мережі для розпізнавання облич	01.11.21-14.11.21	
4	Тестування та оцінка розробленої мережі	15.11.21-30.11.21	
5	Оформлення пояснювальної записки та опис створеної системи. Проходження нормоконтролю	01.12.21-06.12.21	
6	Перевірка пояснювальної записки антиплагіатною системою. Оформлення презентації до роботи, графічного матеріалу та рецензування роботи	07.12.21-20.12.21	
7	Захист роботи	21.12.21	

Студент _____ **Мякшин О. Д.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Башков Є. О.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Мякшин О. Д. Дослідження розпізнавання облич за допомогою згорткових нейронних мереж / Випускна кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки. – ДВНЗ ДонНТУ, Покровськ, 2021.

Випускну магістерську роботу присвячено дослідженню методів побудови та алгоритмів роботи згорткових нейронних мереж в контексті розпізнавання обличь для визначення шляхів покращення вже існуючих моделей.

Мета дослідження – пошук та реалізація підходів удосконалення системи розпізнавання обличь, які дозволяють розпізнавати об'єкти з підвищеною точністю розпізнавання без зниження швидкості.

Об'єкт дослідження даної роботи – процес розпізнавання обличь за допомогою згорткових нейронних мереж.

Предмет дослідження – архітектури та можливості згорткових нейронних мереж для вирішення задачі розпізнавання обличь на двовимірних зображеннях.

Наукова новизна полягає в отриманні більш ефективної архітектури згорткової нейронної мережі, яка дозволяє підвищити точність розпізнавання облич.

Практичне значення полягає в накопиченні рекомендацій щодо розробки та навчання згорткових нейронних мереж в цілях використання в системах розпізнавання обличь на основі аналізу сучасних архітектур та принципів роботи існуючих моделей.

Ключові слова: згорткова, нейронна мережа, дослідження, розпізнавання, детекція, обличь, глибоке навчання, CNN, Python, TensorFlow, Keras.

Список публікацій здобувача:

1. Мякшин О. Д. Архітектури згорткових нейронних мереж для задачі розпізнавання облич / О. Д. Мякшин, П. О. Ануфрієв // Наукові досягнення та відкриття сучасної молоді. Матеріали міжнародної науково-практичної конференції (2021). – С. 62 – 64.

2. Мякшин О. Д. Модифікація архітектури згорткової нейронної мережі OpenFace для системи розпізнавання облич / О. Д. Мякшин, П. О. Ануфрієв // «ТАК»: телекомунікації, автоматика, комп'ютерно-інтегровані технології (2021).

3. Miakshyn A. Face recognition technology improving using convolutional neural networks / A. Miakshyn, P. Anufriev, Y. Bashkov // Advanced Trends in Information Theory (ATIT). Proc. of 2021 IEEE 3rd International Conference (2021).

ABSTRACT

Miakshyn A. Face recognition technology improving using convolutional neural networks / Graduation qualification work for obtaining an educational degree “Master” in specialty 122 “Computer Science“. – DonNTU, Pokrovsk, 2021.

The work is devoted to the study of methods of construction and algorithms of convolutional neural networks in the context of facial recognition to identify ways to improve existing models.

The purpose of the study is to find and implement approaches to improve the face recognition system, which allows to recognize objects with increased recognition accuracy without slowing down.

The object of research in this work is the process of face recognition using convolutional neural networks.

The subject of research is the architecture and capabilities of convolutional neural networks to solve the problem of face recognition in two-dimensional images.

The scientific novelty is to obtain a more efficient architecture of the convolutional neural network, which improves the accuracy of face recognition.

The practical importance is the accumulation of recommendations for the development and training of convolutional neural networks for use in face recognition systems based on the analysis of modern architectures and the principles of existing models.

Keywords: convolutional, neural network, research, recognition, detection, faces, deep learning, CNN, Python, TensorFlow, Keras.

Publisher’s publication list:

1. Miakshyn A. Architectures of convolutional neural networks for the problem of face recognition / A. Miakshyn, P. Anufriev // Scientific achievements and discovery of modern youth. Proceedings of the international scientific-practical conference (2021). – P. 62 – 64.

2. Miakshyn A. Modification of the architecture of the convolutional neural network OpenFace for face recognition system / A. Miakshyn, P. Anufriev // "TAC": telecommunications, automation, computer-integrated technologies (2021).

3. Miakshyn A. Face recognition technology improving using convolutional neural networks / A. Miakshyn, P. Anufriev, Y. Bashkov // Advanced Trends in Information Theory (ATIT). Proc. of 2021 IEEE 3rd International Conference (2021).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – База даних

ЗНМ – Згорткова нейронна мережа

ШІ – Штучний інтелект

GPU – Graphic process unit

CNN – Convolutional neural network

CPU – Central process unit

FN – False negative

FP – False positive

LFW – Labelled Faces in the Wild

NMS – Non max suppression (сколько раз приводится)

TN – True negative

TP – True positive

TPU – Tensor process unit

ВСТУП

З розвитком технологій і істотним збільшенням потужності обчислювальних систем, стали актуальними завдання автоматизації процесів і створення штучного інтелекту (ШІ). Головною метою ШІ є рішення задач, що виникають на практиці.

Однією з таких задач є задача розпізнавання обличь. Системи, що вирішують таке завдання можуть використовуватися для автоматизації ідентифікації людей на зображеннях, для верифікації та авторизації користувачів додатків або співробітників підприємств, а також для безлічі інших завдань.

Під таку задачу важко або навіть неможливо створити послідовний алгоритм, з яким звикли працювати комп’ютери, тому для її вирішення використовуються нейронні мережі. Вони є одним з напрямків ШІ, що ефективно справляються з подібними завданнями, оскільки їх реалізація заснована на розумовій діяльності людини, що дає стійкість перед шумами і спотвореннями в даних.

На сьогоднішній день завдання розпізнавання обличь з використанням нейронних мереж активно розвивається. Нейронні мережі показують найкращі результати якості і швидкості розпізнавання, щороку пропонуються нові архітектури нейронних мереж, алгоритмів, їх модифікацій, які вдосконалюють ефективність роботи своїх попередників. Найбільш придатними для цієї задачі виявляються згорткові нейронні мережі (ЗНМ), що з самого початку були націлені на розпізнавання зображень [1, 2].

Враховуючи все це, тематикою даної роботи стало дослідження та удосконалення технології розпізнавання обличь за допомогою згорткових нейронних мереж.

Мета дослідження – пошук та реалізація підходів удосконалення системи розпізнавання обличь, які дозволяють розпізнавати об’єкти з підвищеною точністю розпізнавання без зниження швидкості.

Об’єкт дослідження даної роботи – процес розпізнавання обличь за допомогою згорткових нейронних мереж.

Предмет дослідження – архітектури та можливості згорткових нейронних мереж для вирішення задачі розпізнавання обличь на двовимірних зображеннях.

Задачами даної роботи є дослідження архітектури та принципів роботи згорткових нейронних мереж, дослідження архітектури системи розпізнавання обличь, проведення аналізу моделей сучасних ефективних ЗНМ для різних цілей, вилучення підходів та алгоритмів, які в них використовуються. Після чого потрібно вибрати архітектуру і модифікації для власної системи розпізнавання обличь на основі сучасного стану досягнень в цій області, навчити нейронну мережу та провести оцінку ефективності її роботи та якості розпізнавання.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД ПРИЦНИПІВ РОЗПІЗНАВАННЯ ОБЛИЧЬ ЗА ДОПОМОГОЮ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ

1.1 Математичний опис задачі розпізнавання обличь

Задача розпізнавання обличь є однією з задач розпізнавання образів, яку можна представити наступним чином. Нехай на вхід системи розпізнавання надходить деякий об’єкт *x*, який необхідно розпізнати. Цей об’єкт, підлеглий до розпізнавання, називають образом, що у контексті задачі розпізнавання обличь безпосередньо є його зображенням. При цьому вважається, що *x* належить до одного з класів з множини всіх класів $A = \{a_1, \ldots, a_n\}$ – кожен клас відповідає ідентифікатору однієї людини, наприклад, її імені. Передбачається, що ці класи є непересічними.

При завданні будь-якого *x* через функцію *f(x)*, яка являє собою систему або алгоритм розпізнавання обличь, на виході потрібно отримати номер або значення відповідного елемента множини *A*, якому належить вхідне зображення *x*, з вказівкою ступеню достовірності результату розпізнавання, або отримати інформацію, що вхідний образ не належить жодному з класів [3].

1.2 Нейронна мережа як сучасний метод машинного навчання

Спочатку ідея нейронних мереж замислювалася як модель роботи людського мозку. Відомо, що в мозку людини присутні нейрони, які весь час передають один одному інформацію у вигляді електричних сигналів по зв’язкам – синапсах. При цьому зв’язки є не між усіма нейронами і відрізняються за силою. Коли людина вчиться чомусь новому, між нейронами, які відповідають за відповідні навички, утворюються нові зв’язки або стають сильнішими, якщо вони вже є. Чим сильніше зв’язок між нейронами, тим простіше інформація передається від одного іншому і, отже, людині з кожним разом простіше виконувати дії, за які відповідають ці нейрони. Наприклад,

дуже міцні зв’язки між тими нейронами, що відповідають за навички, набутими ще в дитинстві, такими, як здатність ходити або говорити. У процесі життя людини зв’язки між нейронами то утворюються і стають сильнішими, то стають слабкішими в залежності від того, набуває людина нових навичок або втрачає їх. Подібна ідея і лягла в основу першої моделі штучних нейронних мереж – персептрону, що також називають повнозв’язною нейронною мережею [4].

Пізніше, на основі відкриттів в області зорової кори людини, було розроблено модель згорткової нейронної мережі, націленої саме на розпізнавання образів [2]. Зазвичай ЗНМ включають в себе повнозв’язну мережу, тому далі розглянуто алгоритм роботи персептрону, а потім згорткової мережі.

1.2.1 Опис архітектури повнозв’язної нейронної мережі

Персептрон складається з послідовних шарів, які, в свою чергу, складаються з нейронів. Кожен нейрон попереднього шару пов’язаний з кожним нейроном наступного шару. Саме через це такий вид нейронних мереж і називається також повнозв’язною (рис. 1.1). При цьому зв’язки мають певні числові ваги, які і показують наскільки міцний зв’язок між нейронами. Чим більше вага у зв’язку, тим більший вплив надають нейрони один на одного.

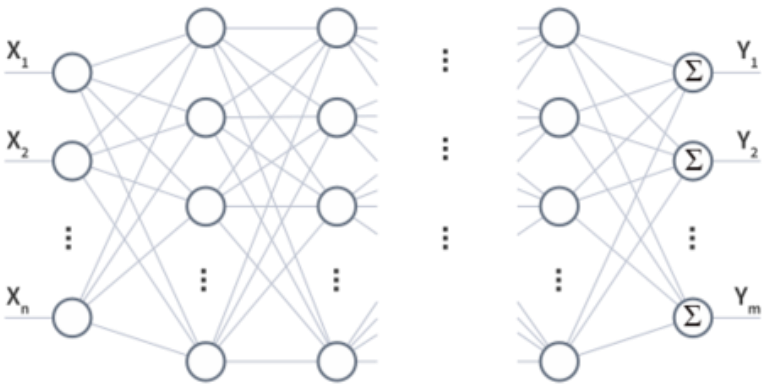


Рисунок 1.1 – Загальний вигляд структури персептрону

Джерело: [5]

З точки зору використання, повнозв’язну нейронну мережу можна представити у вигляді чорного ящика. Перший шар називається вхідним, останній вихідним, а шари між ними – прихованими. На вхід кожного нейрону вхідного шару подається якісь числові значення X і на виходах кожного нейрону вихідного шару можна отримати нові значення Y.

Нейрони першого шару, по суті, просто приймають вхідні значення, які задаються мережі. Нейрони наступних шарів мають більш складну структуру, наведену на рисунку 1.2. У її структурі можна виділити два основних елементи: суматор і функція активації.

Суматор приймає значення з усіх нейронів попереднього шару, з якими він пов’язаний, перемножує їх на ваги відповідних зв’язків і підсумовує їх. При цьому, до суми також додається порогове значення, яке також називають зміщенням. Його можна уявити як вагу додаткового нейрона з фіксованим значенням +1. Воно робить нейрон більш гнучким, додаючи ефект афінного перетворення.

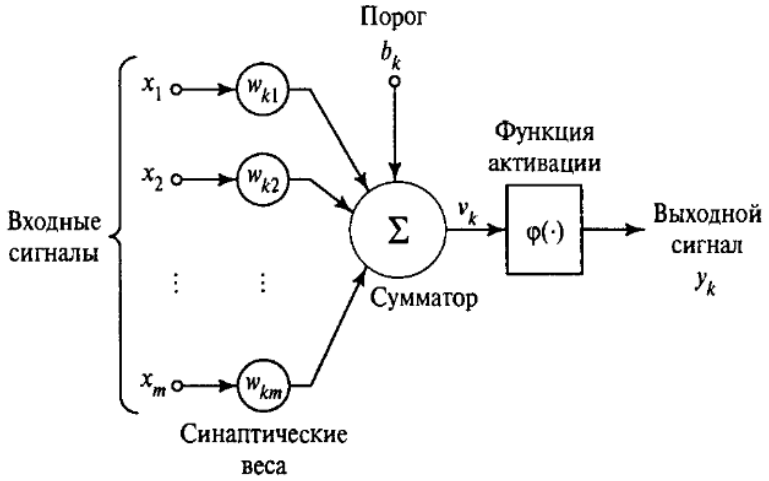


Рисунок 1.2 – Формальна модель нейрону
Джерело: [4]

Функція активації згладжує значення на суматорі, обмежуючи значення вихідного сигналу. Зазвичай, значення на виходах нейронів лежать в інтервалі [0, 1] або [-1, 1] в залежності від використовуваної функції активації. Математично, розрахунок вихідного значення наведено у формулі (1.1).

$$f(w_1x_1 + w_2x_2 + \cdots + w_mx_m + b), \quad (1.1)$$

де f – використовувана функція активації;

w_m – вага між поточним нейроном і нейроном m попереднього шару;

x_m – значення на нейроні m попереднього шару;

b – зміщення на поточному нейроні.

Для всієї мережі в цілому або для кожного шару окремо може бути обрана єдина функція активації, але частіше за все для прихованих шарів вибирається одна функція, а для останнього шару її вибирають інший залежно від завдання.

Існує безліч функцій активації, одними з найбільш поширених є логістична функція або сигмоїда, гіперболічний тангенс, ReLU, PReLU. На питання, яку функцію вибрати немає однозначної відповіді, зазвичай вона підбирається в ході експериментів або з огляду на вже відомий досвід.

Сигмоїда (див. формулу (1.2)) застосовується в мережах з невеликою кількістю прихованих шарів (як правило, не більше шести). Це викликано тим, що у цих функцій значення зазвичай менше одиниці і при перемножуванні безлічі значень нейронів з такою функцією активації виходять дуже маленькі значення, через що нейронна мережа навчається повільніше - по суті більше навчаються останні шари, так як на них в такому випадку помилка впливає більше, а перші майже не змінюються.

$$f(x) = \frac{1}{1+e^{-x}} \quad (1.2)$$

Гіперболічний тангенс (див. формулу (1.3)) за властивостями і застосування збігається з сигмоїдою, але відрізняється від неї тим, що має діапазон значень (-1; 1). Він буває корисний в задачах, де потрібно на виході мережі також отримувати негативні значення.

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \quad (1.3)$$

Для мереж з великою кількістю шарів на прихованих шарах використовують функцію активації ReLU – для таких випадків вона вважається незамінною (див. формулу (1.4)). Але при малій кількості шарів, її також використовують замість інших функцій активації. Раніше, замість цієї функції, використовували гіперболічний тангенс, але в певний момент виявили, що ReLU дає більш кращі результати, завдяки чому вона стала більш розповсюдженою [6].

$$f(x) = \begin{cases} 0, x < 0 \\ x, x \geq 0 \end{cases} \quad (1.4)$$

Іноді, кращі результати в порівнянні з ReLU дає її модифікація PReLU (див. формулу (1.5)). В цю функцію додається параметр, який навчається разом з усіма іншими параметрами мережі.

$$f(x) = \begin{cases} \alpha x, x < 0 \\ x, x \geq 0 \end{cases} \quad (1.5)$$

У задачах класифікації при непересічних класах використовується функція softmax на вихідному шарі (див. формулу (1.6)). Вона переводить значення вихідних нейронів в значення, ближче до імовірнісних, враховуючи значення на всіх вихідних нейронах.

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}, i = 1, \dots, m \quad (1.6)$$

де $\vec{x}$ – вихідний вектор нейронної мережі;

m – кількість значень в вихідному векторі.

У завданнях регресії, тобто в задачах, де вихідне число це і є відповідь і його не потрібно якось інтерпретувати, як, наприклад, в задачах класифікації,

на вихідному шарі може використовуватися лінійна функція активації (див. формулу (1.7)).

$$f(x) = x \tag{1.7}$$

Таким чином, проходячи по шарам і розраховуючи значення на кожному нейроні, нейронна мережа видає результат на останньому шарі. Такий процес називається прямим ходом або процесом прямого поширення.

Навчання нейронної мережі є завданням оптимізації, в якій підбираються параметри ваг і зміщень для досягнення бажаного результату. За допомогою заздалегідь обраної функції помилки (критерію якості) обчислюються помилки на нейронах останнього шару L і за допомогою методу градієнтного спуску виконується пошук нового значення ваг зв'язків, що ведуть до відповідних нейронів [7].

Щоб знайти величину зміни для даної ваги, використовується похідна від функції помилки по цій вазі, яка показує нахил, який вказує в напрямку зменшення помилки. Саме цей нахил і дає напрямок, в якому потрібно рухатися, щоб досягти локальних мінімумів. А для того, щоб підштовхнути вагу в знайденому напрямку потрібно використовувати коефіцієнт швидкості навчання. Його можна розглядати як крок зміни ваги, який дорівнює:

$$\Delta_{ij}^{(l)} = -\eta * g_i^{(l)} * y_j^{(l-1)}, \tag{1.8}$$

де η – коефіцієнт швидкості навчання ($0 < \eta < 1$);

g – градієнт для нейрону i поточного шару;

y – сигнал нейрону j з попереднього шару, з яким пов'язаний нейрон i.

Градієнт залежить від функції помилки, яка, в свою чергу, залежить від помилок сигналів на виходах нейронів. Помилки на нейронах вихідного шару є різницею між отриманими і очікуваним сигналами, тоді градієнт для нейрону вихідного шару дорівнює формулі 1.9.

$$g_j = f'(S_j) * (y_j - d_j), \quad (1.9)$$

де g_j – градієнт нейрону j;

S_j – значення суматору цього нейрону;

$f'(x)$ – похідна функції активації;

y_j – отриманий сигнал на нейроні;

d_j – очікуваний сигнал на нейроні.

Для прихованих шарів неможливо безпосередньо задати критерій якості, тому помилки на нейронах шару $L-1$ обчислюються як зважена сума градієнтів наступного шару L і далі, таким же чином, коригуються ваги нейронів цього шару. В такому випадку градієнт виглядає наступним чином:

$$g_j^{(l)} = -\eta * f'\left(S_j^{(l)}\right) * \sum_{q=1}^{N^{(l+1)}} w_{jq}^{(l+1)} * g_q^{(l+1)}, \quad (1.10)$$

де g_j – градієнт нейрону j прихованого шару l;

η – коефіцієнт швидкості навчання;

S_j – значення суматору нейрону j;

$f'(x)$ – похідна функції активації;

N – кількість нейронів у наступному шарі;

w_{jq} – вага між нейроном j шару l та нейроном q наступного шару;

g_q – градієнт нейрону q наступного шару.

Залежно від використовуваної функції активації, її похідна буде відрізнятися.

При навчанні нейронної мережі все перераховане вище виконується для всіх ваг і зміщень, перебираючи всі приклади навчання. Можна перебирати ті ж приклади багато разів, а як сигнал зупинки вважати досягнення необхідної точності помилки нейронної мережі або ліміту кількості епох – ітерацій по всім прикладам.

Функцію помилки у нейронній мережі, як і функцію активації, вибирають експериментальним шляхом або беруть до уваги вже відомий досвід використання в конкретних завданнях.

Для розпізнавання при класифікації двох класів використовуються хіндж (див. формулу (1.11)), квадратичний хіндж (див. формулу (1.12)) або бінарна крос-ентропія (див. формулу (1.13)). Важливо помітити, що бінарна крос-ентропія працює з вихідними значеннями сигмоїди, а категоріальна – з softmax (версія функції max, що залишає інформацію про немаксимальні значення) [8].

$$E = \frac{1}{N}\sum_{i=1}^{N} \max(1 - d_i y_i, 0)\,, \mathrm{y}_i \in \{-1,1\} \tag{1.11}$$

$$E = \frac{1}{N}\sum_{i=1}^{N} (\max(1 - d_i y_i, 0))^2\,, \mathrm{y}_i \in \{-1,1\} \tag{1.12}$$

$$E = -\frac{1}{N}\sum_{i=1}^{N} [d_i \log y_i + (1 - d_i)\log(1 - y_i)] \tag{1.13}$$

де E – помилка нейронної мережі;

N – кількість нейронів у вихідному шарі;

y_i – вихідне значення на i-му нейроні;

d_i – очікуване значення мережі на i-му нейроні.

При класифікації більше двох класів у якості функції помилки нейронної мережі використовується категоріальна крос-ентропія:

$$E = -\sum_{i=1}^{N} d_i \log(y_i) \tag{1.14}$$

У завданнях регресії використовуються середній квадрат помилок (див. формулу (1.15)), середня сума модулів помилок (див. формулу (1.16)), середній квадрат логарифмічних помилок (див. формулу (1.17)), середній абсолютний відсоток помилок (див. формулу (1.18)).

$$E = \frac{1}{N}\sum_{i=1}^{N} (d_i - y_i)^2 \tag{1.15}$$

$$E = \frac{1}{N}\sum_{i=1}^{N}|d_i - y_i| \quad (1.16)$$

$$E = \frac{1}{N}\sum_{i=1}^{N}(\log(d_i + 1) - \log(y_i + 1))^2 \quad (1.17)$$

$$E = \frac{1}{N}\sum_{i=1}^{N}\frac{|d_i - y_i|}{\max(|y_i|, \varepsilon)} \quad (1.18)$$

Проблемою середнього квадрата помилок є наявність безлічі локальних мінімумів, через що при навчанні більша ймовірність, що градієнтний спуск застрягне в одному з них, що призводить до гіршої якості розпізнавання. Також при використанні даного критерію якості помилка різко зростає, якщо існують значні, але рідкісні помилки, тому його варто використовувати, якщо такі помилки важливі.

При використанні середнього модуля помилок помилка зростає вже не так різко, якщо існують значні, але рідкісні помилки, а середній квадрат логарифмічних помилок ще більше зменшує вплив рідкісних значущих помилок. Середній абсолютний відсоток помилок дозволяє відійти від абсолютних значень до відсотків.

1.2.2 Опис архітектури згорткової нейронної мережі

Згорткові нейронні мережі складаються з шарів згортки і шарів пулінгу (pooling), які також називають шарами субдискретизації (рис. 1.3). Вхідними даними для нейронної мережі, в разі зображення, є 2D-тензор [9] яскравості його пікселів. У разі, якщо зображення кольорове, воно розбивається на канали RGB і подається вже три зображення [10].

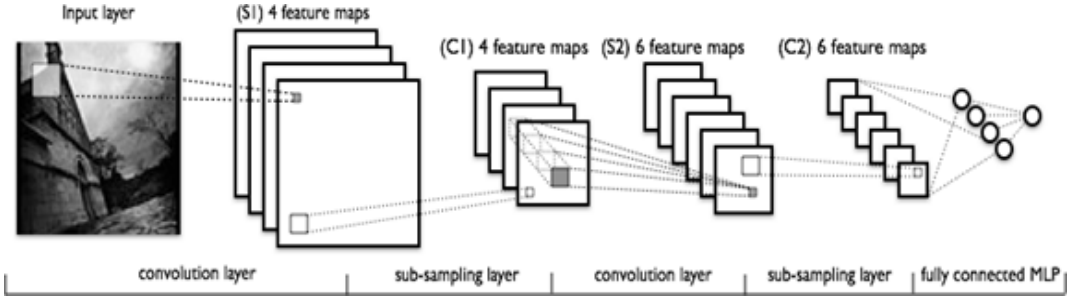


Рисунок 1.3 – Структура згорткової нейронної мережі

Джерело: [11]

Потім для кожного каналу виконується операція згортки за допомогою фільтра згортки (рис. 1.4). Фільтр виділяє певні ознаки на зображенні, формуючи нове зображення, яке називається картою ознак. Наприклад, виділяються горизонтальні або вертикальні лінії, вони стають більш яскравими, в порівнянні з іншими елементами на зображенні. При цьому, фільтрів може бути багато, кожен з яких виділяє на зображенні щось своє і формує свою карту ознак.

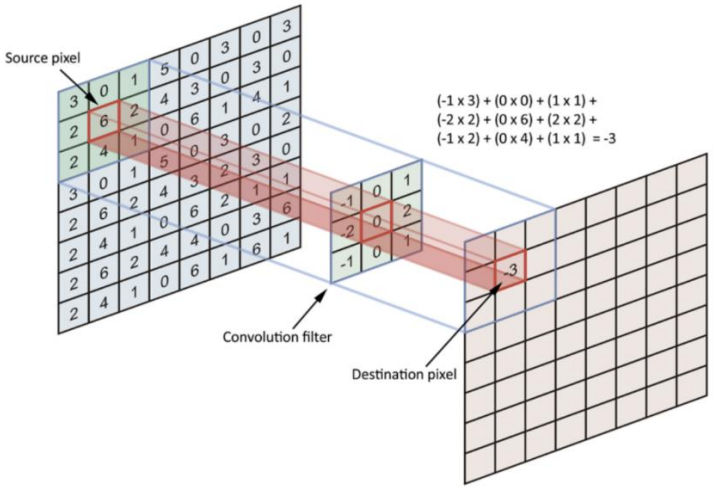


Рисунок 1.4 – Візуалізація процесу згортки
Джерело: [12]

У разі двовимірного тензора фільтр є матрицею невеликого розміру, яка містить вагові коефіцієнти. Під час виконання операції згортки, фільтр за допомогою методу ковзаючого вікна проходить по зображенню, перемножує елементи зображення на відповідні елементи фільтра, підсумовує їх, а також додає ще один ваговий коефіцієнт w_0 у якості зсуву. Це значення також пропускається через функцію активації і утворює нове значення пікселя на зображенні. Формальний вид розрахунку нового значення v пікселя за допомогою фільтра розміром 3x3 наведено у формулі (1.19).

$$v_{k,m} = f\left(\sum_{i=1}^{3}\sum_{j=1}^{3} x_{i+k,j+m} w_{ij} + w_0\right) \qquad (1.19)$$

де k, m – індекси пікселів на зображенні;

f – функція активації;

x – вихідне значення пікселю;

w_{ij} – відповідна вага у фільтрі згортки;

w_0 – зміщення на +1.

Розмір фільтра і крок, з яким він проходить по зображенню, є гіперпараметрами мережі, тобто параметрами, які вибираються заздалегідь як частина конфігурації і потім не змінюються. Кількість таких фільтрів не обмежена і також становить частину конфігурації мережі. Кожен фільтр формує свою карту ознак (канал), обчислюючи її з карт ознак попереднього шару. При чому, так як каналів на попередньому шарі може бути кілька, як у випадку з кольоровими зображеннями, для формування нової карти ознак цей фільтр проходить по кожному з вхідних каналів, а результат підсумовується.

Зазвичай для фільтра згортки вибирається крок, що дорівнює одиниці. Якщо вибирається крок більшого розміру, зображення ставатиме меншого розміру в залежності від кроку. Також важливо помітити, що після операції згортки в залежності від розміру фільтру, розмір зображення стає менше по краях, так як фільтр не може вийти за межі зображення. Якщо такий ефект потрібно прибрати, перед згорткою до зображення по краях додається відступ (padding) з нульових значень.

Після виконання згортки, отримані канали проходять через шар пулінгу. Він також виконується за допомогою ковзного вікна, проходячи по зображенню і з усіх елементів в області вікна обчислює єдине значення. Існує три типи пулінгу – max pooling, min pooling і average pooling. При max пулінгу з усіх елементів у вікні вибирається найбільший елемент, при min пулінгу – найменший, а при average – середній. Зазвичай для шарів пулінгу вибирається крок, рівний розміру вікна пулінгу, щоб зображення стискалося, залишаючи лише найбільш значущі елементи перед тим як перейти до обробки в наступних шарах мережі. Найпоширенішим типом пулінгу є max pooling, візуалізація процесу якого наведено на рисунку 1.5, а приклади зміни зображення при його використанні – на рисунку 1.6.

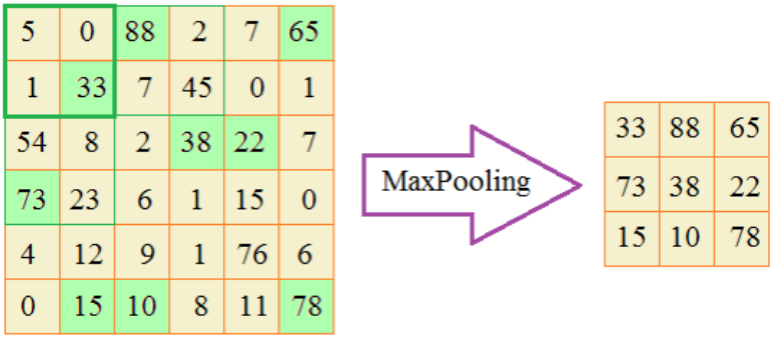


Рисунок 1.5 – Візуалізація процесу max pooling
Джерело: [13]

Рисунок 1.6 – Приклади зображення при використанні max pooling
Джерело: [13]

У згортковій мережі шари згортки і пулінгу зазвичай чергуються. Кожен наступний шар згортки виділяє все більш складні ознаки. Спочатку виділяються прості елементи зображення, а наступні шари виділяють все більш складні. В [14] проводилося дослідження, присвячене візуалізації роботи ЗНМ, виділяючи частини зображення, на яких мережа показує більшу чутливість. Приклади таких областей на зображенні в залежності від шару представлено на рисунку 1.7.

Рисунок 1.7 – Приклади областей, к котрим згорткова нейронна мережа більш чутлива в залежності від шару
Джерело: [14]

Як можна помітити, на першому шарі вона виділяє лінії, градієнти, елементи, пов’язані з кольором і відтінку. На другому шарі виділяються прості форми і текстури. На наступних шарах виділяються частини об’єктів, наприклад, колеса машин. На одному з останніх шарів мережа бачить вже цілі об’єкти.

Після того як зображення пройшло через шари згортки і пулінгу, отримані карти ознак виділяють найбільш значущі частини зображення в компактному вигляді. Далі вони можуть подаватися на вхід іншим алгоритмам, наприклад, для вирішення задачі класифікації або регресії. Зазвичай в ЗНМ після шарів згортки додається повнозв’язна нейронна мережа, яка і вирішує поставлене завдання.

Принцип навчання згорткових шарів нейронної мережі такий же, як і в персептроні. Але в даному випадку за допомогою методу градієнтного спуску оптимізуються вагові коефіцієнти в ядрах згортки.

1.2.3 Відомі проблеми глибокого навчання та методи їх подолання

При розробці глибоких нейронних мереж, що мають велику кількість шарів існує проблема загасання і вибуху градієнта. Для того, щоб пояснити звідки беруться такі ефекти, припустимо, що у нас є мережа з кількістю шарів, наприклад, більше ста, а також для спрощення будемо вважати, що функція активації – лінійна, а вектор зсувів дорівнює нулю. В такому випадку вихідний вектор мережі матиме загальний вигляд, наведений у формулі (1.20).

$$y = w^{[l]}w^{[l-1]}w^{[l-2]} \dots w^{[3]}w^{[2]}w^{[1]}x \tag{1.20}$$

де x – вхідний вектор мережі;

w – матриця ваг на шарі l.

Тоді, якщо значення елементів всіх матриць ваг трохи більше, ніж у одиничної матриці, значення будуть рости з кожним наступним шаром і в підсумку значення вихідного вектора y будуть дуже великими, викликаючи

вибух градієнта. А якщо значення матриць ваг будуть трохи менше одиниці, тоді *y* буде дуже маленькими, через що градієнт буде затухати. Оскільки навчання мережі залежить від градієнта, при таких ефектах мережа просто не зможе навчатися.

Тому, крім стандартних повнозв'язних шарів, а також шарів згортки, пулінгу, в глибоких нейронних мережах можуть використовуватися додаткові, комплексні види шарів, що комбінують в себе інші шари. У таких випадках такі шари можуть складаютися в цілі модулі. Одним з таких видів шарів є residual шари, що в перекладі – залишкові, які зазвичай і використовуються для вирішення проблеми, пов'язаної зі загасанням і вибухом градієнта.

Значення на нейронах residual шарів формуються так, як і на звичайному шарі, але з додаванням значень нейронів якогось попереднього шару перед застосуванням функції активації (див. формулу (1.21)).

$$a^{[l+2]} = f(w^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}) \tag{1.21}$$

де a – вектор значень на нейронах шару l;

w – матриця ваг на шарі l;

b – вектор зміщень на шарі l;

f – функція активації.

Таким чином збільшується вплив результату з попередніх шарів на наступні. Було виявлено, що це дозволяє робити нейронні мережі більш глибокими без втрати точності, наприклад, створювати мережі, що мають в собі більше ста або навіть тисячі шарів. При використанні residual шарів в мережі, така мережа називається ResNet, а групи шарів, що реалізує схему, наведену у формулі вище, називають residual модулем [15].

Також, у згорткових мережах іноді буває складно вибрати які параметри фільтра згортки використовувати, яку їх кількість вибрати, або може замість згортки взагалі використовувати шар пулінгу. Цю проблему вирішує ще один вид комплексних шарів – inception [16]. Їх ідея в тому, щоб з набору карт ознак,

отриманого після попередньої операції, генерувати нові карти ознак за різними параметрами і складати їх усіх в одну стопку. Дану схему ілюструє рисунок 1.8.

Таким чином з попереднього шару можуть виходити декілька відгалужень зі своєю чередою шарів зі своїми параметрами, а потім конкатенуватись в результуючий шар.

В такому випадку мережа під час навчання сама для себе вибирає які параметри і типи шарів краще використовувати. Для непотрібних або менш значних шарів мережа обнулить матриці ваг або зробить їх невеликими так, що вони будуть мало впливати на подальший результат. Головне в даному випадку, щоб розміри результуючих карт признаків збігалися.

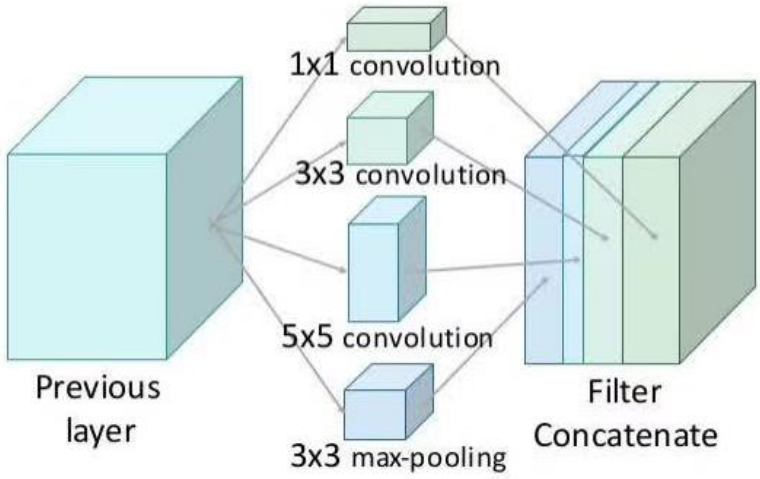


Рисунок 1.8 – Ілюстрація схеми inception модулю
Джерело: [17]

При навчанні у методах ШІ також існує проблема зміщення та дисперсії (bias and variance). У глибокому навчанні вона асоціюється з недонавчанням та перенавчанням мереж. Високе зміщення свідчить, що мережа неспроможна добре підлаштуватися під навчальну вибірку, викликаючи недонавчання, а висока дисперсія показує, що мережа надто підлаштувалася під навчальну вибірку, викликаючи перенавчання. Хорошим балансом є поєднання маленьких зміщень та дисперсії. Візуалізацію цих ефектів наведено на рисунку 1.9.

Рисунок 1.9 – Візуалізація проблеми зміщення та дисперсії
Авторський переклад на основі [18]

Точки кожному графіку представляють якісь навчальні дані, де X – вхідні дані для мережі, а Y – очікувані вихідні значення. При високому зміщенні видно, що багато значень X на лінії, яка утворюється пророкуваннями мережі на тренувальних даних в результаті навчання мають дуже велику відстань до істинно правильних Y. При високому значенні дисперсії лінія, що утворюється після навчання, проходить через всі навчальні приклади, тому на тестовому датасеті та при подальшій експлуатації мережі, де вхідні приклади матимуть трохи інші значення, сумарна помилка мережі матиме велике значення.

На третьому графіку видно, що поєднання низького зміщення та дисперсії дають досить хороший результат – мережа бачить саму закономірність між вхідними та вихідними значеннями, не підлаштовуючись під тренувальні дані, тим самим не перенавчаючись.

Зазвичай проблема зміщення та дисперсії вирішується за допомогою кращого підбору структури мережі та використання великих датасетів. Але можливі ситуації, коли з якихось причин немає можливості покращити структуру мережі або використовувати більший датасет. Тому для вирішення цієї проблеми використовується метод, званий регуляризацією. Існує два види регуляризації: L2 та L1 [19].

L2-регуляризація реалізується за допомогою додавання до функції помилки додаткового доданку таким чином, що утворюється помилка мережі, наведена у формулі (1.22).

$$E = \frac{1}{m}\sum_{i=1}^{m} L\left(d_i, y_i\right) + \frac{\lambda}{2m}\sum_{l=1}^{n}\left\|w^{[l]}\right\|_2^2 \quad (1.22)$$

де m – кількість навчальних прикладів;

L – використовувана функція помилки;

λ – гіперпараметр L2-регуляризації;

n – кількість шарів мережі, виключаючи вхідний шар;

w – матриця ваг шару l.

При цьому $\left\|w^{[l]}\right\|_2^2$ дорівнює виразу, наведеному в формулі (1.23).

$$\left\|w^{[l]}\right\|_2^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2 \quad (1.23)$$

де w – матриця ваг шару l;

n – кількість нейронів шару l.

Додатково до помилки також можна додавати відповідне складене зі зсувами b, але практика показує, що це не дає помітної різниці.

L1-регуляризація реалізується шляхом додавання доданку таким чином, що утворюється помилка мережі, наведена у формулі 1.24.

$$E(w, b) = \frac{1}{m}\sum_{i=1}^{m} L\left(d_i, y_i\right) + \frac{\lambda}{2m}\|w\| \quad (1.24)$$

L1-регуляризація дозволяє «стискати» матрицю ваг, додаючи до неї більше нулів. Оскільки нулі займають менше місця у пам’яті, ваги також займають менше місця. Однак, також практика показує, що це дозволяє зменшити займане місце зовсім небагато, тому набагато частіше використовується L2-регуляризація.

Параметр λ називається параметром регуляризації та підбирається експериментальним чином у межах від 0 до нескінченності. Цей параметр налаштовує чутливість до вхідних даних. Чим більше це значення, тим менш

чутливими до вхідних значень мережі стають її прогнози. Зазвичай оптимальним значенням цього параметра є значення, що дорівнює одиниці.

Беручи на увагу вищеописані методи і комбінуючи їх можна робити нейронні мережі дуже глибокими та більш ефективними за якістю розпізнавання, ніж звичайні мережі, які подібні методи не використовують.

1.3 Опис архітектури системи розпізнавання обличь

Система розпізнавання обличь складається з декількох етапів: детекція обличь на зображенні, виділення ознак кожного обличчя і вже їх безпосереднє розпізнавання [20].

Детекція обличчя, як і детекція будь-якого іншого об’єкта на зображенні, полягає в знаходженні координат його області, що його обмежує (bounding box) в пікселях. За допомогою нейронної мережі, ця задача може вирішуватися як завдання регресії, в якій потрібно знайти ці координати.

На перший погляд, можна навчити мережу, яка на вхід приймає зображення, а на виході видає чотири числа – координати центру або одного з кутів знайденої області, а також її висоту і ширину. Це найбільш очевидний спосіб і його недоліком є те, що він в кращому випадку зможе знаходити тільки один об’єкт. На різних зображеннях може бути різна кількість осіб, а архітектура мережі задається спочатку і не змінюється, тому потрібен інший підхід.

Таким підходом, що в тому чи іншому вигляді використовується у всіх мережах детекції, є реалізація детекції через класифікацію з використанням ковзаючого вікна [21]. Воно проходить по зображенню і класифікує те, що знаходиться в області вікна. Крок і розмір ковзаючого вікна може вибиратися будь-яким, часто прохід по зображенню виконується по кілька разів, з різним розміром вікна або зображення.

Така ідея працює добре в принципі, однак вона може містити величезну кількість областей, кожну з яких необхідно класифікувати, а тому це працює

дуже повільно. Для вирішення такої проблеми виділяють два основних типи методів: двоетапні і одноетапні [22].

Двоетапні методи на першому етапі за допомогою будь-якого допоміжного алгоритму знаходять регіони інтересу, де можуть знаходитись об’єкти, а потім на другому етапі вони перевіряються класифікатором з локалізацією. Локалізація уточнює координати об’єкта в області. Існує безліч алгоритмів знаходження регіонів інтересу [23], порівняння деяких з яких наведено на рисунку 1.10.

Method	Approach	Outputs Segments	Outputs Score	Control #proposals	Time (sec.)	Repea-tability	Recall Results	Detection Results
Bing	Window scoring		✓	✓	0.2	⋆⋆⋆	⋆	·
CPMC	Grouping	✓	✓	✓	250	-	⋆⋆	⋆
EdgeBoxes	Window scoring		✓	✓	0.3	⋆⋆	⋆⋆⋆	⋆⋆⋆
Endres	Grouping	✓	✓	✓	100	-	⋆⋆⋆	⋆⋆
Geodesic	Grouping	✓		✓	1	⋆	⋆⋆⋆	⋆⋆
MCG	Grouping	✓	✓	✓	30	⋆	⋆⋆⋆	⋆⋆⋆
Objectness	Window scoring		✓	✓	3	·	⋆	·
Rahtu	Window scoring		✓	✓	3	·	·	⋆
RandomizedPrim’s	Grouping	✓		✓	1	⋆	⋆	⋆⋆
Rantalankila	Grouping	✓		✓	10	⋆⋆	·	⋆⋆
Rigor	Grouping	✓		✓	10	⋆	⋆⋆	⋆⋆
SelectiveSearch	Grouping	✓	✓	✓	10	⋆⋆	⋆⋆⋆	⋆⋆⋆

Рисунок 1.10 – Методи пропозиції регіонів інтересу
Джерело: [23]

Зазвичай використовується метод селективного пошуку Selective Search або метод Edge Boxes, оскільки вони працюють досить швидко і дають досить добрі результати детекції. Таким чином зменшується кількість областей для перевірки і прискорюється загальна швидкість роботи системи.

Одноетапні методи ґрунтуються на використанні згорткової реалізації ковзаючих вікон за допомогою згорткових нейронних мереж [24]. Її суть в тому, що після проходу зображення по шарах згортки, кожен піксель на результуючої карті ознак відповідає своїй області пікселів на оригінальному зображенні (рис. 1.11).

Згорткова нейронна мережа виконує всі ті ж лінійні перетворення, а значить можна навчити її виконувати завдання регресії таким чином, щоб кожна з останніх карт ознак відповідала певному значенню, як вихідні нейрони в повнозв’язній мережі. Так, перша і друга карти ознак можуть

визначати координату центру області, а третя і четверта – її ширину і висоту. Додатково до координат області в такому методі додається ще значення, що визначає ймовірність або впевненість мережі в тому, що в даній області є шуканий об’єкт. Так як згортка сама по собі реалізується як ковзаючі вікна, то такий механізм уже закладений в згорткові мережі, що дає великий приріст в швидкості.

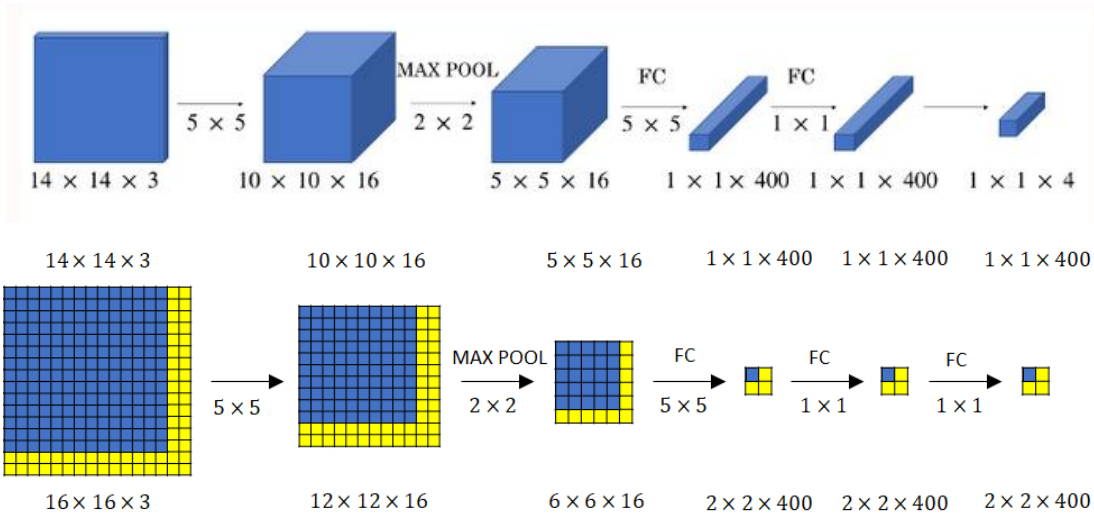


Рисунок 1.11 – Приклад та візуалізація конволюційного підходу детекції об’єктів за допомогою згорткових нейронних мереж

Джерело: [24]

На практиці після виконання класифікації всіх необхідних областей на зображенні нейронна мережа може виділити велику кількість таких прямокутних областей, які можуть перетинатися одна з одною і вказувати на одинакові об’єкти. Але зрештою потрібно визначити лише одну область на одне обличчя. Тому після визначення областей, які містять обличчя людей, застосовується метод придушення областей, з упевненістю присутності обличчя, що дорівнює не максимальній серед усіх знайдених областей. Цей метод має назву Non Maximum Suppression (NMS) [25]. Його алгоритм у тому, що з усіх детектованих областей вибирається лише та, що має максимальний коефіцієнт впевненості, виданий мережею. Але потрібно також враховувати, що на зображенні можуть бути розташовані відразу кілька об’єктів, що

детектуються, при чому відстань між ними буде невеликою, а тоді потрібно розрізняти знайдені області по тому, до кого вони відносяться. Для цього перед вибором області з максимальною впевненістю кожна область порівнюється з усіма іншими і для кожних двох областей обчислюється значення метрики Intersection Over Union (IOU). Ця метрика використовується для порівняння, наскільки області перетинаються одна з одною. Вона розраховується як область перетину, поділена на об'єднану область (див. формулу (1.25)). Більше значення показує більшу міру перетину, причому значення, рівне одиниці, – області повністю відповідають одне одному, а значення, рівне нулю, – області взагалі перетинаються.

$$IOU = \begin{cases} \frac{A \cap B}{A \cup B}, \text{якщо } A \text{ і } B \text{ перетинаються} \\ 0, \text{інакше} \end{cases} \tag{1.25}$$

де *A, B* – прямокутні області, для яких необхідно розрахувати IOU.

Потім у порівнянні ступеня перетину всіх областей вони кластеризуються з допомогою введення порогового значення. Воно є гіперпараметром і підбирається вручну таким чином, щоб загальна точність детекції відповідала найкращим показникам. Якщо ступінь перетину між двома областями більше за поріг, вважається, що вони вказують на той самий об'єкт, тому групуються разом у свій кластер, звідки вже потім буде обрана область з найбільшою впевненістю. Формальний опис вибору знайденої області серед множини областей методом NMS наведено у формулі (1.26).

$$s_i = \begin{cases} s_i, \text{якщо } IOU(M, b_i) < N \\ 0, \text{якщо } IOU(M, b_i) > N \end{cases} \tag{1.26}$$

де s_i – впевненість кожної області;

M – область з найбільшою впевненістю на поточній ітерації;

b_i – одна з інших областей, з якою порівнюється поточна;

N – порогове значення.

Крім цього методу також існують інші, які є його продовженням з певними модифікаціями для досягнення кращих показників точності детекції, такі як Soft NMS, Softer NMS, IOU-Guided NMS, Adaptive NMS, DIOU-NMS, кожен з яких має свої переваги та недоліки [26].

Після детекції обличь на зображенні для кожного обличчя виділяються ознаки, що унікально характеризують його в стислому вигляді. Такі ознаки можуть представлятися у вигляді вектора чисел. При використанні глибокого навчання, ознаки виділяються самими нейронними мережами. ЗНМ виділяють ознаки у вигляді результуючих карт ознак зображення, а якщо в якості другої частини згорткової мережі додається персептрон, то вектором ознак є вихідний вектор мережі.

Безпосереднє розпізнавання обличь в свою чергу, ділиться на дві категорії – ідентифікація і верифікація. Ідентифікацією обличчя є процес порівняння одного обличчя з усіма розміченими зображеннями обличь, які містяться в базі даних (БД), з метою знайти найбільш схоже або прийняти рішення про те, що такого обличчя в базі даних немає. Під розміченими зображеннями мається на увазі, що разом із зображенням обличчя в БД зберігається мітка, що ідентифікує людину, наприклад, його ім’я. Верифікація виконується тільки між двома зображеннями обличь з метою прийняти рішення – одна і та ж це людина, чи ні.

Для вирішення завдання ідентифікації і верифікації, вектори ознак, отримані з двох зображень, порівнюються між собою. Це часто виконується за допомогою сіамських нейронних мереж [27]. Сіамські нейронні мережі складаються з двох однакових нейронних мереж, як за структурою, так і за навченими вагами, тобто, по суті, це два екземпляри однієї і тієї ж мережі. В контексті задачі розпізнавання зображень, на вхід кожної з мереж подається будь-яке зображення, а на виході кожна мережа видає вектор чисел, який в стислому вигляді представляє це зображення. При чому для кожної мережі зображення задається своє, і завдання всієї структури сіамської нейронної мережі – будь-яким чином порівняти вектори на виходах двох її мереж і

зробити висновок про те, наскільки вони схожі. Для безпосереднього прийняття рішення про схожість використовується додатковий пороговий гіперпараметр.

Якщо різниця між векторами менше заданого порогу, значить можна зробити висновок про те, що на зображенні представлений один і той же об’єкт або об’єкт одного і того ж класу. Для ідентифікації і верифікації обличь в якості гіперпараметра встановлюється граничне значення схожості, при якому буде вважатися, що на зображеннях знаходиться одна і та ж людина.

Саму мережу, що кодує зображення в вектори ознак, можна навчати двома способами – навчати безпосередньо метрику схожості векторів або виділяти ознаки через навчання класифікатора.

Під навчанням метрики схожості мається на увазі, що мережу навчають таким чином, щоб вектори ознак для зображень обличь однієї і тієї ж людини були близькі один до одного по відстані або куту, а для різних людей – досить різнилися одна від одної так, щоб можна було з упевненістю прийняти рішення про те, що ці вектори не схожі.

Суть методу виділення векторів ознак із зображень обличь людей через навчання класифікатора полягає в тому, що навчають нейронну мережу, що класифікує зображення обличчя між фіксованим набором людей, після чого просто видаляють останній шар, що класифікує. Вся нейронна мережа, що залишилася, вже вміє виділяти ознаки [28].

При такому підході головним завданням стає правильний вибір функції помилки. Справа в тому, що зазвичай, для класифікації використовується помилка Softmax, але в контексті розпізнавання осіб вона має великий недолік – дана функція помилки не дає чіткого поділу об’єктів, що класифікуються, між їх класами.

Тому, якщо об’єкт одного класу має велику схожість з об’єктами іншого класу, якщо візуалізувати цю задачу як задачу кластеризації, то він буде знаходиться десь на краю кластера, в результаті чого мережа може легко віднести об’єкт, що класифікується, до невірного класу.

1.4 Висновки за розділом

В даному розділі було визначено математичний опис задачі розпізнавання облич як образів, поняття і ідеї нейронних мереж. Далі було розглянуто архітектуру, принципи та процес роботи повнозв’язної та згорткової нейронних мереж. У цьому контексті було досліджено алгоритми їх роботи, деякі з існуючих функцій активації і помилок, а також для яких завдань які функції підходять найкраще.

Додатково було досліджено відомі проблеми, що виникають при глубокому навчанні та методи їх подолання, а саме використання комплексних архітектур шарів – residual та inception, їх ідею і алгоритм роботи, а також методи для досягнення балансу між недонавчанням та перенавчанням мережі. Окрім цього досліджено архітектуру системи розпізнавання облич і принципи роботи кожного з її етапів.

РОЗДІЛ 2
АНАЛІЗ СУЧАСНИХ МОДЕЛЕЙ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ ДЛЯ РОЗПІЗНАВАННЯ ОБЛИЧЬ, ВИБІР МОДИФІКАЦІЇ

2.1 Сучасні моделі згорткових нейронних мереж, їх застосування для різних задачі розпізнавання

Моделі сучасних нейронних мереж по задачі застосування діляться на мережі для детекції та мережі для ідентифікації/верифікації. Спочатку розглянемо мережі для детекції об’єктів. З них можна виділити мережі сім’ї R-CNN (R-CNN, Fast R-CNN, Faster R-CNN), мережі YOLO і MTCNN.

Мережі сім’ї R-CNN відносяться до двохетапних методів. Перша модель R-CNN [29] виконує детекцію через класифікацію областей на зображенні, але для оптимізації швидкості своєї роботи, вона спочатку знаходить тільки ті області для подальшої перевірки, в яких найбільш ймовірно знаходиться об’єкт. Алгоритм роботи цієї моделі наведено на рисунку 2.1.

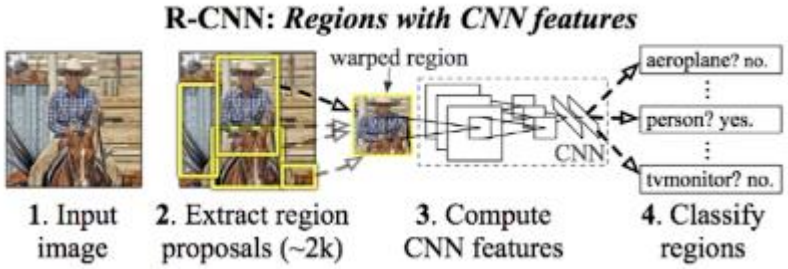


Рисунок 2.1 – Алгоритм роботи R-CNN
Джерело: [29]

Регіони інтересу в цій мережі знаходяться за допомогою методу селективного пошуку [30]. Його суть полягає в тому, щоб виділяти потенційні регіони на зображенні, пов’язаних один з одним за кольорами, відтінками, текстурами, формами. Запропоновані регіони інтересу потім подаються згортковій мережі, яка виділяє ознаки на зображенні, після чого виконується

їх класифікація за допомогою методу опорних векторів SVM [31], а також регресія координат області, уточнюючи місце розташування об’єкта.

Проблемою даної моделі є те, що в ній доводиться виділяти ознаки, класифікувати і виконувати регресію для кожного регіону інтересу окремо, що уповільнює мережу. Також, класифікація і регресія виконується різними моделями, які потрібно навчати окремо. Це збільшує час навчання і розмірність необхідного сховища даних для параметрів кожного з методів.

Для вирішення даної проблеми, було запропоновано мережу Fast R-CNN, яка є подальшим розвитком R-CNN [32]. Вона працює швидше за рахунок того, що тепер замість того, щоб пропускати кожен регіон інтересу через згорткову мережу окремо, мережі задається все зображення цілком, а запропоновані області просто накладаються на результуючі карти ознак. Також в цій моделі методи класифікації і регресії об’єднані в одну повнозв’язну мережу з двома виходами – для класифікації та регресії. Архітектура мережі Fast R-CNN зображена на рисунку 2.2.

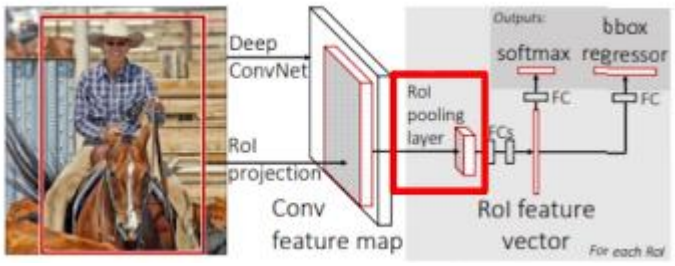


Рисунок 2.2 – Архітектура мережі Fast R-CNN
Джерело: [32]

Надалі мережа Fast R-CNN розвивається в Faster R-CNN [33]. Її поліпшення полягає в тому, що вона повністю відмовляється від методу селективного пошуку для знаходження регіонів інтересу і замість неї використовує згорткову версію ковзного вікна. При чому, для пошуку регіонів інтересу і класифікації об’єктів використовуються окремі згорткові мережі.

Мережа YOLO [34] відноситься до одноетапних методів. Вона дуже схожа на Faster R-CNN тим, що також використовує згорткову версію для

детекції, але YOLO знаходить і класифікує всі об’єкти на зображенні за один прохід по єдиної нейронної мережі.

При згортковому підході ковзного вікна, зображення розбивається на квадратні області, в межах яких мережа повинна знайти центр об’єктів, що детектуються. Але, оскільки, об’єкти можуть перекриватися один одним, центри відразу декількох об’єктів можуть знаходиться в одній і тій же області, що перевіряється, тому мережі, які використовують даний підхід при регресії використовують кілька обмежуючих рамок об’єктів. Вони називаються якорями, оскільки кожна область відповідає за свій конкретний клас об’єкта, що детектується. Такий якірний підхід також реалізований в мережах Faster R-CNN і YOLO.

Ще одним з підходів для детекції об’єктів є каскадний підхід. При такому підході мережі шикуються в каскад, при якому вихід однієї мережі є входом для наступного. При чому, кожна наступна мережа уточнює результат попередньої, оскільки навчається на даних, яка не змогла правильно розпізнати попередня. Прикладом каскадного підходу є мережа MTCNN [35] для детекції обличь. Вона складається з трьох послідовно пов’язаних мереж – P-Net (Proposal Network), R-Net (Refine Network) і O-Net (Output Network). Архітектуру цих мереж представлено на рисунку 2.3.

В роботі MTCNN, зображення спочатку змінюється в розмірах, створюючи так звану піраміду зображень (image pyramid). Потім кожне з цих зображень подається на вхід першій мережі P-Net, з якого вона виконує регресію координат обмежуючих рамок обличь. Знайдені області передаються в мережу R-Net, яка відкидає ті, де обличь точно немає, і передає свої результати мережі O-Net, яка залишає лише одну обмежуючу рамку, а також додатково знаходить п’ять точок на обличчі. Це корисно для вирівнювання обличь за цими точкам, щоб потім здавати їх подальшим більш комплексним системам розпізнавання.

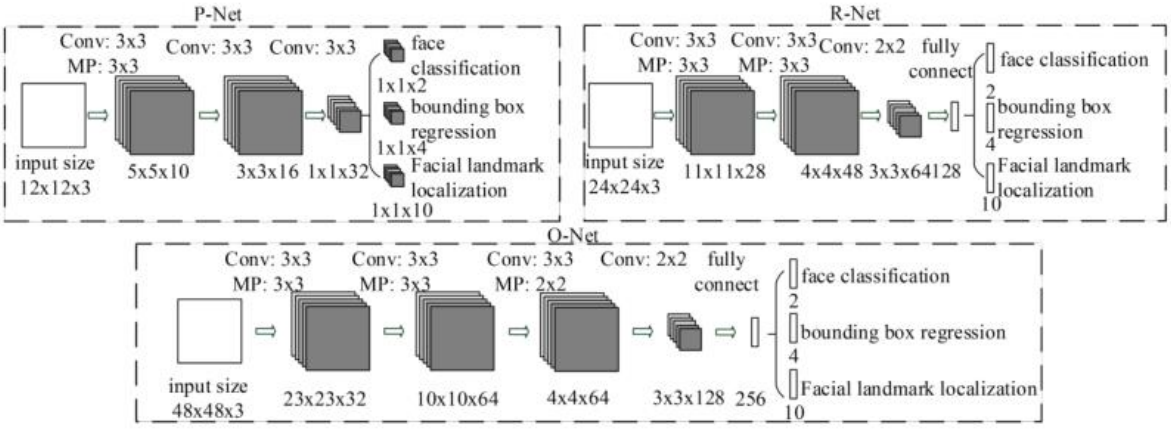


Рисунок 2.3 – Архітектура мережі MTCNN
Джерело: [35]

З огляду на все вищесказане з приводу мереж для детекції, також варто відзначити, що існує залежність між швидкістю роботи нейронної мережі і точністю її розпізнавання. В [36] виконувалося дослідження для того, щоб порівняти одноетапні і двоетапні методи детекції на прикладі деякого набору існуючих мереж для детекції, що використовують різні методи. Все це проводилося на наборі даних COCO [37], що містить 330 тисяч зображень з півтора мільйона об’єктами, кожен з яких відноситься до однієї з 80 категорій. Цей набір був розроблений спеціально для детекції та сегментації об’єктів, які входять до складу представлених в наборі категорій.

Результати дослідження можна побачити на графіку, наведеному на рисунку 2.4. Вісь X позначає час, за яке мережа виконує детекцію в мілісекундах, а вісь Y – середню точністю детекції в процентах. У контексті даного питання детекції за допомогою одноетапних і двоетапних методів, на малюнку головне значення має те, що мережі, які використовують одноетапні методи, працюють швидше, але точність у них гірше, ніж у двоетапних методів, і навпаки.

Мережа RetinaNet, яка була розроблена в результаті даного дослідження, використовує нову функцію помилки, яка називається Focal Loss (див. формулу (2.1)). Це дало мережі можливість використовувати одноетапні методи, щоб бути такою ж швидкою, але при цьому точність мережі не гірше, ніж у двоетапних мереж, таких як Faster R-CNN.

$$FL(p,y) = -(1-p_t)^{\gamma}\log(p_t), p_t = \begin{cases} p, y = 1 \\ 1-p, \text{інакше} \end{cases} \quad (2.1)$$

де p – передбачений мережею результат;

y – очікуваний результат;

γ – гіперпараметр згладжування, чим більше значення, тим більш чутливою буде помилка до прикладів, складних для розпізнавання.

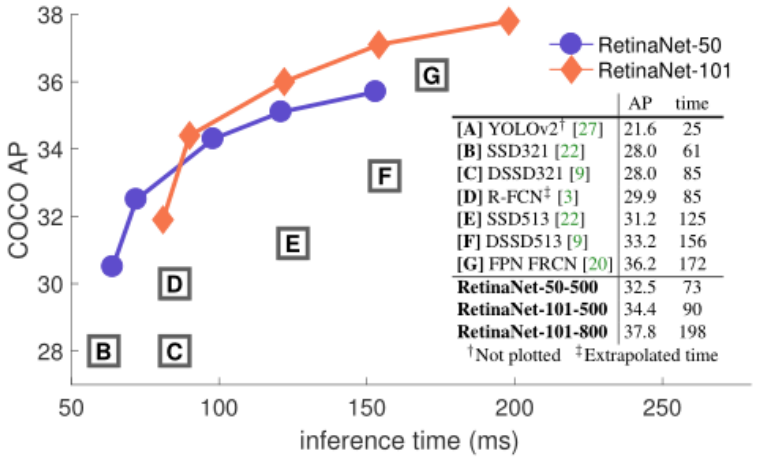


Рисунок 2.4 – Залежність точності детекції від швидкості
Джерело: [36]

Далі розглянемо відому модель мережі для ідентифікації та верифікації осіб – FaceNet [36-37]. Ця мережа є сіамською мережею, яка кодує два вхідних зображення обличь в вектор ознак, а потім порівнює їх між собою по відстані, щоб зрозуміти наскільки схожі ці два обличчя.

В даному випадку головне завдання – навчити мережу таким чином, щоб для однієї і тієї ж людини нейронна мережа видавала найбільш схожий вектор ознак, а для різних людей – різні вектори. Для цього в даній мережі була розроблена спеціальна функція втрат, звана Triplet Loss. Вона оперує трьома зображеннями обличь. На двох із зображень знаходиться одна і та ж людина, а на третьому – інша. Одне з двох зображень тієї ж людини називається якорем (anchor), друге береться за позитивний (positive) приклад, а зображення з іншою людиною – за негативний (negative). Формальний опис даної функції втрат наведено у формулі (2.2).

$$||f(x^a) - f(x^p)||^2 + \alpha < ||f(x^a) - f(x^n)||^2 \quad (2.2)$$

де $f(x^a)$ – вектор ознак, отриманий з якорного зображення;

$f(x^p)$ – вектор ознак, отриманий з позитивного прикладу;

$f(x^n)$ – вектор ознак, отриманий з негативного прикладу;

α – константа, що задає відступ між зображеннями різних людей.

Суть використання цієї функції втрат в тому, щоб навчити нейронну мережу для тієї ж людини видавати близькі по відстані один до одного вектори ознак, а для різних – вектори, що відрізняються на задану константу, яка називається відступом (рис. 2.5). Відступ потрібен, щоб нейронна мережа не намагалася знаходити найбільш тривіальне рішення, роблячи всі вектори однаковими.

Рисунок 2.5 – Процес навчання при використанні Triplet Loss
Джерело: [38]

При цьому важливо відзначити, що одним з важливих аспектів використання Triplet Loss є те, що в якості негативних прикладів варто підбирати середні по складності, тобто зображення людей повинні бути не дуже схожі, але і не дуже різнитися. Виходячи з проведених досліджень було виявлено, що таким чином мережа навчається найкраще і не застряє в екстремумах при навчанні методом градієнтного спуску.

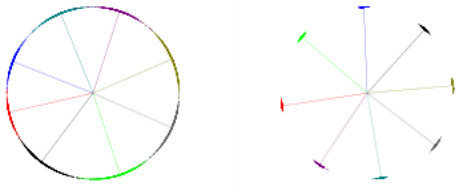
Подальшим розвитком мережі FaceNet стала мережа OpenFace [40]. Її розробка була акцентована на тому, щоб дослідити і довести можливість навчання мережі для розпізнавання облич на відкритому наборі даних, на відміну від її попередників, які зазвичай використовували приватні датасета на мільйони зображень. Також ця мережа передбачалася бути використаною для розпізнавання облич на мобільних пристроях, потужність яких не так

велика, і при цьому бути все ще досить швидкою, щоб можна було використовувати її в режимі реального часу. Тому структура OpenFace є модифікованою структурою мережі FaceNet зі зменшеною кількістю параметрів для більш швидкого навчання і роботи мережі.

Як уже згадувалося раніше, при вирішенні завдання ідентифікації і верифікації, також використовують метод кодування зображень обличь в вектори ознак через навчання класифікатору. При цьому функція помилки Softmax не дає строгого поділу класів між собою на їх границях. В якості вирішення проблеми пропонувалися досконаліші функції помилки для класифікації в контексті виділення векторів ознак із зображень осіб для їх подальшого розпізнавання. Однією з таких функцій помилок є функція ArcFace (див. формулу (2.3)), запропонована і випробувана в однойменній нейронній мережі [41].

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s*\cos\left(\theta_{y_i}\right)-m}}{e^{s*\cos\left(\theta_{y_i}\right)-m}+\sum_{j=1,j\neq y_i}^{C}e^{s*\cos(\theta_j)}} \tag{2.3}$$

Візуалізація прикладу класифікації при використанні помилки ArcFace в порівнянні з Softmax представлено на рисунку 2.6. Кожен колір і лінія на малюнку зображує свій клас. При цьому можна помітити, що у Softmax границі класів мають дуже розмиті переходи з одного класу в інший, на відміну від функції ArcFace, де між кожним класом існує досить відстані, при якій важко віднести об’єкт, що класифікується, до невірного класу.



(a) Softmax (b) ArcFace

Рисунок 2.6 – Візуалізація класифікації зображень при використанні функцій помилок Softmax і ArcFace

Джерело: [41]

2.2 Модификація системи розпізнавання облич

Під час виконання аналізу існуючих мереж було розглянуто множину різних структур мереж.

У реалізації деяких з них основними пропозиціями виступали не стільки сама архітектура мережі, скільки підхід до її навчання. Наприклад, для мереж, що кодують зображення особи в векторне подання, було виділено два основні методи.

Перший метод полягає у використанні спеціальної функції помилки Triplet Loss, що дозволяє скоротити відстань векторів ознак між зображеннями одного і того ж людини і збільшити відстань між векторами зображень різних людей.

Другий метод пропонує навчити класифікатор з подальшим видаленням останнього шару, що класифікує, в результаті чого виходить мережа, яка вже сама по собі вміє правильно виділяти ознаки.

Відомо, що для кращого навчання мережі при використанні функції помилки Triplet Loss потрібно підбирати середні по складності триплети. Однак, це також є складністю, оскільки їх підбір потрібно робити вручну, і якщо датасета зображень дуже великий, а саме такі датасети потрібні для кращого навчання мереж, то створення такого датасету стає довгим рутинним процесом.

Як було виявлено, мережа OpenFace, яка є продовженням мережі FaceNet, використовує в якості функації помилки Triplet Loss, і є однією з сучасних мереж для розпізнавання облич.

Тому в якості модифікації мережі для розпізнавання осіб пропонується взяти мережу OpenFace, і навчити її як класифікатор, додавши в кінець цієї мережі додатковий шар, що класифікує.

Запропонована архітектура OpenFace приведена в таблиці 2.1.

Таблиця 2.1 – Загальна структура мережі

№	Тип шару	Опис
1	вхідний шар	кольорове зображення розміром 96x96 пікселів
2	згортка	96 фільтрів розміром 7x7 пікселів з кроком 2x2
3	макс-пулінг	розмір 3x3 пікселів з кроком 2x2
4	згортка	64 фільтрів розміром 1x1 пікселів з кроком 1x1
5	згортка	192 фільтрів розміром 3x3 пікселів з кроком 1x1
6	макс-пулінг	розмір 3x3 пікселів з кроком 2x2
7	inception (3a)	конкатенує 4 відгалуження шарів з попереднього
8	inception (3b)	конкатенує 4 відгалуження шарів з попереднього
9	inception (3c)	конкатенує 3 відгалуження шарів з попереднього
10	inception (4a)	конкатенує 4 відгалуження шарів з попереднього
11	inception (4e)	конкатенує 3 відгалуження шарів з попереднього
12	inception (5a)	конкатенує 3 відгалуження шарів з попереднього
13	inception (5b)	конкатенує 3 відгалуження шарів з попереднього
14	average-пулинг	розмір 3x3 пікселя з кроком 1x1
15	повнозв’язний шар	128 нейронів в шарі
20	L2-регуляризація	

Структура мережі OpenFace, як і у її попередника FaceNet, має inception архітектуру. Шари inception утворюються з розгалуження шару, коли до нього додаються шари не всі послідовно, а паралельними групами, які вже можуть складатися з послідовних шарів. Після цього inception шар конкатенує всі розгалуження в один цілісний шар. Тому в таблиці 2.1 наведена загальна структура мережі, а на рисунку 2.7 – опис окремих відгалужень, з яких утворюються результуючі inception шари.

Також, крім іншого методу навчання, запропоновано реалізувати модифіковану структуру мережі, а саме – спростити її, але при цьому навчити на досить великому датасеті. Для цього в модифікованої структурі мережі вирішено видалити 4 і 5 групи inception шарів. За аналітичними оцінками

таким чином можна домогтися більш швидкого навчання мережі, не погіршуючи якість розпізнавання. На рисунку 2.7 виділено видалені групи шарів в модифікованій мережі.

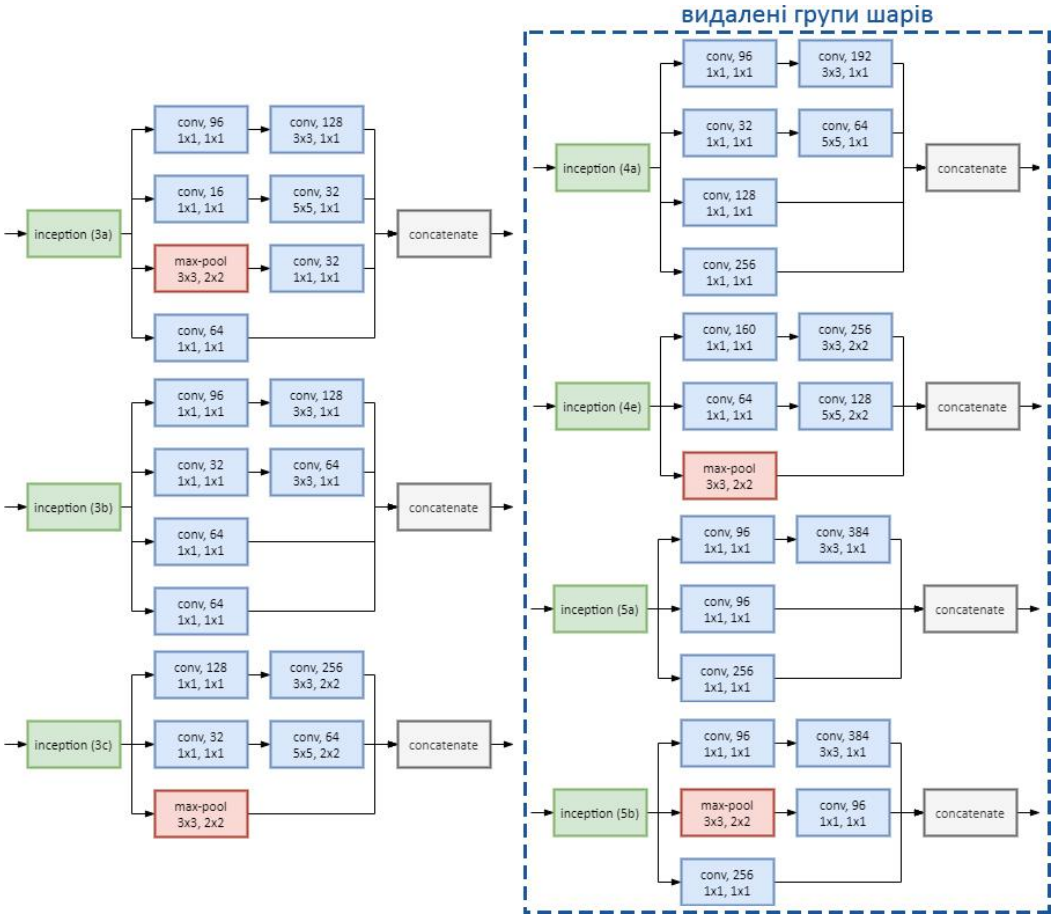


Рисунок 2.7 – Структура inception модулів оригінальної мережі OpenFace та видалена частина шарів в модифікованій версії

Авторський рисунок

При цьому також важливо відзначити, що в наведених таблицях та рисунку шар згортки у всіх випадках складається з трьох шарів – безпосередньо згортка, батч-нормалізація, а потім активація за допомогою функції активації ReLU.

При додаванні класифікуючого шару в кінець цієї мережі додається новий повнозв’язний шар, кількість нейронів в якому дорівнює кількості класів при навчанні мережі. Кожен клас повинен представляти окрему людину і мережа буде навчатися співвідносити різні зображення облич людей до

певного з класів. У свою чергу, кількість класів буде варіюватися відповідно до навчального набору даних, тому немає сенсу ставити заздалегідь визначену кількість класів.

3.3 Висновки за розділом

В даному розділі було розглянуто архітектури, основні ідеї та пропозиції сучасних моделей згорткових нейронних мереж для детекції та ідентифікації, а також методи, які вони використовують. Було вибрано і описано модифіковану структуру мережі для розпізнавання осіб, а також в якості методу навчання мережі вибрано навчання через класифікацію з подальшим видаленням останнього класифікуючого шару.

РОЗДІЛ 3
РОЗРОБКА ТА ОЦІНКА МОДИФІКОВАНОЇ МОДЕЛІ РОЗПІЗНАВАННЯ ОБЛИЧ

3.1 Підготовка середи розробки

Навчання мережі було вибрано виконувати на базі лабораторії ПМІ «ML: Нейронні мережі та машинне навчання». Специфікація робочої станції приведена в таблиці 3.1.

Таблиця 3.1 – Специфікація робочої станції:

Тип	Назва	Специфікація
Назва станції	Комплект науково – дослідного обладнання для лабораторії «Leader-Pro»	Ін. № 1014600338
Процесор	AMD Ryzen Threadripper 2990WX	32 core, 64 treads, 4 channel, 3.0GHz
Відеокарта	MSI GeForce RTX 2080 Ti Gaming Trio	11 GB, GDDR6 1755 MHz / 1350 MHz 4352 cores Підтримка CUDA
Материнська плата	ASUS ROG ZENITH II EXTREME ALPHA	AMD sTRX4, 8 x DDR4, Intel XMP, AMP
Операційна система	Windows 10 Pro	x64

Для машинного навчання часто використовується мова програмування Python. Вона є кросплатформною високорівневою мовою, що спрощує роботу

з ним, а також для нього існує велика кількість готових бібліотек, включаючи бібліотеки, що спрощують виконання математичних розрахунків, бібліотеки роботи з зображеннями, а також бібліотеки для створення і навчання складних архітектур нейронних мереж. З огляду на все це, для реалізації власного модуля штучної нейронної мережі було вирішено використовувати саме цю мову.

При навчанні нейронних мереж необхідно робити дуже багато обчислень і центральні процесори (CPU) справляються з цим набагато повільніше, ніж графічні (GPU). Тому в машинному навчанні зазвичай використовуються GPU і бібліотека TensorFlow [42], що дозволяє безпосередньо взаємодіяти з графічним процесором для виконання обчислень.

Також реалізовувати повністю з нуля комплексні архітектури нейронних мереж є дуже трудомістким завданням, тому для глибокого машинного навчання також існує безліч готових бібліотек для спрощення створення нейронних мереж. Для експериментів було обрано бібліотеку Keras [43]. Вона має простий інтерфейс, який дозволяє зібрати мережу будь-якої складності з готових компонентів – шарів різного типу, безлічі функцій активації, помилок, різних оптимізацій градієнтного спуску, що використовується при навчанні, і безліч інших корисних можливостей. Крім побудови архітектури нейронної мережі, вона також дозволяє навчати її і оцінювати якість навчання по її підсумковій помилці. Сама бібліотека Keras включається в стандартну поставку TensorFlow з версії 2.0.

На додаток до основних бібліотеках для машинного навчання TensorFlow і Keras було вибрано використовувати бібліотеки Numpy, Matplotlib, OpenCV. Numpy є популярною бібліотекою для математичних розрахунків, Matplotlib дозволяє малювати графіки, що корисно при виведенні результатів навчання і їх аналізі, а бібліотека OpenCV корисна при роботі з зображеннями, оскільки вона надає можливість завантажувати зображення будь-якого популярного формату, виводити їх, а також малювати

на них , що може знадобитися, наприклад, для малювання обмежувальних рамок осіб на зображеннях.

Для проведення експериментів з параметрами нейронної мережі корисний інтерактивний пакет розробки Jupyter Notebook [44]. Він дає можливість писати код Python в спеціальних блокнотах, де можна запускати його блоками з висновком результатів після кожного з них. При чому все результати зберігаються, щоб потім їх можна було відкрити і подивитися знову, щоб, наприклад, проаналізувати або згадати в якому форматі повертає результат та чи інша функція. Корисною можливістю є ще те, що в ньому можна відразу виводити всі зображення і графіки без необхідності створення цілих віконних додатків для цього. Також блоки можуть використовуватися не тільки для коду, але і для описів тексту в форматі Markdown, який додає можливості повноцінного редактора тексту, дозволяючи використовувати різний розмір шрифту тексту, використовувати різні виділення тексту, такі як жирність або курсив, різний колір тексту, так і в цілому Markdown дозволяє також використовувати всю мову розмітки HTML, додаючи ще більше можливостей форматування. Це корисно, коли потрібно, наприклад, задокументувати, описати, виділити завдання, процес або результати проведених експериментів прямо в ході програмування. Таким чином, блокноти широко використовуються в світі Python-розробки і спрощують життя дослідникам вивчати особливості якихось бібліотек і проводити власні дослідження зі збереженням і описом результатів без необхідності створювати цілі додатки, концентруючи увагу тільки на завданнях досліджень.

В області програмування на Python також широко відоме поняття віртуального середовища. Воно дозволяє створювати для додатків, що розробляються, своє власне, незалежне від інших проектів середовище з виконуваними файлами і бібліотеками. Туди входять сам інтерпретатор Python заданої версії та всі встановлювані через менеджер пакетів Pip бібліотеки. Це все дозволяє легко переносити проекти з однієї машини на

іншу, а також спрощує встановлення залежностей, оскільки всі версії встановлених бібліотек залишатимуться тими самими в межах одного середовища, гарантуючи сумісність версій бібліотек, інших залежностей та інтерпретатора Python між собою.

Таким чином, було виконано процес налаштування віртуального середовища та встановлення всіх необхідних бібліотек Python для подальшої роботи, команди якого наведено на рисунку 3.1. Для роботи було використано версію Python 3.7.2.

```
# Перехід у директорію проекту
cd D:/Projects/face-recognition-cnn

# Створення та активація віртуального оточення у поточній директорії
python -m venv .
scripts/activate

# Встановлення залежностей через Pip
pip install tensorflow==2.4.1
pip install Keras==2.4.3
pip install numpy==1.19.5
pip install matplotlib==3.4.0
pip install jupyterlab==3.0.12
pip install opencv-python==4.5.1.48
```

Рисунок 3.1 – Команди процесу налаштування середовища розробки
Авторський рисунок

Бібліотека TensorFlow для роботи з GPU вимагає бібліотеку CUDA, яка використовується в роботі з графічними відеокартами NVIDIA, тому для роботи з цією бібліотекою було вирішено використовувати комп’ютери, оснащені саме цими відеокартами. Для того, щоб TensorFlow міг використовувати GPU за допомогою CUDA, також було необхідно встановити відповідні бібліотеки – CUDA Toolkit та cuDNN, інакше під час виконання програми, яка використовує Tensorflow, будуть виводитися попередження та помилки про те, що він не зміг знайти відповідні DLL бібліотеки та замість GPU використовуватиме CPU.

Інсталятор CUDA Toolkit надається безкоштовно і його можна завантажити з офіційного сайту NVIDIA [45]. На сторінці завантаження необхідно вибрати операційну систему (ОС), версію ОС та тип інсталятора. У цьому випадку було вибрано ОС Windows версії 10. NVIDIA підтримує лише x64 архітектуру процесора і оскільки робоча станція відповідає цій вимогі, було обрано саме цю архітектуру. Тип інсталятора надає вибір між локальним інсталятором і через мережу Інтернет. У першому випадку CUDA Toolkit містить всі необхідні компоненти для установки, а в другому – вони будуть завантажуватися вже самим установником в процесі його роботи. Як тип інсталятора було вибрано локальний тип, щоб у разі розриву Інтернет-з'єднання можна було спокійно продовжити інсталяцію. Процес установки CUDA наведено рисунку 3.2.

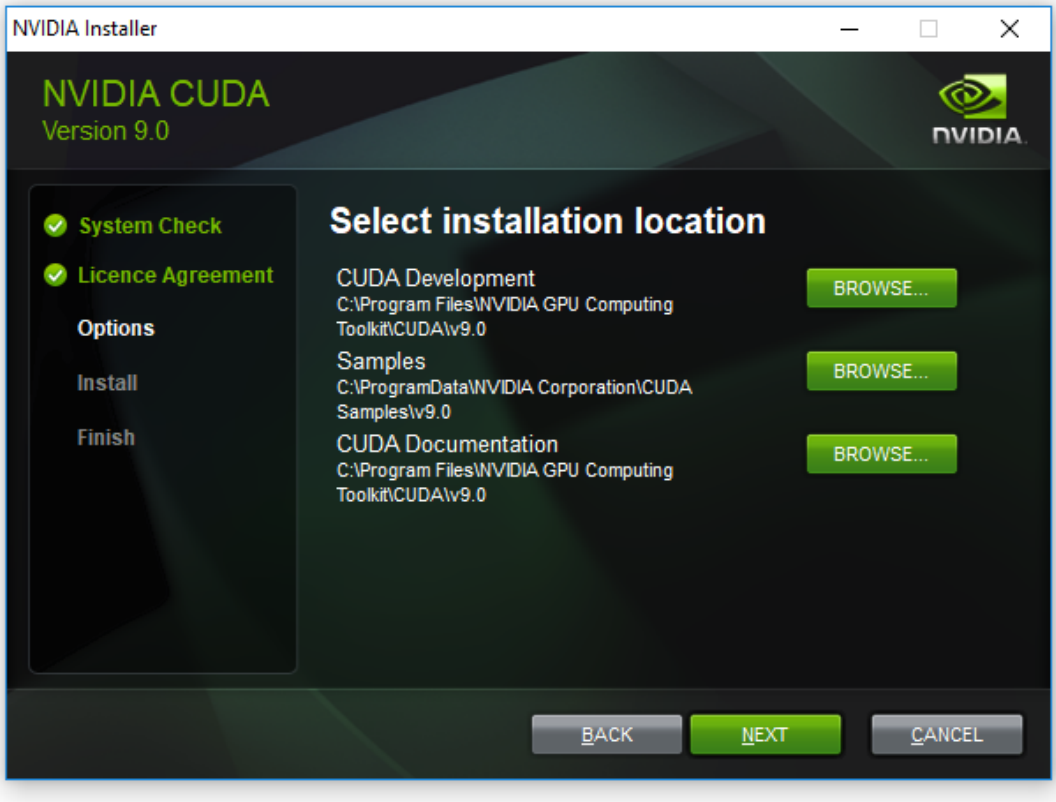


Рисунок 3.2 – Процес установки CUDA
Авторський рисунок

Бібліотека cuDNN працює безпосередньо разом із CUDA, але її необхідно встановлювати окремо від неї. Її також було завантажено з

офіційного сайту CUDA у вигляді архіву з файлами C++ та DLL, які потрібно було вручну перенести у відповідні підкаталоги у директорії із встановленим CUDA. Перед завантаженням та вибором версії cuDNN також було виконано перевірку сумісності версій за допомогою команди наведеної на рисунку 3.3.

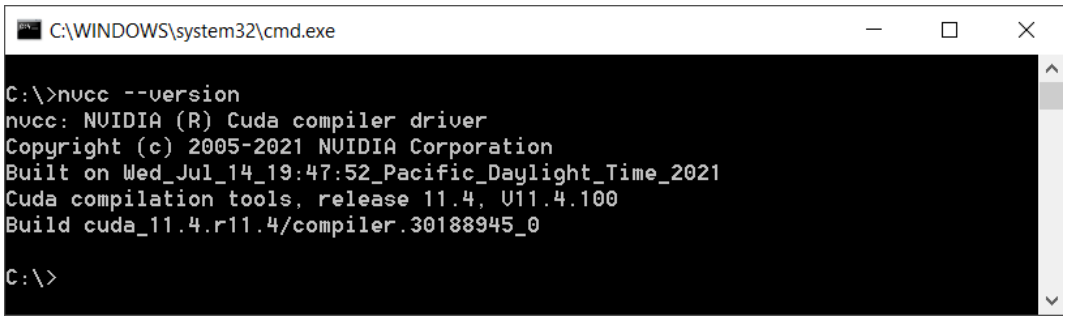


Рисунок 3.3 – Перевірка версії встановленого CUDA
Авторський рисунок

Крім того, TensorFlow працює з поняттям тензорів. Вони визначаються розмірністю і зазвичай використовуються розмірності від 0 до 3. Наприклад, 0D-тензор – це число, 1D – вектор чисел, 2D – матриця [46]. Саме розрахунки з цими тензорами і виконуються на процесорі. Для прискорення і поліпшення точності при роботі з тензорами Google розробила власні процесори, звані тензорними (TPU), і дозволяє експериментувати з ними на своєму сервісі Google Colab, де також можна використовувати і GPU.

Сервіс Google Colab [47] надає дослідникам безкоштовні ресурси для обчислень з попередньо встановленими TensorFlow, Keras і всіма іншими бібліотеками, які часто застосовуються для машинного навчання і аналізу даних. Цей сервіс представлений у вигляді Jupyter Notebook в браузері, який дозволяє виконувати власний код Python. Самі екземпляри Jupyter Notebook запускаються у віртуальних машинах з системою Linux і крім коду в блоках Notebook можна вводити більшість доступних в Bash команд, наприклад, для переходу по каталогам системи або, щоб при необхідності доустановити будь-які додаткові бібліотеки через менеджер пакетів, які не встановлені спочатку.

Лімітами Colab є кількість використовуваної пам’яті, час життя процесу Python, але Google не публікує чіткий перелік лімітів, так як він заявляє, що

вони динамічні і весь час змінюються в залежності від потреб користувачів і навантаження на сервер. Тим не менш, це відмінний інструмент, щоб експериментувати і навчати власну мережу на продуктивних серверах з сучасними процесорами, розробленими спеціально для машинного навчання. Вигляд інтерфейсу Google Colab із прикладами процесу роботи наведено на рисунку 3.4.

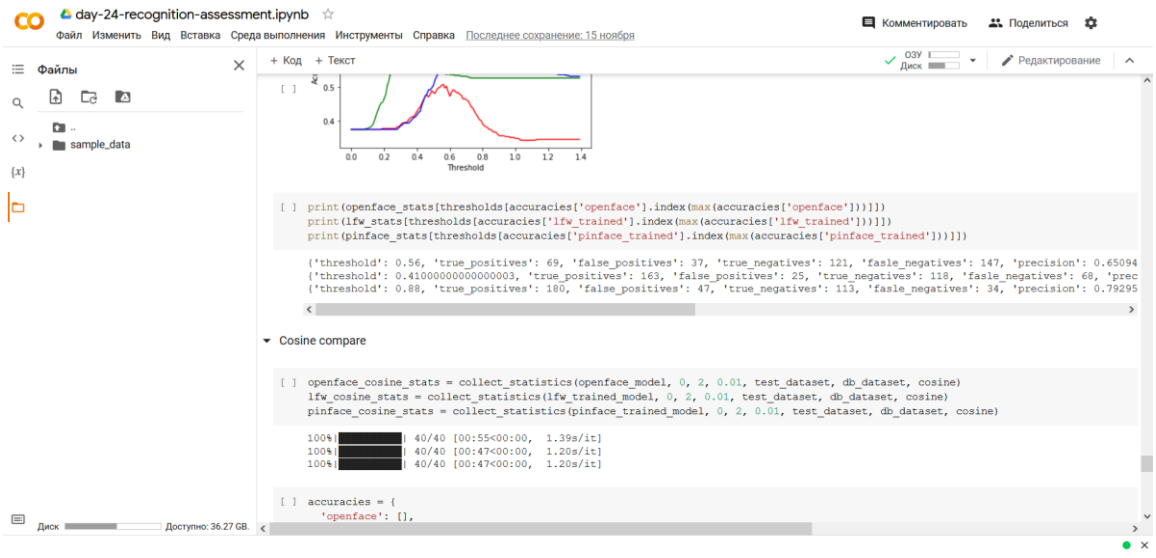


Рисунок 3.4 – Вигляд інтерфейсу Google Colab
Авторський рисунок

Для написання кінцевого коду для використання нейронної мережі як середовище розробки було вибрано безкоштовний, легкий, але розширюваний редактор вихідного коду від Miscrosoft – Visual Studio Code [48]. Він підтримує величезну кількість мов, в тому числі і Python у вигляді спеціального плагіна, який і був встановлений для подальшої роботи. Це дозволяє використовувати редактор як повноцінний IDE, який підтримує автодоповнення коду і налагодження.

3.2 Визначення набору навчальних та тестових даних

Для навчання нейронних мереж для розпізнавання обличь потрібні великі набори зображень. При цьому вибірку розбивають на навчальну і тестову. На навчальній вибірці нейронна мережа навчається, а приклади з

тестового набору є для мережі новими, тобто тими, які вона ще не бачила, тому її використовують для тестування якості роботи мережі. З навчальної вибірки часто також виділяють певну частину прикладів для валідації під час навчання. Безпосередньо на вибірці валідації мережу не навчають, але після кожної епохи навчання на ній перевіряють помилку мережі. Якщо помилка на навчальній вибірці продовжує падати, а на вибірці валідації в якийсь момент помилка починає рости, це є ознакою того, що пора закінчувати навчання, інакше мережа може перенавчитися.

Можна створювати свій власний набір даних, але це дуже довго і витратно, тим більше, що для того, щоб дійсно добре навчити нейронну мережу, набір для навчання мережі повинен бути дуже великим, різноманітним, а також збалансованим за всіма характеристиками прикладів. Тому зазвичай використовують вже готові набори, на яких навчають, тестують і оцінюють моделі нейронних мереж. Всі ці набори знаходяться в публічному доступі, деякі з них є умовними стандартами, на яких часто порівнюють існуючі моделі мереж один з одним для отримання більш правильного показника ефективності, оскільки якість розпізнавання сильно залежить від навчального набору даних і не зовсім правильно порівнювати мережі, навчені на різних наборах даних.

Також перевагою використання готових наборів є те, що деякі з них спеціально створювалися для того, щоб ускладнити розпізнавання осіб за допомогою нейронних мереж, додаючи різного роду спотворення і перекриття особи. Якщо навчати нейронну мережу на таких наборах, вона повинна навчитися краще розпізнавати обличчя, звертаючи увагу лише на їх дійсно значущі елементи, що відрізняють одну особу від іншої.

Існує величезна кількість наборів даних для різних цілей, а також набори, створені спеціально для тестування моделей розпізнавання осіб. При розробці власної системи розпізнаванн було розглянуто та використано набори описані нижче.

Набір Labeled Faces in the Wild (LFW) [49] складається з 13233 зображень 5749 осіб. Він був отриманий з фотографій людей, доступних в мережі Інтернет, за допомогою детектора осіб, заснованому на OpenCV реалізації методу Віоли-Джонса. Всі зображення є кольоровими, мають розмір 255x255 пікселів і містять область навколо голови людини. На кожну людину припадає різна кількість зображень, від 1 до 530. Файли зображень мають формат JPG і розподілені по 5749 директоріям, кожна з яких визначає конкретну людину і має назву відповідну імені та прізвищу цієї людини розділених між собою через знак підкреслення. Усередині директорій, ім'я кожного файлу відповідає імені директорії з доповненням номера зображення в цій директорії, доданий через знак підкреслення. Приклад фотографій облич із датасету LFW наведено рисунку 3.5.

Рисунок 3.5 – Приклади зображень із датасету LFW
Джерело: [50]

Набір PinsFace [51] складається із зображень осіб знаменитостей, зібраний з мережі Інтернет. Всього в датасеті 17534 зображень, які можна класифікувати між 105 людьми. Всі зображення також кольорові і містять особи в різних кутах, освітлення, емоціях, макіяжі або без нього. У датасеті всього 105 директорій, кожна з яких містить більше сотні файлів із зображеннями облич людей, відповідним директоріям. Назви директорій відповідають іменам людей, а кожен файл всередині директорій має назву

схожу з директорією, в якій знаходиться, з додаванням номера зображення в цій директорії. Приклади зображень із датасету PinsFace наведено на рисунку 3.6.

Рисунок 3.6 – Приклади зображень із датасету PinsFace
Джерело: [51]

Набір ORL Faces [52] містить набір зображень облич, зроблених в лабораторних умовах. У ньому є десять різних зображень кожного з 40 різних людей. Для деяких людей зображення були зроблені в різний час, варіюючи освітлення, вираз обличчя (відкриті/закриті очі, посміхається/не посміхається) і деталі особи (окуляри/без окулярів). Всі зображення були зроблені на темному однорідному фоні, коли випробовувані перебували у вертикальному фронтальному положенні з допуском на деякий бічний рух. Файли мають формат PGM і розмір кожного зображення в наборі дорівнює 92x112 пікселям, при цьому всі зображення в датасеті чорно-білі. Зображення організовані в 40 каталогів по одному для кожної людини, які мають імена в формі sX, де X вказує номер людини від 1 до 40. У кожному з цих каталогів є десять різних зображень цього об’єкта, які мають імена в формі Y.pgm, де Y - номер зображення для цього об’єкта від 1 до 10. Приклади зображень з датасету ORL Faces наведено на рисунку 3.7.

Рисунок 3.7 – Приклади зображень із датасету ORL Faces
Джерело: [53]

Вищенаведені набори зображень можна використовувати для навчання і тестування власної системи розпізнавання обличь. Для збільшення розміру навчального набору зображень також існує метод, званий аугментація даних (data augmentation) [54]. Його суть полягає в додаванні нових зображень на основі вже наявних за рахунок різних трансформацій, додаванні різного роду ефектів. Наприклад, можна відобразити зображення по вертикалі, збільшити або зменшити його в розмірах, додати розмиття, шуми. Також можна змінити яскравість, контрастність, колірний баланс зображення. Останнє може добре допомогти з розпізнаванням осіб при висвітленні будь-яким кольором, відмінного від денного. Якщо застосовувати зрушення і, наприклад, перемістити зображення в сторону на якусь частину його розміру, можливо, вдасться домогтися розпізнавання по половині обличчя.

Додавання зображень з використанням аугментації даних може виконуватися прямо перед навчанням в програмі за допомогою будь-яких бібліотек, тому цей спосіб є дуже ефективним з точки зору витрат за часом, оскільки не потрібно проробляти це все вручну.

3.3 Визначення алгоритмів оцінки мережі для розпізнавання облич

Для оцінки систем розпізнавання існує набір метрик, які ґрунтуються на матриці заплутаності (confusion matrix) [20]. Вона має вигляд, наведений в таблиці 3.2. Колонки в таблиці означають істинні значення, а рядки – передбачені мережею.

Таблиця 3.2 – Матриця заплутаності

	Позитивні	Негативні
Позитивні	Істинно позитивні (True Positive, TP)	Помилково позитивні (False Positive, FP)
Негативні	Помилково негативні (False Negative, FN)	Істинно негативні (True Negative, TN)

В контексті розпізнавання осіб з певною базою даних із зображеннями облич набору людей, ця матриця буде інтерпретуватися наступним чином:

– істинно позитивні приклади позначають зображення, які були правильно співвіднесені з людьми, які знаходяться в БД, тобто співвіднесення вірно;

– помилково позитивні приклади позначають зображення, які були співвіднесені з будь-якими людьми з БД, але насправді ці зображення до них не належать, тобто співвіднесення невірно;

– істинно негативні приклади позначають зображення, які не були співвіднесені з будь-якими людьми з БД, тому що в дійсності вони до них не належать, тобто співвіднесення вірно;

– помилково негативні приклади позначають зображення, які не були співвіднесені з будь-якими людьми з БД, але насправді вони до них відносяться, тобто співвіднесення невірно.

Перед оцінюванням якості розпізнавання мережі тестовий датасет розбивається на дві частини: одна частина зображень буде відходити до бази даних людей, до яких потрібно буде співведнести або неспівведнести

зображення, що перевіряються, а друга частина буде безпосередньо складати ці зображення, які будуть перевірятися на відповідність будь-якого з людей в базі даних або невідповідність нікому з них.

Потім, в процесі оцінювання виконується підрахунок кількості зображень, які потрапили в одну з груп матриці заплутаності. З цієї кількості в подальшому розраховуються метрики, які дають більш високорівневе уявлення про якість розпізнавання. Одними із самих основних і найбільш часто використовуваних метрик є метрики точності, чуйності, випадання і акуратності.

Точність (precision) – це частка виявлених зображень, квадратна міра якої відповідає потребам користувача. Це додатково називається надійністю або повторюваністю і полягає в тому, що вимірювання, які повторюються в незмінних умовах, показують еквівалентні результати. У бінарній класифікації точність також називається позитивним прогностичним значенням. Розраховується точність за формулою (3.1).

$$Precision = \frac{TP}{TP+FP} \tag{3.1}$$

Чуйність (recall) – це відсоток позитивних випадків, які були правильно ідентифіковані. Це частка успішно виявлених релевантних зображень. Це додатково називається істинно позитивним коефіцієнтом. У бінарній класифікації часто також називається чутливістю. Розраховується ця метрика по формулі (3.2).

$$Recall = \frac{TP}{TP+FN} \tag{3.2}$$

Випадання (fallout) – це частка нерелевантних зображень, які визначені як позитивні, серед всіх нерелевантних зображень. Це часто також інтерпретується як імовірність того, що нерелевантне зображення буде виявлено як позитивне. Розраховується випадання за формулою (3.3).

$$Fallout = \frac{TN}{TN+FP} \quad (3.3)$$

Акуратність (accuracy) – це частка класифікацій по всім прикладам, які були правильно розпізнані. Акуратність визначається як «частка правильної класифікації по всій кількості зразків». Значення акуратності розраховується з усіх компонент матриці заплутаності, тому часто саме це значення в першу чергу використовується для порівняння різних моделей розпізнавання. Розраховується акуратність за формулою (3.4).

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (3.4)$$

Після визначення метрик, за якими буде виконуватися оцінка навчених моделей і їх порівняння, необхідно вибрати тестовий датасет для цього процесу. Так як потрібно порівнювати кожне зображення всіх людей з кожним зображенням інших людей, доведеться виконувати дуже багато операцій. Тому було вирішено вибрати невеликий датасет і виконувати тестування і оцінку саме на ньому. Таким датасетом став набір ORL Faces. Він містить всього сорок різних людей, у кожного по десять зображень. Для різних людей фотографії були зроблені в різних умовах – при різному освітленні, часу зйомки, повороті голови, виразу обличчя, а також при наявності або відсутності додаткових аксесуарів таких як окуляри. Оскільки розмір цього датасета досить малий, на ньому не варто навчати мережу для розпізнавання осіб, якщо потрібно досягти хороших показників, але завдяки його невеликому розміру, він відмінно підійде для тестування мережі.

Таким чином, цей датасет було вирішено випадковим чином розбити на дві частини: базу даних і тестові зображення з ймовірністю попадання людини в базу даних, що дорівнює 50%. При чому, в базу потрапляє тільки одне зображення людини, всі інші автоматично потрапляють в тестовий датасет. Тому в базі даних з обличчями людей присутнє тільки по одному зображенню

на людину і мережа повинна співвіднести або неспіввіднести кожне тестове зображення з відповідним зображенням в базі даних.

Псевдокод алгоритму генерації тестового датасету та датасету бази даних представлено на рисунку 3.8.

Оскільки після видалення останнього шару, що класифікує, мережа видає вектор ознак, такі вектори, отримані з різних зображень потрібно порівняти. Існує два основних методи для порівняння векторів ознак: обчислення Евклідової відстані або порівняння по косинусу кута між векторами. Формальне визначення Евклідової відстані між двома векторами p і q, що мають розмірність n, наведено у формулі (3.5).

$$d(p,q) = \sqrt{\sum_{k=1}^{n}(p_k - q_k)^2} \tag{3.5}$$

Розрахунок косинуса між двома n-розмірними векторами p і q наведено у формулі (3.6).

$$cos(\theta) = \frac{x \cdot y}{\|x\| \cdot \|y\|} \tag{3.6}$$

Початок
  Дано *шлях до директорії датасету;*
  Дано *вірогідність попадання в базу даних;*
  список піддиректорій = завантажити із *шляху ло директорії датасету*
  тестовий датасет = пустий ассоціативний масив;
  датасет БД = пустий ассоціативний масив;
  Для кожного *імені людини* **из** *списка піддиректорій*:
    тестовий датасет[*ім'я людини*] = пустий масив;
    випадкове число = згенерувати випадкове число в межах [0; 1);
    Якщо *випадкове число* < *вірогідності попадання в базу даних*:
      додати в базу даних = **True**;
      датасет БД[*ім'я людини*] = пустий масив;
    Інакше:
      додати в базу даних = **False**;
    Для кожного *зображення* в піддиректорії *імені людини*:
      Якщо *додати в базу даних*:
        додати *зображення* в масив *датасет БД*[*ім'я людини*];
        додати в *базу даних* = **False**;
      Інакше:
        додати *зображення* в масив *тестовий датасет*[*ім'я людини*];
Кінець

Рисунок 3.8 – Алгоритм генерації датасетів для оцінки мереж

Авторський рисунок

Тоді формула розрахунку схожості між цими векторами матиме вигляд, наведений у формулі (3.7).

$$\text{Відстань} = \frac{\cos^{-1}(\cos(\theta))}{\pi} \tag{3.7}$$

Кожен із цих методів розрахунку схожості між векторами ознак може вибиратися залежно від задачі, оскільки в різних сценаріях використання вони можуть давати різні результати. З огляду на це було вирішено виконати додаткове дослідження та провести тестування мереж використовуючи обидва методи.

Псевдокод загального алгоритму оцінки мережі наведено далі, на рисунку 3.9.

```
Початок
    Дано тестовий датасет, датасет бази даних, поріг;
    True Positives = 0, False Positives = 0, True Negatives = 0, False Negatives = 0;
    Для кожного імені людини testPerson із тестового датасету:
        Для кожного зображення testImage із тестового датасету[testPerson]:
            відстані = пустий масив;
            testVector = розрахувати вектор признаків з testImage;
            Для кожного імені людини dbPerson із датасету БД:
                Для кожного зображення dbImage із датасету БД[dbPerson]:
                    dbVector = розрахувати вектор признаків из dbImage;
                    відстань = Евклідова відстань між testVector і dbVector;
                    додати кортеж (відстань, ім'я dbPerson) в масив відстаней;
            відсортувати масив відстаней за зростанням;
            мінімальна відстань = перший елемент масива відстаней;
            ім'я людини dbPerson = перший елемент масива відстаней;
            Якщо мінімальна відстань < поріга:
                Якщо testPerson и dbPerson це одна й та сама людина:
                    True Positives = True Positves + 1;
                Інакше:
                    False Positives = False Positives + 1;
            Інакше:
                Якщо ім'я людини testPerson відсутня в датасеті бази даних:
                    True Negatives = True Negatives + 1;
                Інакше:
                    False Negatives = False Negatives + 1;
Кінець
```

Рисунок 3.9 – Алгоритм тестування розпізнавання згорткової мережі та підрахунку елементів матриці заплутаності

Авторський рисунок

Після отримання значення метрик схожості між векторами потрібно прийняти рішення про те, чи є отримане значення досить маленьким, щоб прийняти ці два вектори як такі, що описують одну і ту ж людину. Для цього вводиться додатковий гіперпараметр алгоритму порівняння – поріг. Для кожної моделі мережі розпізнавання його потрібно підбирати індивідуально експериментальним чином. Метою підбору даного параметра є підібрати його так, щоб отримані результати метрик давали найкращий результат для моделі.

Порівняння в ньому виконується по Евклідовій відстані, але замість нього може бути використаний будь-який інший метод порівняння двох векторів.

3.4 Проєктування класів для навчання та тестування мережі

Структура програмної частини з розробки, навчання та оцінювання нейронної мережі для розпізнавання облич відповідає діаграмі класів, наведених на рисунках 3.10 – 3.11. На першому рисунку представлено архітектуру базових класів, що задіяні в алгоритмі навчання, а на другому – безпосередньо архітектура тієї частини, що стосується алгоритмів навчання та оцінювання мережі.

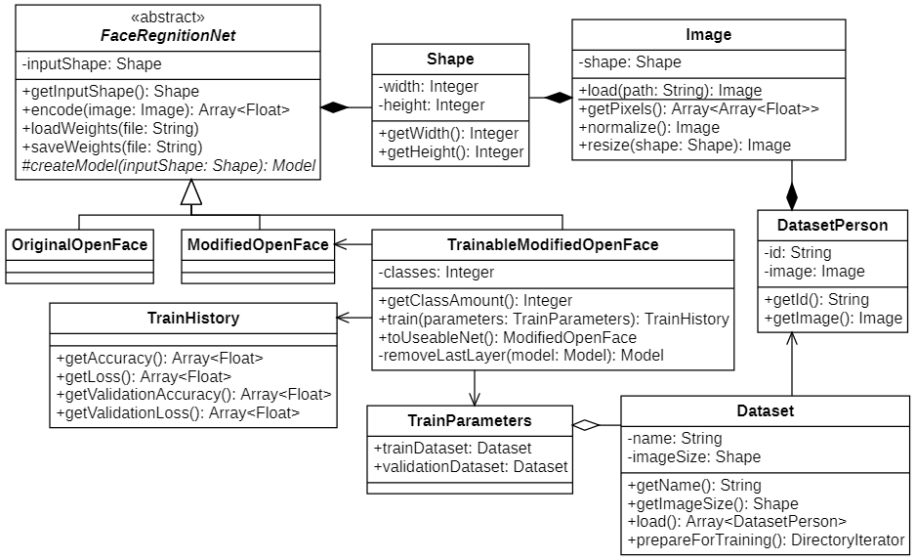


Рисунок 3.10 – Діаграма загальних класів та класів для реалізації навчання модифікованої мережі розпізнавання облич

Авторський рисунок

Одним із головних класів є клас FaceRecognitionNet. Він є абстрактним класом оскільки його мета – надати інтерфейс нейронної мережі для розпізнавання осіб для класів, які будуть успадковані від нього, а також спростити їх, виділивши загальну реалізацію. Такими класами-спадкоємцями є реалізація оригінальної мережі OpenFace та модифікованої мережі. Таким чином, за допомогою поліморфізму досягається можливість використовувати будь-яку реалізацію класу мережі для розпізнавання без зміни самого алгоритму.

Клас FaceRecognitionNet містить такі поля та методи:

– поле inputShape зберігає розмір зображення, яке має подаватися на вхід мережі і може бути встановлено фіксовано в спадкоємці класу або, наприклад делегуватися ззовні як параметр через конструктор або сеттер спадкоємця, тобто бути динамічним;

– метод getInputShape повертає значення розміру зображення, встановлене в полі inputShape;

– метод encode приймає на вхід зображення, що є фотографією обличчя будь-якої людини, яку необхідно розпізнати, і кодує це зображення у вектор ознак, який потім повертається з цього методу у вигляді масиву з числами речовинного типу;

– метод loadWeights приймає на вхід рядкове значення, що є шляхом до файлу, з якого завантажує ваги в модель нейронної мережі для їх подальшого використання, замінюючи поточні;

– метод saveWeights приймає на вхід рядкове значення, що є шляхом до файлу, в який зберігає поточні ваги моделі нейронної мережі;

– метод createModel є абстрактним методом, який має бути реалізовано у спадкоємцях і має наступний опис:

1) приймає на вхід розмір зображень, який використовуватиметься в моделі нейронної мережі;

2) конструює і повертає сам екземпляр цієї моделі;

3) модель представлена у вигляді класу Model із бібліотеки Keras;

4) цей метод буде використаний виключно всередині класу FaceRecognitionNet для внутрішнього зберігання та делегування необхідних операцій.

Клас FaceRecognitionNet при цьому композує допоміжний клас Shape для зберігання розмірів зображень. Цей клас містить такі властивості та методи:

– поле width зберігає ширину зображення;

– поле height зберігає висоту зображення;

– метод getWidth повертає ширину, що знаходиться у полі width;

– метод getHeight повертає висоту, яка знаходиться у полі height.

Від класу FaceRecognitionNet успадковується три класи: OriginalOpenFace, ModifiedOpenFace і TrainableModifiedOpenFace. Кожен з них реалізує абстрактний метод створення моделі, який створює нативну Keras модель нейронної мережі. OriginalOpenFace представляє оригінальну мережу OpenFace, ModifiedOpenFace – модифіковану мережу OpenFace, придатну для використання, а TrainableModifiedOpenFace – модифіковану мережу OpenFace, придатну для навчання. З погляду архітектури, мережа, яка навчається через класифікатор, має структуру, відмінну від структури мережі, яка буде використовуватися безпосередньо для кодування зображень і розпізнавання осіб, оскільки при навчанні додається ще один шар, який не потрібен для мережі після навчання. Тому має сенс реалізувати цю мережу у двох окремих класах – один для навчання, а інший – для використання. У такому випадку клас TrainableModifiedOpenFace має такі поля та методи:

– поле classes містить кількість класів для класифікації, щоб використовувати це значення для визначення кількості нейронів в останньому шарі, що класифікує;

– метод getClassAmount повертає значення поля classes;

– метод train виконує навчання мережі із заданими в екземплярі класу TrainParameters параметрами та повертає статистику з історією навчання, представленою класом TrainHistory;

– метод toUseableNet виконує побудову моделі поточної мережі для використання, видаляючи останній класифікуючий шар та повертаючи мережу як екземпляр класу ModifiedOpenFace;

– метод removeLastLayer безпосередньо бере нативну модель Keras і створює копію цієї моделі без останнього шару, що класифікує;

Клас TrainParameters представляє параметри, що передаються в метод train класу TrainableModifiedOpenFace для навчання мережі. Він містить такі поля:

– поле trainDataset містить датасет для навчання;

– поле validationDataset містить датасет для валідації.

Значення датасетів у системі представлені класом Dataset, який має такі поля та методи:

– поле name містить ім’я датасету та використовується як ім’я директорії, в якій розташовані зображення облич людей; задається у конструкторі класу;

– поле imageSize містить розмір у вигляді екземпляра класу Shape, до якого необхідно приводити зображення під час їх завантаження;

– метод getName повертає значення властивості name;

– метод getImageSize повертає значення властивості imageSize;

– метод load виконує завантаження зображень датасету, повертаючи масив чи список із екземплярами класу DatasetPerson;

– метод prepareForTraining має наступні характеристики:

1) створює та повертає нативний Keras об’єкт класу DirectoryIterator, який використовується для навчання;

2) дозволяє не завантажувати в пам’ять весь датасет повністю, а завантажувати кожне зображення при необхідності, оскільки датасет може бути дуже великим і не поміститься в пам’яті;

3) доповнює тренувальний датасет, виконуючи аугментацію.

Клас DirectoryIterator надається бібліотекою Keras спеціально для процесу навчання мережі. Він працює як генератор Python, повертаючи на

кожній ітерації кортеж, що складається з двох елементів – вхідне значення у мережу та очікуване значення на її виході. Вхідним значенням у разі мережі для розпізнавання облич є матриця пікселів зображення обличчя людини у потрібному розмірі, а вихідне значення генерується як вектор розміром, що дорівнює кількості класів. У цьому векторі для кожного класу всі елементи – нулі, крім того, що представляє цей клас за номером. Цей елемент у вихідному векторі дорівнює одиниці.

Для того, щоб отримувати інформацію про елементи датасету у вигляді, більш придатному для використання, наприклад, вже при розпізнаванні, є клас DatasetPerson. Він представляє певне зображення будь-якої людини в датасеті, асоційоване з його ідентифікатором. Цей клас має такі поля і методи:

– поле id містить ідентифікатор людини у рядковому вигляді;

– поле image містить зображення цієї людини у вигляді об’єкта Image;

– метод getId повертає значення поля id;

– метод getImage повертає значення поля зображення.

Клас Image представляє зображення та містить такі поля:

– поле shape містить розмір зображення як об’єкт класу Shape;

– статичний метод load приймає шлях до зображення, завантажує з нього матрицю пікселів, створює екземпляр класу Image та повертає його;

– метод getPixels повертає матрицю пікселів зображення;

– метод normalize виконує нормалізацію зображення шляхом поділу кожного пікселя на 255, роблячи значення кожного елемента матриці пікселів у межах [0, 1], для подальшого завдання нормалізованої матриці для розпізнавання;

– метод resize приймає новий розмір зображення, до якого приводить поточне зображення та повертає новий екземпляр Image з новим зображенням, що має заданий розмір.

Нативна модель Keras після навчання повертає історію зміни помилки та акуратності залежно від епохи. Тому цю історію також повертає і метод

навчання класу TrainableModifiedOpenFace у вигляді екземпляра класу TrainHistory.

Цей клас має наступні методи:

– метод getAccuracy повертає історію зміни акуратності під час навчання на тренувальному датасеті;

– метод getLoss повертає історію зміни помилки під час навчання на тренувальному датасеті;

– метод getValidationAccuracy повертає історію зміни акуратності під час перевірки на датасеті валідації;

– метод getValidationLoss повертає історію зміни помилки під час перевірки на датасеті валідації.

Далі, на рисунку 3.11 зображено діаграму класів для алгоритмів розпізнавання та оцінювання роботи мереж. Частина цих класів вже було описано вище, тому нижче було описано інші класи.

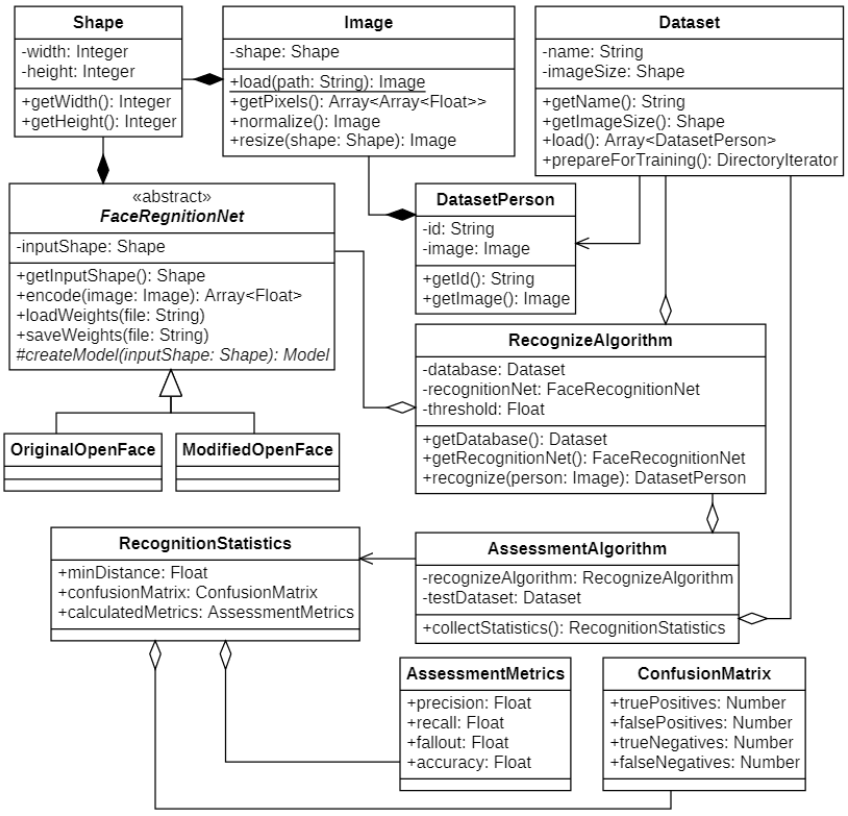


Рисунок 3.11 – Діаграма класів для реалізації алгоритмів розпізнавання облич та оцінювання нейронної мережі

Авторський рисунок

Клас RecognizeAlgorithm реалізує алгоритм розпізнавання. Він має такі атрибути та методи:

– поле database містить датасет у вигляді класу Dataset, що представляє базу даних із зображеннями людей, яких потрібно розпізнавати;

– поле recognitionNet має наступний опис:

1) містить екземпляр мережі для розпізнавання у вигляді об’єкта класу FaceRecognitionNet;

2) оскільки клас FaceRecognitionNet є абстрактним, то як значення може використовуватися будь-який спадкоємець цього класу, наприклад оригінальна згорткова мережа OriginalOpenFace або модифікована ModifiedOpenFace;

– поле threshold містить мінімальне граничне значення, при якому необхідно вважати різницю у відстані між векторами ознак зображень людей достатньою для прийняття цих зображень як таких, що описують одну й ту саму людину;

– метод getDatabase повертає значення поля database;

– метод getRecognitionNet повертає значення поля recognitionNet;

– метод recognize приймає зображення людини, а на виході видає об’єкт класу DatasetPerson, який містить ідентифікатор розпізнаної людини, або порожнє значення, якщо зображення не було розпізнане та співвіднесене до будь-якої людини у заданій датасеті бази даних.

Клас AssessmentAlgorithm реалізує алгоритм оцінки мережі. Його поля та методи такі:

– поле recognizeAlgorithm містить алгоритм розпізнавання, який оцінюватиметься на якість;

– поле testDataset містить датасет, зображення якого задаватимуться на вхід алгоритму розпізнавання;

– метод collectStatistics виконує тестування мережі та збирає корисну для подальшого аналізу статистику розпізнавання.

Клас RecognitionStatistics власне і представляє статистику, що збирається у процесі тестування мережі класом AssessmentAlgorithm. Цей клас містить такі поля:

– поле minDistance є пороговим значенням, при якому виконувалось тестування мережі та збиралася статистика;

– поле confusionMatrix представляє значення матриці заплутаності у вигляді класу ConfusionMatrix;

– поле calculatedMetrics представляє більш високорівневі метрики, зібрані з урахуванням матриці заплутаності як екземпляр класу AssessmentMetrics.

Клас ConfusionMatrix представляє значення матриці заплутаності та містить наступні атрибути:

– поле truePositives містить кількість істинно позитивних прикладів;

– поле falsePositives містить кількість помилково позитивних прикладів;

– поле trueNegatives містить кількість істинно негативних прикладів;

– поле falseNegatives містить кількість помилково негативних прикладів.

Клас AssessmentMetrics представляє метрики, зібрані на основі значень ConfusionMatrix, і містить наступні атрибути:

– поле precision містить метрику точності;

– поле recall містить метрику чуйності;

– поле fallout містить метрику випадання;

– поле accuracy містить метрику акуратності.

3.5 Програмна реалізація та навчання системи розпізнавання обличь

Під час програмування нейронної мережі для розпізнавання облич, основна розробка проводилась в Google Colab, оскільки даний сервіс вже має попередньо встановлені бібліотеки для машинного навчання. Після програмування реалізації та проведення невеликих експериментів, що

дозволяють переконатися, що не виникає ніяких помилок під час виконання коду Python, подальше навчання проводилося на власному комп’ютері.

Навчання проводилося два рази на двох різних датасетах: PinFaces і LFW. Всього процес навчання мережі зайняв два дні. Один день було витрачено на навчання на датасеті PinFaces, які мають всього 105 класів, а ще день – на навчання на датасеті LFW, який має 5749 класів. Незважаючи на таку різницю в кількості класів між цими двома датасетами, вони мають приблизно однакову загальну кількість зображень, тому навчання на кожному з них зайняло приблизно однаковий час. Додатково кожен датасет розширювався виконанням аугментації за допомогою Keras. Зображення зсувалися в сторони в межах 10%, поверталися в межах 40 градусів, масштабувати в межах 20%, а також дзеркально відображалися по вертикалі. Це все виконувалося автоматично випадковим чином спеціальним завантажувачем даних Keras. Він потрібен для того, щоб завантажувати зображення з диска поступово оскільки великі датасети можуть не вміститися в оперативній пам’яті, а також він виконує внутрішні оптимізації для прискорення навчання в TensorFlow.

При навчанні на кожному з датасетів, зображення випадковим чином поділялися на навчальну вибірку і вибірку валідації, де 80% зображень відходило до вибірки для навчання, а 20% – для валідації. За допомогою спеціальних функцій Keras було налаштовано автоматичне збереження ваг при досягненні кращого показника точності на вибірці валідації після завершення кожної епохи. Якщо результат погіршився або не змінився, ваги на черговий епосі навчання не зберігаються.

Тому сигналом про зупинку навчання під час цього процесу навчання послужив показник досягнення найкращої точності на вибірці валідації. Якщо при навчанні цей показник не поліпшується досить велику кількість епох поспіль, значить навчання зупинялося. При цьому за замовчуванням для навчання виставлялося 500 епох. Якщо було видно, що точність ще поліпшується, мережа навчалася ще 500 епох, до тих пір поки не буде видно, що навчання пора припиняти. Всього для навчання на датасеті LFW

знадобилося 794 епохи, а на PinFaces – 906 епох, домігшись точності на вибірках валідації – 97% і 98% відповідно.

На рисунку 3.12 представлено динаміку зміни втрати на датасеті валідації для набору LFW. Як можна побачити, помилка змінювалася досить рівно протягом усього навчання і при зупинці мережа досягла рівня помилки, близької до 0,25%.

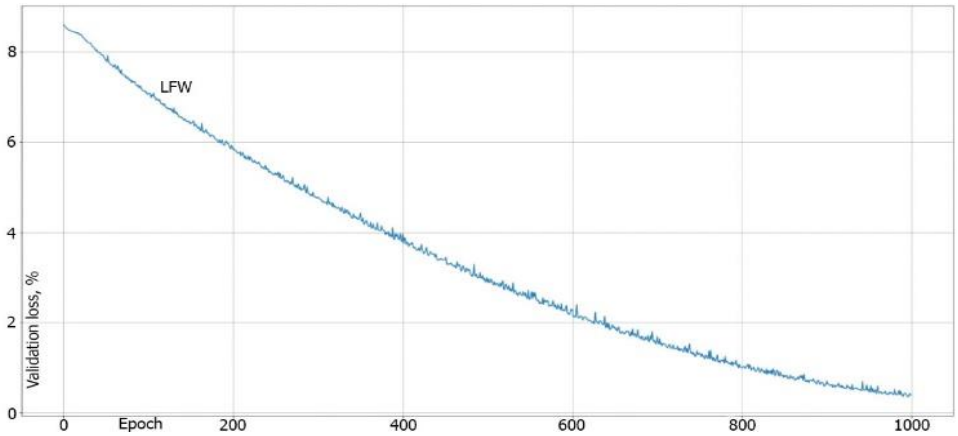


Рисунок 3.12 – Валідація на датасеті LFW
Авторський рисунок

Динаміку зміни помилки на датасеті валідації для набору PinsFace представлено на рисунку 3.13. PinsFace має досить невелику кількість класів у порівнянні з LFW, тому на графіку можна помітити, що мережа досягла маленького відсотка помилки швидше – приблизно на рівні 300 епох і далі залишалася приблизно на одному рівні.

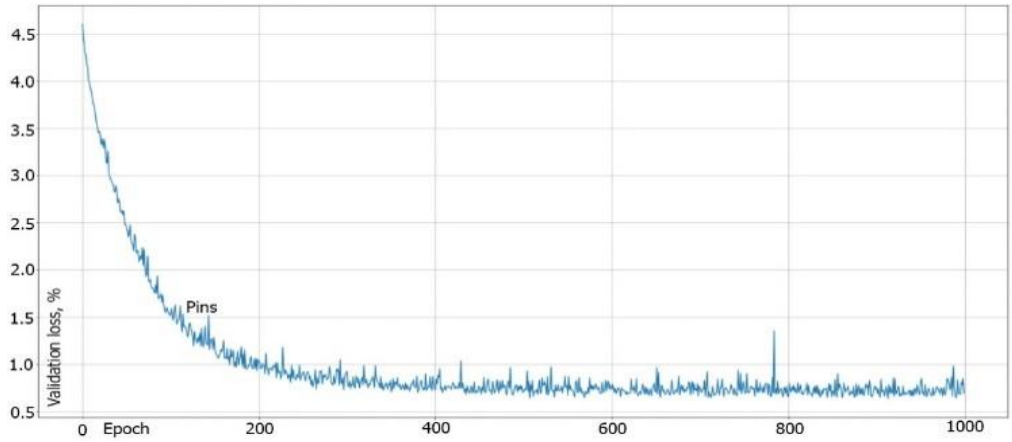


Рисунок 3.13 – Валідація на датасеті PinsFace
Авторський рисунок

Такі ж висновки можна зробити і з графіків з динамікою зміни значення акуратності класифікації для використаних датасетів, представлених на рисунку 3.14. Мережа під час навчання на PinsFace досягає акуратності, близької до 100%, майже вдвічі швидше, ніж LFW. Звідси можна дійти до висновку, що для навчання на PinsFace необхідно близько 500 епох, а LFW достатнім буде 850-900 епох.

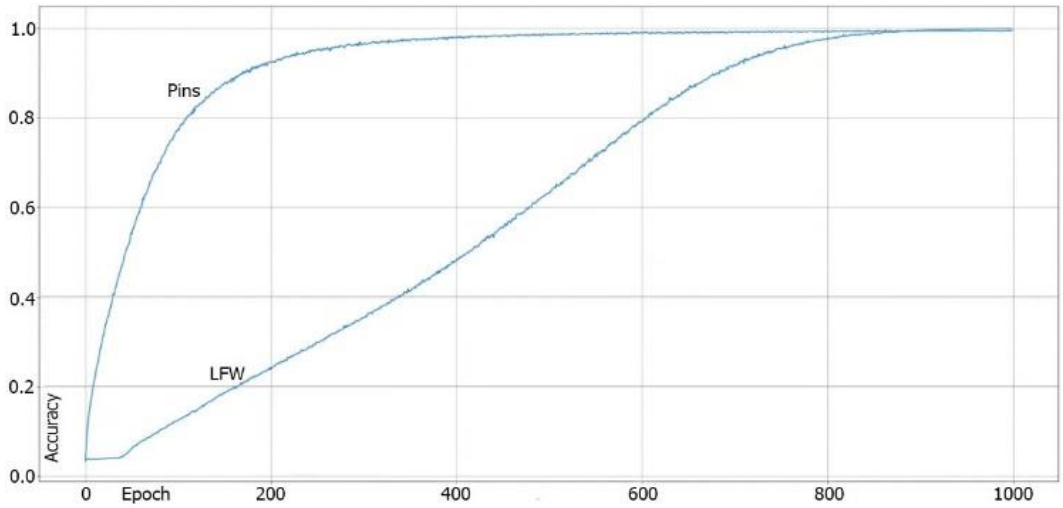


Рисунок 3.14 – Точність навчання на використаних датасетах
Авторський рисунок

3.6 Оцінка розробленої системи розпізнавання обличь

Під час оцінки крім двох навчених моделей також оцінювалася оригінальна заздалегідь навчена мережа OpenFace. Її вихідний код був знайдений на GitHub [55]. При чому варто також відзначити, що спочатку вона була написана за допомогою бібліотеки Pytorch [56], яка є аналогом Tensorflow і також використовується для машинного навчання. Pytorch безпосередньо несумісний з бібліотекою Keras, на якій виконувалася розробка та навчання власної мережі, тому використовувалася її конвертована в Keras версія з тими ж вагами.

Під час оцінки мереж підбирати поріг можна вручну, але це довго і можна випадково упустити потрібне значення. Тому його знаходження було вирішено реалізувати шляхом грубого перебору в циклі. Першими ручними експериментами було визначено можливі значення порога, тобто в якому

приблизному діапазоні його шукати. З'ясувалося, що потрібне значення може бути як менше нуля, так і більше одиниці, але не більше двох. Тому було обрано шукати найкращі граничні значення в діапазоні [0; 2) з кроком 0.01. Таким чином, провівши експерименти і домігшись найкращих параметрів алгоритму порівняння для кожної з моделей, було отримано результати, описані далі.

На рисунку 3.15 наведено графік залежності акуратності розпізнавання від порогового значення для кожної з мереж, отримані при порівнянні векторів ознак за допомогою Евклідової відстані. З отриманого графіка добре видно, що акуратність розпізнавання при найкращому порозі для модифікованої мережі вища, ніж оригінальна. Різниця між двома навченими мережами невелика, хоча можна помітити різницю у формі графіків. Графік мережі, яка навчалася на датасеті LFW вужчий, коли графік для мережі, навченої на PinsFace має ширшу форму. Це означає, що якість розпізнавання менше залежить від порогу для PinsFace, де кількість класів менше. Звідси можна дійти до висновку, що залежність якості розпізнавання від порога корелює з кількістю класів, тобто чим менше класів – тим менше залежність.

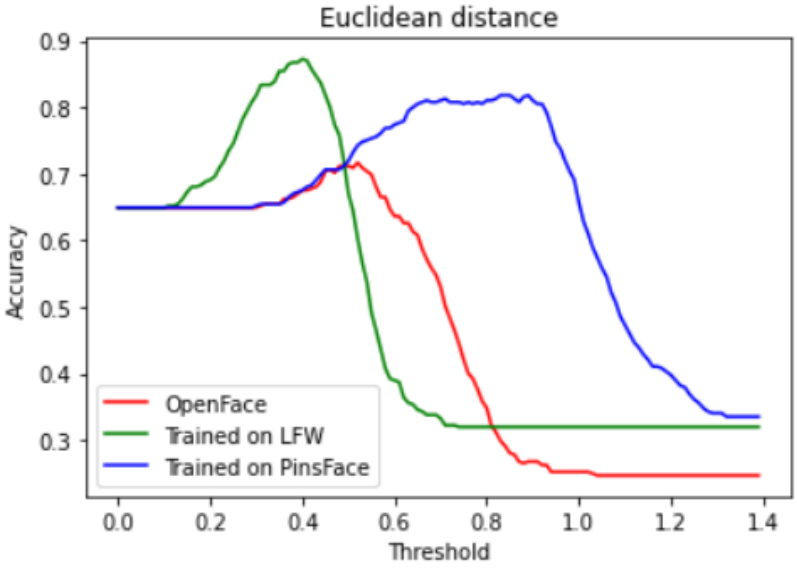


Рисунок 3.15 – Залежність акуратності розпізнавання від порога порівняння векторів по Евклідовій відстані

Авторський рисунок

Далі для кожної мережі було розраховано весь набір метрик у точці, що відповідає максимальній акуратності. Отримані значення наведено в таблиці 3.3. Зліва результати оригінальної мережі OpenFace, посередині – модифікованої мережі, навченої на датасеті LFW, а справа – модифікованої мережі, навченої на датасеті PinFaces.

Таблиця 3.3 – Порівняння найкращих результатів моделей при порівнянні векторів за допомогою Евклідової відстані

	Оригінальна OpenFace	Навчена на LFW	Навчена на PinFaces
Поріг	0.52	0.4	0.83
Число TP прикладів	34	104	102
Число FP прикладів	10	21	41
Число TN прикладів	242	232	213
Число FN прикладів	99	28	29
Точність	77.27%	83.2%	71.33%
Чуйність	25.56%	78.79%	77.86%
Випадання	96.03%	91.7%	83.86%
Акуратність	71.69%	87.27%	81.81%

Для кожної моделі на першому рядку представлений параметр порогу, при якому вектори повинні вважатися такими, що описують одну і ту ж людину. Для кожної моделі він є різним оскільки кожна модель навчалася за допомогою різних методів і/або на різних датасетах, тому підбирати його потрібно для кожної окремо. Далі перераховані значення категорій матриці заплутаності, після чого результати метрик точності, чуйності, випадання і акуратності відповідно.

Як можна побачити при навчанні власних моделей вдалося підвищити загальну акуратність розпізнавання приблизно на 10-16%. При чому мережа, яка навчалася на датасеті LFW дала на 6% кращий результат, ніж при навчанні

на PinFaces. Можливо це пов’язано з тим, що LFW має більшу кількість класів, а саме 5749, в порівнянні з PinFaces, в якому їх всього 105, тому мережа могла навчитися відрізняти один від одного більш складні приклади, тобто зображення облич людей, схожих між собою.

Відповідні висновки відносно впливу на точність розпізнавання було розглянуто в [57]. Це дослідження підтверджується порівнянням акуратності модифікованих моделей.

Що ще примітно, за кількістю нарахованих значень TP, FP, TN і FN видно, що оригінальна мережа при найкращому показнику своєї акуратності віддає більшу перевагу негативному розпізнаванню, тобто відносить зображення облич до категорій тих, кого немає в базі даних. При чому робить вона це як правильно, так і ні. У свою чергу, модифіковані мережі справляються з цим завданням краще – у них розподіл позитивно та негативно розпізнаних прикладів більш збалансований, що робить загальну акуратність цих мереж вище.

Також було виконано аналогічні дії зі збирання та аналізу статистики при порівнянні векторів ознак за допомогою кута косинуса між ними. Методом перебору в циклі було знайдено найкращі значення порогового значення та розраховано інші метрики у відповідних точках. Графік залежності акуратності від порога наведено рисунку 3.16.

Як можна побачити, результати майже ідентичні. Різницю можна помітити лише в дуже невеликих деталях, таких як форма кутів, що формуються лініями.

Звідси можна дійти до висновку, що з завдання розпізнавання осіб вибір методу порівняння векторів ознак немає значення – обидва методи порівняння векторів ознак придатні для використання при вирішенні даної задачі. Відповідний набір метрик для найкращих значень порога порівняно з куту косинуса наведено в таблиці 3.4.

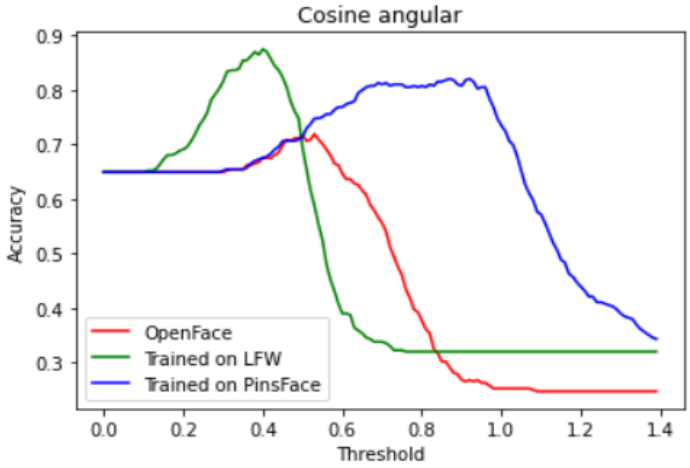


Рисунок 3.16 – Залежність акуратності розпізнавання від порога при порівнянні векторів по куту косинуса

Авторський рисунок

Таблиця 3.4 – Порівняння найкращих результатів моделей при порівнянні векторів за допомогою косинуса кута

	Оригінальна OpenFace	Навчена на LFW	Навчена на PinFaces
Поріг	0.53	0.4	0.87
Число TP прикладів	35	103	106
Число FP прикладів	10	19	44
Число TN прикладів	242	234	210
Число FN прикладів	98	29	25
Точність	77.77%	84.43%	70.66%
Чуйність	26.32%	78.03%	80.92%
Випадання	96.03%	92.5%	82.68%
Акуратність	71.95%	87.53%	82.08%

Під час тестування також було виконано вимірювання швидкості розпізнавання для кожної мережі. Результати порівняння наведено на графіку, зображеному на малюнку 3.17. Наведені значення відображають середній час, що займає розпізнавання одного тестового зображення з усіма зображеннями в базі даних, тобто розпізнавання однієї людини. Як можна побачити,

оригінальна мережа займає приблизно 1,5 секунд, коли в модифікованій структурі мережі вдалося зменшити цей час приблизно на 0.25 секунд.

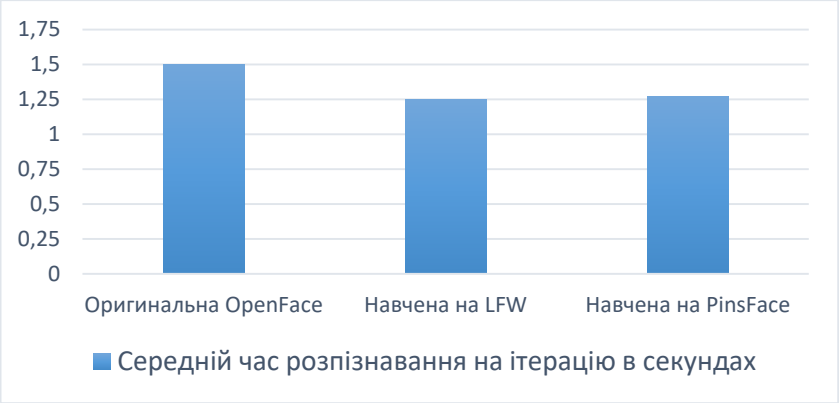


Рисунок 3.17 – Середній час розпізнавання на одну ітерацію
Авторський рисунок

Додатково до оцінки точності розпізнавання мереж було виконано експеримент порівняння швидкості навчання на різних процесорах. За результатами навчання мереж на Tensorflow (рис. 3.18) видно, що час навчання на всіх епохах займає приблизно однаковий час, тому, щоб не перенавчати мережу цілком, було вирішено навчити модифіковану мережу на 100 епохах на CPU і GPU. При цьому навчання виконувалося на датасеті LFW, який показує найкращі характеристики якості розпізнавання. Таким чином час навчання 100 епох на CPU зайняв у сумі 23146 секунд, а на GPU – 4606 секунд, тобто приблизно 6,4 години і 1,3 відповідно.

```
Epoch 1/100
410/410 [==============================] - 405s 982ms/step - loss: 8.6107 - accuracy: 0.0381 - val_loss: 8.5949 - val_accuracy: 0.0025
Epoch 2/100
410/410 [==============================] - 212s 518ms/step - loss: 8.4912 - accuracy: 0.0384 - val_loss: 8.5653 - val_accuracy: 0.0025
Epoch 3/100
410/410 [==============================] - 213s 520ms/step - loss: 8.4243 - accuracy: 0.0384 - val_loss: 8.5447 - val_accuracy: 0.0025
Epoch 4/100
410/410 [==============================] - 214s 522ms/step - loss: 8.3643 - accuracy: 0.0384 - val_loss: 8.5265 - val_accuracy: 0.0025
Epoch 5/100
410/410 [==============================] - 216s 526ms/step - loss: 8.3074 - accuracy: 0.0384 - val_loss: 8.5104 - val_accuracy: 0.0025
Epoch 6/100
410/410 [==============================] - 215s 524ms/step - loss: 8.2533 - accuracy: 0.0384 - val_loss: 8.4959 - val_accuracy: 0.0025
Epoch 7/100
410/410 [==============================] - 214s 523ms/step - loss: 8.2018 - accuracy: 0.0384 - val_loss: 8.4838 - val_accuracy: 0.0025
Epoch 8/100
410/410 [==============================] - 216s 528ms/step - loss: 8.1534 - accuracy: 0.0384 - val_loss: 8.4729 - val_accuracy: 0.0025
Epoch 9/100
410/410 [==============================] - 217s 529ms/step - loss: 8.1080 - accuracy: 0.0384 - val_loss: 8.4634 - val_accuracy: 0.0025
Epoch 10/100
410/410 [==============================] - 218s 532ms/step - loss: 8.0661 - accuracy: 0.0384 - val_loss: 8.4580 - val_accuracy: 0.0025
Epoch 11/100
410/410 [==============================] - 218s 533ms/step - loss: 8.0274 - accuracy: 0.0385 - val_loss: 8.4513 - val_accuracy: 0.0027
Epoch 12/100
410/410 [==============================] - 220s 536ms/step - loss: 7.9927 - accuracy: 0.0385 - val_loss: 8.4463 - val_accuracy: 0.0030
```

Рисунок 3.18 – Процес навчання мереж на Tensorflow з Keras
Авторський рисунок

Також було виконано спробу навчання мережі на сервісі Google Colab на процесорах TPU. Оскільки цей сервіс є безкоштовним і їм одночасно користується безліч людей, він був створений лише для інтерактивної роботи. Тому в процесі роботи він регулярно вимагає від користувача підтвердження присутності, видаючи вікно з відповідною кнопкою, через що повноцінне навчання мережі на ньому стає трудомістким. Враховуючи це, експерименти на ньому було виконано лише на 10-ти епохах із приблизним розрахунком часу, який би знадобився для навчання на 100 епохах. При цьому для розрахунку часу решти епох було взято лише час епох, наступних після першої, оскільки, як можна побачити на рисунку 3.18, при навчанні на Tensorflow з Keras перша епоха може займати найбільший час, коли наступні вимагають набагато менше часу завдяки кешуванню даних під час навчання на першій епосі. Таким чином, час навчання на Google Colab на CPU склав приблизно 46900 секунд, на GPU – 8285 секунди, а на TPU – 47778 секунди, тобто приблизно 13 годин, 2,3 години і 13,3 годин відповідно.

Сумарні результати досліджень швидкості навчання на CPU, GPU робочої станції кафедри ПМІ та CPU, GPU, TPU виділеної машини Google Colab зведено у таблиці 3.5.

Таблиця 3.5 – Приблизний час навчання на робочій станції ПМІ та виділеній машині Google Colab на CPU, GPU та TPU

	Робоча станція кафедри ПМІ	Виділена машина Google Colab
100 епох на CPU	6,4 годин	13 годин
100 епох на GPU	1,3 годин	2,3 годин
100 епох на TPU	–	13,3 годин

За результатами дослідження швидкості навчання мережі можна дійти до висновку, що у робочої станції кафедри ПМІ мережу можна навчити у кілька разів швидше, виконуючи навчання на GPU-процесорах. Також можна

помітити, що TPU на Google Colab працює дуже повільно в порівнянні з класичним GPU-процесором і навіть трохи повільніше за CPU. Тому, експериментуючи з глибоким навчанням на Google Colab, краще віддавати перевагу саме GPU, ніж CPU або TPU.

3.7 Висновки за розділом

В даному розділі було описано процес підготовки середовища розробки для створення, навчання моделей нейронних мереж для розпізнавання облич. Далі було вибрано навчальні та тестові датасети, після чого було обрано та описано методи та метрики оцінки мереж для розпізнавання осіб, а також було розроблено алгоритми складання датасетів для тестування мереж та безпосереднього збору статистики з розрахунком відповідних метрик.

Потім було розроблено класи для програмної реалізації алгоритмів навчання та оцінки мереж, після чого було навчено дві моделі на різних датасетах. За допомогою спеціальних метрик було виконано оцінку якості розпізнавання навчених моделей, які було порівняно між собою, а також оригінальною мережею без модифікацій. За результатами було відзначено, що модифіковані мережі дають кращу загальну акуратність розпізнавання осіб, при чому з них найкращою виявилась та, що навчалася на датасеті з більшою кількістю класів.

Також було виконано вимірювання середньої швидкості розпізнавання однієї людини, внаслідок чого з’ясувалося, що в модифікованій структурі мережі вдалося підвищити середню швидкість розпізнавання на 0,25 секунди.

Додатково було виконано дослідження швидкості навчання мереж на CPU та GPU робочої станції кафедри ПМІ, а також на CPU, GPU та TPU сервісу Google Colab. За результатами, найкращі показники дає GPU, при чому навчання виконується швидше саме на комп’ютері кафедри, в порівнянні з виділеною машиною Google Colab.

ВИСНОВКИ

Під час виконання дипломної роботи було поставлено завдання виконати дослідження на тему розпізнавання облич за допомогою згорткових нейронних мереж і далі навчити власну модель мережі з модифікацією, яка повинна поліпшити вже існуючу технологію.

Спочатку було вивчено теоретичну інформацію про предметну область, достатню для розробки власної ЗНМ, а також для того, щоб визначитися, що саме можна покращити. Це включило наступне:

– визначено математичну постановку задачі розпізнавання облич;

– вивчено принципи роботи нейронних мереж в цілому, структуру і алгоритм конкретно згорткової мережі, що також включає в себе повнозв’язну мережу, а також вивчено використовувані функції активації, функції помилки і додаткові архітектури та методи в нейронних мережах, які можуть допомогти в розробці глибоких нейронних мереж і дати краще розуміння їх роботи;

– вивчено архітектуру системи для розпізнавання облич, її базові компоненти, їх алгоритм роботи і як вони взаємодіють між собою;

– досліджено теоретичну інформацію про вже існуючі сучасні моделі нейронних мереж, які використовуються в сфері розпізнавання облич.

На основі дослідження існуючих нейронних мереж було вибрано модифікацію для власної мережі розпізнавання облич, а також підхід до її навчання. Після цього було необхідно реалізувати мережу з модифікацією та протестувати її. Для виконання цього завдання було виконано наступну послідовність дій:

– підготовлено середовище розробки, що включило вибір робочої станції та її характеристик для навчання мережі, а також встановлення всіх необхідних залежностей і бібліотек;

– визначено датасети для навчання та тестування мережі, з яких було вибрано два датасети для навчання мережі та один для оцінки якості розпізнавання;

– визначено методи та метрики оцінки мереж для розпізнавання осіб, а також розроблено алгоритми для їх тестування;

– розроблено архітектуру та класи для реалізації навчання мережі, її використання та тестування;

– реалізовано модифіковану версію ЗНМ і навчено її на різних датасетах, після чого було виконано її оцінку та порівняння з оригінальною мережею;

В результаті отримано дві модифіковані мережі, навчені на двох різних датасетах, які показали якість розпізнавання на 10-16% вище в порівнянні з оригінальною мережею без модифікації. Крім цього, загалом модифікована мережа дає досить високі показники точності та швидкості розпізнавання для використання у виробничому середовищі.

Також виконано порівняння швидкості навчання мережі на робочій станції кафедри ПМІ та безкоштовного сервісу для експериментів із глибоким навчанням Google Colab на різних процесорах, які вони дозволяють використовувати. В результаті цього порівняння навчання на GPU показало найкращі показники швидкості навчання на обох комп’ютерах, причому навчання на машині кафедри ПМІ виконується швидше.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ramakrishnan S. Introductory Chapter Face Recognition - Overview, Dimensionality Reduction, and Evaluation Methods / S. Ramakrishnan // Face Recognition - Semisupervised Classification, Subspace Projection and Evaluation Method: IntechOpen, 2013. P. 137-144.

2. Путь в Machine Learning и Deep Neural Networks [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/company/oleg-bunin/blog/470904/.

3. Лепский А. Е. Математические методы распознавания образов: Курс лекций / А. Е. Лепский, А. Г. Броневич. – Таганрог: Изд-во ТТИ ЮФУ, 2009. – 155 с.

4. Хайкин С. Нейронные сети: полный курс, 2-е издание / С. Хайкин. – М. : Издательский дом «Вильямс», 2006. – 1104 с.

5. Нейронные сети, перцептрон [Електронний ресурс]. – Режим доступу до ресурсу: https://neerc.ifmo.ru/wiki/index.php?title=Нейронные_сети,_перцептрон.

6. ReLU [Електронний ресурс]. – Режим доступу до ресурсу: https://congyuzhou.medium.com/relu-b1a6ba5455b.

7. Рашид Т. Создаем нейронную сеть / Т. Рашид. – СПб. : ООО «Альфа-книга», 2017. – 272 с.

8. Brownlee J. Loss and Loss Functions for Training Deep Learning Neural Networks // J. Brownlee // Mach. Learn. Mastery, 2019. P. 1–19.

9. What Are Tensor Cores? Mixed-Precision Computing. [Електронний ресурс]. – Режим доступу до ресурсу: https://www.techspot.com/article/2049-what-are-tensor-cores/.

10. Masters T. Deep Belief Nets in C++ and CUDA C Volume 3: Convolutional Nets / T. Masters. – Apress Media, 2018. – 184 p.

11. Сверточная нейронная сеть, часть 1: структура, топология, функции активации и обучающее множество. [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/post/348000/.

12. Самое главное о нейронных сетях. [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/company/yandex/blog/307260/.

13. Как работают сверточные нейронные сети. [Електронний ресурс]. – Режим доступу до ресурсу: https://proproprogs.ru/neural_network/kak-rabotayut-svertochnye-neyronnye-seti.

14. Zeiler M. D. Visualizing and understanding convolutional networks / M. D. Zeiler, R. Fergus // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): arXiv, 2014. P. 1-11.

15. He K. Deep residual learning for image recognition / K. He, X. Zhang, S. Ren, J. Sun // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2016. P. 770–778.

16. Szegedy C. Going deeper with convolutions / C. Szegedy // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2015. P. 1–9.

17. Три преобразующие архитектуры: ResNet, Inception и Xception [Електронний ресурс]. – Режим доступу до ресурсу: https://russianblogs.com/article/355119630/.

18. Understanding the Bias-Variance Tradeoff [Електронний ресурс]. – Режим доступу до ресурсу: https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229.

19. Kukačka J. Regularization for Deep Learning: A Taxonomy / J. Kukačka, V. Golkov, D. Cremers // Technical University of Munich: arXiv. 2017. P. 1–23.

20. Sundaram M. Face Recognition: Demystification of Multifarious Aspect in Evaluation Metrics / M. Sundaram, A. Mani // Face Recognition - Semisupervised Classification, Subspace Projection and Evaluation Method: IntechOpen, 2013. P. 137–144.

21. Spatial Localization and Detection [Електронний ресурс]. – Режим доступу до ресурсу: http://cs231n.stanford.edu/slides/2016/ winter1516_lecture8.pdf.

22. Задача нахождения объектов на изображении [Електронний ресурс]. – Режим доступу до ресурсу: https://neerc.ifmo.ru/wiki/index.php?title= Задача_нахождения_объектов_на_изображении.

23. Hosang J. What Makes for Effective Detection Proposals? / J. Hosang, R. Benenson, P. Dollar, B. Schiele // IEEE Transactions on Pattern Analysis and Machine Intelligence: arXiv, 2016. P. 814–830.

24. Convolutional Neural Networks [Електронний ресурс]. – Режим доступу до ресурсу: https://www.coursera.org/learn/convolutional-neural-networks.

25. Hosang J. Learning non-maximum suppression / J. Hosang, R. Benenson, B. Schiele // Proceedings - 30th IEEE Conference on Computer Vision Pattern Recognition: arXiv, 2017. P. 6469–6477.

26. Gong M. A review of non-maximum suppression algorithms for deep learning target detection / M. Gong, D. Wang, X. Zhao, H. Guo, D. Luo, M. Song // Seventh Symposium on Novel Photoelectronic Detection Technologyand Application 2020, Kunming, China: SPIE, 2020. P. 133-141.

27. One Shot learning, Siamese networks and Triplet Loss with Keras [Електронний ресурс]. – Режим доступу до ресурсу: https://medium.com/@crimy/one-shot-learning-siamese-networks-and-triplet-loss-with-keras-2885ed022352.

28. Функции потерь для задачи распознавания лиц [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/company/ ntechlab/blog/531842/.

29. Girshick R. Rich feature hierarchies for accurate object detection and semantic segmentation / R. Girshick, J. Donahue, T. Darrell, J. Malik // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2014. P. 580–587.

30. Uijlings J. R. R. Selective search for object recognition / J. R. R. Uijlings, K. E. A. Van De Sande, T. Gevers, A. W. M. Smeulders // International Journal of Computer Visision: Springer, 2013. P. 154-171.

31. Evgeniou T. Support vector machines: Theory and applications / T. Evgeniou, M. Pontil // Lecture Notes in Computer Science: Springer, 2001. P. 249–257.

32. Girshick R. Fast R-CNN / R. Girshick // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2015. P. 1440–1448.

33. Ren S. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks / S. Ren, K. He, R. Girshick, J. Sun // IEEE Transactions on Pattern Analysis and Machine Intelligence: arXiv, 2017. P. 1137–1149.

34. Redmon J. You only look once: Unified, real-time object detection / J. Redmon, S. Divvala, R. Girshick, A. Farhadi // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2016. P. 1-10.

35. Zhang K. Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks / K. Zhang, Z. Zhang, Z. Li, Y. Qiao // IEEE Signal Processing Letters: arXiv, 2016. P. 1499–1503.

36. Lin T. Y. Focal Loss for Dense Object Detection / T. Y. Lin, P. Goyal, R. Girshick, K. He, P. Dollar // IEEE Transactions on Pattern Analysis and Machine Intelligence: arXiv, 2020. P. 318–327.

37. Lin T. Y. Microsoft COCO: Common objects in context / T. Y. Lin // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): arXiv, 2014. P. 740–755.

38. Schroff F. FaceNet: A unified embedding for face recognition and clustering / F. Schroff, D. Kalenichenko, J. Philbin // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2015. P. 815–823.

39. FaceNet — пример простой системы распознавания лиц с открытым кодом Github [Електронний ресурс]. – Режим доступу до ресурсу: https://neurohive.io/ru/tutorial/raspoznavanie-lica-facenet/.

40. Amos B. Openface: A general-purpose face recognition library with mobile applications / B. Amos, B. Ludwiczuk // School of Computer Science, Carnegie Mellon University, Pittsburgh. – 2016. – №16. – P. 1–18.

41. Deng J. ArcFace: Additive angular margin loss for deep face recognition / J. Deng, J. Guo, N. Xue, S. Zafeiriou // Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition: arXiv, 2019. P. 4685–4694.

42. TensorFlow [Електронний ресурс]. – Режим доступу до ресурсу: https://www.tensorflow.org.

43. Keras [Електронний ресурс]. – Режим доступу до ресурсу: https://keras.io/.

44. Project Jupyter [Електронний ресурс]. – Режим доступу до ресурсу: https://jupyter.org/.

45. NVIDIA Developer [Електронний ресурс]. – Режим доступу до ресурсу: https://developer.nvidia.com/.

46. Что такое тензорные ядра: вычисления со смешанной точностью [Електронний ресурс]. – Режим доступу до ресурсу: https://habr.com/ru/post/512816/.

47. Google Colaboratory [Електронний ресурс]. – Режим доступу до ресурсу: https://colab.research.google.com.

48. Visual Studio Code [Електронний ресурс]. – Режим доступу до ресурсу: https://code.visualstudio.com/.

49. Labeled Faces in the Wild [Електронний ресурс]. – Режим доступу до ресурсу: http://vis-www.cs.umass.edu/lfw/.

50. Ouanan H. Development of Deep Learning-Based Facial Recognition System / H. Ouanan, A. Gaga, O. Diouri, M. Ouanan, B. Aksasse // Advances in Intelligent Systems and Computing: Springer, 2020. P. 45–52.

51. Pins Face Recognition [Електронний ресурс]. – Режим доступу до ресурсу: https://www.kaggle.com/hereisburak/pins-face-recognition.

52. AT&T Database of Faces [Електронний ресурс]. – Режим доступу до ресурсу: https://www.kaggle.com/kasikrit/att-database-of-faces.

53. Xu W. A hybrid method based on dynamic compensatory fuzzy neural network algorithm for face recognition / W. Xu, E. J. Lee // International Journal of Control, Automation and Systems: Springer, 2014. P. 688–696.

54. Wang J. The effectiveness of data augmentation in image classification using deep learning / J. Wang, L. Perez // Computer Vision and Pattern Recognition: arXiv, 2017. P. 1-8.

55. Keras OpenFace [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/iwantooxxoox/Keras-OpenFace.

56. Pytorch [Електронний ресурс]. – Режим доступу до ресурсу: https://pytorch.org.

57. Ануфрієв П. О. Дослідження впливу тестових баз зображень на точність систем розпізнавання облич з різними методами класифікації/ П. О. Ануфрієв, Э. О. Башков // Наукові праці ДонНТУ. Серія «Інформатика, кібернетика та обчислювальна техніка». – 2020. – №2 (31). – С. 74-83

Додаток А

Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження

Додаток Б

Лістинг основного програмного коду

```
class AssessmentAlgorithm:

  def __init__(self, recognize_algorithm, test_dataset):
    self._recognize_algorthm = recognize_algorthm
    self._test_dataset = test_dataset

  def collect_statistics(self):
    confusion_matrix = self._collect_confusion_matrix()
    TP = confusion_matrix[‘TP’]
    FP = confusion_matrix[‘FP’]
    TN = confusion_matrix[‘TN’]
    FN = confusion_matrix[‘FN’]
    precision = self._divide_or_zero(TP, TP + FP)
    recall = self._divide_or_zero(TP, TP + FN)
    fallout = self._divide_or_zero(TN, TN + FP)
    accuracy = self._divide_or_zero(TP + TN, TP + TN + FP + FN)
    return {
      ‘threshold’: threshold,
      ‘true_positives’: TP,
      ‘false_positives’: FP,
      ‘true_negatives’: TN,
      ‘fasle_negatives’: FN,
      ‘precision’: precision,
      ‘recall’: recall,
      ‘fallout’: fallout,
      ‘accuracy’: accuracy
    }

  # Collects confusion matrix essentials. The test dataset is the images that are
  # recognized and compared with the database dataset
  def _collect_confusion_matrices():
    result[threshold] = {
      ‘TP’: 0, # true positives
      ‘FP’: 0, # false positives
      ‘TN’: 0, # true negatives
      ‘FN’: 0  # false negatives
    }
    for person in tqdm(self._test_dataset):
      for image in self._test_dataset[person]:
        recognized = self._recognize_algorithm.recognize(image)
        db_dataset = self._recognize_algorithm.get_database()
        is_db_person = person in db_dataset
        if recognized:
          if person == recognized:
            result[‘TP’] += 1
          else:
            result[‘FP’] += 1
        else:
          if person not in db_dataset:
```

```
            result['TN'] += 1
          else:
            result['FN'] += 1
    return result

  def _divide_or_zero(a, b):
    try:
      return a / b
    except ZeroDivisionError:
      return 0

from tensorflow.keras.preprocessing.image import ImageDataGenerator
import Image from Image
import DatasetPerson from DatasetPerson
import os

class Dataset:

  def __init__(self, name, image_size):
    self._name = name
    self._image_size = image_size

  def get_name(self):
    return self._name

  def get_image_size(self):
    return self._image_size

  def get_directory(self):
    return f'datasets/{self._name}'

  def load(self):
    result = []
    dataset_dir = self.get_directory()
    for directory in os.listdir(dataset_dir):
      if not os.path.isdir(f'{dataset_dir}/{directory}'):
        continue
      person_name = directory
      for filename in os.listdir(f'{dataset_dir}/{directory}'):
        image = Image.load(f'{dataset_dir}/{directory}/{filename}').resize(self._image_size)
        item = DatasetPerson(filename, image)
        result.append(item)
    return result

  def prepare_for_training():
    generator = ImageDataGenerator(rescale=1.0/255.0,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   rotation_range=40,
                                   width_shift_range=0.1,
                                   height_shift_range=0.1)
    return generator.flow_from_directory(self.get_directory,
                                         target_size=input_shape[:2],
                                         class_mode='categorical')

class DatasetPerson:
```

```
  def __init__(self, id, image):
    self._id = id
    self._image = image

  def get_id(self):
    return self._id

  def get_image(self):
    return self._image

import abc
import numpy as np

class FaceRecognitionNet(abc.ABC):
  def __init__(self, input_shape):
    self._input_shape = input_shape
    self._model = self._createModel(input_shape)
  def get_input_shape(self):
    return self._input_shape
  def encode(self, image):
    normalized_image = image.normalize()
    predictions = model.predict(np.array(normalized_image.getPixels()))
    return predictions[0]
  def load_weights(self, file):
    self._model.load_weights(file)
  def save_weights(self, file):
    self._model.save_weights(file)
  @abs.abstractmethod
  def _create_model(self, input_shape):
    pass

import numpy as np
import cv2

class Image:
  def load(path):
    data = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)
    return Image(data)
  def __init__(self, data):
    self._data = data
  def normalize(self):
    data = np.array(self._data, dtype='float32') / 255
    return Image(data)
    def resize(self, shape):
    data = cv2.resize(self._data, (shape.width(), shape.height())), interpolation=cv2.INTER_AREA)
    return Image(data)

from FaceRecognitionNet import FaceReconitionNet

class ModifiedOpenFace(FaceReconitionNet):

  def __init__(self, input_shape, model):
    super(input_shape)
    self._model = model

  def _create_model(self, input_shape):
```

```
        return self._model

    from FaceRecognitionNet import FaceReconitionNet
    from tensorflow.keras import Model
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dense, Flatten, Dropout, GlobalAveragePooling2D, AveragePooling2D, ZeroPadding2D, BatchNormalization, Activation, Lambda, concatenate
    from tensorflow.keras.optimizers import Adam, SGD
    from keras import backend as K

    class OriginalOpenFace(FaceReconitionNet):

      def _create_model(self, input_shape):
        inputs = Input(shape=input_shape)
        x = ZeroPadding2D(padding=(3, 3), input_shape=(96, 96, 3))(inputs)
        x = Conv2D(64, (7, 7), strides=(2, 2), name='conv1')(x)
        x = BatchNormalization(axis=3, epsilon=0.00001, name='bn1')(x)
        x = Activation('relu')(x)
        x = ZeroPadding2D(padding=(1, 1))(x)
        x = MaxPooling2D(pool_size=3, strides=2)(x)
        x = Lambda(LRN2D, name='lrn_1')(x)
        x = Conv2D(64, (1, 1), name='conv2')(x)
        x = BatchNormalization(axis=3, epsilon=0.00001, name='bn2')(x)
        x = Activation('relu')(x)
        x = ZeroPadding2D(padding=(1, 1))(x)
        x = Conv2D(192, (3, 3), name='conv3')(x)
        x = BatchNormalization(axis=3, epsilon=0.00001, name='bn3')(x)
        x = Activation('relu')(x)
        x = Lambda(LRN2D, name='lrn_2')(x)
        x = ZeroPadding2D(padding=(1, 1))(x)
        x = MaxPooling2D(pool_size=3, strides=2)(x)

        # Inception3a
        inception_3a_3x3 = Conv2D(96, (1, 1), name='inception_3a_3x3_conv1')(x)
        inception_3a_3x3 = BatchNormalization(axis=3, epsilon=0.00001, name='inception_3a_3x3_bn1')(inception_3a_3x3)
        inception_3a_3x3 = Activation('relu')(inception_3a_3x3)
        inception_3a_3x3 = ZeroPadding2D(padding=(1, 1))(inception_3a_3x3)
        inception_3a_3x3 = Conv2D(128, (3, 3), name='inception_3a_3x3_conv2')(inception_3a_3x3)
        inception_3a_3x3 = BatchNormalization(axis=3, epsilon=0.00001, name='inception_3a_3x3_bn2')(inception_3a_3x3)
        inception_3a_3x3 = Activation('relu')(inception_3a_3x3)

        inception_3a_5x5 = Conv2D(16, (1, 1), name='inception_3a_5x5_conv1')(x)
        inception_3a_5x5 = BatchNormalization(axis=3, epsilon=0.00001, name='inception_3a_5x5_bn1')(inception_3a_5x5)
        inception_3a_5x5 = Activation('relu')(inception_3a_5x5)
        inception_3a_5x5 = ZeroPadding2D(padding=(2, 2))(inception_3a_5x5)
        inception_3a_5x5 = Conv2D(32, (5, 5), name='inception_3a_5x5_conv2')(inception_3a_5x5)
        inception_3a_5x5 = BatchNormalization(axis=3, epsilon=0.00001, name='inception_3a_5x5_bn2')(inception_3a_5x5)
        inception_3a_5x5 = Activation('relu')(inception_3a_5x5)

        inception_3a_pool = MaxPooling2D(pool_size=3, strides=2)(x)
        inception_3a_pool = Conv2D(32, (1, 1), name='inception_3a_pool_conv')(inception_3a_pool)
        inception_3a_pool = BatchNormalization(axis=3, epsilon=0.00001,
```

```
name='inception_3a_pool_bn')(inception_3a_pool)
        inception_3a_pool = Activation('relu')(inception_3a_pool)
        inception_3a_pool = ZeroPadding2D(padding=((3, 4), (3, 4)))(inception_3a_pool)

        inception_3a_1x1 = Conv2D(64, (1, 1), name='inception_3a_1x1_conv')(x)
        inception_3a_1x1 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3a_1x1_bn')(inception_3a_1x1)
        inception_3a_1x1 = Activation('relu')(inception_3a_1x1)

        inception_3a = concatenate([inception_3a_3x3, inception_3a_5x5, inception_3a_pool,
inception_3a_1x1], axis=3)

        # Inception3b
        inception_3b_3x3 = Conv2D(96, (1, 1), name='inception_3b_3x3_conv1')(inception_3a)
        inception_3b_3x3 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_3x3_bn1')(inception_3b_3x3)
        inception_3b_3x3 = Activation('relu')(inception_3b_3x3)
        inception_3b_3x3 = ZeroPadding2D(padding=(1, 1))(inception_3b_3x3)
        inception_3b_3x3 = Conv2D(128, (3, 3), name='inception_3b_3x3_conv2')(inception_3b_3x3)
        inception_3b_3x3 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_3x3_bn2')(inception_3b_3x3)
        inception_3b_3x3 = Activation('relu')(inception_3b_3x3)

        inception_3b_5x5 = Conv2D(32, (1, 1), name='inception_3b_5x5_conv1')(inception_3a)
        inception_3b_5x5 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_5x5_bn1')(inception_3b_5x5)
        inception_3b_5x5 = Activation('relu')(inception_3b_5x5)
        inception_3b_5x5 = ZeroPadding2D(padding=(2, 2))(inception_3b_5x5)
        inception_3b_5x5 = Conv2D(64, (5, 5), name='inception_3b_5x5_conv2')(inception_3b_5x5)
        inception_3b_5x5 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_5x5_bn2')(inception_3b_5x5)
        inception_3b_5x5 = Activation('relu')(inception_3b_5x5)

        inception_3b_pool = Lambda(lambda x: x**2, name='power2_3b')(inception_3a)
        inception_3b_pool = AveragePooling2D(pool_size=(3, 3), strides=(3, 3))(inception_3b_pool)
        inception_3b_pool = Lambda(lambda x: x*9, name='mult9_3b')(inception_3b_pool)
        inception_3b_pool = Lambda(lambda x: K.sqrt(x), name='sqrt_3b')(inception_3b_pool)
        inception_3b_pool = Conv2D(64, (1, 1), name='inception_3b_pool_conv')(inception_3b_pool)
        inception_3b_pool = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_pool_bn')(inception_3b_pool)
        inception_3b_pool = Activation('relu')(inception_3b_pool)
        inception_3b_pool = ZeroPadding2D(padding=(4, 4))(inception_3b_pool)

        inception_3b_1x1 = Conv2D(64, (1, 1), name='inception_3b_1x1_conv')(inception_3a)
        inception_3b_1x1 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3b_1x1_bn')(inception_3b_1x1)
        inception_3b_1x1 = Activation('relu')(inception_3b_1x1)

        inception_3b = concatenate([inception_3b_3x3, inception_3b_5x5, inception_3b_pool,
inception_3b_1x1], axis=3)

        # Inception3c
        inception_3c_3x3 = conv2d_bn(inception_3b,
                                     layer='inception_3c_3x3',
                                     cv1_out=128,
                                     cv1_filter=(1, 1),
                                     cv2_out=256,
```

```
                                cv2_filter=(3, 3),
                                cv2_strides=(2, 2),
                                padding=(1, 1))

        inception_3c_5x5 = conv2d_bn(inception_3b,
                                layer='inception_3c_5x5',
                                cv1_out=32,
                                cv1_filter=(1, 1),
                                cv2_out=64,
                                cv2_filter=(5, 5),
                                cv2_strides=(2, 2),
                                padding=(2, 2))

        inception_3c_pool = MaxPooling2D(pool_size=3, strides=2)(inception_3b)
        inception_3c_pool = ZeroPadding2D(padding=((0, 1), (0, 1)))(inception_3c_pool)

        inception_3c = concatenate([inception_3c_3x3, inception_3c_5x5, inception_3c_pool], axis=3)

        #inception 4a
        inception_4a_3x3 = conv2d_bn(inception_3c,
                                layer='inception_4a_3x3',
                                cv1_out=96,
                                cv1_filter=(1, 1),
                                cv2_out=192,
                                cv2_filter=(3, 3),
                                cv2_strides=(1, 1),
                                padding=(1, 1))
        inception_4a_5x5 = conv2d_bn(inception_3c,
                                layer='inception_4a_5x5',
                                cv1_out=32,
                                cv1_filter=(1, 1),
                                cv2_out=64,
                                cv2_filter=(5, 5),
                                cv2_strides=(1, 1),
                                padding=(2, 2))

        inception_4a_pool = Lambda(lambda x: x**2, name='power2_4a')(inception_3c)
        inception_4a_pool = AveragePooling2D(pool_size=(3, 3), strides=(3, 3))(inception_4a_pool)
        inception_4a_pool = Lambda(lambda x: x*9, name='mult9_4a')(inception_4a_pool)
        inception_4a_pool = Lambda(lambda x: K.sqrt(x), name='sqrt_4a')(inception_4a_pool)
        inception_4a_pool = conv2d_bn(inception_4a_pool,
                                layer='inception_4a_pool',
                                cv1_out=128,
                                cv1_filter=(1, 1),
                                padding=(2, 2))
        inception_4a_1x1 = conv2d_bn(inception_3c,
                                layer='inception_4a_1x1',
                                cv1_out=256,
                                cv1_filter=(1, 1))
        inception_4a = concatenate([inception_4a_3x3, inception_4a_5x5, inception_4a_pool,
inception_4a_1x1], axis=3)

        #inception4e
        inception_4e_3x3 = conv2d_bn(inception_4a,
                                layer='inception_4e_3x3',
                                cv1_out=160,
                                cv1_filter=(1, 1),
```

```
                        cv2_out=256,
                        cv2_filter=(3, 3),
                        cv2_strides=(2, 2),
                        padding=(1, 1))
inception_4e_5x5 = conv2d_bn(inception_4a,
                        layer='inception_4e_5x5',
                        cv1_out=64,
                        cv1_filter=(1, 1),
                        cv2_out=128,
                        cv2_filter=(5, 5),
                        cv2_strides=(2, 2),
                        padding=(2, 2))
inception_4e_pool = MaxPooling2D(pool_size=3, strides=2)(inception_4a)
inception_4e_pool = ZeroPadding2D(padding=((0, 1), (0, 1)))(inception_4e_pool)

inception_4e = concatenate([inception_4e_3x3, inception_4e_5x5, inception_4e_pool], axis=3)

#inception5a
inception_5a_3x3 = conv2d_bn(inception_4e,
                        layer='inception_5a_3x3',
                        cv1_out=96,
                        cv1_filter=(1, 1),
                        cv2_out=384,
                        cv2_filter=(3, 3),
                        cv2_strides=(1, 1),
                        padding=(1, 1))

inception_5a_pool = Lambda(lambda x: x**2, name='power2_5a')(inception_4e)
inception_5a_pool = AveragePooling2D(pool_size=(3, 3), strides=(3, 3))(inception_5a_pool)
inception_5a_pool = Lambda(lambda x: x*9, name='mult9_5a')(inception_5a_pool)
inception_5a_pool = Lambda(lambda x: K.sqrt(x), name='sqrt_5a')(inception_5a_pool)
inception_5a_pool = conv2d_bn(inception_5a_pool,
                        layer='inception_5a_pool',
                        cv1_out=96,
                        cv1_filter=(1, 1),
                        padding=(1, 1))
inception_5a_1x1 = conv2d_bn(inception_4e,
                        layer='inception_5a_1x1',
                        cv1_out=256,
                        cv1_filter=(1, 1))

inception_5a = concatenate([inception_5a_3x3, inception_5a_pool, inception_5a_1x1], axis=3)

#inception_5b
inception_5b_3x3 = conv2d_bn(inception_5a,
                        layer='inception_5b_3x3',
                        cv1_out=96,
                        cv1_filter=(1, 1),
                        cv2_out=384,
                        cv2_filter=(3, 3),
                        cv2_strides=(1, 1),
                        padding=(1, 1))
inception_5b_pool = MaxPooling2D(pool_size=3, strides=2)(inception_5a)
inception_5b_pool = conv2d_bn(inception_5b_pool,
                        layer='inception_5b_pool',
                        cv1_out=96,
                        cv1_filter=(1, 1))
```

```
    inception_5b_pool = ZeroPadding2D(padding=(1, 1))(inception_5b_pool)

    inception_5b_1x1 = conv2d_bn(inception_5a,
                                 layer='inception_5b_1x1',
                                 cv1_out=256,
                                 cv1_filter=(1, 1))
    inception_5b = concatenate([inception_5b_3x3, inception_5b_pool, inception_5b_1x1], axis=3)

    av_pool = AveragePooling2D(pool_size=(3, 3), strides=(1, 1))(inception_5b)
    reshape_layer = Flatten()(av_pool)
    x = Dense(128, name='dense_layer')(reshape_layer)
    x = Lambda(lambda  x: K.l2_normalize(x, axis=1))(x)
    model = Model(inputs=inputs, outputs=x)
    return model

import numpy as np

def euclidean(x, y):
  return np.linalg.norm(x - y)

def unit_vector(vector):
  """Returns the unit vector of the vector"""
  return vector / np.linalg.norm(vector)

def cosine(v1, v2):
  """Returns the angle in radians between vectors 'v1' and 'v2'::

    >>> angle_between((1, 0, 0), (0, 1, 0))
    1.5707963267948966
    >>> angle_between((1, 0, 0), (1, 0, 0))
    0.0
    >>> angle_between((1, 0, 0), (-1, 0, 0))
    3.141592653589793
  """
  v1_u = unit_vector(v1)
  v2_u = unit_vector(v2)
  return np.arccos(np.clip(np.dot(v1_u, v2_u), -1.0, 1.0))

class RecognizeAlgorithm:

  def __init__(self, database, recognition_net, threshold):
    self._database = database
    self._recognition_net = recognition_net
    self._theshold = threshold

  def get_databse(self):
    return self._database

  def get_recognition_net(self):
    return self._recognition_net

  def get_threshold(self):
    return self._threshold

  def recognize(self, person):
    comparison = self._compare(person, euclidean, disable_progress=True)[0]
    if comparison['distance'] < self._threshold:
```

```
            return comparison
          return None

        # Helper function that compares a test image against the test dataset
        # and returns comparatons sorted by distance in ascending order
        def _compare(test_image, test_dataset, comparator, disable_progress=False):
          model = self._recognition_net
          test_dataset = self._database
          test_vector = model.encode(np.array([test_image.get_pixels()]))[0]
          comparations = []
          input_data = []
          for person_name in tqdm(test_dataset, disable=disable_progress):
            for i in range(len(test_dataset[person_name])):
              compare_image = test_dataset[person_name][i]
              comparations.append({'person': person_name, 'file': compare_image})
              input_data.append(compare_image.get_pixels())

          predictions = model.encode(np.array(input_data))
          for comparison, compare_vector in zip(comparations, predictions):
            distance = comparator(test_vector, compare_vector)
            comparison['distance'] = distance

          comparations = sorted(comparations, key=lambda x: x['distance'])
          return comparations

      class Shape:

        def __init__(self, width, height):
          self._width = width
          self._height = height

        def get_width(self):
          return self._width

        def get_height(self):
          return self._height

      from FaceRecognitionNet import FaceReconitionNet
      from tensorflow.keras import Model
      from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dense, Flatten, Dropout,
GlobalAveragePooling2D, AveragePooling2D, ZeroPadding2D, BatchNormalization, Activation,
Lambda, concatenate
      from tensorflow.keras.optimizers import Adam, SGD
      from keras import backend as K
      from TrainHistory import TrainHistory
      from ModifiedOpenFace import ModifiedOpenFace

      class TrainableOpenFace(FaceReconitionNet):

        def __init__(self, input_shape, class_amount):
          super(input_shape)
          self._class_amount = class_amount

        def get_class_amount(self):
          return self._class_amount
```

```
        def train(self, parameters):
          train_dataget = parameters[‘train_dataset’].prepare_for_training()
          test_dataset = parameters[‘validation_dataset’].preprare_for_training()
          history = model.fit(train_generator, epochs=500, verbose=1,
                    validation_data=test_generator)
          return TrainHistory(history)

        def to_useable_net(self):
          useable_net = self._remove_last_layer(self._model)
          return ModifiedOpenFace(self._input_shape, useable_net)

        def _remove_last_layer(self, model):
          return Model(model.input, model.layers[-2].output)

        def _create_model(self, input_shape):
          inputs = Input(shape=input_shape)
          x = ZeroPadding2D(padding=(3, 3), input_shape=(96, 96, 3))(inputs)
          x = Conv2D(64, (7, 7), strides=(2, 2), name=‘conv1’)(x)
          x = BatchNormalization(axis=3, epsilon=0.00001, name=‘bn1’)(x)
          x = Activation(‘relu’)(x)
          x = ZeroPadding2D(padding=(1, 1))(x)
          x = MaxPooling2D(pool_size=3, strides=2)(x)
          x = Lambda(LRN2D, name=‘lrn_1’)(x)
          x = Conv2D(64, (1, 1), name=‘conv2’)(x)
          x = BatchNormalization(axis=3, epsilon=0.00001, name=‘bn2’)(x)
          x = Activation(‘relu’)(x)
          x = ZeroPadding2D(padding=(1, 1))(x)
          x = Conv2D(192, (3, 3), name=‘conv3’)(x)
          x = BatchNormalization(axis=3, epsilon=0.00001, name=‘bn3’)(x)
          x = Activation(‘relu’)(x)
          x = Lambda(LRN2D, name=‘lrn_2’)(x)
          x = ZeroPadding2D(padding=(1, 1))(x)
          x = MaxPooling2D(pool_size=3, strides=2)(x)

          # Inception3a
          inception_3a_3x3 = Conv2D(96, (1, 1), name=‘inception_3a_3x3_conv1’)(x)
          inception_3a_3x3 = BatchNormalization(axis=3, epsilon=0.00001,
name=‘inception_3a_3x3_bn1’)(inception_3a_3x3)
          inception_3a_3x3 = Activation(‘relu’)(inception_3a_3x3)
          inception_3a_3x3 = ZeroPadding2D(padding=(1, 1))(inception_3a_3x3)
          inception_3a_3x3 = Conv2D(128, (3, 3), name=‘inception_3a_3x3_conv2’)(inception_3a_3x3)
          inception_3a_3x3 = BatchNormalization(axis=3, epsilon=0.00001,
name=‘inception_3a_3x3_bn2’)(inception_3a_3x3)
          inception_3a_3x3 = Activation(‘relu’)(inception_3a_3x3)

          inception_3a_5x5 = Conv2D(16, (1, 1), name=‘inception_3a_5x5_conv1’)(x)
          inception_3a_5x5 = BatchNormalization(axis=3, epsilon=0.00001,
name=‘inception_3a_5x5_bn1’)(inception_3a_5x5)
          inception_3a_5x5 = Activation(‘relu’)(inception_3a_5x5)
          inception_3a_5x5 = ZeroPadding2D(padding=(2, 2))(inception_3a_5x5)
          inception_3a_5x5 = Conv2D(32, (5, 5), name=‘inception_3a_5x5_conv2’)(inception_3a_5x5)
          inception_3a_5x5 = BatchNormalization(axis=3, epsilon=0.00001,
name=‘inception_3a_5x5_bn2’)(inception_3a_5x5)
          inception_3a_5x5 = Activation(‘relu’)(inception_3a_5x5)

          inception_3a_pool = MaxPooling2D(pool_size=3, strides=2)(x)
          inception_3a_pool = Conv2D(32, (1, 1), name=‘inception_3a_pool_conv’)(inception_3a_pool)
```

```
        inception_3a_pool = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3a_pool_bn')(inception_3a_pool)
        inception_3a_pool = Activation('relu')(inception_3a_pool)
        inception_3a_pool = ZeroPadding2D(padding=((3, 4), (3, 4)))(inception_3a_pool)

        inception_3a_1x1 = Conv2D(64, (1, 1), name='inception_3a_1x1_conv')(x)
        inception_3a_1x1 = BatchNormalization(axis=3, epsilon=0.00001,
name='inception_3a_1x1_bn')(inception_3a_1x1)
        inception_3a_1x1 = Activation('relu')(inception_3a_1x1)

        inception_3a = concatenate([inception_3a_3x3, inception_3a_5x5, inception_3a_pool,
inception_3a_1x1], axis=3)

        av_pool = AveragePooling2D(pool_size=(3, 3), strides=(1, 1))(inception_3a)
        reshape_layer = Flatten()(av_pool)
        x = Dense(128, name='dense_layer')(reshape_layer)
        x = Lambda(lambda  x: K.l2_normalize(x, axis=1))(x)
        x = Dense(classes, activation='softmax')(x)
        model = Model(inputs=inputs, outputs=x)
        model.compile(loss='categorical_crossentropy',
                optimizer=Adam(learning_rate=0.0001),
                metrics=['accuracy'])
        return model

    class TrainHistory:

      def __init__(self, history):
        self._history = history

      def get_accuracy(self):
        return self._history['accuracy']

      def get_loss(self):
        return self._history['loss']

      def get_validation_accuracy(self):
        return self._history['validation_accuracy']

      def get_validation_loss(self):
        return self._validation_loss['validation_loss']
```

Додаток В

Роздатковий матеріал

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
ФАКУЛЬТЕТ КОМП'ЮТЕРНО-ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА АВТОМАТИЗАЦІЇ
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

Випускна кваліфікаційна робота магістра
на тему:

«Удосконалення технології розпізнавання облич за допомогою згорткових нейронних мереж»

Автор:
ст. гр. КНм-20
Мякшин О. Д.

Керівник:
проф. каф. ПМІ, д.т.н.
Башков Є. О.

Рисунок В.1 – Титульний аркуш презентації

1 ЗАГАЛЬНА ПОСТАНОВКА ЗАДАЧІ

Мета дослідження: пошук та реалізація підходів удосконалення системи розпізнавання облич, які дозволяють розпізнавати об'єкти з підвищеною точністю розпізнавання без зниження швидкості

Об'єкт дослідження: процес розпізнавання облич за допомогою згорткових нейронних мереж

Предмет дослідження: архітектури та можливості згорткових нейронних мереж для вирішення задачі розпізнавання облич на двовимірних зображеннях

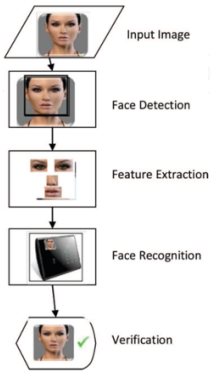
Наукова новизна: отримання більш ефективної архітектури згорткової нейронної мережі, яка дозволяє підвищити точність розпізнавання облич

Завдання:

- вивчити теоритетичні відомості про предметну область
- вибрати архітектуру мережі та модифікації, виконати її навчання
- виконати оцінку якості розпізнавання розробленої мережі

Рисунок В.2 – Перший плакат презентації

2 АРХІТЕКТУРА СИСТЕМИ РОЗПІЗНАВАННЯ ОБЛИЧ



Джерело: [1]

Детекція облич на зображенні

- знаходження координат обмежуючих рамок
- підходи реалізації поділяються на одно- та двоетапні

Виділення признаків кожного обличчя

- найбільш значимі признаки, представлені у вигляді вектору чисел, що сжато характеризує обличчя

Ідентифкація/верифікація

- порівняння векторів признаків для зображень облич між собою за відстаню або кутом між векторами

Рисунок В.3 – Другий плакат презентації

3 ВИДІЛЕННЯ ПРИЗНАКІВ

Головна задача: навчити нейронну мережу для зображень однієї людини видавати схожі вектори признаків, а для різних людей – різні вектори.

Існуючі підходи

1. Використання функції помилки Triplet Loss (використовується у мережі FaceNet и OpenFace)

$$\mathcal{L}(A,P,N) = \max\left(\|\mathrm{f}(A)-\mathrm{f}(P)\|^2 - \|\mathrm{f}(A)-\mathrm{f}(N)\|^2 + \alpha, 0\right)$$

Джерело: [3]

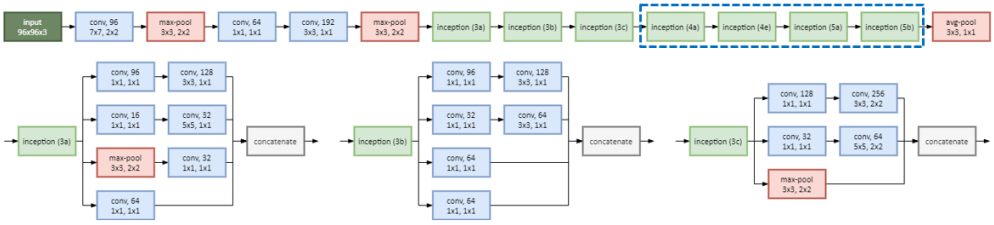
2. Навчання класифікатору

- навчити мережу класифікувати фіксований набір людей
- після навчання видалити останній (класифікуючий) шар
- мережа, що залишилась, вже вміє правильно виділяти признаки

Рисунок В.4 – Третій плакат презентації

4 МОДИФІКАЦІЯ МЕРЕЖІ

БАЗОВА СТРУКТУРА МЕРЕЖІ OPENFACE



Джерело: Авторський рисунок на основі [4]

Модификація:

- навчання через Triplet Loss замінено на навчання через класифікатор
- спрощено структуру мережі: видалені 4 и 5 группи inception модулів

Рисунок В.5 – Четвертий плакат презентації

5 ВИКОРИСТАНІ ДАТАСЕТИ

1. Pins Face (навчання)

- зображення облич знаменитостей, зібраний з мережі Інтернет
- 17534 кольорових та чорно-білих зображень облич 105 людей (класів)

2. Labeled Faces in the Wild (LFW) (навчання)

- зібран з фотографій людей, доступних в мережі Інтернет, за допомогою детектору
- 13233 кольорових зображень облич 5749 людей (класів)

3. ORL Faces (тестування)

- створений в лабораторних умовах в різний час, освітленні, виразі обличчя
- 40 людей (классів), у кожного по 10 чорно-білих зображення

Рисунок В.6 – П’ятий плакат презентації

6 РОЗРОБКА ТА НАВЧАННЯ МОДЕЛЕЙ

Основні бібліотеки та засоби:

- модель мережі створена та навчена на Keras
- розробка та експерименти проводились на Google Colab
- навчання виконувалось на GPU комп’ютера кафедри ПМІ за допомогою Tensorflow

Технічні характеристики робочої станції кафедри ПМІ

Тип	Назва	Специфікація
Назва станції	Комплект науково – дослідного обладнання для лабораторії «Leader-Pro»	Ін. № 1014600338
Процесор	AMD Ryzen Threadripper 2990WX	32 core, 64 treads, 4 channel, 3.0GHz
Відеокарта	MSI GeForce RTX 2080 Ti Gaming Trio	11 GB, GDDR6 1755 MHz / 1350 MHz 4352 cores Підтримка CUDA
Материнська плата	ASUS ROG ZENITH II EXTREME ALPHA	AMD sTRX4, 8 x DDR4, Intel XMP, AMP
Операційна система	Windows 10 Pro	x64

Рисунок В.7 – Шостий плакат презентації

6 РОЗРОБКА ТА НАВЧАННЯ МОДЕЛЕЙ

ОЦІНКА ШВИДКОСТІ НАВЧАННЯ

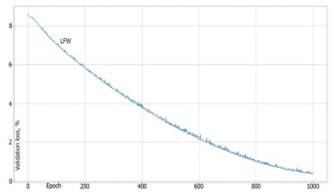
	Робоча станція кафедри ПМІ	Виділена машина Google Colab
100 епох на CPU	6,4 годин	13 годин
100 епох на GPU	1,3 годин	2,3 годин
100 епох на TPU	–	13,3 годин

Результати оцінки:

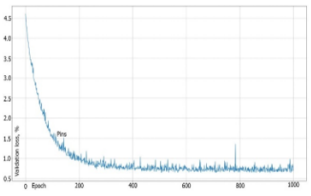
- ✓ GPU показує найкращі показники швидкості на обох машинах
- ✓ На комп’ютері кафедри ПМІ мережа навчається швидше
- ✓ При експериментах на Google Colab краще віддавати перевагу GPU

Рисунок В.8 – Сьомий плакат презентації

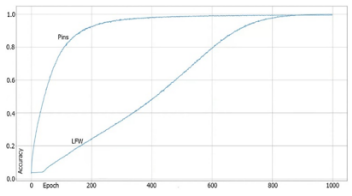
6 РОЗРОБКА ТА НАВЧАННЯ МОДЕЛЕЙ



Помилка на датасеті валідації LFW



Помилка на датасеті валідації PinsFace



Точність навчання на використаних датасетах

Результати навчання:

- всього було навчено дві моделі: на LFW та PinsFace
- навчання кожної з моделей тривало 1000 епох (приблизно день)
- **умова зупинки**: припинення покращення точності протягом великої кількості епох навчання
- на датасеті LFW достатнім може бути 850-900 епох
- на датасеті PinsFace достатнім може бути близько 500 епох

Рисунок В.9 – Восьмий плакат презентації

7 ТЕСТУВАННЯ МОДЕЛЕЙ

МЕТРИКИ ОЦІНКИ – CONFUSION MATRIX

	Істинно позитивні	Істинно негативні
Позитивне розпізнавання	True Positive (TP) Розпізнавання вірно	False Positive (FP) Розпізнавання невірно
Негативне розпізнавання	False Negative (FN) Розпізнавання невірно	True Negative (TN) Розпізнавання вірно

- ✓ True Positive приклади мережа вірно співвіднесла до людей в БД
- ✓ True Negative приклади мережа не співвіднесла до людей, яких немає в БД
- ✗ False Positive приклади мережа невірно співвіднесла до людей в БД
- ✗ False Negative приклади мережа не співвіднесла до людей, що є в БД

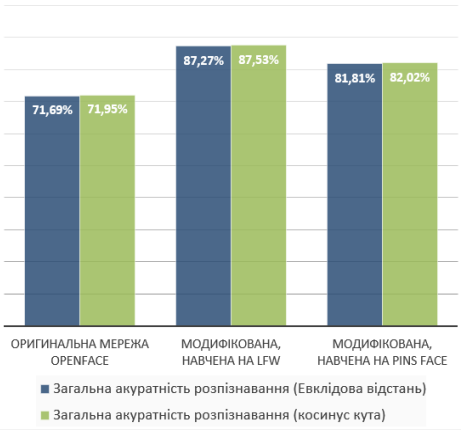
$$\text{Загальна акуратність розпізнавання} = \frac{TP+TN}{TP+TN+FP+FN}$$

Джерело: [1]

Рисунок В.10 – Дев’ятий плакат презентації

7 ТЕСТУВАННЯ МОДЕЛЕЙ

РЕЗУЛЬТАТИ ОЦІНКИ ЯКОСТІ РОЗПІЗНАВАННЯ



Умови тестування:

- навчені моделі порівнювались з оригінальною навченою мережею OpenFace
- використовувалось порівняння векторів признаків за Евклідовою відстанню та косинусом кута між ними

Результати оцінки:

- ✓ загальна акуратність мережі покращилась на 10-16%
- ✓ мережа, навчена на датасеті з більшою кількістю класів (LFW) дає кращі результати в порівнянні з навченою мережею на Pins Face, де класів менше
- ✓ між використанням Евклідової відстані та косинуса кута майже немає різниці

Рисунок В.11 – Десятий плакат презентації

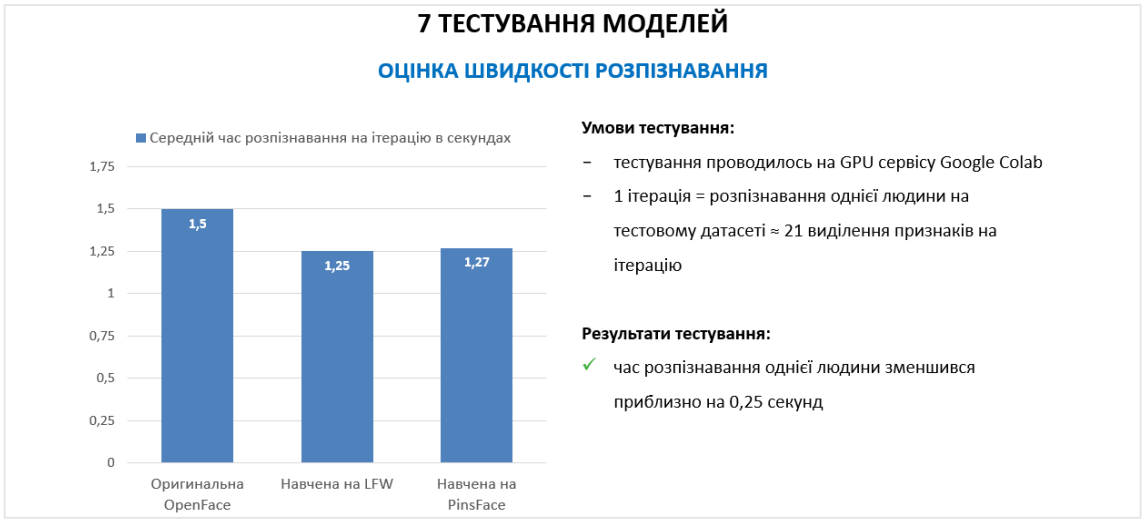


Рисунок В.12 – Одинадцятий плакат презентації

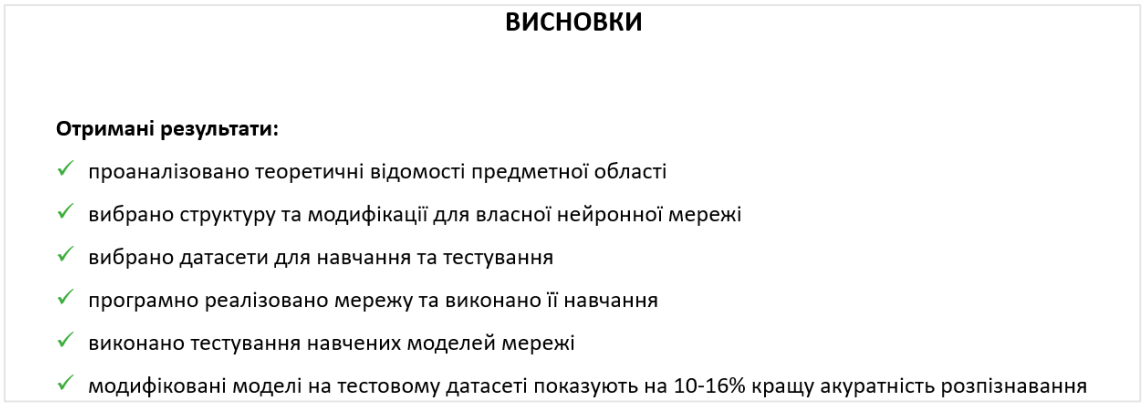


Рисунок В.13 – Дванадцятий плакат презентації

Рисунок В.14 – Тринадцятий плакат презентації

СПИСОК ПУБЛІКАЦІЙ

1. Мякшин О. Д. Архітектури згорткових нейронних мереж для задачі розпізнавання облич / О. Д. Мякшин, П. О. Ануфрієв // Наукові досягнення та відкриття сучасної молоді. Матеріали міжнародної науково-практичної конференції (2021). – С. 62 – 64.

2. Мякшин О. Д. Модифікація архітектури згорткової нейронної мережі OpenFace для системи розпізнавання облич / О. Д. Мякшин, П. О. Ануфрієв // «ТАК»: телекомунікації, автоматика, комп'ютерно-інтегровані технології (2021).

3. Miakshyn A. Face recognition technology improving using convolutional neural networks / A. Miakshyn, P. Anufriev, Y. Bashkov // Advanced Trends in Information Theory (ATIT). Proc. of 2021 IEEE 3rd International Conference (2021).

Рисунок В.15 – Чотирнадцятий плакат презентації